ONTOLOGY MERGING USING SEMANTICALLY-DEFINED MERGE CRITERIA AND OWL REASONING SERVICES: TOWARDS EXECUTION-TIME MERGING OF MULTIPLE CLINICAL WORKFLOWS TO HANDLE COMORBIDITIES

by

Borna Jafarpour

Submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia
December 2013

To my parents – I hope I have made you proud.

Contents

List of Figures

List of Tables

Abstract

Semantic web based decision support systems represent domain knowledge using ontologies that capture the domain concepts, their relationships and instances. Typically, decision support systems use a single knowledge model—i.e. a single ontology—which at times restricts the knowledge coverage to only select aspects of the domain knowledge. The integration of multiple knowledge models—i.e. multiple ontologies—provides a holistic knowledge model that encompasses multiple perspectives, orientations and instances. The challenge is the execution-time merging of multiple ontologies whilst maintaining knowledge consistency and procedural validity. Knowledge morphing aims at the intelligent merging of multiple computerized knowledge artifacts—represented as distinct ontological models—in order to create a holistic and networked knowledge model. In our research, we have investigated and developed a knowledge morphing framework—termed as ***OntoMorph***—that supports ontology merging through: **(1) Ontology Reconciliation** whereby we harmonize multiple ontologies in terms of their vocabularies, knowledge coverage, and description granularities; **(2) Ontology Merging** where multiple reconciled ontologies are merged into a single merged ontology. To achieve ontology merging, we have formalized a set of semantically-defined merging criteria that determine ontology merge points, and describe the associated process-specific and knowledge consistency constraints that need to be satisfied to ensure consistent ontology merging; and **(3) Ontology Execution** whereby we have developed logic-based execution engines for both execution-time ontology merging and the execution of the merged ontology to infer knowledge-based recommendations. We have utilized OWL reasoning services, for efficient and decidable reasoning, to execute an OWL ontology. We have applied the *OntoMorph* framework for clinical decision support, more specifically to achieve the dynamic merging of multiple clinical practice guidelines in order to handle comorbid situations where a patient may have multiple diseases and hence multiple clinical guidelines are to be simultaneously operationalized. We have demonstrated the execution time merging of ontologically-modelled clinical guidelines, such that the decision support recommendations are derived from multiple, yet merged, clinical guidelines such that the inferred recommendations are clinically consistent. The thesis contributes new methods for ontology reconciliation, merging and execution, and presents a solution for execution-time merging of multiple clinical guidelines.

List of Abbreviations and Symbols Used

$\boldsymbol{\mathcal{A}}$	= Set of newly created ontology instances
$\boldsymbol{\mathcal{C}}$	= Set of ontology concepts
$\boldsymbol{\mathcal{I}}$	= Set of ontology instances
$\boldsymbol{\mathcal{L}}$	= Set of ontology literals
$\boldsymbol{\mathcal{M}_p}$	= Set of Meta properties
$\boldsymbol{\mathcal{M}_c}$	= Set of Meta classes
$\boldsymbol{\mathcal{O}}$	= An ontology
$\boldsymbol{\mathcal{R}}$	= Set of ontology relations
$\boldsymbol{\mathcal{V}}$	= Vocabulary of the ontology

$\mathbb{I}$	= Set of transformed ontology instances
$\mathbb{D}$	= Set of mappings between two ontologies discovered by a reasoning algorithm
$\mathbb{M}$	= Set of mappings between two ontologies
$\mathbb{P}$	= Set of morphing constructs between two ontologies
$\mathbb{R}$	= Set of inferred triples based on a reasoning algorithm

AF	= Atrial Fibrillation
C-OWL	= Contextualizing OWL
CDSS	= Clinical Decision Support System
CHF	= Chronic Heart Failure
CP	= Clinical Pathway
CPG	= Clinical Practice Guideline
DKO	= Domain Knowledge Ontology
DSS	= Decision Support System
DL	= Description Logic
GLIF	= Guideline Interchange Format
HL7	= Health Level 7

KMO	= Knowledge Mapping Ontology
KPO	= Knowledge Morphing Ontology
OWL	= Web Ontology Language
RDF	= Resource Description Format
SWRL	= Semantic Web Rule Language
TIA	= Transient Ischemic Attack
URI	= Unique Resource Identifier

Acknowledgements

I express my deepest gratitude to my supervisor, Dr. Raza Abidi, for his excellent guidance, caring and patience. He provided me with an excellent atmosphere for research, patiently corrected my writing, and technically supported my research.

I would also like to thank my examining committee: Dr. Samina Abidi, Dr. Christian Blouin and Dr. Michael Shepherd and my external examiner Dr. David Riano for their interest in my research and their valuable feedback. I am especially thankful to Samina Abidi for her constructive discussions and providing me with several useful test cases without which I could not have evaluated my research.

I am also grateful to my parents, Dr. Mojtaba Jafarpour and Zahra Mazloomdoost as well as my brother Seena Jafarpour; without their resolute patronage, I would not have reached this standing. It was difficult to be 10,000 kilometers apart from my loved ones but they made me feel close to their minds and hearts at all times.

CHAPTER 1: INTRODUCTION

1.1. Problem Description

Decision Support Systems (DSS) aim to facilitate decision making in many areas such as medicine [1], business [95], management [96], etc. Computerized DSS can be categorized as data-driven and knowledge-based DSS [5]. In knowledge-based DSS, the knowledge artifacts are modeled in a computer understandable format and used for decision making. Knowledge artifacts are explicit/published knowledge sources in a specific domain area. Knowledge-based DSS use the computerized knowledge sources to develop a model using logic-based knowledge representation languages, and then use logic-based knowledge reasoners to derive recommendations based on the problems' state. Examples of ontology based DSS can be seen in many domain areas such as medicine [107][134], law [8], network security [6], oil-and-gas production [7], electronic issue management systems [9]. Ontologies are among the most popular languages for computerized knowledge representation [2]. Web Ontology Language (OWL) [142], has gained great popularity in recent years as a knowledge representation formalism in knowledge-based DSS [3] specially clinical DSS [4].

Knowledge artifacts used in decision support in every domain may come in different formats and representation formalisms. For instance, Clinical Pathways (CP) and Clinical Practice Guidelines (CPG) are two types of medical knowledge artifacts used for decision making in health-care. These knowledge artifacts and the ontologies represent them in a computer understandable format are different [37] in their (1) vocabularies, (2) coverage of concepts; (3) description granularities—this happens when two ontologies try to cover the same part of the domain knowledge with different levels of details and (4) points of view: knowledge sources of the same domain may be described differently according to the point of view of the user of the ontology. For instance, CPG developed for physicians and nurses for the same disease are different. Thus, different knowledge artifacts complete each other and collectively provide a holistic view of the domain knowledge [10][12].

In the same way that a domain expert takes into consideration several heterogeneous knowledge sources for decision making, a DSS can benefit from using multiple computerized knowledge sources; however, conventional knowledge-based DSS are able to make their decisions based on only a single knowledge source and do not accommodate the use of multiple knowledge sources [12]. Research in knowledge synthesis, referred as knowledge morphing aims at integrating several heterogeneous knowledge sources and creating a consistent holistic view of the domain knowledge. Knowledge Morphing is defined as "**The intelligent and autonomous fusion/integration of contextually, conceptually and functionally related knowledge objects that may exist in different representation modalities and formalisms, in order to establish a comprehensive, multi-faceted and networked view of all knowledge pertaining to a domain-specific problem**" [97]. Therefore, we argue that the key element in providing decision making based on several knowledge sources is to morph the knowledge encapsulated within different knowledge artifacts, and use the morphed knowledge as the knowledge base in a DSS. We call such a DSS environment as a Knowledge Morphing Framework. Developing a knowledge morphing framework is challenging because challenges with regards to integration of heterogeneous semantic web ontologies, formally capturing the procedural aspects of the domain knowledge (i.e. execution semantics) and morphing ontologically modelled knowledge artifacts to realize a unified knowledge object.

1.2. Research Objectives and Challenges

The objective of this thesis is to develop a semantic web based knowledge morphing framework that allows the specialized merging of knowledge artifacts modeled as ontologies—i.e. to achieve the merging of multiple ontologies along user-specified and domain-specified merging constraints. From an applied perspective, the objective of this thesis is to pursue the problem of merging multiple Clinical Practice Guidelines (CPG), in a CDSS setting, to provide recommendations that conform to comorbid (i.e. the simultaneous presence of multiple diseases within a patient) situations. Concerning the research objectives, in this thesis we aim to pursue the following research challenges:

1.2.1. Knowledge Mapping

The knowledge pertaining to a problem-specific decision making activity is typically encapsulated in different types of knowledge artifacts and across several knowledge artifacts of the same type [12]. In a semantic web framework, knowledge is represented using domain-specific ontologies, therefore for a given domain one may need to work with knowledge that is represented across heterogeneous ontologies, where these ontologies may differ in their domain interpretation, concept formalization, vocabularies and usage [37]. The diversity of the knowledge representation—i.e. the encapsulation of domain-specific knowledge across different ontologies referred here as Local Knowledge Ontologies (LKO)—makes it rather infeasible to effectively reason over multiple ontologies to derive knowledge-driven decision support. Even if one pursues reasoning over a single LKO to derive a recommendation/action based the knowledge encapsulated in that particular LKO, there is still the need to intelligently synthesize the inferential results from these multiple LKO-based reasoning exercises in order to derive a consistent and comprehensive recommendation/solution that takes into account the available domain knowledge represented in terms of multiple LKO.

In a semantic web framework, ontology reconciliation techniques are used to map heterogeneous ontologies and transfer instances between them [13]. Literature reports on two different approaches for ontology reconciliation leading to reasoning over several heterogeneous LKO [13][37]: (1) Every two ontologies can be mapped to each other so that a network of connected ontologies is created; (2) All LKO can be mapped to and have their instances transformed to a unified knowledge object such as a more detailed ontology, referred as a Domain Knowledge Ontology (DKO). Please refer to section 2.2 for more details, advantages and disadvantages of each of approaches. In the latter approach, inferring a recommendation/solution based on a more comprehensive DKO is regarded to entail knowledge from multiple LKO. Therefore, the second approach seems a natural choice for our knowledge morphing framework. Unifying the representation of ontologies, with the intent to reason over them to infer recommendations/solutions, is not a straightforward task as it raises the following challenges:

1. Insufficiency of the existing ontology mapping representation languages in representing complex mappings between ontologies to facilitate automatic instance transformation between multiple LKO to a common DKO.

2. The conceptual and representational sufficiency of the DKO should be (a) rich enough to capture the available domain knowledge; and (b) detailed enough so that it is possible to align every concepts in LKO to a concept in DKO

1.2.2. Ontology Merging

Ontology merging in an ontology-based DSS can be regarded as creating a holistic view of the domain area by integrating several knowledge sources represented in different heterogeneous LKO [14]. Although each individual LKO may encapsulate a specific aspect of the domain knowledge, yet a LKO does not represent the merging constructs that may potentially exist with other LKO. Merging constructs—also referred as merging constraints—describe how the several LKO are semantically related to instantiations of other LKO and how these relations should affect the results of reasoning on them when put together as a single ontology. Existing ontology merging approaches can handle representational differences between heterogeneous LKO but do not take into account the semantic relations between instantiations of LKO that are not directly asserted in them or cannot be inferred [13][14][37]. Therefore, traditional ontology merging approaches do not facilitate knowledge morphing from a decision support perspective because even though a LKO presents a specific decision it does not render any information about how the inferred decision should be modified/interpreted in the presence of multiple LKO-driven decisions. Note that the synthesis of multiple decisions is an important aspect of knowledge morphing as one needs to reconcile the decisions from multiple LKO so that the eventual decision does not lead to a conflict situation; and that the synthesis of decisions may only be possible after the satisfaction of problem-specific metrics such as duration or cost. The challenge therefore is to synthesise, through a process of decisional reconciliation, decisions derived from inferring different LKO in terms of unified conflict-free decision. We note the following challenges with regards to ontology merging for DSS:

1. To perform automatic decision reconciliation during decision making, the necessary modifications are needed to be identified and represented in a computer interpretable format so that a computer program can load, parse, verify and execute them. The choice of the representation language and how the decision reconciliation process is encoded in that language are two challenges to be addressed. The language constructs that represent these modifications and describe the conditions under which the modification should be performed are called *morphing construct*.

1. The representation language developed for knowledge morphing in a specific domain area should be instantiated to represent the decision reconciliation process for a specific set of input ontologies used in a computerized DSS. Due to subtlety of each domain and each problem, we believe this step should be performed manually with the help of a domain expert.

2. If a reasoner is utilized to infer when and how modifications are performed, semantics of the morphing constructs should be represented in a formal language such as Description Logic. The formal description of how the recommendations by different knowledge sources, for the same input scenario, are reconciled in a reasoner to ensure a mutually consistent recommendation is called formal semantics and it is a challenge to define it.

1.2.3. Ontology Execution

Knowledge execution in an ontology-based DSS involves reasoning on the ontology (which serves as the knowledge base) to infer decisions in line with the current problem state [6][7][8][9][107][134]. In a semantic web framework, it is important to define the formal semantics of the domain knowledge in terms of a formal knowledge representation language such as Description Logic [50]. Formal semantics enables a reasoner to interpret the knowledge encapsulated in the ontology and derive the most relevant result pertaining to the current state of the domain variables [15]. It is a challenge to implement an ontology execution engine based on execution semantics represented in OWL due to the following limitations in OWL:

(1) Lack of an expression language to support data type expressivity needed for knowledge execution

(2) Decidability and performance issues

(3) Non-unique naming and open world assumptions

These limitations need to be addressed to define the semantics of a domain knowledge ontology in OWL. This task becomes more challenging in a knowledge morphing framework. Not only the execution semantics of the knowledge sources contribute to the decision made for the current state of the problem, but also the morphing semantics defined during knowledge morphing step. A knowledge execution engine should be able to dynamically find the effect of the defined morphing constructs on the reasoning process in each of knowledge sources and to combine those results in order to come up with a single decision to be shown to the user as the most appropriate decision to follow.

1.3. Solution Approach

In this thesis, we use semantic web technologies [16] to develop a knowledge morphing framework in the decision support context. Semantic Web technologies are suitable to address research challenges discussed in section 1.2 because they offer (a) a standard and formal knowledge representation language called Web Ontology Language (OWL) that can be used to capture the concepts and their underlying semantic relations in a domain of interest; (b) efficient and decidable reasoning algorithms that produce sound and complete results [17]. Our solution framework is realized through the following five general tasks as shown in Figure 1.1.

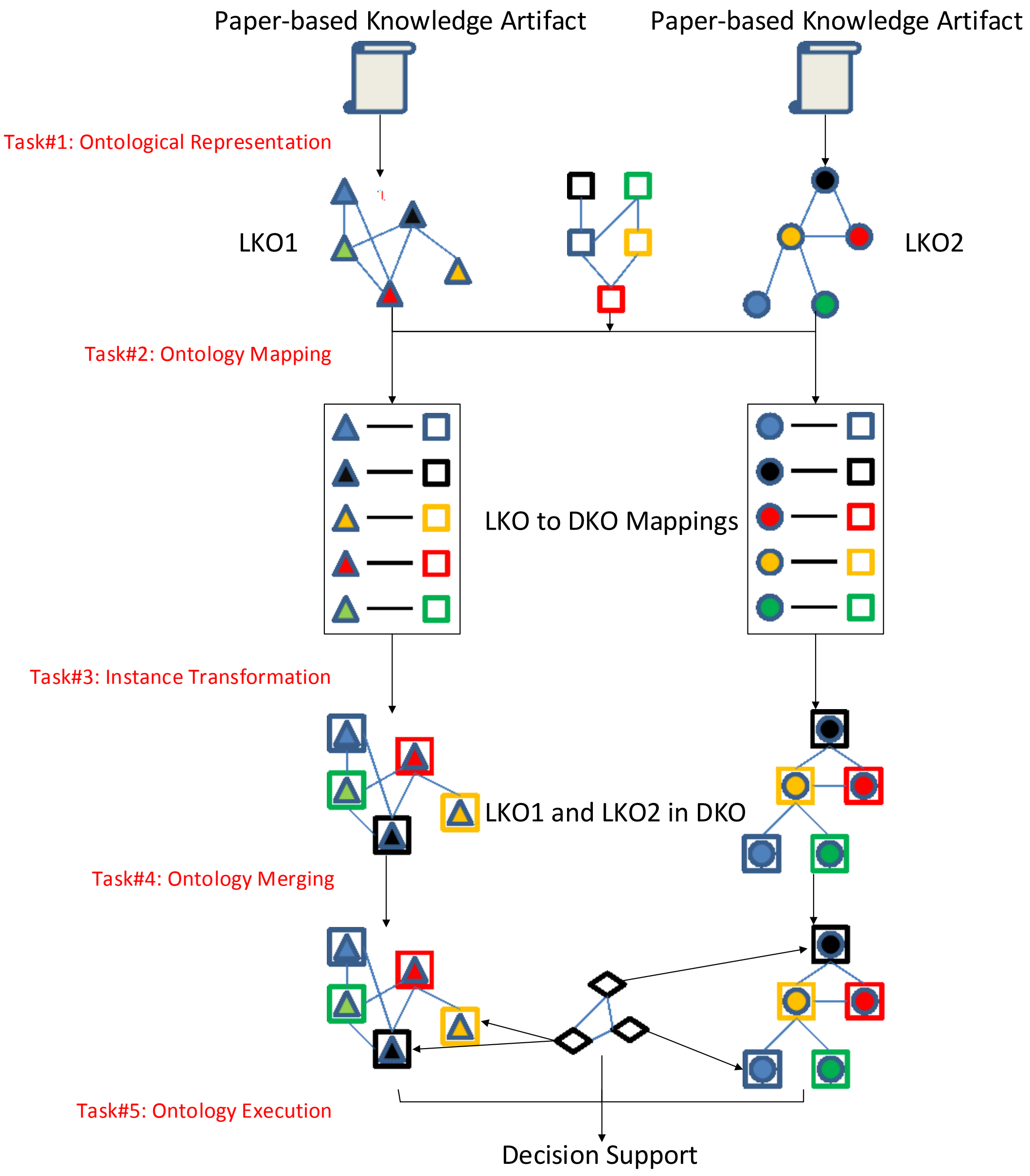

Paper-based Knowledge Artifact
Paper-based Knowledge Artifact
Task#1: Ontological Representation
LKO1
LKO2
Task#2: Ontology Mapping
LKO to DKO Mappings
Task#3: Instance Transformation
LKO1 and LKO2 in DKO
Task#4: Ontology Merging
Task#5: Ontology Execution
Decision Support

Figure 1.1 *OntoMorph* Solution Approach

- **Task #1**: Ontological representation of knowledge artifacts

Knowledge artifacts should be represented in a computer interpretable format in order to be used in DSS for decision making. Due to existence of several ontologies to represent knowledge in a domain of interest, we assume these knowledge sources might be represented in different LKO. This task in our framework is done manually with the help of a domain expert.

- **Task #2**: Mapping LKO to DKO

LKO are of different levels of details and coverage. In order to standardize the representation of these LKO, we develop a comprehensive *Domain Knowledge Ontology (DKO)* that all LKO can be mapped to and have their instantiations transformed to it. We represent the mapping using an OWL-Full ontology called *Knowledge Mapping Ontology (KMO)*. We manually instantiate this ontology to represent mappings between two ontologies.

- **Task#3**: Transformation of instantiations of LKO to DKO

We perform reasoning on the mapped ontologies and the mapping ontology in order to discover new mappings that do not exist in the original mappings and transform instances of the source and the target ontologies. In order to avoid undecidability during the reasoning process of the ontology mapping, we translate instantiations of KMO from OWL-Full to OWL-DL using a translation algorithm. Automatic instance transformation is achieved by reasoning on the mappings and the mapped ontologies.

- **Task#4**: Merging transformed LKO in DKO

To help a domain expert to identify and formally represent the decision reconciliation process, we developed an ontology called *Knowledge morPhing Ontology* (*KPO*). An instantiation of this ontology represents the criteria that should be met during concurrent execution of knowledge in several LKO. We call these criteria the *morphing criteria*. We define the morphing criteria as constraints that should be respected during recommendation generation in a DSS using several LKO in order to achieve decision reconciliation. We call these constraints the *Merging Constraints* in the remainder of this thesis. Therefore, morphing criteria and merging constraints are used interchangeably in the rest of the thesis. This ontology is instantiated manually.

- **Task#5**: Ontology Execution

In order to provide decision support using ontologies, they need to be executed. Execution is the computerized interpretation of the ontology knowledge in regards to the current state of the decision making problem in order to provide the user of DSS with the most suitable recommendation. We exploit OWL reasoning services to (1) provide decision support by executing a single LKO and (2) perform decision reconciliation during concurrent execution of several LKO according to their merging constraints.

1.4. *OntoMorph* Framework Application: Merging Clinical Practice Guidelines

Clinical Practice Guidelines (CPG) are evidence-based medical algorithms that assist health-care professionals in diagnosis, management and follow-up of medical conditions. Studies [18] show that following CPG by health-care professional are beneficial for patient care by standardization of the diagnostic and treatment procedures, reducing the care costs and increasing the quality and consistency of care. To improve the utilization of CPG in point of care, there have been numerous efforts to computerize CPG in ontologies and incorporate them within CDSS. PROforma [136][138], GLIF [133][140], GASTON [134][135], Asbru [108], EON [137][155], SAGE [34] and COMET [107] are a few to name. In these frameworks, the domain knowledge related the disease (i.e. medical concepts) and the procedural aspects (i.e. medical actions) are captured in terms of ontology elements.

CPG are designed to address only one medical condition in patients [107]. However, a patient can have comorbidities—i.e. existence of several concurrent medical conditions.

Older patients are more likely to have more comorbidities. For instance, hypertension and diabetes are two diseases that commonly co-exist in elderly patients. Comorbidities are complex and expensive to be treated and comorbid patients are associated with worse outcomes than patients with only one medical condition. From a CDSS perspective, concurrently using several CPG as inputs for treatment of two or more comorbid diseases is not a feasible solution as it leads to unnecessary duplication of tasks, visits and conflicts. Rather, a solution to handle comorbid conditions in a CDSS is to systematically merge the independent CPG of the comorbid conditions to generate a mutually consistent comorbid CPG [107]. We call this task *CPG Merging* henceforth. However, it may be noted that CPG do not give a detailed account of strategies and recommendations to handle comorbid conditions. There are only a few CPG computerization frameworks that offer a number of functionalities to handle comorbidities [24][26][30][99][107]. We use our knowledge morphing framework to address the problem of CPG Merging to deliver clinical decision support to comorbid patients. We call this domain-specific knowledge morphing framework the *CPG Merging Framework*. Contrary to the existing CPG merging frameworks that perform the merge before CPG execution, we decide on CPG merging on the fly and generate more relevant recommendations pertaining to the patients' status. The role of each components of the *OntoMorph* framework in CPG merging is discussed in section 3.5.

1.5. Research Contributions

In this thesis, we developed a knowledge morphing framework capable of dynamically morphing the knowledge encapsulated in several heterogeneous ontologies. We have contributed to the following areas of semantic web research:

a. Ontology Mapping

We developed an ontology that can be used as an ontology mapping representation language. This ontology is instantiated to represent complex mappings between any two

OWL ontologies. Our ontology is superior to ontology mapping representation languages in terms of capturing meta-data, providing an expression language and expressivity. To improve expressivity, we have improved upon support for mapping patterns, mapping operators, conditions, constraints, and relations between mappings.

Another novel aspect of our ontology mapping approach is the translation of the mappings to OWL + SWRL in order to define formal semantics for the mappings. Formal semantics enables us to utilize an OWL reasoner to perform further mapping discovery and instance transformation between the mapped ontologies. This feature is missing in most of the mapping representation languages or is not fully implemented.

b. Ontology merging

We have also contributed to the field of semantic-web based knowledge morphing frameworks. We proposed to develop an ontology for expressing morphing constructs between semantic web ontologies. Moreover, we define the formal semantics of these morphing constructs to be used in tandem with the domain knowledge semantics in order to perform dynamic execution time knowledge merging. We implemented these knowledge morphing constructs and their semantics for a CPG Merging Framework.

c. Knowledge Execution

We have made two advancements in this research area. Our first contribution is in the area of knowledge execution engines. The novel aspect of our knowledge execution engine is to define the formal semantics in OWL + SWRL and to use OWL reasoning services to derive the best decisions as opposed to conventional knowledge execution engines that hardcode the semantics of the domain knowledge in a programming language. Our knowledge execution engine is not tied to any programming language, OWL reasoner or API. This independence has the following benefits:

1. Improvement of the shareability of the domain knowledge ontology.

2. Ease of switching to new semantics web technologies.

3. Ease of developing tailor-made knowledge execution engines as the execution semantics is encoded in the domain ontology and does not need to be implemented in a programming language.

We also developed a Knowledge Morphing Execution Engine capable of delivering decision support using several relevant semantic web ontologies. The novel aspect of this engine is to use the semantics of the morphing constructs to dynamically merge source ontologies as opposed to existing knowledge morphing frameworks that perform the merge before the execution at the modelling level [30]. As we discussed before, modelling level merging is inferior to execution level merging for two reasons:

1. Every possible state of the problem should be evaluated for a possible morphing scenario. However, a problem may have a large number of state that makes this process tedious or even impossible.

2. To deal with the huge problem state, a number of frameworks [24][26][30][107] make some assumptions about the problem state that might not be true during knowledge execution.

Table 1.1 summarizes the semantic web ontologies and algorithms developed for our CPG merging framework in this thesis.

Table 1.1 Semantic Web ontologies and algorithm developed in this thesis

Ontologies	**Algorithms**
CPG Domain Ontology (CPG-DKO)	Knowledge Morphing Execution Algorithm
Expression Ontology	KMO to OWL + SWRL Algorithm
Knowledge Mapping Ontology (KMO)	CPG-DKO and CPG-KPO pre-processing Algorithm
CPG Knowledge morPhing Ontology (CPG-KPO)	

1.6. Thesis Outline

This thesis is outlined as follows: Chapter 2 provides an overview of the existing approaches for ontology reconciliation. This includes a review of the existing ontology mapping representation languages, instance transformation approaches and the features that a domain knowledge ontology should have in order to be considered as 'comprehensive' DKO. Since we are applying our framework to the problem of CPG merging, we also review the CPG computerization frameworks and present the list of features that a comprehensive CPG domain knowledge ontology should have. Moreover, we review the decision reconciliation approaches proposed in the literature pertaining to CPG merging research. In chapter 3 we thoroughly discuss the steps of our framework, the module that we have developed to carry out those steps and their interactions. In chapter 4, we describe the CPG-DKO and the workflow patterns that have been included in it in order to be used as a 'comprehensive' CPG-DKO. Chapter 5 discusses an extension that we have made to CPG-DKO in order to provide the CDSS with more coverage of the domain knowledge. This ontology can be used towards representation and execution of the decision logic in computerized CPG. We provide the full description of our OWL-based CPG execution engine in chapter 6. Our ontology mapping framework is discussed in chapter 7. This chapter is divided into two subsections describing the KMO and tis translation algorithm. Chapter 8 explains our KPO and the Knowledge Morphing Execution engine. Evaluation and conclusion are discussed in chapter 9 and 10 respectively.

CHAPTER 2: RELATED WORK

In this chapter, we review research pertaining to (a) ontology-driven Clinical Decision Support Systems (CDSS) based on computerized Clinical Practice Guidelines (CPG), (b) ontology mapping frameworks and existing ontology mapping representation languages; and (c) merging of computerized CPG for handling comorbidities.

2.1. Ontology-based Clinical Decision Support Systems

A computer-based clinical decision support system is defined as the use of computer software to assist the patients, physicians, nurses or any other healthcare professional involved the care of a patient with the diagnosis and treatment of patients [1]. CDSS usually have the following components: (1) A knowledge base containing specialized medical knowledge, represented in a computer interpretable and operable representation, describing the diagnostic and therapeutic processes; (2) A knowledge execution engine for deriving relevant recommendation based on the available knowledge in line with the patient data; and (3) several information interfaces to interact with the user, electronic health records and hospital information systems.

CDSS are typically categorized as knowledge-based systems and data-driven systems [1]. Data driven CDSS mainly use soft computing techniques to 'learn' decision models from available healthcare data, and use the learnt model to derive decisions. Typical model learning methods include Genetic Algorithm [80][81], Particle Swarm Optimization [84][85], and Neural Networks [82][83] and Bayesian networks [93]. The source of medical knowledge in these systems is the data that is usually stored in databases or electronic health records and are retrieved by the analyzing engine during execution.

Knowledge based CDSS use explicit/published knowledge to develop a knowledge model using logic-based knowledge representation approaches, and then use logic-based knowledge execution engines (also referred as reasoning engines) to derive pertinent recommendations based on the patients' information. Based on the knowledge base of the CDSS they can be categorized as (1) Rule-Based Systems [86][87] and (2) Ontology based systems. The knowledge base in rule based approaches is composed of classical condition-action rules, such as If-Then-Else rules, which are used to infer new knowledge. The knowledge base in ontology based approaches is an ontology which captures the domain knowledge in terms of a semantic representation of concepts, relationships and axioms. A logical reasoned is used as the knowledge execution engine, whereby patient information is provided in terms of instances of concepts and relationships and the output is a set of recommendations.

For knowledge-based CDSS, one of the prominent and validated sources of knowledge is Clinical Practice Guidelines (CPG). To incorporate CPG in CDSS, several CPG modeling frameworks have been developed for computerization and execution of CPG. The literature reports on a number of solutions for computerizing CPG, such as PROforma [136][138], GLIF [133][140], GASTON [134][135], Asbru [108], EON [137][155], SAGE [34] and some OWL-based methods [106][143]0[144][145][168]. The goal for computerizing CPG is to both model the CPG in a computer interpretable format and subsequently to execute the computerized CPG in order to give evidence-based advice to the health care professionals [88][89][90][91][145].

In this thesis, we review the PROforma [136][138], GLIF [133][140], GASTON [134][135], Asbru [108], EON [137][155] CPG modelling frameworks along the axis of CPG representation ontology and CPG execution engine. We chose these CPG modelling frameworks for two reasons: (1) Their CPG representation languages are among the most expressive CPG representation languages [88][89][90][91][145] and (2) They cover both approaches of CPG execution engine (graph parsing and transformation). These approaches are discussed in section 2.1.2.

Our rationale for this review is to identify the salient features of the CPG representation languages, and the capabilities of existing CPG execution engines associated with these

CPG representation languages. In reviewing the CPG representation languages, our focus mainly is to identify the workflow elements supported by these languages, since the goal of this thesis is to include CPG related workflow elements in the CPG-DKO and within our execution engines for both CPG execution and merging.

2.1.1. CPG Representation Languages

Ontology-based CDSS need a knowledge representation language to formally represent the CPG knowledge which is used towards clinical decision making. Most of the CPG representation languages have specialized constructs to (1) model the organization of plans and nesting components, (2) model sequential, parallel and cyclical tasks, (3) model decision models, and (4) provide the user with an expression/criterion language to specify criteria and goals. We review the CPG representation language of the PROforma [136][138], GLIF [133][140], GASTON [134][135], Asbru [108], EON [137][155] along the axis of the abovementioned criteria.

2.1.1.1. PROforma

PROforma was developed at Imperial Cancer Research centre. The name is a combination of the words *proxy* and *formalize*.

- Organization of Plans and Nesting Components in PROforma

CPG are modeled as a directed graph in which the nodes are instances of classes that model guideline elements. *roottask* is the main class of this language. *roottask* class has the following three important set of attributes: (1) *Components*: This attribute holds a set of tasks that should be executed (e.g., Follow-up, Therapy, and Diagnosis); (2) *Scheduling constraints*, *Temporal constraints* and *Termination conditions* attributes are used to define the order in which the tasks are execute in the CPG, what temporal constraints exist between them and the conditions under which a task is regarded as terminated respectively; (3) Trigger condition: A task can be triggered according to a temporal event or evaluation of a Boolean expression. If the trigger condition is satisfied and the goal of the task is not present, the task will be considered for enactment. Sub-classes of this class are 1. *Plan*, 2. *Decision* 3.*Action* and 4.*Enquiry*. All of them except for the *Plan* task are atomic and are not further extensible. Figure 2.1 shows the class hierarchy of the PROforma.

$$\mathit{Root\ task} \Rightarrow \begin{cases} \mathit{Plan} \\ \mathit{Decision} \\ \mathit{Action} \\ \mathit{Enquiry} \end{cases}$$

Figure 2.1 Task Ontology (Class hierarchy) in PRO*forma*

The *Plan* class represents a (sub) guideline in PROforma. PROforma incorporates a model of decision making that can work under uncertainty [157]. The decision making logic is modeled using the *Decision* class. Instances of the *Action* class are tasks that ask an external agent to act upon for completion of the plan. (e.g. Administration of a drug). Actions are atomic and not decomposable in PRO*forma*. Enquiries represented by the *Enquiry* class are used for requesting information from a source. This source can be a nurse, an electronic medical record, etc.

- Sequential, Parallel and Cyclical Tasks in PROforma

PROforma does not support branching explicitly but it is achievable through scheduling constraints [92]. PROforma provides a cycle description sub-language to define the constraints under which the repetition should be continued. It can repeat a task for a defined interval, until a specific state or a predefined number of iterations is reached or the termination condition becomes true. Multiple entry points are possible by setting pre-conditions for tasks of the PROforma.

- Decision Model in PROforma

Priority of options are defined by a set of schema that have rules, qualitative variables, quantitative weights and certainty factor and support (+) and oppose (-) candidates (options). For instance, “if (diagnosis = disease X) and (disease Y = absent) then (medicine

Z+)" is a rule that will be a schema by adding certainty factor and quantitative weights to the available options. Some other rules can be associated with the same options. Commitment to the available options depends on the overall balance of oppose and support schemas.

PRO*forma* does not provide explicit mechanism for switch construct but it is possible to have it using aforementioned schema. After an option wins the argumentation, the precondition of that task must be satisfied to be executed. The *goal* of a decision can be processed in the precondition of a *task*. When the argumentation is not needed, the decision making can be based solely on the commitment mechanism which is not clearly discussed in [136][138].

- Expression/Criterion Language to Specify Criteria and Goals in PROforma:

The expression language of PROforma that is defined in Backus Naur Form. This language allows use of usual Boolean, arithmetic, and comparison operators, as well as functions that evaluate the execution states of tasks (e.g. active, completed etc.) as well as the values of data items.

The *root task* class has the attribute *goal* that represents the goal of a task. If the matching term of the goal of a task is inserted into patient record before or during execution of that task, enactment of the task will stop because the goal is already achieved.

The *Precondition* attribute is used to assign preconditions to medical actions. The value of the *Postcondition* attribute of a task is considered to be true after its completion. It is possible to write rules in the form of if-then-else to set post conditions.

2.1.1.2. GASTON

GASTON is a joint effort of the Department of medical Informatics of Maastricht University and the Signal Processing Systems group of the Eindhoven University of

Technology. The goal of the project is to create a standard formalism to boost the acceptance of computerized CPG and CDSS.

- Organization of Plans and Nesting Components in GASTON

This CPG representation formalism uses a frame-based model as an underlying mechanism. Two ontologies are defined for modeling CPG in a slightly modified version of Open Knowledge Base Connectivity (OKBC) as the underlying knowledge model in protégé [134]: 1. Domain Ontology and 2. Method Ontology.

Domain ontology is the domain-specific knowledge represented in terms of medical concepts, attributes and their relations. Method ontology is designed to capture the workflow structure of the CPG. The important concepts used in the Method ontology are *Guideline*, *Primitives* and *Problem Solving Methods* (PSM). Figure 2.2 shows the class hierarchy of the method ontology.

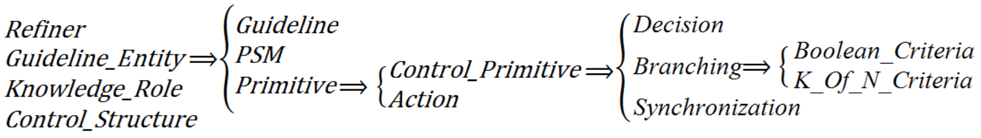


Figure 2.2 Method Ontology (Class hierarchy) in GASTON

PSMs are decomposable CPG elements which contain an instance of *Control_Structure* class that defines sub-components of the PSM. It can, for example, have references to a *branch* primitive that expresses that two specific tasks should be enacted in parallel and a *synchronization* point that waits for that two primitives (or PSMs) to meet. To use a PSM for solving a problem two tasks should be done: 1.refining the PSMs by changing some attributes of them, like messages that are going to be shown to the user or for example the number of clusters for a PSM that does clustering of data. 2. Mapping PSM knowledge roles to the domain ontology concepts. Three kind of knowledge roles are defined for PSM: Input knowledge roles, Output knowledge roles and Intermediate knowledge roles. Output and Input roles serve as inputs and outputs for other PSMs respectively and Intermediate roles are used in the internal procedure of PSMs. PSMs are decomposable into sub-tasks that can be executed by sub-methods. When a sub-method is no longer decomposable it is called *primitive-PSM*. Separating the PSM from the domain knowledge increases the usability of PSM

The *Primitive* class represents a single non-decomposable step in the CPG and contains the following four attributes: (1) *Procedure*: contains execution time information; (2) *Refiners*: specifies which attributes of PSM are used to further refine the primitives; (3) *Mappings*: Contains mappings to parameters from other primitives, domain ontology concepts and knowledge roles in a PSM and (4) *Visualization*: contains information about how to visualize the primitive in the user interface. The *Primitive* class has the following four sub-classes:

1. *Action*: specifies a clinical action. An action can be an instantaneous or persistent over time. An Activity is an action which is persistent over time and have some values assigned to *Start_Activity* and *End_Acitivity* attributes which refer to instances of *Time_annotation* to specify the temporal constraint for that *Action*.

2. *Decision*: will be discussed in later sections.

3. *Branching*: directs the flow of the CPG into parallel branches of that should be enacted in parallel.

4. *Synchronization*: the primitive in which the previously branched execution paths meet.

5. A *Guideline* represents the CPG as a whole and can only contain primitives and PSM. Similar to PSM, *Guideline* contains a control structure that defines constituent elements of it. Eligibility criteria of a patient and the criteria for abortion of the guideline are represented by *Eligibility_Criteria* and *Abort_Criteria* attributes

respectively. The values of these two attributes come from the class *K_of_N_Criteria* which will be discussed in the next sub-section.

- Decision Model in GASTON

GASTON has two types of decision model primitives namely; *K_of_N Primitives* and *Boolean_Criteria. K_of_N Primitives* has a *Criteria* attribute that refers to one or more *Criterion* class instances and a *K* attribute. If K out of N criteria of this decision primitive is true, then the whole logical statement is evaluated as true and the designated primitive is enacted. *K* is defined via refiners attribute that exist in *Primitive* class. *Boolean_Criteria* acts exactly the same as "if-then-else".

- Expression/Criterion Language to Specify Criteria and Goals in GASTON

GASTON has a rule representation model developed separately from the method ontology. Each guideline can contain several If-Then-Else rules. Two types of rules can be written in GASTON guidelines: 1.Action Rule and 2. Intermediate rules. Action rules ask a health professional to perform a medical task to continue with the course of treatment and the intermediate rules changes the internal values of the CPG without leading to any immediate actions. The conditions of the rules can be composed of instantiated conditions in the domain ontology combined with Boolean operators (AND, OR and NOT). However, the existing expression language does not support any type of mathematical, string and numerical computations and comparison.

Goal attribute of each primitive or PSM is an instance of *K_of_N_Critera* that defines whether the goal of the task or primitive is reached or not.

2.1.1.3. EON

EON that is developed in Stanford University is a methodology for building DSS based on clinical workflows. The EON approach is accompanied with several tools that facilitate acquisition and execution of CPG. It has an object oriented CPG model that is capable of describing complex scheduling and temporal constraints between tasks to capture the structure of CPG.

- Organization of Plans and Nesting Components in EON

EON's architecture is composed of four components that work together for modeling CPG. 1.Domain Independent Problem Solving Methods (PSM) 2.Temporal Reasoning System that can infer higher level abstractions form time stamped patient data. 3. A temporal query system that can perform time-oriented queries on time stamped patient data and 4. A domain ontology.

Domain independent problem solving method is very similar to PROforma's and GASTON's problem solving methods. Problem solving methods are reusable domain independent procedures that can be utilized for domain specific problems with appropriate mappings between the concepts in the PSM and the domain knowledge. PSM that are not further decomposable are called *mechanism*s in EON. It has 5 main constructs for modeling the workflow of the CPG: 1.*ClinicalAlgorithm* 2.*Scenario* 3.*Decision* 4.*Action* and *Activity* and 5.*Goal* that form the core elements of the CPG. In contrast to GLIF and PROforma they can be further specialized.

An instance of the *ClinicalAlgorithm* class represents a CPG. Instances of this class may contain *scenarios*, *decisions*, *actions* and sequential and temporal constraints. It is similar to *Plan* in PROforma and PSM in GASTON. A *Scenario* shows the state of the patients that are being treated with an eligibility criterion that specifies the necessary conditions for the patient to be in that state. A scenario is always followed by a decision or an action. It can also be an exception handler when the situation is out of control of the CPG and needs human attention. The *Decision* class will be discussed in later sections.

Action instances are instantaneous acts that should be done in the care process like enquiring information from the user or showing a message regarding an injection. They may change the status of the other components too. An *Activity* is an action that is persistent processes over time. *Actions* can start, stop and change the state of an *Activity*.

The authors of the EON have developed a PSM for Protocol-directed therapy which is called Episodic Skeletal-Plan Refinement (ESPR) [137]. Episodic Skeletal-Plan Refinement **(**ESPR**)** is an extremely general problem solving method that can be specialized for different clinical problems. For instance, the ESPR can be "fleshed out" to be a therapy planner that generates relevant recommendations based on the patients' state and the related therapy protocol. The following tasks can be achieved using ESPR problem solver:

1. **Propose plan**: determine the appropriate therapy plan for the patient according to the patient status and the therapy protocol (Domain Knowledge).

2. **Identify Problem**: find the problem that may occur because of applying the selected therapy in the previous subtask.

3. **Revise Plan:** modify the standard treatment that is suggested by *Propose Plan* subtask according to the Identify problem *subtask*'s suggestions.

- Decision Model in EON

Two types of decisions are provided in EON. The first one supports if-then-else and in the second decision model each option has a rule-in and a rule-out associated to it. If the option's rule-in is true, then the option is selected and if the same thing happens to the rule-out the option will be rejected. The latter decision method is similar to PROforma's. In the case that both of them are false, a default option is selected.

- Expression/Criterion Language to Specify Criteria and Goals in EON

EON uses Protégé 2000 constraint language (PAL) to define decision criteria in a sub-set of first order predicate logic written in knowledge interchange format [155]. It is possible to check presence or absence of data items, perform numeric comparison, and make Boolean combination of statement in the expression language. Every step has a goal that can be true or false indicating the achievement of the goal after the task is completed.

2.1.1.4. GLIF 3

GLIF is collaboratively developed at Columbia, Stanford and Harvard universities. The goal is to develop a standard representation format for sharable computer interpretable CPG. GLIF 3 is a revision of GLIF2 [33] which tries to alleviate the shortcomings of the older version by adding flowchart constructs, improving the expression language and handling iterations, etc.

- Organization of Plans and Nesting Components in GLIF 3

GLIF 3 can encode CPG in three different layers: (1) Conceptual Flowchart, (2) Computation Level and (3) Implementation specification. GLIF3 uses an object-oriented language that models concepts in the form of classes, attributes and relations. CPG in GLIF3 are represented as flowcharts that have temporal or sequential transitions between CPG steps. Figure 2.3 shows the class hierarchy in GLIF 3.

$$\mathit{GuidelineModelEntity} \Rightarrow \begin{cases} \mathit{Guideline} \\ \mathit{Algorithm} \\ \mathit{Guideline\_Step} \Rightarrow \begin{cases} \mathit{Action\_Step} \\ \mathit{Decision\_Step} \\ \mathit{Patient\_State\_Step} \\ \mathit{Branch\_Step} \\ \mathit{Synchronization\_Step} \end{cases} \end{cases}$$

Figure 2.3 Method Ontology (Class hierarchy) in GLIF3

The first step in modeling a CPG is to create an instance of the *Guideline* class. This class that represents a guideline or a sub-guideline contains attributes pertaining to intention of the CPG, eligibility criteria, and exceptions that may interrupt normal flow of the CPG. CPG recommendations are modeled by an instance of *Algorithm* class that is an attribute of *Guideline* class and contains a collection of *Guideline_Step*. *First_Step* of the *Algorithm* shows where the execution should start form. The flow between these CPG steps are established via explicitly linking CPG steps with the *next_step* attribute of the *Action_Step.*

Action_Step is similar to all of the action steps in previously discussed models. It contains a strength value that shows the necessity of the recommendations provided in that step as well as attributes that can define the duration of the task, events that trigger or abort the task and whether it has medical or programming purposes. *Decision_Step* is used to model the decisions in the CPG. It is possible to add more tailored decision models to the provided decision models in GLIF3. For this mean, the sub-classes of *Decision_Step* are further specialized. Decision steps will be discussed in more detail in later sections.

Patient_State_Step has two purposes. The first purpose is to show the state of the patient after each successful step of the CPG. The CPG can be imagined as a network of patient states that are connected by CPG steps. Another purpose of this step is to serve as entry points to the CPG. When a CPG is about to be executed, all the patient states are evaluated to check the eligibility of the patient for that state. If the patient is eligible for one of the states, the CPG can be executed from that point rather than from the starting point.

Branch_Step is used to define parallel paths in the CPG workflow. The *continuation* attribute of the *Synchronization_Step* defines how the synchronization step should wait for parallel branches that are going meet in that step. A synchronization step may wait for any or all of the parallel paths to be finished in order to pass the execution flow to the next task. Iteration is possible in *Decision* and *Actions* steps via *Iteration_Specification* attribute. It is possible to define frequency, stopping criteria (normal) and abortion criteria (abnormal).

Decision, action and patient state steps have an instance of the class *Triggering_Event* that defines the condition for triggering of that task. When the control flow reaches one of these steps, the step will be considered for execution only if one of its triggering events occurs. If more than one trigger is occurring, one of them is selected according to their priorities.

Different types of events can be defined, for example end of execution of the previous step, availability of needed information can be a *Triggering_Event*.

It is possible to nest the CPG in action and decision steps. For example, a decision step can consist of several sub-guidelines. Decision steps are nested using the *decision_detail* attribute that can refer to a sub-guideline. With adding an instance of *subguideline_action* to the *task* attribute of an *action* step, it is possible to refer to a sub-guideline for further nesting

- Decision Model in GLIF 3

Each decision step has a set of *Decision_Option*s. Each *Decision_Option* has a *Destination_Step* that determines the next step that is going to be executed if the corresponding option is selected. Two types of decision models are developed: *rule-in-choice* and *weighted-choice.* In *rule-in-choice* rules are defined for and against each option. There are four categories of such rules: 1.rule-in 2.strict-rule-in 3.rule-out 4.strict-rule-out. In the *weighted-choice* model, each option has a number of weighted criteria. Weight of a choice is equal to the sum of the weights of the associated criteria that are true. Choices are ranked and selected according to these associated weights. The Boolean variable called *automatic_deicion* defines whether the user should confirm the decision or it can be made automatically.

- Expression/Criterion Language to Specify Criteria and Goals in GLIF 3

The object oriented expression language GELLO [92] which is a superset of Arden syntax is proposed to specify decision and eligibility criteria in GLIF. There is a structured grammar for specifying these expressions and criteria. The grammar can specify logical criteria, numerical expressions, temporal expressions, and strings operations.

2.1.2. CPG Execution Engines

CPG execution engines developed in CPG modelling frameworks can be categorized in to two main approaches:

In the first approach, the core of the CPG execution engine offers the functionality to parse the computerized CPG to render it as a graph-based model for execution purposes, where the graph represents the execution workflow of the CPG. GLIF's [140] and EON's [137] execution engines are two examples of this execution approach. This approach is referred as graph parsing approach.

The second approach, referred as the transformation approach, involves a CPG pre-processing stage before execution where the computerized CPG is specifically transformed to an intermediate (knowledge) representation that is amenable for execution purposes. The intermediate knowledge representation formats are usually complex and not suitable for CPG encoders, hence the need for transformation of the computerized CPG. For instance, Petri-net is an intermediate representation formalism that is used by some researchers to translate computerized CPG to because it offers a well-understood execution semantics [145][156]. Another example is the Arezzo [157] (a PROforma engine) execution engine which transforms the CPG encoded in the Red Representation Language (R^2L) [157] to R_{R2L}(Logic of R^2L) format that can be interpreted by a PROLOG-like interpreter. This approach is regarded as the transformation approach.

There has been some attempts to use semantic web technologies for executing CPG modeled in OWL, where SWRL rules are used to determine the execution logic [143][149][150]. In these approaches, medical knowledge and execution semantics (e.g. Tasks' states) are represented through rules which makes the execution engine a CPG specific execution engine that is not capable of executing any other CPG.

We review GLIF's [140] and EON's [137] execution engines as two examples of the graph parsing execution approach and PROforma's [157] and GASTON's [121] execution engines as two examples of the transformation execution approach.

- *PROforma's Execution Engine*

Two main implementation of PROforma approach are Arezzo^{TM} [157] a commercial tool which is developed by *Infer*Med Ltd. (London UK) and Tallis which is developed by Cancer Research UK [138]. CPG are encoded in terms of instances of classes using the Red Representation Language (R^2L) [19]. This language is transformed to another language which is called R_{R2L}(Logic of R^2L) and then a PROLOG-like engine is used to parse and execute the CPG in an execution engine [157]. PROforma is one of the few reported approaches in the literature that does a pre-processing on the original encoded CPG similar to our method to prime them for execution. Both execution engines can be integrated with the legacy systems using the provided APIs We describe Tallis in this section of the report in more detail. The enactment engine is responsible for execution (enactment) of the CPG.

Each medical action goes through four states during execution: 1. *dormant* 2. *in_progress* 3. *discarded* and 4. *completed*. Figure 2.4 shows the possible transitions between these states. All tasks are initially in the dormant state. Tasks that are in dormant state are not ready for execution yet. *in_progress* state means that the task is in progress either by the engine or the healthcare professional. A task is *discarded* if the it is not going to be executed or it is interrupted in the middle of execution and *completed* means successful execution of the task. There are transitions from *completed* and *discarded* state to other states to be able to handle cycle during execution. A task will go from *completed* to *in_progress* if the task is cyclic itself and it goes to *dormant* if the super-task is cycling. Figure 2.4 shows the possible transitions between the abovementioned execution states.

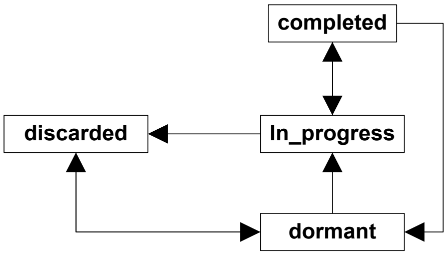


Figure 2.4 *Task* state transitions in PROforma [138]

The CPG execution engine has four important internal variables.

1. Properties Table: A three column table containing properties' values of all components. Each row in the table is of the form (C,P,V) where C is the component, P is the property and V is the value.

2. Change Table: A table which has the same form as Properties Table. This table contains the new values for properties of components that are to be assigned as a result of the operations that have been performed in the CPG. Having two different rows with the same C and P but different V will result in setting an exception flag.

3. Exception Flag: it is set when something abnormal has happened during execution.

4. EngineTime: This is set by setEngineTime public function of the CPG execution engine. It does not necessary corresponds to the real time.

The execution engine also has a number of public operations that an external system can use to connect to it and guide the engine for proper execution. Here are some examples of these public operations: 1.LoadGuideline 2.ConfrimTask: Informs the CPG execution engine that a given task is executed 3.commitCandidate: commits to a particular candidate or candidate of a decision etc.

- *GASTON's Execution Engine*

To execute the CPG, execution engine loads the control structure of each the guidelines and creates a workflow structure that is composed of primitives only. This structure along with the information that is attached to each primitive regarding its execution is compiled into an efficient form for execution. The execution time representation is optimized in order to make execution as fast as possible.

Their execution engine is composed of an execution-time scheduler and four other components that connect to it: (1) Procedure manager: This component traverses the CPG and informs of the execution-time scheduler what tasks are going to be executed and in what order; (2) Data Source Manager: This component makes use of existing communication standards to connect to hospital information systems and electronic health records; (3) Event Manager: this component enable the execution-time scheduler to act upon triggers that may be generated by the user or by outside sources such as a hospital information system; (4) Action Manager: This component is the interface for the user. It shows the user the tasks that should be executed and their related information. User can use this component to inform the scheduler of his decisions, outcomes of actions and states of the medical actions.

All the components connect to the scheduler with the same standard interface. This makes it effortless to expand the execution engine by creating new components and connect them to the execution-time scheduler with the standard interface. Figure 2.5 shows the architecture of the GASTON's execution engine.

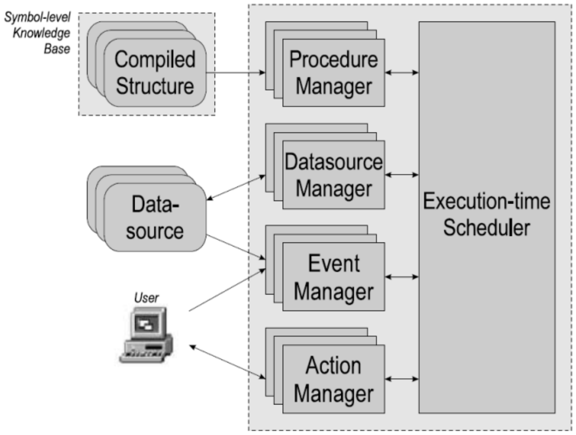


Figure 2.5 Components of the Execution-Time Scheduler in GASTON [134]

- *EON's Execution Engine*

The algorithm for EON model execution engine is described in [155]. A task can have four executional states namely; *active***,** *completed***,** *aborted, suspended.* Possible state transitions are between the *active* and *aborted* states and between *active* and *completed* states. State transitions to and from *suspended* state is not discussed in [155]. A task can get activated if (a) All the preconditions are satisfied; (b) the predecessor of the task is active and (c) all the interventions associated with predecessor of the node is completed or aborted. Revision rules are instructions regarding how to change the attribute values and states of an intervention. You can see the execution algorithm in Table 2.1.

Table 2.1 EON modeling language execution algorithm [155]

1. If the procedure has not been activated before, activate the start node. If the procedure has been activated before, go to 2.
2. Evaluate the revision rules of all active nodes, if there are any.
3. For each node that is a successor of an activated node, check to see if the node can be activated. If yes, then add the node to the list (L) of nodes to activate.
4. If L is empty, then exit.
5. Deactivate those nodes whose successors are in L.
6. Activate the nodes in L.
7. Go to 2.

- ***GLIF 3’s Execution Engine***

GLIF3 Guideline Execution Engine (GLEE) [140] is the only tool developed for executing CPG encoded in GLIF3. This CPG execution engine can be categorized under graph parsing methods. The execution engine considers four different states for each task in the CPG. These states are 1. *prepared*, 2. *started*, 3. *finished* and 4. *stopped*.

A task is in the *prepared* state when it is suggested to the user for execution by the engine. *started* state means that the task is in progress. Tasks that are stopped by the user before or during the execution will have the *stopped* state and finally, the *finished* state means normal completion of a task. Figure 2.6 shows these states, the state transitions and the actors that may cause these state transitions.

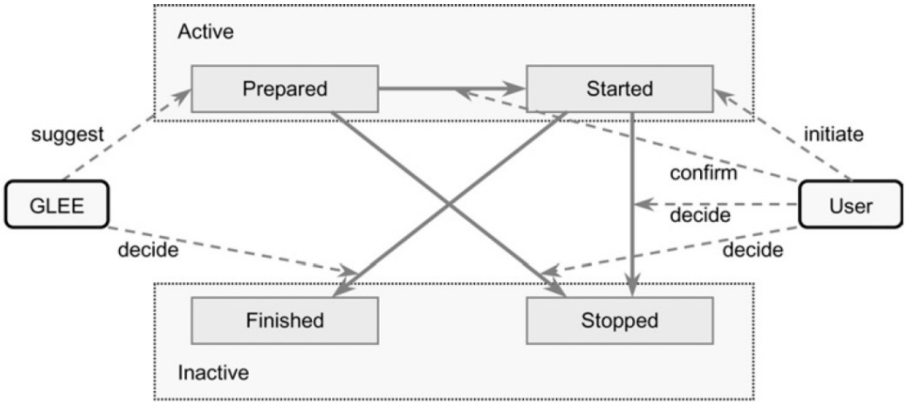


Figure 2.6 *Task* state transition in GLEE [140]

An interesting behavior that is not reported in any other CPG execution engines is the ability to decide whether to go to sub-steps of a step for completion of it or skipping over them if the goal of the step is already achieved. If the user decides to go through the steps, the first step is activated for execution. During the execution of the sub-steps, the super-step will remain in the *started* state and execution returns to the super-step after finishing all the sub-steps. A tracing system is also provided in the CPG execution engine that can keep track of the time of the activation, start, finish and discarding of the steps during execution.

2.1.3. Conclusion

Review of the CPG computerization frameworks informs us about common features of CPG representation languages and CPG execution engines. Our review shoes that the elements of the CPG representation languages serve two different purposes: 1. Modelling of the medical knowledge encapsulated in the paper-based CPG and 2. Capturing the workflow structure of the paper-based CPG and making the necessary mappings between the workflow variables and their counterpart variables in the domain knowledge. Since modelling the medical knowledge of CPG is out of scope of this thesis, we focused on modelling of the workflow structure of CPG. We created a list of most commonly representable workflow patterns within existing CPG representation languages:

1. Task ordering: It represents the order in which the tasks should be followed during the treatment of a patient.

2. Task nesting: CPG may have different levels of nesting of tasks—i.e. a high-level composite task may contain multiple levels of sub-tasks.

3. Conditions: Execution of tasks in a CPG may depend on satisfaction of a certain set of conditions. Boolean conditions are the most common type of the representable conditions in these languages.

4. Condition satisfaction criteria: The criterion for satisfaction of conditions of medical tasks can be rather complex. This criterion expresses how many of the conditions of a task should be satisfied before execution.

5. Loops: CPG representation languages should have the ability to express iterative medical tasks—i.e. tasks that are to be repeated till a specified loop termination condition are satisfied.

6. Decisions: Several decision mechanisms such as if-then-else, switch and argumentation rules are present in the decision logic of CPG.

7. Branches: At any given point during patient care multiple clinical tasks may get activated and executed simultaneously. A branch point is where the execution flow diverges into these parallel tasks.

8. Synchronization: Clinical tasks executing in parallel may need to eventually merge at a point to realize a singular execution flow. A synchronization point merges multiple branches to streamline the execution flow.

9. Expression Language: During the course of execution of a CPG, several mathematical, Boolean and string computations might be needed in order to interpret the execution logic. A language describing these expressions is necessary.

We also reviewed the existing engines developed for execution of CPG representation languages. Regardless of the approach that is followed for execution, they mostly share the same capabilities in regard with CPG execution. Below we present our list of desired CPG execution engine functionalities:

1. Functions for loading and storing the computerized CPG and modifying the patient information.

2. Executing the workflow structure in line with the patients' information. CPG execution engines need to follow the stipulated order of the clinical tasks indicated in the CPG in order to generate recommendations relevant to the current states of the patients. This involves execution of all the devised workflow structures in the CPG representation language. In order to execute a CPG a state transition system should be implemented. This state transition mechanism is needed to (a) control the execution order of the tasks as per the execution semantics, and (b) provide the current state of the tasks at any given point in time.

3. Executing the expression language. Execution of the instances of the expression language is pivotal for proper interpretation of the workflow of the CPG. This is the ability of the execution CPG to understand and process: (a) Restrictions to a subset of data type values, e.g. Numbers that are bigger than say 20, (b) Relationships between values of datatype properties on different resources, e.g. Blood pressure measurements that are greater than their previous one, (c) Mathematical, string and Boolean functions, e.g. difference between two values or increasing a counter for loop handling.

4. Handling Conditions: A CPG execution engine should have the functionality to evaluate the conditions of a task based on the available information.

5. An interface that allows users to interact with the execution engine, enter input values and outcomes and follow the instructions.

6. Easy to use data interface methods that are needed to communicate with hospital information systems/electronic medical records.

7. A mechanism to handle decisions automatically or with the help of the user during execution.

2.2. Ontology Mapping

2.2.1. Introduction

Ontologies are expressive knowledge representation languages used in many areas of science. They can create a semantically rich representation of the concepts within a specific domain in a sharable and computer understandable format. The ability to share knowledge while ensuring that the parties have the same understanding of the meaning of the shared knowledge is called semantic interoperability [94]. Ontologies can increase semantic interoperability among heterogeneous systems and semantic web applications if accepted as the standard knowledge representation formalism between all parties. However, the distributed and the open nature of the semantic web have led to slightly dissimilar ontologies describing the overlapping or similar domains. This dissimilarity between ontologies is also referred as heterogeneity. If heterogeneous ontologies are used by different parties knowledge sharing and as a result semantic interoperability will not be achieved. Four types of heterogeneities exist between ontologies describing the same topic [37]: **(a) Syntactic heterogeneity**: when ontologies are described in different representation languages (e.g. OWL [142] and F-Logic [35]). **(b) Terminological heterogeneity**: same concepts described using different words (e.g. synonyms, etc.). **(c) Semiotic heterogeneity:** this heterogeneity can be explained by the fact that ontologies are interpreted and understood differently by different individuals. Even though ontologies are used to increase interoperability and shareability, heterogeneity among ontologies developed by different parties on the same subject stops them from interacting successfully. **(d) Semantic heterogeneity**: this sort of heterogeneity is the differences in techniques and approaches that used in modeling the same concepts. For instance, CPG representation languages described in section 2.1.1 use different modelling techniques for capturing encapsulated knowledge in paper-based CPG. Therefore, the same CPG will be represented differently in various CPG representation ontologies. Three reasons are identified in [36] for semantic heterogeneity:

(1) Difference in the coverage of the concepts: different ontologies about the same topic may have different levels of coverage of the concepts within the domain knowledge. For instance, a workflow ontology may be able to express 10 workflow patterns while another workflow ontology can only support 5 of those patterns.

(2) Difference in granularity: This happens when two ontologies try to cover the same part of the domain knowledge with different levels of detail. For instance, subclasses of a class can be identified up to 2 or 3 levels.

(3) Difference in point of view: Ontologies of the same domain may be described differently according to the point of view of the user of the ontology. For instance, ontologies developed for physicians and nurses for the same disease are different. This is different from coverage as ontologies with different overages are developed with the same point of view and ideally should be exactly the same. On the contrary, the ontologies with different points of views should not be the same and must be suitable for their intended users.

We are interested in merging several existing computerized CPG and executing them concurrently in a comorbidity CPG execution engine. However, these CPG may be represented in different ontologies. Merging CPG in heterogeneous ontologies is difficult as the similar concepts might be modeled differently. As a result, it might not be possible to find the corresponding concepts in different ontologies in order to establish the necessary merging alignments. Moreover, each of the CPG ontologies has their own dedicated CPG execution engine and executing all of them is not possible in a single merging execution engine. Therefore, semantic interoperability which is CPG merging in our case, can be achieved by removing semantic heterogeneity and unifying the representation of all the CPG that are going to be merged in a single representation format suitable for merging and execution.

A solution to reduce semantic heterogeneity between ontologies is ontology matching. There are three central concepts concerning ontology matching relevant to our work:

- **Ontology matching** is the process of finding similarities among different ontologies that are going to be mapped. This process can be either performed manually or automatically. Either way, the outcome can be a similarity matrix filled with similarity measures from the [0,1] range. For instance, the output of an ontology matching process can express that *Human* and *Person* classes from two different ontologies are similar. Ontology matching results can be used towards finding the mappings among those ontologies.

- **Ontology mapping** is the process of aligning elements from different ontologies either directly or according to the ontology matching results. As an example, the result of mapping two ontologies in OWL can express that *Human* and *Person* are equivalent classes using *owl:equivalentClass* construct. The ontology mapping representation must be rich enough to capture all the aspects of the mapping. It should also have clear syntax.

- **Instance Transformation:** Mapping two ontologies enables us to perform instance transformation. Instance transformation is the process of transforming instances of one ontology in to the terms of another ontology based on the identified mappings. Continuing from our previous example, instance transformation will cause all the instance of the class *Human* in the source ontology to belong to the class *Person* in the target ontology as well. By transforming instance of the source ontology to the target ontology the semantic heterogeneity is removed and all the concepts are represented in a unified ontology. Therefore, instance transformation can be used towards unification of representation CPG modeled in different ontologies.

Ontology matching eliminates the semantic heterogeneity as it finds the differences between ontologies and makes the necessary mappings needed for instance transformation. Two approaches can be taken in order to reduce semantic heterogeneity among different ontologies by ontology matching [1]:

(a) Mapping local ontologies to a pre-existing background ontology: in this approach, a background ontology exists that every local ontology can be mapped to. It is easier to find mapping between a local ontology and the global background ontology because it is assumed that the global ontology provides a comprehensive, easy to understand and standard vocabulary of the domain knowledge. No direct links exists between local ontologies; however, those links can be inferred via reasoners. This approach is not suitable for highly dynamic environment as updating the background ontology frequently in a highly dynamic environment is not always a possibility. This approach can provide applications with a unified knowledge model in order to access the knowledge of all local ontologies. Since no direct mappings exist between the local ontologies they can contain inconsistent pieces of knowledge. Since we are not dealing with a highly dynamic environment, we believe that this is an appropriate solution for the problem of unifying the CPG ontologies.

(b) Mapping local ontologies directly: in this methodology, direct mappings are established among elements of local ontologies and no global ontology takes part in the process. Dynamic and distributed systems can benefit from this mapping, as there is no need to keep a background knowledge updated frequently. However, it is harder to find mappings between these local ontologies because of lack of enough common vocabulary and enough overlap in concepts. Another disadvantage is that incremental development of systems using this approach will be complicated, as new mappings should be created between the newly added ontology to the systems and every other existing ontology. On the contrary, in approaches using a background ontology, only a new mapping between the newly added ontology and the background ontology should be made. When interoperability is necessary and no background knowledge exists or cannot be agreed upon, this mapping approach should be followed.

2.2.2. Mapping Techniques

Ontology mapping techniques can be categorized under 4 classes [37] namely; **(a)** Name-based, **(b)** Structure-based **(c)** Extensional and **(d)** Semantic-based approaches. These methods are described in the next four sub-sections.

2.2.2.1. Name-Based Approaches

In this approach, names, comments and other forms of textual information are used for finding links between the concepts. Our research does not concern this sub category of ontology matching therefore we just review the general ideas in this area. Two sub-categories exist based on this approach: **(a)** string and **(b)** linguistic-based approaches.

In string-based approaches, ontology mapping is performed based on the similarity of strings that represent concepts in ontologies. All of the existing approaches follow these two general steps (a) Find similarities among elements of the two ontologies based on their names or comments, etc. and (b) use a threshold value to decide on the validity of the created links. Similarity of names can be based on the distance of the two strings. Several distance measures exist that can assist the ontology matching. Hamming distance, n-gram distances, cosine similarity are few to name. Interested reader may refer to [38] for further details on distance functions and their use in name matching.

Linguistic methods rely on Natural Language Processing (NLP) techniques. Natural languages are hard to understand for computers because they are ambiguous, vague and are created to describe anything. For instance, it is possible that the same concept can have different forms with the same meaning. A word can undergo different morphological processes that turn the word into another word with almost the same meaning but different string. It is hard for string-based approaches to detect this correctly. Inflection, derivation and compounding are amongst the possible morphological processes. In order to remove the effect of these processes, a text undergoes different steps to be transformed into a unified format. Some of these steps are **(a)** Stemming**:** This is the process in which stems of the words are found e.g., "foxes" is mapped to "fox", or the word "library" is mapped to the stem "library". It is highly possible that the words with the same stem are semantically related. String matching algorithms now can be used conveniently; **(b)** Lemmatization: In this step, a word is mapped to its base form considering the context in which the word appears in the text. For instance, The word "merging" can be either the base form of a noun or a form of a verb ("to merge") depending on the context, e.g., "in this approach of merging" or "We are merging CPG". Unlike stemming, lemmatisation understands the difference and selects the right base form as it appears in the dictionary; **(c)** Elimination: certain categories of words are repeated frequently no matter what the texts is about and are not of much value for most of language processing related tasks. Articles, prepositions, conjunctions, etc. are few of such categories to name.

Sometimes the text itself is not enough and we have to resort to external sources of information like dictionaries or thesauri. A thesaurus is a dictionary to which some information regarding relations of the words are added. For instance, "fast" is an antonym of "slow" and "raging" is a synonym of "angry". Knowing synonym can greatly enhance our success rate in finding matches that string based approaches fail. Homonyms (similar words with different meanings) can also be useful in removing some of the non-relevant mappings.

WordNet [39] is a thesaurus for English language and based on the concept of "synsets" which is a set of synonym names. It also contains information regarding the super and sub concepts and "part of" relation. It is one of the most widely used thesauri in NLP research area. Several similarity measures have been proposed based on WordNet. For instance, a distance between to concepts can be calculated based on the distance that they have from each other based on super concept and sub concept relations. Five different proposed measures are compared in [40] for the task of finding semantic similarity. Publication [48] described a matching method which makes use of WordNet to find different meanings (senses) that a word might have. Relevant senses are kept according to background knowledge and used to drive relations between concepts between two hierarchies that are matched.

2.2.2.2. Structure-Based Approaches

The structure of ontologies can contain a great amount of information that can be beneficial to ontology matching algorithms. This source of information can be accessed from two different perspectives: (a) Properties associated with concepts (Internal Structure) and (b) their relationships with other concepts in the ontology (External Structure).

The methods relying on internal structure of the concepts use properties and their different features such as their type (datatype or object properties), ranges, cardinality, restrictions, transitivity, etc. to evaluate the similarity of two concepts. For instance, a datatype property with the range “real” is more similar to a property with the range “integer” than a property with the range of “string”. Some other similarity measures are developed based on the range of the properties [42], their cardinalities [43]. It is also proposed in [41] that use of constructs like “sets”, “bags”, “arrays”, etc. can be used for further similarity measurements. This sort of matching usually creates inaccurate “m:n” matching clusters. Even though the output of this approach is not accurate enough to be used in isolation it can significantly reduce the match possibilities for other methods.

An algorithm to find similarities between entities of OWL-Lite ontologies is described in [41]. This methods works purely on the internal structure of the OWL-lite ontologies and no domain knowledge is needed to be incorporated in the process. However, there is no proposed algorithm in the literature that takes into account the peculiarities of OWL 2 and SWRL technologies. As an example, Qualified Cardinality Restrictions or user defined data types are constructs that are not available in OWL 1 and can be used towards improvement of internal structure based ontology mapping algorithms for OWL 2. SWRL rules can also contain significant amount of knowledge that is currently totally neglected. It seems that further research is needed in this area due to increasing popularity of these technologies.

External methods are based on the fact that the more similar two concepts, the more similar their relationships with other concepts are. Therefore, another perspective is to evaluate similarity of concepts with respect to the relationships that they have with other concepts of the ontology. Two sorts of properties can be used for this mean:

1. Tree making properties (taxonomy making properties): These properties make a tree out of the ontology graph. Absence of cycle makes calculating the similarity values an easy job based on either top-down or bottom-up methods. One of the most important set of properties that make a tree is the set of properties that create the backbone of the ontology's taxonomy (such as *owl:subClassOf*, *owl:superClassOf*, etc.). One of the most obvious ways of comparing objects in any taxonomy (no matter if it is an ontology or not) is to calculate the distance that they have in the taxonomy structure[44].

2. Cycle making properties: These properties make a graph in the ontology. The difficulty with these sorts of properties is that if we consider that similarity of a concept depends on the similarity of its neighbors (according to the selected property) we may end up going around the loops infinitely in order to compute the similarity. One of the approaches that deals with this problem is presented in [45] as a graph matching algorithm.

Choosing properties other than the taxonomy making ones is highly dependent on the domain. For instance, in the medical domain, *hasSymptom* property might be a very important property for finding similarities between classes of diseases.

2.2.2.3. Instance-Based Approaches

On the contrary to the other approaches discussed so far, this one needs the ontologies to be instantiated. This approach, works based on the shared instances. The more instances two ontology classes share, the more overlap these two concepts have. The first step is to identify the shared instances. Name based approaches are the most natural way of finding these correspondences between instances. The next step is to compare similarity of the concepts in ontologies based on their known shared instances. As an example, Jaccard index between two sets can be used to measure the similarity of two concepts based on their instances: $\delta(c_1, c_2) = \frac{|c_1 \cup c_2 - c_1 \cap c_2|}{|c_1 \cup c_2|}$. Another way of leveraging existence of instances is to perform statistical analysis on them. For instance, minimum, maximum, mean and variance of values of data type properties can be computed utilized towards the measurements of the similarity [37]. Publication [46] presents **(a)** an empirical comparison of different similarity measures (Hamming Distance, Point wise Mutual Information, Information Gain or Log-likelihood ratio) that can be used for evaluating the similarity of two concepts based on their common instances and **(b)** how to tune thresholds to make decision on the overlap of the concepts in different ontologies on order to deal with data sparseness problem.

This approach is not of interest to us as we are dealing with ontologies which do not have many instantiations and those instantiations hardly share instances.

2.2.2.4. Semantic-Based Techniques

This approach is composed of two main sub-steps **(a)** anchoring and **(b)** deriving relations. In the first step, other ontology mapping (name based methods for instance) are used to create an initial set of mapping between ontologies. After creating anchors between ontologies, they can be considered as one ontology and a reasoner can derive new relations between the concepts in both ontologies based on the anchors and the existing relations in the ontologies. Sometimes it is not possible to map two ontologies to each other directly because of the semantic heterogeneity. Background ontologies can be used as an Interlingua between two ontologies. Reasoning techniques that used to derive new relations can be categorized under three classes: (1) Propositional logic and satisfiability, (2) Description logic and (3) algorithm developed for specific mapping representations and frameworks.

- **Propositional Logic and Satisfiability**

Prepositional logic [51] is a formal system in which elements of a language are defined as propositions in terms of elements of a finite set of alphabets, operators, and initial points. A set of transformation (inference) rules is used to derive new formulas. Three steps should be taken to apply this logic to ontology mapping problem:

(1) An anchoring process will be performed based on existing matching techniques. The result is formulated using the alphabets and operators of the propositional logic. For instance, a name based matching technique may say that *o1:Brain* is equal to *o2:brain* and *o1:neoplasm* is equal to *o2:tumour*. Translating these anchors to prepositional logic axioms will result in: $(o1\!:Brain \leftrightarrow o2\!:brain) \wedge (o1\!:neoplasm \ \leftrightarrow o2\!:tumor)$.

(2) A matching formula is built for each possible pair of classes in which elements of the pair are from different ontologies. Suppose that class *c1* is intersection of *o1:Brain* and *o1:neoplasm* and *c2* is the intersection of *o2:brain* and *o2:tumour*. To discover to correct relation of *c1* and *c2* all the possible relations between *c1* and *c2* should be considered and formulated.

(**3**) These matching formula are used as inputs to prepositional SATisfiability solver that evaluates the validity of the formula. BerkMin [52] and GRASP [53] are two examples of SAT solvers. A new relation between two classes is valid if and only if the negation is unsatisfiable.

To find the correct relation between classes an exhaustive search over all classes and relations is necessary and this makes it a computationally expensive algorithm. The other disadvantage of this approach is the lack of expressivity in prepositional logic. CTXMATCH mapping framework that is explained in [48] uses SAT solvers to find the correct mapping between ontologies. Publications [47][49] are some other examples of this approach.

- **Description Logic**

Description Logic (DL)[50] is a family of knowledge representation languages which are more expressive than the propositional knowledge. In this approach, first, an anchoring step is performed and an initial set of mappings are created. After creating the initial mappings, ontologies can be considered as a single ontology and a reasoner can perform reasoning for further mapping discovery. For instance, suppose that OWL ontology 1 expresses: All *Vehicles* have at least 2 *Wheel*s and OWL ontology 2 expresses: all *Cars* have exactly 4 *Tire*s. If we map the two ontology by saying that "*O1:Wheel owl:equivalentClass O2:Tire*" then the reasoner will infer that "*o2:Car owl:subClassOfO1:Vehicle*".

There are two disadvantages to this approach: (1) It can be computationally expensive because merging all ontologies to one can result in a huge ontology that is very time consuming for the reasoner to perform reasoning on; (2) It is not possible to perform reasoning if ontologies contain inconsistent knowledge.

To overcome these problems in OWL which is the most popular ontology language, an extension of OWL which is called Contextualized OWL (C-OWL) is proposed in [58]. In this approach, instead of sharing the knowledge of all of the merged ontologies, their contents are contextualized. In other words, ontologies can keep their content local and have mappings with other ontologies with the explicit constructs (semantic bridges) that are devised in C-OWL. Five types of semantic bridges are defined: 1. Subsumption, 2. Equivalence, 3. Containment, 4. Disjunction and 5. Intersection. Using this language, ontologies can contain inconsistent pieces of knowledge because ontologies do not have to share them with others during the reasoning process. Being computationally expensive is not an issue either because the reasoning engine performs on a small fraction of all of the ontologies to derive new intra-ontology relations.

The second disadvantage (impossibility of reasoning because of inconsistency) can be an advantage for domains that we merely want to check the correctness of the mappings. In these domains, it is assumed that the ontologies do not contain any inconsistent knowledge and any issues with inconsistency have occurred because of the incorrect mappings. Incorrect mapping will be detected and changed with the help of a DL reasoner. A method that uses description logic to encode mapping for integration of spatio-temporal database schema is described in [54]. A DL reasoner is used to debug mappings using inconsistency detection. Another example is discussed in [55]. Authors of this paper present an algorithm for finding inconsistencies in mappings using Distributed Description Logic.

Another approach which pursues application of DL reasoning is explained in [57]. This paper contains a theoretical work which shows how description logic can be used for reasoning about the mapping themselves to check their attributes. These attributes are containment, minimality, consistency and embedding. Another approach that leverages DL reasoning services is the OIS framework [56].

Authors of [59] introduce a new language which is called WordNet Description Logic (WDL). In this approach, meaning of each element is extracted from the domain knowledge (WordNet) and represented by DL. A word may have multiple senses (meanings) in WordNet. They suggest removing some of the senses from the meaning of the words by a process called semantic elicitation. With the help of the structure of the ontology irrelevant senses of the words are removed. They use a DL reasoner to find the semantic relation of two elements from two different ontologies based on the meaning of those elements described in DL. Since WordNet works as the background knowledge (common vocabulary), there is no need to create any direct mapping between the ontologies and it is the DL reasoner that infers the relationships.

Even though DL has a high level of expressivity and many of the computational tasks can be performed efficiently, it is still not enough because of lack of expressivity to capture some simple and useful mappings. An example which shows DL falls short is the following mapping [60]: If an *article* has a *book_title* and the *article* is included in a *book*, that *book* has the same *book_ title* as the *article*. This issue emanates from lack of variables in DL.

- **Other Approaches**

There are other methodologies that use neither of the abovementioned reasoning formalism. They have developed their own reasoning algorithms for their representation.

OntoMerge [61] methodology has a solution for the problem of ontology translation. Ontology translation is the process of transformation of an ontology (The ontology itself and its instances) to another ontology according the specified mappings. They encode the mappings using First Order Logic in an XML based syntax. They have implemented an inference engine (OntoEngine) which processes the mapping in either a demand driven (backward chaining) or a data driven (forward chaining) manner. An interesting point mentioned in the paper is that fully automatic ontology merging and instance transformation is not possible yet due to the limitations of the existing mappers. The existing automatic mappers cannot produce 100% accurate mapping and can only infer simple mappings such as *equivalentClass*, *subClassOf*, etc. whereas far more complex mappings may exist that can be only detected by a domain expert.

MAFRA [62] is an interactive and dynamic framework for ontology mapping and transformation. It has five vertical activities that should be performed in order to achieve the ontology transformation goal. These activities that can be seen in Figure 2.7 are: 1.Lift and Normalization: in this step, a text pre-processing step is performed on the texts (names, descriptions, etc.) in order to translate them to a unified format 2. Similarity: after normalization of the text, a string similarity measure is used to find a confidence factor for potential mappings. 3. Semantic Bridging: This step has 5 sub steps namely; a. class and b. property mapping identification: Based on the similarities, an algorithm will figure out the valid mappings. The most straightforward approach is to have a cut off value. c. inferencing step which suggest the use of a reasoner to acquire the mappings that are implicitly stated in the ontology, d. refinement step: an optional step in which the mappings are checked for quality and e. transformation specification step: It specifies a transformation procedure to the mappings. 4. Execution: execution is the process of transforming the source ontology to the target ontology based on the specified mappings. Publication [62] contains some details on how the execution engine should be implemented in regard with their mapping representation language. 5. Post-processing: in this step, quality and consistency of the transformed ontology will be checked.

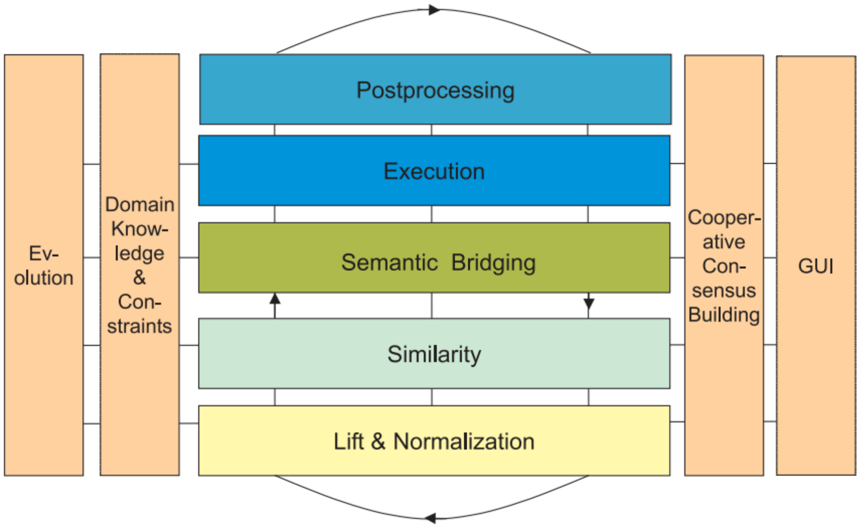


Figure 2.7 Conceptual Architecture of MAFRA [62]

Vertical activities are the ones that should be performed in parallel with all of the sequential steps. Evolution synchronizes the mappings with the changes in the source and the target ontologies in dynamic environments. Domain knowledge & Constraints: making use of the domain ontology can drastically improve the results in each of the sequential steps. For instance, domain knowledge may become handy when the similarity measure is computed. Cooperative Consensus Building is bringing the two parties practicing in the mapping to an agreement regarding the correct mappings. A Graphical User Interface can facilitate all of the above-mentioned steps.

Another ontology mapping framework is introduced in [63]. They believe that the methodology introduced in MAFRA does not cover some necessary aspects of the ontology mapping life cycle. For instance, they believe MAFRA does not support detection of ontology mismatch. Another issue with MAFRA is the assumption that ontologies are used to represent mappings while this is not the case in many mapping methodologies [64]. Their methodology is composed of three steps: 1.Mapping Discovery, 2.Mapping Representation and 3.Mapping execution. These steps are shows in Figure 2.8 taken from [97].

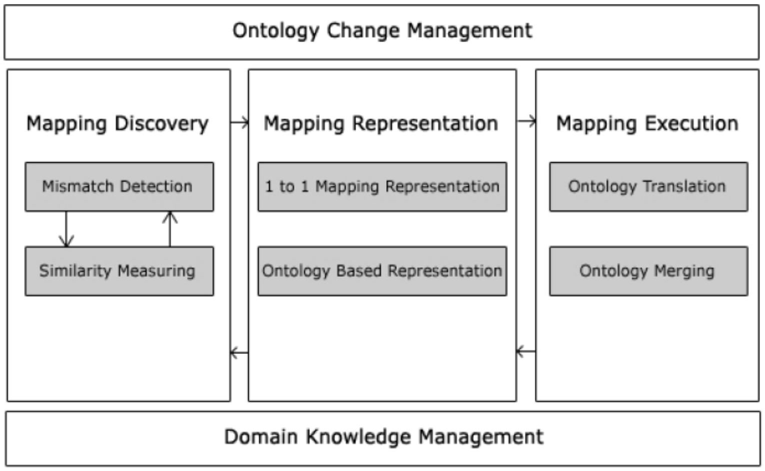


Figure 2.8 Ontology Mapping Framework discussed in [63]

Mapping discovery is finding mismatches and similarities. The next step is mapping representation. They have classified different representation approaches into two classes of 1-to-1 mapping and ontology based mapping. In 1-to-1 mapping representation, two concepts from the two ontologies are directly connected to each other without adding any semantics to the link. They mention that since this approach is a good solution for direct translation of instances, it can be easily used for ontologies that share the same level of details and have enough overlap. Different levels of abstraction or the need for complex mappings can be a big challenge for this approach. On the other hand, ontology based representations are capable of expressing very complex mappings and are useful for other purposes. The last step is Mapping Execution that is categorized under the Ontology Translation and Ontology Merging tasks. Similar to MAFRA, they suggest the use of domain knowledge and management of changes in source and target ontologies in two umbrella activities.

Two other papers that develop their own inference algorithm are discussed in [65][109]. They both use very similar approaches to solve the mapping problem. They try to find a CPG-DKO (selected either manually or automatically) and use it as an Interlingua between the two mapped ontologies. Both of the approaches use very basic string comparisons to make very simple mapping (equivalence, subsumption, etc.) between ontologies and the background ontology. They also introduce their own basic reasoning algorithms to infer new relations between elements of the mapped CPG.

Anchor-Prompt [66] is another approach which performs reasoning on the ontologies and their mappings. This algorithm takes two ontologies and the mappings as the input and finds the pair-wise similarity measure between all elements of the ontologies. This algorithm treats classes in the ontology as nodes and properties as links. After anchoring, all of the existing paths between anchor points in one ontology are traversed and compared with the paths in the other ontology. For instance, suppose that *a* is mapped to *b* and *c* is mapped to *d* (*a* and *c* from ontology 1 and *b* and *d* from ontology 2). Every existing path between *a* and *c* in ontology 1 and every existing path between b and d in ontology 2 are traversed. If node e from ontology 1 and f from ontology 2 have the same position in the paths traversed in their ontologies (e.g. being the second node in the path for instance) their similarity is increased in a similarity matrix. This approach does not fully exploit the expressivity of ontologies and can only infer extremely simple mappings.

2.2.3. Conclusion

In this section, we reviewed four approaches of ontology mapping discovery: 1. Name based, 2. Structure based, 3. Instance based and 4. Semantic based approaches. Semantic-based approaches have two steps (a) anchoring and (b) reasoning relations. Anchoring is done with the help of another mapping technique and the reasoning step is performed via a reasoner. Description logic, propositional and proprietary reasoning formalisms have been used for the reasoning step.

We believe that propositional logic is not a good option for reasoning due to its lack of expressivity for representation of the complex mappings. Moreover, using proprietary algorithms has following limitations: (1) a reasoning engine should be developed from scratch; (2) since these engines can only perform reasoning on the mappings and not the ontology representation languages (e.g. OWL) they cannot exploit the internal structure of the mapped ontologies to draw new mappings

Review of the semantic based mapping literature showed that DL has been widely used for debugging and deriving simple mappings (equivalence, etc.) but no attempt has been made to derive more complex mappings. We believe that lack of an expressive mapping representation language in DL is putting this limit on the capabilities of the semantic based merging techniques.

2.3. Representation of Mappings

Most of the research in the area of ontology mapping has been focused on discovery of *mappings* using different sources of knowledge that can come from external resources (domain knowledge or thesauri) or the internal structure of the ontologies. The output of the existing mapping discovery algorithms are usually in the following form of (e_1, e_2, R, c). $e_1$and e_2 are the elements that are mapped from ontology 1 and ontology 2 respectively and R is the relationship that they have (e.g. equivalence) and c is the level of the confidence that is usually between 0 and 1. However, automatic mapping discovery algorithms are not capable of finding complex m-n mappings that may contain structural modification and mathematical operators. These mappings can only be detected by a domain expert familiar with ontologies. In order to perform automatic instance transformation and ontology merging these mappings identified by the domain expert should be modelled in a computer understandable format. Since the mappings can be quite complex, an expressive mapping representation language is needed. In this section, we first review the identified requirements of these representation languages in section 2.3.1 and then review some of the existing languages in section 2.3.2.

2.3.1. Requirements for Mapping Representation Languages

We reviewed the publications that identify requirements of the mapping representation languages [67][68][69][70][71][72][73][74]. These publications present overlapping sets of required features using different vocabularies. Below we present the list 8 general features identified in these papers in a unified wording:

1. Clear syntax

Clear syntax is important for both human and computer understanding. It makes it possible to create, store, parse, modify and share the instantiations of these languages using computer programs. This features has been identified in all of the reviewed literature [67][68][69][70] [71][72][73][74].

2. Formal Semantics

Semantics of the mappings should be formally represented in a computer interpretable format. Defining semantics enables us to perform reasoning on the mappings and the mapped ontologies. Formal semantics for reasoning on the mapped ontologies are necessary for the reasoning step of the semantic-based ontology mapping approaches discussed in section 2.2.2.4. With no formal semantics defined for the ontology mapping representation language, automatic instance transformation and ontology merging is not possible. This feature has been mentioned in [67][74]. Moreover, reasoning on the mappings can help us find several relations between the mappings such as equivalence and entailment. Finding these relations can help us to optimize the mappings. For instance, there is no need to have two equivalent mappings and one of them can simply be discarded.

Query re-writing is another identified feature in [70] that is closely related to formal semantics. Imagine ontology A is mapped to ontology B and there is an application that makes use of the knowledge encoded in ontology B by querying it. If we are interested in using the contents of the ontology A in the application at hand, the queries written for ontology B should be re-written in terms of the ontology A based on the existing mappings and then the results should be translated back to ontology B. This is called query re-writing. To the best of our knowledge, no research related to ontology query re-writing exists. However, we believe transforming instances of the ontology A to ontology B can solve the problem of query re-writing and translating the knowledge back to the ontology B. As we discussed earlier, formal semantics is needed for instance transformation. Therefore, query re-writing can be regarded as a sub-feature of the formal semantics.

3. Expressivity

Expressivity is the capability of the mapping representation language in representation of complex mappings. The more expressive the mapping language, the more detailed and accurate the mappings are. All the reviewed publications [67][68][69][70][71][72][73][74] discuss the necessity of expressivity of the mapping representation languages and introduce a number of its sub-features:

a. Accommodation of incompleteness

Incompleteness in mapping refers to the case when the target ontology does not contain the piece of knowledge that corresponds to the entity from the source ontology or the mapping is too complex to be encoded and an incomplete mapping suffices. Other than [67] no other publication mentions this as an expressivity need.

b. Expressing functions to manipulate numbers, structures, string, dates, etc.

It is often the case that mathematical, string and date calculations on the data values of the source ontologies should be performed in order to accurately express the mappings. For instance, the values of the height and weight of a person in the source ontology may be mapped to the value of the BMI (Body Mass Index) in the target ontology. As another example, weight of a person may be in pounds in the source ontology whereas it is in kilograms in the target ontology. String, date and numerical computational needs of ontology mapping are mentioned in [69][70][72].

c. Expressing *m*-to-*n* mappings:

Any mapping more complex than 1-to-1 mapping including class-to-property and property-to-class mapping can be categorized as m-to-n mapping [69][70]. Therefore, this feature is simply being able to express mappings more complex than 1-to-1 mappings.

d. Capturing recurrent mapping patterns

Even though each mapping problem has its own peculiarities, the same mapping patterns are repeated frequently in many of them. In order to speed up the process of mapping by using the existing templates for the mappings and reuse the existing mappings, these patterns should be captured and modeled in the mapping representation language [70]. This makes the mapping process more manageable and modularized. They are similar to functions in programming languages. Instead of writing the mappings multiple times, you write them once and use them multiple times. An instance of one of these mapping patterns is Property-to-Class mapping pattern. In this mapping pattern, two instances connected to each other using a property are mapped to an instance of a class. This property will be replaced by an instances of a class that connects those two instances using two other properties. For instance, mapping of *hasiWife* property to class *Marriage* will cause the triple ":john :hasWife :jane" to be mapped to the following triples in an OWL ontology.

```
:janeAndJohnMarriage a :Marriage;
     :hasMalePartner :john;
     :hasFemalePartner :jane;
```

A comprehensive library of the mapping patterns has been listed and discussed in [165].

e. Expressing conditional mappings

The importance of conditional mappings in terms of the type and values of the attributes associated to classes is pointed out in [73]. A mapping should only be used for ontology merging and instance transformation if its conditions are satisfied. A mapping representation language should be able to model a wide range of conditions and conditional mappings. Conditions of a mapping might be based on the (1) values of properties, (2) Number of different values that properties have. For example, in order to map the class *Person* from the source ontology to the class *Adult* in the target ontology, the condition is that the value of the property *hasAge* in the source ontology should be greater than 18.

f. Variable:

A mapping representation language should support variables. Variables are useful for three purposes: **(1)** They can be used to address a subset of the ontology rather than a specific elements of the ontology [69]. For instance, variables can represent all the instance of the class *Person* with at least two values for the property *ownsCar*. This variable represents people that own at least two cars. This feature helps with the representation of the m-to-n mappings mentioned previously; **(2)** Variables are needed for value transformations during the mapping [70]. For instance, imagine a mapping scenario in which the value of BMI is computed using the supported math functions in the source ontology and it is supposed to be assigned as the value of the property *hasBMI* in the target ontology. If the BMI was a value of a property in the source ontology, let us say *hasBodyMassIndex* we could just simply create a equivalence relation between the properties *hasBMI* and *hasBodyMassIndex*. However, since BMI is not the value of a property it should be transferred as a variable to the target ontology and be assigned as the value of the property *hasBMI*. **(3)** Creating rich semantics among the transformed entities: using variables makes the transformed elements in the target ontology available for creating relations between them using target ontology properties [74].

g. Expressing relations between the mappings

Mappings can have several relations with each other. The identified relations in literature are Specialization, Abstraction, Composition and Alternatives. Composition has been identified in [62][68] and the rest of them are identified only in [62]. Specialization relation express that a mapping is a specialized form of another mapping. For instance, we can first define a general mapping that maps two super classes and then define specialized mappings that take care of the subclasses of these two more general classes. Abstraction is similar to specialization with the difference that the more general mapping does not participate in the mapping an act as an abstract mapping that other mappings can specialize. Composition relation shows that a mapping is actually composed of several other mappings. For instance, if m12 is mapping class *C1* to class *C2* and m23 is mapping class *C2* to class *C3*, m13 that is a mapping between classes C1 and C3 is actually the composition of the mappings m12 and m23. Alternative relation shows that several mappings are mutual exclusive alternatives and only one of them should be considered for mapping if its conditions are satisfied. For instance, mappings m1 and m2 that map the instances of the class *Person* in the source ontology to the disjoint classes of *MalePerson* and *FemalePerson* classes in the target ontology are alternative mappings.

h. Expressing the necessary structural transformation

Besides the data values in an ontology, structure of the ontology is sometimes needed to be modified during the mapping [69]. The following sub-features are reported in the literature as the possible structural modification operations:

i. Counting:

By counting we mean counting the number of values of a property in an ontology and assigning this number to value of an object property in the source ontology [70]. As an example, imagine two ontologies about the research faculty of a department. In the first ontology, the property *hasPublication* indicates the publications of a researcher while in the second one there is a property called *numberOfPublications* that only holds the number of publications. In order to transform an instance from the source ontology to the target ontology, the number of values of the property *hasPublication* should be counted and assigned as the value of the property *numberOfPublications*.

ii. Treating classes as instances and vice versa:

Because of different modeling techniques that are taken and different views of the world that people have, it is possible to see a similar concept modeled as a class or as an instance in two different ontologies. For instance, airplane may be class in the source ontology while it is an instance of the class *TransportationMedium* in the target ontology. Therefore, the mapping representation language should be able to map classes to instance and vice versa [70].

iii. Creating classes, properties, instances:

During the instance transformation or ontology merging, new classes, properties or instance may be created to alleviate the problem of lack of overlap or to cope with the different approaches in design. A mapping expression language should facilitate creating these necessary ontology elements [71] .For instance, in the property-to-class transformation discussed earlier, an instance of a specific class should be created in the target ontology in order to deal with different modelling approaches.

4. Language independence

Mapping representation languages should ideally be able to map any two ontologies regardless of their representation languages [68][74]. This requirement is almost impossible to be achieved since each language has its own peculiarities and no mapping representation language can cope with all of those differences. Even if in an ideal world such a language existed, there was no reasoner that could perform reasoning on multiple representation languages in a semantic-based ontology mapping approach at the same time. This feature has been called "handling heterogeneity" in [67] which is a reference to syntactic heterogeneity.

5. Have existing applications that support loading, saving, manipulation of and reasoning on the mappings

A mapping representation language needs applications for loading, manipulation, visualization and storage of the mappings [69]. Availability of these applications means less implementations and easier integration with the existing systems. Therefore, availability of these applications can greatly improve the acceptance of a mapping representation language among researchers, developers and users. This feature has been called "being web ready" in [68].

6. Capture the meta-data that can help reusing the mappings

It is suggested that a mapping representation language should capture the meta-data related to the mappings. Authors' information, the mapping algorithm and parameters used in the algorithm and versioning information [69][70] are among the meta-data that are identified useful to be captured. Capturing meta-data (1) improves the shareability of the mappings and (2) facilitates the maintenance of the mappings.

7. Be multipurpose

Mapping representation languages should be able to map two ontologies regardless of the domain knowledge they cover [68][70]. Therefore, the mapping representation language should be as general as possible.

8. Be able to map OWL ontologies

OWL that is a sub-set of the Description Logic has gained great popularity in recent years as an ontology representation language. This popularity has led to development of a huge number of working ontologies in this language. Therefore, it is a great advantage for a mapping representation language to be able to map these already developed OWL ontologies [70]. However, it is not discussed what features a mapping representation language should have in order to be able to map OWL ontologies.

2.3.2. Existing Representation Languages for Mappings

Six languages that are designed or can be used for ontology mapping representation including OWL, C-OWL, SWRL, MAFRA are discussed in this section and their advantages and disadvantages in the context of the mapping representation are described. Our evaluations show that all these language are multipurpose and are not proprietary to any special domain and can be used for mapping OWL ontologies. Therefore, we do not discuss these aspects of the ontology mapping in these languages.

2.3.2.1. OWL

OWL is the W3C standard language for representation of the ontologies. It also contains constructs that can be used to map ontologies. Some of the examples are the following constructs that can be used for mapping of ontology classes and properties: *owl:subClassOf*, *owl:equivalenClass*, *owl:subPropertyOf* and *owl:equivalentProperty*. For instance, the following triples show a mapping between two ontologies about cars: *o1:SportCarowl:subClassOf o2:Car*. Some shortcomings in regard with ontology mapping have been noticed in several publications:

A disadvantage which is mentioned in [68] is the language dependency. They believe that OWL is not language independent in the sense that in order to perform reasoning on the mappings and benefit from the internal structure of the ontologies, ontologies should be in OWL. This is true but none of the existing representation languages is capable of performing reasoning on the mappings and taking into account the content of the ontologies regardless of their representation format. Therefore, this is a disadvantage that is not specific to OWL. Some other disadvantages in regard with ontology mapping are mentioned in [70]:

a. **Tight coupling between ontologies:** If we import an ontology directly in another ontology and then articulate the mappings, the definition of the entities in two ontologies rely on each other and a tight coupling which is not desirable exists between them. This makes it necessary to have access to both of the ontologies when we are performing local reasoning on one of them. Dependency of ontologies is highly undesirable in the semantic web and this is one of the most important reasons of adoption the Open World Assumption in semantic web tools.

b. **Lack of expressivity**: Certain sorts of mappings are not expressible in OWL. For instance, transforming a value from a property in an ontology to a property in another ontology is not possible.

c. **Epistemologically inadequacy:** It is also mentioned in [70] that "*We argue that the mapping constructs provided by OWL are also epistemologically inadequate, mainly because the OWL language was not conceived as a mapping language, but as an ontology modeling language.*" They believe that the DL constructs available in OWL are more useful for describing merged ontologies rather than the mappings between them.

Two other issues are mentioned in [58]: (1) to perform reasoning on the ontologies and mappings in OWL they should be considered as a single ontology. This ontology may become computationally expensive to perform reasoning on. (2) Ontologies may contain inconsistent pieces of knowledge. Reasoning on these ontologies is not possible in semantic web.

Advantages of this language in the context of the ontology mapping representation are 1. Clear syntax, 2. Formal semantics, and 3. Existence of applications and API for loading, saving, manipulation of and reasoning on the mappings

2.3.2.2. C-OWL

As we mentioned in previous section, reasoning on the ontologies that are mapped using OWL constructs can be expensive in terms of computation or even impossible due to inconsistencies in the mapped ontologies. To overcome these problems, an extension of

OWL which is called Contextualized OWL (C-OWL) is proposed in [58]. This version of OWL enables the ontology designers to design contextualized ontologies. Contextualized ontologies can keep their content local and establish mappings with other ontologies using the constructs (semantic bridges) that are devised in C-OWL for this mean. Having inconsistent knowledge is not a problem as there is no need to perform reasoning on a global interpretation of the ontologies for mapping and only the pieces of knowledge that are shared with other ontologies will be used for reasoning. This solves the problem of being computationally expensive as well. Five types of semantic bridges are defined: 1. Subsumption, 2. Equivalence, 3.Containement, 4. Disjunction and 5. Intersection. Syntax and semantics of this ontology representation language is formally defined in[58].

The advantages that this mapping representation language has are 1. Clear syntax, 2. Formal semantics and 3. The ability to represent mappings between OWL ontologies.

The biggest disadvantage that this language has is the simplicity of the constructs that are defined for mapping representation. This language is hardly capable of expressing any complex mappings. Moreover, no application supporting this language exists and no meta-data support is provided.

2.3.2.3. SWRL

SWRL is a proposal for representation of rules in semantic web. SWRL allows writing rules in terms of OWL constructs, classes, properties and instances in order to achieve a higher level of expressivity and reasoning. The only publication that discusses the possibility of using SWRL for ontology mapping is [75]. A useful feature of SWRL in ontology mapping is its built-in functions that make it possible to execute aggregation, mathematical and string functions etc. However, this language lacks the necessary constructs needed for representing complex mappings. This language lacks the expressivity to represent Boolean functions, conditions, structural transformation and meta-data.

Advantages of this language in the context of the ontology mapping representation are 1. Clear syntax, 2. Formal semantics, 3. Existence of applications and API for loading, saving, manipulation of and reasoning on the mappings and 4. The ability to define variables and express date, mathematical and string computations.

2.3.2.4. MAFRA

MAFRA [62] (MAppingFRAmework) is a semantic bridging ontology. Two important classes of the ontology are the *SemanticBridge* and *Service*. Semantic bridges indicate a mapping between two elements of the ontology. A *SemanticBridge* is further specialized to *ConceptBridge*, *AttributeBridge* and *RelationBridge*. The class service represents a class that is used to reference resources that are responsible to connect to, or describe transformations. This class is intended to be used to describe these transformations resources. Transformations that are performed by services are not executed by the transformation engine and they are external to it. Their inputs can be literals, classes, properties or an array of them. *Condition* and *Alternative* classes can be used to define conditional mappings and the different alternatives that exist for a decision. Conditions can be combined using *and*, *or*, *xor* and *not* operators. For instance, an *Individual* from ontology 1 can be mapped to either one of the *Man* or *Woman* classes of ontology 2 based on the gender. Another example from [62] can be found below:

```
<Transformation rdf:ID="copyName">
        <mapSourceArgument>
                <MapArg><from
        rdf:resource="#name"/><to>sourceString</to></MapArg>
        </mapSourceArgument>
        <mapTargetArgument>
                <MapArg><from>targetString</from><to
        rdf:resource="#name"/></MapArg>
        </mapTargetArgument>
        <inService>CopyString</inService>
</Transformation>
```

This mapping makes use of a service called "copyString". The other service explained in the same paper is *CountRelation* service that is used for counting the number of different values that an entity has for a specific property. To find out how expressive MAFRA is, we downloaded MAFRA Toolkit from http://mafra-toolkit.sourceforge.net/. Most of the operators available in the toolkit designed for string manipulation and no operators exist for mathematical transformations. The only other service that is useful for structural modification is the *CountRelation* that supports counting. Additionally, no formal semantics or meta-data support is provided. This language does not support pre-defined mapping patters.

The advantages of this mapping representation language are 1. Clear syntax, 2. Existence of tools for loading, saving, manipulation of the mappings and 3. String manipulation functions. Moreover, MAFRA is the only mapping representation language that can express composition, specialization, abstraction relations between the mappings.

2.3.2.5. A Language to Specify Mappings between Ontologies

This language [73] contains constructs to express mappings between classes, relations and attributes. Table 2.2 shows the predefined relations that can be established between entities of ontologies.

Table 2.2 Constructs of the mapping language in [73] and their description

Language Construct	**Description**
ClassMapping	Mapping between two classes
AttributeMapping	Mapping between two attributes
RelationMapping	Mapping between two relations
ClassAttributeMapping	Mapping between a class and an attribute
ClassRelationMapping	Mapping between a class and a relation
ClassInstanceMapping	Mapping between a class and an instance
IndividualMapping	Mapping between two instances

Each type of the constructs in Table 2.3 has its own set of operators that can be combined with entities of the same type. For instance, a PhD student is a subclass of both *Researcher* **and** *Student* Classes. Table 2.3 shows the set of operators that can be applied to each type of the constructs in the ontologies.

Table 2.3 Set of ontology manipulation operators presented in [73] and their associated input entity types

Entity	**Operator**
Class	and, or, not, join
Attribute	and, or, not, inverse, symmetric, reflexive, transitive, closure, join
Relation	and, or, not, join

Another important aspect of the mapping which is covered in this paper is the support for conditional mappings. Mappings can be conditional based on the existence of attributes, their types and their values. A programming interface is developed to facilitate the use of this language. As you can see in Table 2.2 this language provides a user with some frequently used mapping patterns.

No mathematical or string manipulation functions are defined and formal semantics does not exist for this language .

2.3.2.6. An Instance Mapping Ontology for the Semantic Web

Another language is introduced by Lei [74]. They discuss that many of the existing representation mapping languages are expressive enough to be used for articulation of complex mappings that create proper classes, instances and properties in the target ontology. What is missing is the ability to create rich semantics between the created entities. To fill this gap, they have boosted their representation language with constructs that make it possible to use the transformed entities to the target ontology and incorporate them in the mapping process. For instance, suppose that in an ontology mapping scenario in which an instance of the class *Publication* (p1) and an instance of the class *Researcher* (r1) are being created in the target ontology. This language makes it possible to create the relation *hasPublication* between instances p1 and r1 in the target ontology. Table 2.4 shows the language constructs.

Table 2.4 Constructs of the instance mapping ontology in [74]

Construct	Description	Slots
OntologyMappings	Grouping the mapping instructions for generating semantic data entries.	has-instance-mapping
InstanceMapping	Describing how to generate instances from the specified source data objects	has-source-dataset has-target-dataset has-source-class-name has-target-class-name has-instance-constraint has-slot-value-mapping has-correspondent-slot
InstanceConstraint	Describing constraints on the instances of the specified source class	has-constrained-source-dataset has-constrained-class-name has-constrained-expression
SlotValueMapping	Expressing how to generate values for the specified target slot from the specified sources	has-target-slot values-from-slot values-from-existing-instances values-from
DataSet	Describing how to access databases, knowledge bases, or XML documents.	has-db-name has-db-url has-db-server
ConstraintExpression	Expressing constraints	has-constrained-slot-name has-constrained-operator has-constrained-value has-logic-operator

No formal syntax, meta-data modelling solution or supporting tools are provided for this language. This language also does not support mathematical, date Boolean or string computations. This language has clear syntax and supports many of the commonly used mapping patters.

2.3.3. Conclusion

In this section, we reviewed the literature concerning requirements for an ontology mapping representation language. Our review and existing review of these languages [69] show that they suffer from lack of expressivity in several aspects. These languages mostly support equivalence and subclass relations only, are not able to represent variables, complex conditional mappings and mathematical functions. We also observed that even though the intended semantics of these languages can be conveyed from their descriptions and names, they do not have formal semantics that can be used for reasoning. Availability of tools for loading, saving and manipulation of mapping representation languages is another limitation of the reviewed mapping representation languages. OWL is the only language that is widely agreed upon as a standard. Several tools are developed to load, save, manipulate, visualize and reason over OWL ontologies.

Most of these existing representation languages are designed to be utilized by automatic mapping discovery algorithms that are unable to find complex relationships between elements of the ontologies. However, domain exerts are able to find intricate mappings that cannot be represented by these languages. Developing an expressive mapping representation language that is supported by existing tools and defining its formal semantics to be used by reasoners can be two great steps forward in this research area.

2.4. Merging of Computerized Clinical Practice Guidelines

2.4.1. Introduction

As we discussed earlier, computerized CPG have been developed for patients with just one disease and do not usually take into account the simultaneous presence of other medical conditions—the clinical term for such conditions is *comorbid conditions*. To handle comorbid conditions within a CDSS, it is not possible to simultaneously operate on multiple CPG as one needs to be cognizant of the interactions between multiple CPG—the recommendations from one CPG may counter the recommendations from another CPG resulting in a situation that is harmful to the patient. Therefore, to handle comorbidities in a CPG within an CDSS, one solution is to systematically synthesize/merge the computerized CPG of the individual comorbid diseases in order to obtain a unified CPG which resolves the potential interactions between multiple CPG and in turn offers recommendations that are consistent with the comorbid conditions.

Merging CPG is best defined in [107] as "*The challenge is to align multiple CPG of the comorbid diseases whilst maintaining the integrity of medical knowledge and task pragmatics, and ensuring patient safety*". In order to successfully merge several computerized CPG, the *Morphing constraints* should be captured by interviewing the domain expert. Merging constraints represent the knowledge morphing constructs in the CPG merging framework that describe how the medical actions in each of the CPG should be modified for comorbid patients in order to avoid conflicts and adverse interactions. Domain expert will notify the user of possible conflicts, similarities and any other aspect that should be considered during treatment of a patient using several CPG. This knowledge will be used to guide the merging of multiple CPG.

There have not been many attempts to merge computerized CPG in order to develop a CDSS for patients with comorbidities. Merging CPG in general can happen at four levels [107]: **(a)** Guideline level (before computerization) **(b)** Modeling Level (during computerization) **(c)** Pre-Execution Level (after computerization, before execution) and **(d)** Execution Level (after computerization, during execution).

Merging computerized CPG can have several benefits. The following list shows benefits of merging computerized CPG for comorbidities [20][21][22][24][25][26][29][30][107]:

1. Avoiding unnecessary duplication of tasks, visits and medical tests and reusing their results.

2. Identifying potential adverse interactions that may arise because of simultaneous application of several treatments on the patient and preventing from them since they may compromise patient's safety.

3. Standardization of care across multiple institutions for comorbid patients.

4. Timely and cost effective treatment of patients by reusing medical tests and tasks and avoiding adverse interactions.

2.4.2. Guideline Level Merging

Textual CPG can be merged by physicians on the paper. Then the merged CPG can be computerized like any other CPG. If merging happens at this level, the CPG computerization methodology does not need to provide a tool for merging and many of leading computerization methodologies can model and execute such merged CPG. There might be a need for some extra level of expressivity for modeling these comorbidity CPG as they can be more complex than the CPG for single diseases. No research exists about this topic.

2.4.3. Computerization Level Merging

The next level of merge can happen in the computerization level. In this level, multiple paper-based CPG are merged into a single computerized CPG. The result is a unified computerized CPG that contains the knowledge of the all of the paper-based CPG. Physicians and health informaticians can identify the common steps and inconsistencies among several CPG, establish the necessary connection between them and model all CPG in a unified computerized CPG. This makes it possible for the execution engine to take advantage of this pre-execution merging and execute the unified CPG to generate a therapy plan which is concordant to all of the CPG participating in the treatment of the patient.

The literature [20][21] describe a method for merging ontologically modeled CPG in OWL language. The authors' goal is not to merge computerized CPG of multiple diseases but to merge location-specific CPG for a specific disease into one unified CPG that can be utilized by all the locations. They indicate that the same CPG may have different tasks, treatments, follow ups, clinicians, task intervals and frequencies at different institutions and provide the necessary ontology elements to model the CPG and their location based merge. They successfully merge three prostate cancer CPG from three locations into one unified CPG by finding the common tasks. This unified CPG can be used in each of the three locations by using branch nodes to deviate from the unified CPG and to go to institution specific tasks and come back to the unified CPG by using synchronization points. We have not come across any other publication that performs computerization level merge.

2.4.4. Pre-Execution Merging

The next level is to merge the CPG after they are computerized. These computerized CPG are merged into a single computerized CPG. They can either be merged automatically or manually. In the manual merge, physicians and health informaticians identify the potential incompatibilities, adverse interactions and common tasks among the computerized CPG and merge them into a unified computerized CPG manually. These methods are explained in section 2.4.4.1. The other approach is to try to solve the problem using techniques from computer science. Using these techniques can automate the merging process. Section 2.4.4.2 describes these methods.

2.4.4.1. Manual Merging of Clinical Practice Guidelines

In this level of the merge, at least two CPG that are already computerized are merged into a single CPG. This merge is done using special constructs devised in the CPG modeling language for this purpose.

An example of this research is [107]. The researchers first computerize Chronic Heart Failure (CHF) and Atrial Fibrillation (AF) CPG using the OWL language. These separately computerized CPG are then merged into a unified model based on the domain experts' opinion. The unified model is composed of three parts: (a) CHF tasks, (b) AF tasks and (c) common tasks. The common tasks part represents the identical shared tasks between the two CPG. A set of preconditions are set for the common tasks to make sure that they are executed solely for the patients who have the comorbidity and their patient record indicates the need. In a similar research [100], parts of guidelines are re-used and put together manually in order to create a therapy plan for comorbid patients.

A conceptual framework for merging clinically modeled CPG is designed in [22]. It suggests the use of SWRL to represent the similarity of elements from different CPG. Capturing the commonalities of CPG using SWRL rule enables the execution engine to use a SWRL engine for dynamic merge the common element in line with patient data. No further details that can be used towards implementation of this framework are provided. It is mentioned in [30] that utilizing representation formalisms that allow defining CPG elements once and reusing and re-instantiating them multiple times (like OWL) can pave the way for merging CPG at the pre-execution level but no further detail is provided.

2.4.4.2. *Automatic Merging of Clinical Practice Guidelines*

Computer programs can merge the computerized CPG automatically. To the best of our knowledge, [26] describes the first proposed method for automatic merging of CPG. This merge algorithm works with the SDA* representation language [27]. SDA* language is based on three clinical elements: *States*, *Actions* and *Decisions*. A state describes the patient's status. For instance, it can show that patient is taking a specific medication or have a specific medical condition. These states can be used as precondition of medical actions or as entry points to the CPG. Actions are always followed by states and other elements can be followed by any other type of elements. This is how the CPG are merged:

A. For each pair of the (*State*,*Action*) in the CPG, all of the paths between them are enumerated and stored in the State-Decision-Action structure (SDA structure) by considering different decisions that can be made and ignoring the actions in the path. Therefore if two binary decisions exist between a state and an action, four paths will be enumerated regardless of the number of actions in between. Each SDA structure shows that action A should be performed if the decision D has been made in state A. Figure 2.9 shows an example of the SA generation step. In this step the SD part of the SDA structures are (A^B) and (A^~B) and the A part is Action1 and Action2 respectively.

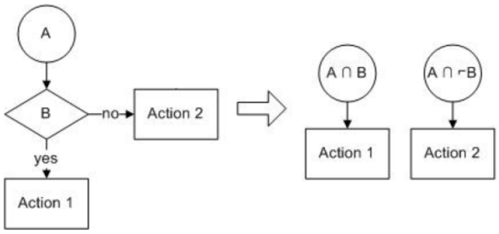


Figure 2.9 CPG to SDA transformation process [26]. State primitives are shown as circles, decision primitives as diamonds, and action primitives as squares.

B. After computing the SDA structure of the all of the CPG, they are combined into a single SDA structure. Every SDA structure of the first CPG is combined with all of the SDA structures of the second one. Combining of two SDA structures is equal to computing the union of SD part of the SDA structures and union of the A part of the structures. Having *m* and *n* SDA structures in CPG will result in *m.n* SDA structures in the combination of the CPG.

C. Two sets of rules are applied to the combination of these two SDA sets. *Restriction Rules* are used to remove the possibility of having contradicting State-Decision or State-State pairs in a patient. For instance, a restriction rules may state that state *Anorexy = true* and the decision *isPatientObese = true* are not consistent. S*ubstitution Rules* are used to remove the possibility of having adverse effects because of undesired interactions. For instance, it may state that if a patient has fever and is receiving two specific medications, he should stop taking one of them and rest. These rules are applied to the SDA structure repeatedly until no further change is possible.

D. Domain experts are asked to predict what would be the state after execution of A in every SDA structure. Therefore, SDA structures are transformed to State-Decision-Action-State (SDAS) structures. These structure are merged into a unified CPG based on the similarity of beginning state and ending state of SDAS structures [28].

The CPG merging method described above suffers from several limitations: **(a)** The outcome state of each SDA structure in each of the merged CG should be foreseen by domain expert prior the merge [29];**(b)** It is not very straightforward and sometimes even not possible to predict comorbid patients' state for all of the SDA structures [29]; **(c)** During a merge, CPG are broken down into SDA structures merged and put together to form a single merged CPG with a possibly totally different order. This is highly undesirable since these CPG are developed by a group of experienced domain experts based on the evidence based literature of the related field and rearranging these CPG can be very dangerous for the patients; **(d)** In this approach, CPG are not merged based on the similarity of the tasks but based on the similarity of the outcomes and states. Therefore, it can't merge similar tasks into one task in order to save time and money in treatment of the patients; **(e)** Temporal aspects are neglected. To alleviate the first two deficiencies, it is proposed to use machine learning techniques to automatically extract the necessary information regarding SDAS structures to relax knowledge dependencies of the algorithm from clinical data [29]. This makes situation grimmer as the role of the domain expert is removed from the process.

Another research that describes a method to solve the adverse interaction sub-problem of merging comorbidities' CPG is discussed in [25]. They propose a solution based on constraint logic programming that performs modifications on CPG to remove the adverse interactions and contradicting states according to the patient information. When it is not possible to remove the threats to the patients' wellbeing via a modification, appropriate warnings are generated for the healthcare professionals. This method can only be used for removing adverse interactions and cannot be used to avoid duplication of tasks, etc. The key principle of their method is the use of *mitigation operators*. These operators indicate what changes should be made to the first CPG in the case of occurrence of a specific pattern in the first or second or both of CPG. For instance, a mitigation operator may indicate that *asprin* should be replaced with *clopidogrel* in the first CPG when *asprin* is prescribed in the absence of *antiplatelet* drugs (in any of the CPG). These mitigation operators are iteratively applied and the corresponding tables containing enumerated paths are modified accordingly and a solution that does not violate the constraints is sought. If a solution is not found another mitigation operator is applied and a solution is sought again. Two sets of constraints should be respected by a solution: (1) The set of constraints that mitigation operators represent and (2) The set constraints that indicate that if there are common tasks in the two CPG they must have the same binary value for their execution variable. Same binary value for two CPG elements means that if they are tasks they must either co-occur (not necessarily at the same time) or do not occur at all and if they are decisions or enquiries they must have the same outcome in both of the CPG. They use ECLiPSe [23] to solve this constraint satisfaction problem.

This method has the following limitations: **(a)** their workflow model is unrealistically simple. The only workflow patterns that they support are binary decisions and simple sequential tasks; **(b)** only 2 CPG can be merged using their algorithm; **(c)** temporal aspects are not considered; **(d)** a solution generated by their algorithm is two execution paths (One for each of the merged CPG) that respect all the constraints. Existence of such an execution path for each of the CPG means that if we make specific decisions and task have specific outcomes we are sure that CPG can be applied to the patient simultaneously without any contradictions and constraint violations. This solution is only useful for that specific scenario because the decisions that are going to made by the healthcare professional and the outcome of tasks are almost impossible to be foreseen in advance. As a result, if the user does not make those specific decisions or tasks do not have those specific outcomes indicated in the solution adverse interactions or violation of constraints are possible to happen. The same research is discussed in an earlier paper [24]. The different is the introduction of mitigation operators in [25] to remove the inconsistencies in addition to finding them. In a more recent publication [98], the ability to handle iterative tasks and numerical measurements is added to this framework.

2.4.5. Execution-Time Merging

In this approach, a unified CPG is not created beforehand and it is the execution engine's responsibility to merge the CPG during the execution dynamically. Even though there are publications [21][30][99][188] that propose methods for execution-time merging of CPG, there is no implementation of them and to the best of our knowledge there is no other computer system that are capable of performing execution time merging. These are the only publications that introduce the idea of merging CPG at the execution level. Two scenarios under which execution time merging (reusing tasks and their outcomes) is possible are introduced in publications [21][30]: **(a)** There are common tasks among CPG and they should be executed at the same time. If all the CPG reach that point at the same time, the common task is executed once but all of the contributing CPG can proceed to the next step. If a CPG is ahead of others in terms of execution of the common tasks, it should wait (if possible) for the rest to catch up. It is necessary to know how long it takes for the behind CPG to reach the common step and how long the CPG ahead is capable of waiting; **(b)** common tasks may not need to happen at the same time but they can have results or effects lasting long enough to be reused by other CPG. A mechanism should be devised in order to ensure the validity of the result when the other CPG are using them.

Publication [99] proposes a multi-agent framework for dynamic CPG merging. In this framework, each agent is responsible for generating a therapy plan for a patients based on a disease-specific CPG while they share their associated CPG status with the rest of the agents in order to avoid harmful interactions. Each agent creates a comorbidity therapy plan and the best one is chosen among all the therapy plans to be followed. However, several implementation details and research questions are left unanswered in this framework. The

GLINDA project's [188] goal is also to develop a multi-agent system for comorbidity decision making based on several clinical guidelines. They suggest using an ontology to represent medical interactions such as drug-drug interactions between several CPG. No actual implementation of the guideline interaction ontology or their multi-agent system exists to the best of our knowledge.

We argue that execution level merging, although a challenging exercise is the most viable way of merging two CPG based on the current state of the patient and the current state of a CPG's execution. In other words, merging is decided upon in response to the current state of the CPG execution as opposed to pre-execution CPG merging with priori assumptions about the outcomes of decisions, values of variables and CPG starting times.

2.4.6. Conclusion

Literature review shows that merging CPG can happen at four levels: (1) before computerization (CPG level), (2) during computerization, (3) before execution and (4) during execution. Lack of research in this area can be greatly felt. No research related to merging CPG at the guideline level is reported in the literature. There exists only one publication that tries to merge CPG during computerization [20]. The most studied topic is merging CPG after computerization (Pre-execution) using CPG merging algorithms. These algorithms still need many improvements as they can only reconcile conflicts but no attempt has been made to reuse the tasks and their results. These methods suffer from other deficiencies that stop them from practically being used in real clinical settings. Only one proposal exists for merging CPG during the execution [21]. However, no implementation exists for this conceptual framework. We have not come across any other CPG computerization and execution framework that is able to perform execution time merging.

A general shortcoming of the existing CPG merging methodologies is the simplicity of merging constraints. We believe that a much broader range of merging constraints should be identified and incorporated in CPG merging methodologies. Additionally, in order to utilize the new identified merging constraints in automatic CPG merging algorithms, they should be represented in an expressive computer understandable format. None of the existing approaches has such an expressive language due to simplicity of their merging constraints.

CHAPTER 3: *ONTOMORPH:* A SEMANTIC WEB BASED KNOWLEDGE MORPHING FRAMEWORK

3.1. Solution Approach

In this section, we propose a Semantic Web architecture in order to address research challenges discussed in section 1.2 to develop a knowledge morphing framework.

Semantic web technologies provide us with (a) Web Ontology Language (OWL) that is used for formalizing and organizing knowledge in a domain area; (b) improvement in knowledge sharing between different parties and semantic operability between computer systems that use ontologies by providing a standard representation; (c) a rule language called Semantic Web Rule Language (SWRL). SWRL improves expressivity of OWL in many aspects such as numeric, date, string and Boolean comparison and computation functions and (d) Several open-source and publicly available software tools for manipulation, loading and saving, visualization of ontologies and reasoning on them. These features make OWL a suitable logical framework for knowledge morphing. Our Semantic Web based Knowledge Morphing Framework called *OntoMorph* is realized through the following main tasks (as shown in Figure 1.1):

- **Task #1**: Ontological representation of knowledge artifacts
- **Task #2**: Mapping ontologically modeled knowledge artifacts to the domain knowledge ontology
- **Task#3**: Transformation of local ontologies (LKO) to the domain knowledge ontology (DKO)
- **Task#4**: Merging instantiation of LKO transformed to instantiations in DKO

- **Task#5**: Knowledge Execution

We discuss the details of these steps in section 3.3 after we discuss the necessary preliminaries in section 3.2. Modules that have been developed to perform these steps are discussed in section 3.4.

3.2. Preliminaries

In order to discuss steps and modules of our solution we need to discuss some preliminaries first. These preliminaries are OWL ontologies, reasoning process on OWL ontologies, and ontology semantics.

3.2.1. OWL Ontologies

Definition 1 (OWL Ontologies) An OWL ontology $\mathcal{O}$ defined on the vocabulary $\mathcal{V}$ = $(\mathcal{C}, \mathcal{R}, \mathcal{I}, \mathcal{L}, \mathcal{M}_p, \mathcal{M}_c)$ is composed of a set of concepts $\mathcal{C}$, a set of properties $\mathcal{R}$, a set of instances $\mathcal{I}$ and a set of literals $\mathcal{L}$, a set of meta properties $\mathcal{M}_p$ and a set of meta classes $\mathcal{M}_c$. An OWL ontology can be represented as a set of triples of the form <s,p,o> $\in \mathcal{O} \subseteq (\mathcal{C} \cup \mathcal{I} \cup \mathcal{R}) \times (\mathcal{C} \cup \mathcal{I} \cup \mathcal{R} \times \mathcal{M}_p) \times (\mathcal{C} \cup \mathcal{I} \cup \mathcal{R} \cup \mathcal{L} \times \mathcal{M}_c)$. s, p and o are called *subject*, *predicate* and *object* in an OWL triple respectively. As an example, in the triple ":Patient rdf:type owl:class" "Patient" is a member of $\mathcal{C}$, "rdf:type" is a member of $\mathcal{M}_p$ and "owl:class" is a member of $\mathcal{M}_c$. As another example, in triple ":john :hasAge 22", "john" is a member of $\mathcal{I}$, "hasAge" is a member of $\mathcal{R}$ and 22 is a member of $\mathcal{L}$.

Depending on the specie of OWL, the membership of s, o and p can be restricted to specific sets or subsets of the vocabulary. For instance, in both OWL-lite and OWL-DL there exists a strict separation of classes, properties, instances and literals.

Definition 2 (Ontology Instantiation) Instantiation of an ontology $\mathcal{O}$ based on the vocabulary $\mathcal{V}$ = $(\mathcal{C}, \mathcal{R}, \mathcal{I}, \mathcal{L})$ is the process of creating a set of new instances $\mathcal{A}$, assigning them to classes in $\mathcal{C}$ and creating relations between the new and the existing instances in the ontology using properties in $\mathcal{R}$. Therefore, an instantiation of the ontology $\mathcal{O}$ is a set of the form {<s,p,o>} $\subseteq \mathcal{O} \cup (\mathcal{A} \times (\mathcal{R} \cup \mathcal{M}_p) \times (\mathcal{I} \cup \mathcal{L} \cup \mathcal{A})) \cup ((\mathcal{I} \cup \mathcal{A}) \times (\mathcal{R} \cup \mathcal{M}_p) \times \mathcal{A}) \cup (\mathcal{A} \times \{\text{rdf:type}\} \times \mathcal{C})$

Please note that *rdf:type* $\in \mathcal{M}_p$. Instantiations can have labels in order to be distinguished from other instantiations. The instantiation with the label "j" from ontology $\mathcal{O}$ is represented as $\mathcal{O}^j$.

3.2.2. OWL Reasoning and Ontology Semantics

The following list shows a subset of tasks that OWL reasoners perform on an OWL ontology [31]:

- **Consistency checking**: Checking the ontology for contradictory facts.
- **Concept satisfiability:** Checking the concepts for possibility of having any instances. Defining an instance for an unsatisfiable class will results in inconsistency.
- **Classification:** Computing all of the direct super and sub classes of classes and all of the direct super and sub properties of properties in an OWL ontology.
- **Realization:** Finding all the classes that instances belong to. This step can only be performed when the classification step is finished, as the complete class hierarchy is needed for this task.
- **Property fillers**: Given a specific property *p* and a specific individual *s*, all the values and individuals that are related to *s* via *p* are found.

OWL has model-theoretic semantics written for its abstract syntax [32]. Reasoners will utilize the formal semantics of OWL language to perform the abovementioned reasoning tasks on an ontology and its instantiations. This process results in addition of triples that do not exist in the original ontology and its instantiation. Reasoning algorithm is represented as the function $\mathbb{R}$. The new knowledge-base after reasoning on instantiation $\mathcal{O}^i$ is represented as $\mathbb{R}(\mathcal{O}^i) \supseteq \mathcal{O}^i$. An OWL ontology formalizes the concepts and their relations in a specific domain area. Semantics of an ontology formally defines the meaning of the ontology in a computer understandable format so that a logical reasoner can utilize them to derive new facts that did not already exist. Formal semantics should be defined in such a way the reasoner derives the desired facts for each specific instantiation of the ontology. From a decision support perspective, an ontology instantiation can represent a problem and its state in a domain area and the new derived facts by the reasoner can represent the decision corresponding to the current state of the problem. Since we are using our ontologies for decision support, we discuss ontology semantics along this line of thinking. We define semantics of our ontologies in terms of OWL constructs.

Definition 3 (Ontology Semantics) Semantics of an ontology is a set of OWL triples that are used in a reasoner along with an instantiation of an ontology $\mathcal{O}^i$ that represent the current state of the domain problem in order to reason the solution. Therefore, semantics of ontology $\mathcal{O}$ represented by $S_{\mathcal{O}}$ is a triple set of the form {<s,p,o>} $\subseteq (\mathcal{C} \cup \mathcal{I} \cup \mathcal{R}) \times (\mathcal{C} \cup \mathcal{I} \cup \mathcal{R} \times \mathcal{M}_p) \times (\mathcal{C} \cup \mathcal{I} \cup \mathcal{R} \cup \mathcal{L} \times \mathcal{M}_c)$ such that $\mathbb{R}(\mathcal{O} \cup S_{\mathcal{O}})$ - $\mathbb{R}(\mathcal{O})$ ontologically represents the solution for that specific state of the problem. Formal semantics may be dependent on or independent of the instantiation representing the problem in a specific domain area. A more complex domain area needs generation of formal semantics for each instantiation in order to achieve the desired results using OWL reasoners.

3.3. *OntoMorph* Knowledge Morphing Framework Solution Steps

In this section, we discuss the steps of our solution for morphing the knowledge encapsulated in several heterogeneous ontologies.

- **Task #1: Ontological representation of knowledge artifacts**

In order to use knowledge artifacts relevant to a specific problem in a computer-based DSS, the knowledge encapsulated in several knowledge artifacts need to be modeled in a computer understandable format. We propose to use OWL based ontologies to capture the domain concepts inherent within the knowledge artifacts and their relations in terms of OWL classes, instances and properties. We call this ontology as the *Domain Knowledge Ontology* (DKO). Our solution approach is to manually instantiate DKO in order to computerized each of the knowledge artifacts utilized in the decision making process. Therefore, each knowledge artifact is an instantiation of the DKO. An instantiation of this ontology with label i is represented as $\mathcal{O}_{DKO}^{i}$. The *Knowledge Representation Module* is responsible for ontologically modelling the knowlege encapsulated in the relevant knowledge artifacts.

- **Task #2: Mapping ontologically modeled knowledge artifacts to the domain knowledge ontology**

We also recognize that there may already exist a wide range of specialized ontologies to represent knowledge artifacts in a domain area—these can be regarded as Local Knowledge Ontologies (LKO). It is our intent to reuse the available knowledge artifacts represented in LKO and hence to achieve semantic interoperability by mapping them to a common representation formalism—i.e. the DKO. In order to represent mappings between two ontologies, we developed an ontology called *Knowledge Mapping Ontology* (KMO) represented as $\mathcal{O}_{map}$ on vocabulary $(\mathcal{C}_{map}, \mathcal{R}_{map}, \mathcal{I}_{map}, \mathcal{L}_{map})$. Ontology mapping in our framework is the process of manually aligning elements from different ontologies using m-to-n relations. We use the ontology $\mathcal{O}_{map}$ to represent mappings between any two OWL ontologies regardless of their domain. To create the ith m-to-n alignment between the local ontology $\mathcal{O}_j$ and $\mathcal{O}_{DKO}$ represented by $Map_{io_j o_{DKO}}$, a new set of instances $\mathcal{A}_{\mathrm{i}}$ is created and added to $\mathcal{I}_{map}$. Then, this mapping between ontologies $\mathcal{O}_j$ and $\mathcal{O}_{DKO}$ is represented as a set of the form {<s,p,o>} $\subseteq$ $(\mathcal{A}_i \times \mathcal{R}_{map} \times (\mathcal{I}_{map} \cup \mathcal{L}_{map} \cup \mathcal{C}_j \cup \mathcal{R}_j \cup \mathcal{I}_j \cup \mathcal{C}_{DKO} \cup \mathcal{R}_{DKO} \cup \mathcal{I}_{DKO})) \cup (\mathcal{A}_i \times \{\text{rdf:type}\} \times \mathcal{C}_{map})$.

The complete mapping between $\mathcal{O}_j$ and $\mathcal{O}_{DKO}$ that is composed of k n-to-m mapping is represented as $\mathcal{O}_{map}^{\mathcal{O}_j\mathcal{O}_{DKO}} = \bigcup_{i=1}^{k} \mathcal{M}ap_{\mathrm{i}\mathcal{O}_\mathrm{j}\mathcal{O}_\mathrm{DKO}}$ is the output of the function $\mathbb{M}(\mathcal{O}_j, \mathcal{O}_{DKO})$.

- **Task #3: Transformation of LKO to DKO**

In our framework, we propose to use ontology mapping techniques in order to map and transform the knowledge within multiple LKO to the common representation of DKO. We define the semantics of the mapping language in OWL order to use it for discovering new mappings between the mapped ontologies and then use the existing and the new mappings to transfer instances between the source and the target ontologies. Therefore, discovering new mappings between $\mathcal{O}_j$ and $\mathcal{O}_{DKO}$ with respect to the mapping $\mathcal{O}_{map}^{o_j o_{DKO}}$ is the process of computing the union of $\mathcal{O}_j, \mathcal{O}_{DKO}$, and $\mathcal{O}_{map}^{o_j o_{DKO}}$ and treating them as a single ontology in a logical reasoner in order to find new relations between classes and properties of $\mathcal{O}_j$ and $\mathcal{O}_{DKO}$ that do not already exist in $\mathcal{O}_j$ or $\mathcal{O}_{DKO}$. Thus, the output of this process which is represented by $\mathbb{D}(\mathcal{O}_j, \mathcal{O}_{DKO}, \mathcal{O}_{map}^{o_j o_{DKO}})$ will be a triple set of the form {<s,p,o>} $\subseteq (\mathcal{R}_j \times \mathcal{M}_{p-property} \times \mathcal{R}_{DKO}) \cup (\mathcal{C}_j \times \mathcal{M}_{p-class} \times \mathcal{C}_{DKO}) \subseteq \mathbb{R}(\mathcal{O}_j \cup \mathcal{O}_{DKO} \cup \mathcal{O}_{map}^{o_j o_{DKO}})$. Please note that $\mathcal{M}_{p-property} \subset \mathcal{M}_p$ represents the set of possible relations between OWL properties such as owl:subPropertyOf and $\mathcal{M}_{p-class} \subset \mathcal{M}_p$ represents the set of possible relations between OWL classes such as owl:disjointWith.

The new discovered mappings and the existing mappings are used towards automatic instance transformation from the source ontology to the target ontology. Instance transformation from $\mathcal{O}_j$ to $\mathcal{O}_{DKO}$ with respect to the mapping $\mathcal{O}_{map}^{o_j o_{DKO}}$ is the process of (1) creating new instances and literals in the target ontology in order to prepare the target ontology to accommodate the source ontology instances and (2) to utilize the mapping representation language semantics to find (a) membership of instances of $\mathcal{O}_j$ in classes of $\mathcal{O}_{DKO}$; (b) relation of instances of $\mathcal{O}_j$ with instances and literals of $\mathcal{O}_j$ and $\mathcal{O}_{DKO}$ using properties of $\mathcal{O}_{DKO}$. Therefore, instance transformation in our framework from $\mathcal{O}_j$ to $\mathcal{O}_{DKO}$

with respect to the mapping $\mathcal{O}_{map}^{o_j o_{DKO}}$ is the process of computing the union of $\mathcal{O}_j, \mathcal{O}_{DKO}$, $\mathcal{O}_{map}^{o_j o_{DKO}}$ and $\mathbb{D}(\mathcal{O}_j, \mathcal{O}_{DKO}, \mathcal{O}_{map}^{o_j o_{DKO}})$ and treating them as a single ontology in a logical reasoner. Results of instance transformation which is represented by $\mathcal{O}_{DKO}^{o_j} = \mathbb{I}(\mathcal{O}_j, \mathcal{O}_{DKO}, \mathcal{O}_{map}^{o_j o_{DKO}} \cup \mathbb{D}(\mathcal{O}_j, \mathcal{O}_{DKO}, \mathcal{O}_{map}^{o_j o_{DKO}})\ )$ is a set of the form $\{<\mathrm{s,p,o}>\} \in \{<\mathrm{s,p,o}>| < s, o > \in (\mathcal{I}_j \times \mathcal{I}_{DKO} \cup \mathcal{I}_{DKO} \times \mathcal{I}_j \cup \mathcal{I}_j \times \mathcal{I}_j) \wedge \mathrm{p} \in \mathcal{R}_{DKO}\ \} \cup (\mathrm{s} \times \{\mathrm{rdf{:}type}\} \times \mathcal{C}_{DKO}) \subseteq \mathbb{R}(\mathcal{O}_j \cup \mathcal{O}_{DKO} \cup \mathcal{O}_{map}^{o_j o_{DKO}})$.

Knowledge Mapping Module is used to perform ontology mapping and instance transformation. This module is described in section 3.4.2.

- **Task#4: Merging instantiation of LKO transformed to instantiations in DKO**

If we assume *n* ontologies that are transformed to DKO are used as the knowledge sources in our knowledge morphing framework, the input of this step is $\bigcup_{j=1}^{n} \mathcal{O}_{DKO}^{o_j}$. The knowledge encapsulated in each of the instantiations of the DKO represents a specific view of the domain knowledge. Knowledge morphing, however, cannot be achieved by merely transferring all the local ontologies to a common representation format due to lack of definition of morphing constructs in the transformed LKO. From a decision support perspective, these morphing constructs representing the necessary modifications to be performed on the reasoned decisions in each of the transformed LKO to achieve a conflict-free and improved set of decisions.

Identifying and capturing the morphing constructs among local ontologies in a formal representation format enables a logical reasoner to derive more relevant and accurate results by reasoning on the ontologies and their morphing constructs as opposed to a reasoner that only reasons over LKO.

In our framework, we propose to use an OWL ontology called *Knowledge morPhing Ontology* (KPO) represented by $\mathcal{O}_{morph}$ on the vocabulary

$(\mathcal{C}_{moprh}, \mathcal{R}_{morph}, \mathcal{I}_{morph}, \mathcal{L}_{morph})$ to represent the problem-specific morphing constructs between several instantiations of DKO. Defining a morphing construct between two ontologies $\mathcal{O}_1$ and $\mathcal{O}_2$ is performed by creating new instances of $\mathcal{C}_{moprh}$ represented by $\mathcal{A}_i$ and creating relations between them and other instances in $\mathcal{O}_{morph}$, $\mathcal{O}_{DKO}^{o_1}$ and $\mathcal{O}_{DKO}^{o_2}$ using properties in $\mathcal{R}_{morph}$. Therefore, the ith morphing construct between ontologies, $\mathcal{O}_{DKO}^{o_1}$ and $\mathcal{O}_{DKO}^{o_2}$ using the morphing ontology $\mathcal{O}_{morph}$ that is called $\mathcal{M}orph_{\mathrm{io_1o_2}}$ is a set of the form <s,p,o> $\in \mathcal{A}_i \times \mathcal{R}_{morph} \times (\mathcal{I}_{morph} \cup \mathcal{L}_{morph} \cup \mathcal{I}_{\mathcal{O}_{DKO}^{o_1}} \cup \mathcal{I}_{\mathcal{O}_{DKO}^{o_2}}$. All the morphing constructs represented by the triple set $\mathcal{O}_{morph}^{o_1o_2} = \cup_{i=1}^{k} \mathcal{M}orph_{\mathrm{io_1o_2}}$ that is the output of the function $\mathbb{P}(\mathcal{O}_{DKO}^{o_1}, \mathcal{O}_{DKO}^{o_2})$. Please note that due to complexity of the morphing constructs between two ontologies, we generate them manually with the help of a domain expert.

If we assume $\mathcal{O}_1$ and $\mathcal{O}_2$ are two independent problem-solving ontologies for a decision support problem, reasoning on $\mathcal{O}_{DKO}^{o_1}$ and $\mathcal{O}_{DKO}^{o_2}$ together as one ontology or separately produces the same decisions in either scenario because there are no semantic relations between their instances. However, addition of the morphing constructs $\mathcal{O}_{morph}^{o_1o_2}$ that creates semantic relations between instances of $\mathcal{O}_{DKO}^{o_1}$ and $\mathcal{O}_{DKO}^{o_2}$ may results in obtaining new facts in a logical reasoner. These new facts are used to modify the values of the properties in $\mathcal{O}_{DKO}^{o_1}$ and $\mathcal{O}_{DKO}^{o_2}$. These new facts that are obtained by computing $\mathbb{R}\left(\mathcal{O}_{DKO}^{o_1} \cup \mathcal{O}_{DKO}^{o_2} \cup \mathcal{O}_{morph}^{o_1o_2}\right) - \mathbb{R}\,(\mathcal{O}_{DKO}^{o_1} \cup \mathcal{O}_{DKO}^{o_2}) - \mathcal{O}_{morph}^{o_1o_2}$ are of the following form: {<s,p,o> | <s,o>$\in (\mathcal{I}_1 \times (\mathcal{I}_{\mathcal{O}_{DKO}^{o_1}} \cup \mathcal{L}_{\mathcal{O}_{DKO}^{o_1}}) \cup \mathcal{I}_{\mathcal{O}_{DKO}^{o_2}} \times (\mathcal{I}_{\mathcal{O}_{DKO}^{o_2}} \cup \mathcal{L}_{\mathcal{O}_{DKO}^{o_2}})) \wedge \exists \mathrm{p}^{\prime}, \mathrm{URI(p)} = \mathrm{URI}(\mathrm{p}^{\prime}) + \text{"New"}$ }. Please note that URI function returns the unified resource identifier (URI) of its input and "+" represents the string concatenation function. Value of each property p$^{\prime}$ is replaced by the value of the property p, if $\mathrm{URI}(\mathrm{p}^{\prime}) = \mathrm{URI(p)} + \text{"New"}$. In this way, the solutions reasoned in $\mathcal{O}_{DKO}^{o_1}$ and $\mathcal{O}_{DKO}^{o_2}$ are modified according to the morphing construct in order to merge $\mathcal{O}_{DKO}^{o_1}$ and $\mathcal{O}_{DKO}^{o_2}$. The result of morphing for ontologies $\mathcal{O}_{DKO}^{o_1}, \mathcal{O}_{DKO}^{o_2} \ldots \mathcal{O}_{DKO}^{o_n}$, n>2 that is represented by $\mathbb{P}(\mathcal{O}_{DKO}^{o_1}, \mathcal{O}_{DKO}^{o_2}, \ldots .\ \mathcal{O}_{DKO}^{o_n})$ is equal to $\cup_{i=1}^{n} \mathbb{P}(\mathcal{O}_{DKO}^{o_i}, \cup_{j=1}^{n} \mathcal{O}_{DKO}^{o_j}))$.

Please note that this reasoning and modification is not a onetime task in the decision support context and needs to be executed repeatedly for each state of the problem until an equilibrium is reached. Due to the subtleties of each domain area, a problem-specific KMO should be developed for morphing the knowledge of LKO. Moreover, for each set of input LKO, the domain expert who is familiar with the domain knowledge and its morphing constraints creates an instantiation of this ontology manually. Using OWL as the language for representing this ontology enables us to perform reasoning on the transformed local ontologies and the instantiation of the KMO using a single OWL reasoner. *Ontology Merging Module* that is responsible for this step of our framework is described in section 3.4.3.

- **Task#5: Ontology Execution**

If we assume the source ontologies of our knowledge morphing framework are *n* ontologies called $\mathcal{O}_1, \mathcal{O}_2 \dots \mathcal{O}_n$, the input of this step is $\mathbb{P}(\mathcal{O}_{DKO}^{o_1}, \mathcal{O}_{DKO}^{o_2}, \dots.\ \mathcal{O}_{DKO}^{o_n})$. Please note that $\mathcal{O}_{DKO}^{o_j} = \mathbb{I}(\mathcal{O}_j, \mathcal{O}_{DKO}, \mathbb{D}(\mathcal{O}_j, \mathcal{O}_{DKO}, \mathbb{M}(\mathcal{O}_j, \mathcal{O}_{DKO})) \cup \mathbb{M}(\mathcal{O}_j, \mathcal{O}_{DKO}))$ represents the instances that are transformed from $\mathcal{O}_j$ to $\mathcal{O}_{DKO}$. The ultimate purpose of knowledge morphing is to use the morphed knowledge in a DSS to deliver decision support based on several heterogeneous knowledge sources that complete each other. The assumption is that using multiple knowledge sources as opposed to only one leads to making more accurate and relevant problem-specific decisions. We use the algorithm in Table 3.1 in our DSS to provide decision support based on several morphed ontologies transformed to DKO.

Table 3.1 Knowledge Execution Algorithm

1. Acquire the current state of the problem. This can be done through querying a data-base, interacting with the user, etc.

2. If the desired state of the decision making problem has reached, execution is finished otherwise go to next.

3. Encode the current state of the problem in DKO. This happens by modification of values of certain properties in that ontology.

4. Perform reasoning on $(\bigcup_{n}^{i=1} \mathcal{O}_{DKO}^{o_i}) \cup \mathbb{P}(\mathcal{O}_{DKO}^{o_1}, \mathcal{O}_{DKO}^{o_2}, \ldots . \mathcal{O}_{DKO}^{o_n})$.

5. If ∃ <s, p´, o´> ∈ $\mathbb{R}((\bigcup_{i=1}^{n} \mathcal{O}_{DKO}^{o_i}) \cup \mathbb{P}(\mathcal{O}_{DKO}^{o_1}, \mathcal{O}_{DKO}^{o_2}, \ldots . \mathcal{O}_{DKO}^{o_n}))$ ∧ ∃ <s,p ,o > ∈ $\mathbb{R}((\bigcup_{i=1}^{n} \mathcal{O}_{DKO}^{o_i}) \cup \mathbb{P}(\mathcal{O}_{DKO}^{o_1}, \mathcal{O}_{DKO}^{o_2}, \ldots . \mathcal{O}_{DKO}^{o_n}))$ ∧ $\mathrm{URI(p)} = \mathrm{URI}(\mathrm{p}') + \text{"New"}$, replace <s,p,o> with <s,p, o´>, delete <s, p´, o´> from the ontology and go to previous. Otherwise go to next.

6. Query the values of properties that represent the decision made for the current state of the problem and represent them to the user through a user interface.

7. Wait for the user to act upon the made decisions.

8. Go to 1.

In our framework, knowledge morphing is dynamically guided by the morphing constructs represented in KMO during running the execution algorithm as opposed to altering the transformed local ontologies manually to reflect the morphing before knowledge execution. We argue dynamic knowledge morphing during running the execution algorithm according to the morphing constructs is superior to knowledge morphing before running the execution algorithm for two reasons:

(1) The elements that define the state of a problem can create a huge problem space that may not be easily accounted for during the pre-execution morphing process. For instance, in case of merging several medical workflow algorithms, problem state may contain several variables from an Electronic Medical Records (EMR) and the Hospital Information Systems (HIS). Morphing the knowledge in these clinical workflow ontologies should be concordant to every possible state that a patient might have during his treatment.

(2) In order to deal with the huge problem state, assumptions about the problem state can be made to reduce it to a smaller and more manageable one. However, many of the assumptions may not hold true during the execution and cause the generation of decisions that does not apply to the current problem at hand.

Knowledge Execution Module is responsible for execution of the abovementioned steps. This module is described in section 3.4.4.

3.4. Solution Modules

The following modules are needed in order to perform the abovementioned tasks: 1. Knowledge Representation, 2. Knowledge Mapping, 3. Knowledge Morphing and 4. Knowledge Morphing Execution. Figure 3.1 shows the system architecture of our semantic web based knowledge morphing framework.

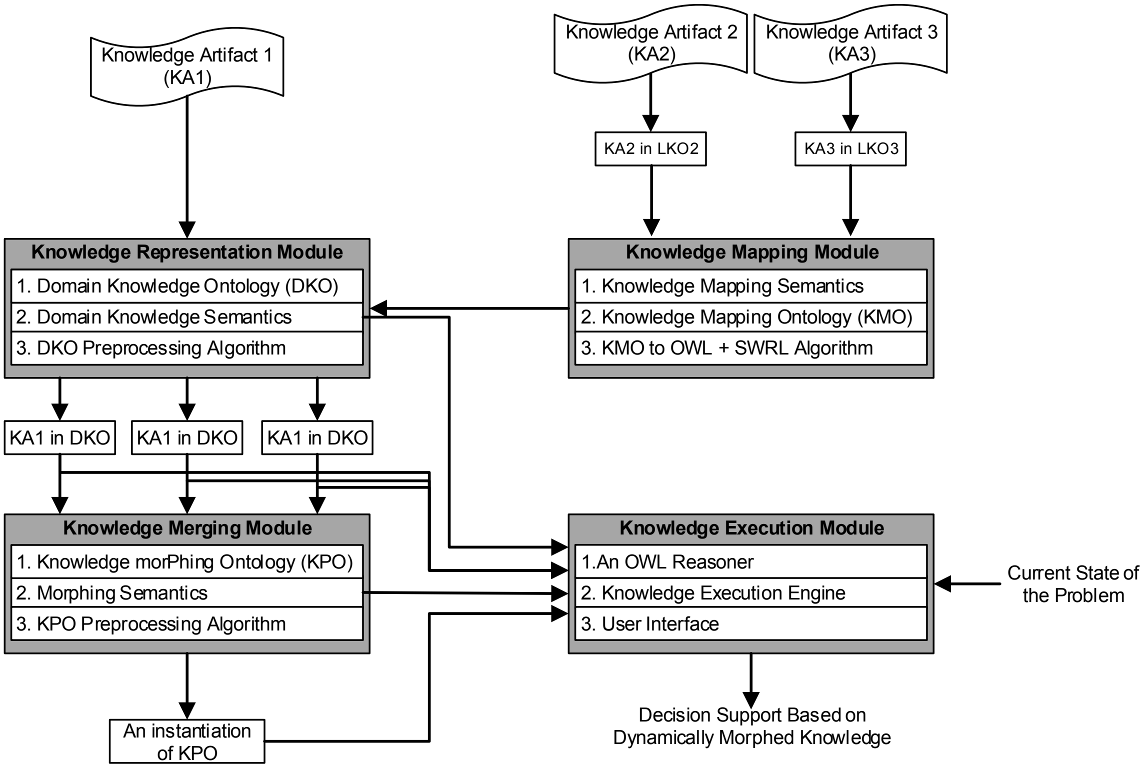

Knowledge Artifact 1 (KA1)
Knowledge Artifact 2 (KA2)
Knowledge Artifact 3 (KA3)
KA2 in LKO2
KA3 in LKO3
Knowledge Representation Module
1. Domain Knowledge Ontology (DKO)
2. Domain Knowledge Semantics
3. DKO Preprocessing Algorithm
Knowledge Mapping Module
1. Knowledge Mapping Semantics
2. Knowledge Mapping Ontology (KMO)
3. KMO to OWL + SWRL Algorithm
KA1 in DKO
KA1 in DKO
KA1 in DKO
Knowledge Merging Module
1. Knowledge morPhing Ontology (KPO)
2. Morphing Semantics
3. KPO Preprocessing Algorithm
Knowledge Execution Module
1.An OWL Reasoner
2. Knowledge Execution Engine
3. User Interface
Current State of the Problem
An instantiation of KPO
Decision Support Based on Dynamically Morphed Knowledge

Figure 3.1 The systems architecture of *OntoMorph*

3.4.1. Knowledge Representation Module

In order to use the knowledge encapsulated in knowledge artifacts for computerized decision support, this knowledge should be captured in a computer understandable format. The inputs of this module are knowledge artifacts and the output is several instantiations of the DKO ontology each modelling a knowledge artifact. Knowledge Representation Module that is responsible for this task has the following sub-components:

a. Domain Knowledge Ontology (DKO)

DKO is an OWL ontology that models concepts, instances and their relations. This OWL ontology should be expressive enough to capture all the knowledge pertaining to the decision making process. For instance, if the knowledge sources of a CDSS are paper-based disease-specific CPG, DKO captures the all the medical and procedural aspects of these knowledge artifacts that participate in the clinical decision making. Thus, it is a problem-specific ontology. If a knowledge source is mapped to this ontology, instantiations will happen automatically during the instance transformation process. Otherwise, it is manually instantiated by the help of a domain expert in order to represent a paper-based knowledge source.

b. Domain Knowledge Semantics

The knowledge modeled in DKO is meant to be used towards decision making. A solution is to use an OWL reasoner to perform reasoning on the instantiation of the DKO that represents the participating knowledge artifacts in order to derive the decisions. To do so, meaning of the DKO should be described in a formal language that can be understood by OWL reasoners. Formal semantics are meaning of the DKO constructs in a computer understandable format. For instance, imagine a property called *hasNextTask* in DKO ontology that represents the sequence of actions in a workflow ontology. If the instantiation of DKO expresses task2 should be executed when task1 is completed using OWL triple ":task1 :hasNext :task2" and task1 is completed, reasoning on DKO should derive the fact that task2 is the active task at this moment. The formal semantics of DKO are expressed using OWL triples and SWRL rules in our framework.

c. DKO Pre-processing algorithm

Each of the instantiations of DKO undergoes a pre-processing phase that deals with the non-unique naming and the open world assumptions and addresses the lack of qualified cardinality restriction, property chaining and data type expressivity in OWL in order to prepare the knowledge artifact for execution in the knowledge execution module. *DKO pre-processing algorithm* is responsible for this process.

3.4.2. Knowledge Mapping Module

As we discussed earlier, knowledge artifacts represented in different heterogonous ontologies are mapped and then have their instances transformed to a common representation language in our knowledge morphing framework for two reasons: (1) Morphing the knowledge in different ontologies is not possible as it is difficult and sometimes impossible to find corresponding concepts and relations in heterogeneous ontologies and (2) *execution algorithm* utilized for the decision making in the morphing execution engine is not able to process several heterogeneous ontologies. The knowledge-mapping module is responsible for mapping local heterogeneous ontologies to DKO and then transforming their instances to instances of DKO. This module is composed of two sub-components:

a. Knowledge Mapping Ontology (KMO)

Knowledge Mapping Ontology (KMO) is developed for ontology mapping representation. This OWL-Full ontology can represent a range of simple mappings to complex mappings that may contain complex mathematical computation and structure modification. This ontology is in OWL-Full as it does not respect the separation of classes, properties and instances in OWL-DL. An instantiation of this ontology represents how classes, instances and properties of the two ontologies are related to each other and how the source ontology instances are transformed to instances of the target ontology. Since mappings can be quite complex, we generate them manually. Please note that this ontology is not problem-specific and can be used to map any two OWL ontologies.

We have also developed an ontology called *Expression Ontology* that can be instantiated to model and calculate the expressions needed for decision support. For instance, the formula how to calculate Body Mass Index (BMI) based on the weight and height of a person can be modelled in this ontology. Moreover, this ontology is equipped with OWL triples and SWRL rules that enable computing the values of the expression in this language using OWL reasoners.

b. Knowledge Mapping Semantics

Semantics of the KMO enables a reasoner to perform reasoning on the mapped ontologies and the reasoning in order to perform instances transformation and discovering new mappings. However, since KMO is in OWL-Full, it is undecidable and cannot be used for these two tasks in a logical reasoner. Therefore, instead of representing the mappings in OWL-DL + SWRL, we represent the mappings OWL-Full and then translate them to OWL-DL + SWRL using the *KMO to OWL + SWRL Algorithm* for the following reasons: (a) The expressivity of KMO being OWL-Full—i.e. using properties and classes as instances—makes the ontology mappings more readable and less verbose—i.e. with fewer triples compared to OWL-DL; (b) It enables us to support conditional mappings and complex condition satisfaction criteria, meta modelling, Boolean operators and converting ontology elements and creating new ones which are not directly supported by either OWL or SWRL. These aspects of ontology mapping are supported by automatic generation of several OWL triples and SWRL rules that simulate the lacking feature during the translation process; (c) SWRL rules are difficult to write and can easily become undecidable if not written correctly. In our translation algorithm, DL-Safe SWRL rules are generated automatically thus relieving the user about decidability concerns.

3.4.3. Knowledge Merging Module

Ontology instantiations transformed to DKO are ready to be morphed. Morphing in our framework is achieved by expressing the dependencies between the knowledge encapsulated in several instantiations of DKO. This module is composed of two sub-components:

a. Knowledge morPhing Ontology (KPO)

KPO is a domain-specific ontology for knowledge morphing that can express the dependencies between the knowledge encapsulated in the knowledge artifacts of a specific domain are. These dependencies are often ignored in individual knowledge artifacts and the domain expert should help to define the morphing constructs. This ontology is composed of two parts. These first is used to define the morphing constructs between several LKO. We call this part of the ontology the *independent part*. This part of the ontology is independent of DKO in the sense that it does not share any classes, instances or properties with it. The second part of this ontology that is shared between DKO and KPO represent the key elements of the domain knowledge. We call this part the *shared part*. Therefore, an instantiation of the KPO uses the *independent part* to define morphing constructs between concepts represented in the *shared part*. As an example, imagine a CDSS that uses two ontologies as its knowledge sources each describing a drug family. These two ontologies may not contain the knowledge regarding the possible adverse interactions between them in infant patients. Any decisions made regarding use of these drugs in an infant will be affected by the morphing constructs that are not encapsulated in any of the drug ontologies. In this example, the independent part of the KPO contains morphing constructs that can represent the adverse interactions and their conditions. The shared part of KPO in this example contains classes representing drugs in local ontologies.

The source of knowledge encapsulated in an instantiation of this ontology can be domain experts' personal experience participating in the ontology instantiation or their interpretation of published knowledge sources that contain knowledge regarding the how to merge LKO participating in the merge.

b. Knowledge Morphing Semantics

Morphing constructs in an instantiation of the KPO are used towards modification of the results of reasoning on the local ontologies. Morphing semantics enables a reasoner to automatically infer these modifications for each state of the problem in a computerized DSS. For instance, reasoning on two local ontologies that have not been morphed may result in administration of two adversely interacting drugs. However, if we perform reasoning on these local ontologies and an instantiation of the KPO, the reasoning result will provide us with necessary modifications to avoid adverse interactions. Since local ontologies and KPO are all in OWL, we define the semantics of the morphing ontology in

OWL and SWRL. This choice of representation enables us to perform reasoning on the local ontologies and the instantiation of DKO simultaneously.

c. KPO Pre-processing algorithm

Each of the instantiations of KPO undergoes a pre-processing phase that deals with the non-unique naming and the open world assumptions and addresses the lack of qualified cardinality restriction, property chaining and data type expressivity in order to be prepared for execution in the *knowledge execution module*. This module is described next.

3.4.4. Knowledge Execution Module

After transforming the local ontologies to a common representation format and then morphing the knowledge encapsulated in them using an instantiation of the KPO, they can be used for delivering decision support. The *Knowledge Execution Module* is responsible for interacting with the user and reasoning on the transformed ontologies in DKO and an instantiation of KPO. As it is shown in Figure 3.1, reasoning on the transformed ontologies and the knowledge morphing constructs is performed in regard with their formal semantics in OWL and SWRL. This module is composed of the following sub-components:

a. An OWL reasoner

Knowledge Execution Module makes use of an OWL reasoner that performs reasoning on instantiations of the DKO and KMO in order to derive the best decision according to the current state of the problem. We use Pellet reasoner [161] as it supports both OWL and SWRL in an integrated environment.

b. A Knowledge Execution Engine

A *Knowledge Morphing Execution Engine* is a program that runs the execution algorithm in order to deliver decision support based on the ontologies encoded in DKO and an instantiation of KMO. Our execution algorithm has been previously shown in Table 3.1. This engine is also capable of execution of one or more instantiations of DKO independently if an instantiation of the KMO is not provided. We refer to this engine as the *Knowledge Execution Engine* if no morphing is performed during execution of one or concurrent execution of several instantiations of the DKO.

c. A User Interface

This user interface has the following responsibilities: (1) Showing the decision reasoned by the execution algorithm and (2) Accepting inputs from user that indicates the problem state and his preferences and reflecting them in DKO.

3.4.5. Functional Description of our Knowledge Morphing Framework

In this section, we briefly discuss the tasks that are performed in each of the modules in order to prepare several paper-based and ontologically modelled knowledge artifacts as knowledge sources in our knowledge morphing framework. Figure 3.2 displays a functional view of our framework.

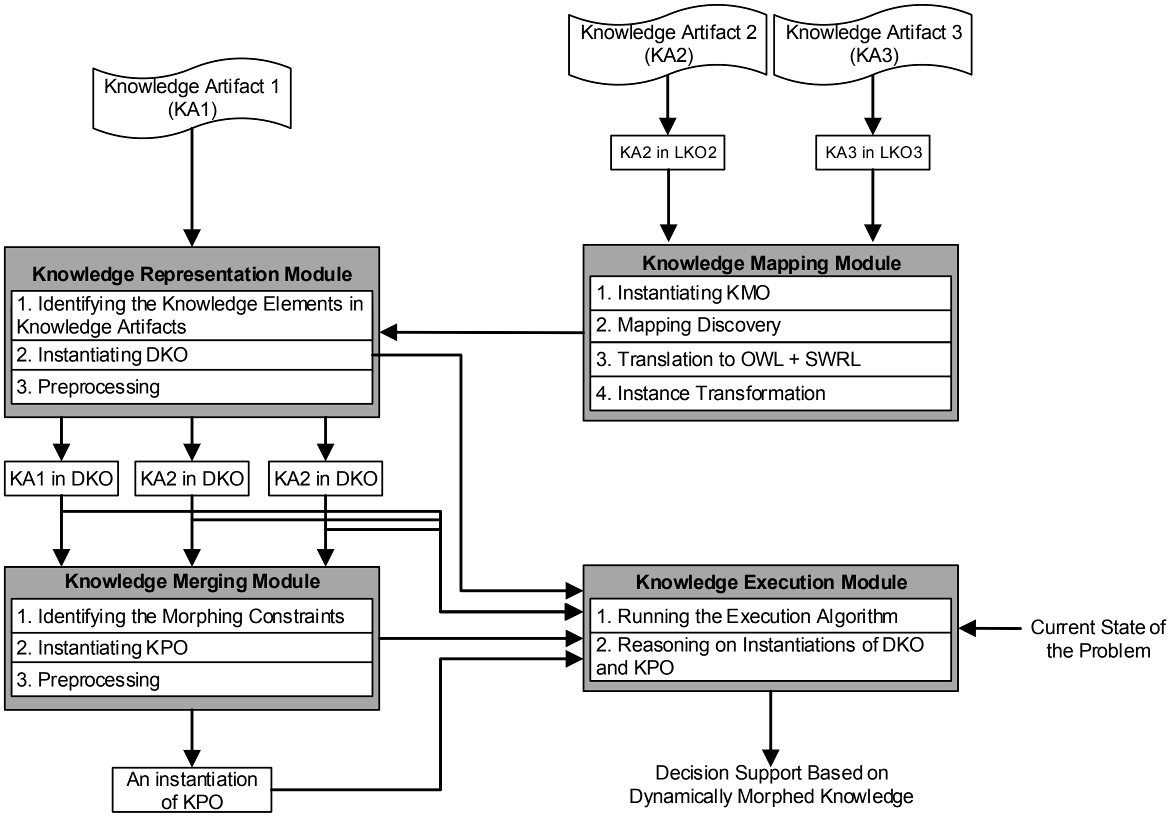

Knowledge Artifact 1
(KA1)
Knowledge Artifact 2
(KA2)
Knowledge Artifact 3
(KA3)
KA2 in LKO2
KA3 in LKO3
Knowledge Representation Module
1. Identifying the Knowledge Elements in Knowledge Artifacts
2. Instantiating DKO
3. Preprocessing
Knowledge Mapping Module
1. Instantiating KMO
2. Mapping Discovery
3. Translation to OWL + SWRL
4. Instance Transformation
KA1 in DKO
KA2 in DKO
KA2 in DKO
Knowledge Merging Module
1. Identifying the Morphing Constraints
2. Instantiating KPO
3. Preprocessing
Knowledge Execution Module
1. Running the Execution Algorithm
2. Reasoning on Instantiations of DKO and KPO
Current State of the Problem
An instantiation of KPO
Decision Support Based on Dynamically Morphed Knowledge

Figure 3.2 Tasks accomplished in the modules of *OntoMorph*

a. Tasks Performed in the Knowledge Representation Module

In the first step, the main concepts of the domain knowledge and the procedural aspect represented in the knowledge artifact are identified with the help of a domain expert. In the next step, the domain knowledge and the procedural aspects covered in the knowledge artifact are represented by an instantiation of the DKO. Therefore, each knowledge artifact will be an instantiation of the DKO in our framework. The last step is executing the pre-processing algorithm on instantiations of DKO.

b. Tasks Performed in the Knowledge Mapping Module

In order to unify the representation of the heterogeneous ontologies representing different knowledge artifacts, we first need to identify the relations between classes, properties and instances of those ontologies and DKO. These relations can be represented as an instantiation of KMO. Therefore, an instantiation of KMO is manually created for each knowledge artifact that is not represented in DKO. Instantiation of the KMO are translated to a combination of OWL axioms and SWRL rules using a translation algorithm. Reasoning on the translated mappings and the mapped ontologies will result in discovering new mapping and subsequently transforming the instances of the LKO to DKO.

c. Tasks Performed in the Knowledge Merging Module

In the first step, domain expert identifies the morphing constructs between the knowledge artifacts represented in DKO. Then, KPO is instantiated to represent those morphing constructs in terms of merging constraints. The last step is to execute the KPO pre-processing algorithm on instantiations of KPO in order to prepare it for execution.

d. Tasks Performed in the Knowledge Execution Module

The only tasks that are needed to be performed in this module are to enter the current state of the problem to the engine using the provided user interface and act upon the provided decisions. These two tasks are repeatedly performed until the desired problem state is reached.

3.5. Knowledge Morphing Framework Application

We use our knowledge morphing framework to address the problem of CPG Merging to deliver clinical decision support to comorbid patients. We call this domain-specific knowledge morphing framework the *CPG Merging Framework*. In the rest of this section, we briefly describe how each of the modules of the knowledge morphing framework has been customized in order to handle the task of the CPG merging.

CPG modeled in heterogeneous ontologies are the knowledge sources of our CPG merging framework that come in different representation ontologies. CPG represented in different ontologies are all needed to be mapped to a common representation ontology. We developed an ontology called *CPG Domain Knowledge Ontology* (CPG-DKO). We map each of the ontologically modeled CPG to CPG-DKO using an instantiation of the Knowledge Mapping Ontology (KMO) and then transform all their instances to instances of the CPG-DKO.

We first developed an engine called *CPG Execution Engine* that is able to execute a CPG represented in CPG-DKO in order to deliver decision support to patients with only one medical condition. Using this engine, we can concurrently execute several CPG independently. However, this engine is not capable of avoiding duplications and conflicts; therefore, it cannot be used for CPG Merging.

The knowledge encapsulate in several CPG are ready to be morphed after unifying the representation in CPG-DKO. We developed a problem-specific Knowledge Morphing Ontology called *CPG Knowledge morPhing Ontology* (CPG-KPO) with the help of medical domain experts and reviewing the related literature. An instantiation of this ontology expresses the morphing constructs between two CPG modeled in CPG-KPO. For example, an instantiation of this ontology may describe the condition under which two medical actions may be conflicting and their simultaneous application on the patient may be harmful. The source of knowledge in a manually created instantiation of this ontology is physicians personal experience participating in ontology instantiation and their interpretation of published knowledge sources regarding

In order to dynamically merge several CPG during their execution, we modified our CPG execution algorithm so that it can also take into consideration the semantics of the merging constraints. This engine that is called *CPG Merging Execution Engine* uses semantics of the merging constraints to modify the medical actions recommended by individual CPG and therefore delivering a conflict-free and improved decision support for comorbid patients. For instance, if two medical actions are conflicting and one of them is already performed on the patient in one of the CPG, the other task will be discarded if recommended by any other

CPG. Table 3.2 shows the modules of our CPG Merging Framework and their counterpart modules in our Knowledge Morphing Framework.

Table 3.2 The modules of our CPG Merging Framework, their purposes and their counterpart modules in *OntoMorph*

Knowledge Morphing Framework Modules	**CPG Merging Framework Modules**	**Purpose in CPG merging framework**
Domain Knowledge Ontology (DKO)	CPG Domain Knowledge Ontology (CPG-DKO)	Unifying the representation of several ontologically modeled CPG
Knowledge Morphing Ontology (KMO)	CPG Knowledge MorPhing Ontology (CPG-KPO)	Capturing the comorbidity related morphing constructs between several single-disease CPG
Knowledge Execution Engine	CPG Execution Engine	Delivering decision support for patients with one medical condition based on one CPG
Knowledge Morphing Execution Engine	CPG Merging Execution Engine	Delivering Decision Support for comorbid patients based on several merged CPG

We successfully used our CPG merging framework to merge several comorbidity related CPG in order to deliver decision support for the following comorbidities:

- Chronic Heart Failure - Atrial Fibrillation
- Transient Ischemic Attack - Duodenal Ulcer
- Osteoarthritis **-** Diabetes
- Osteoarthritis – Hypertension

- Hypertension – Diabetes
- Osteoarthritis – Hypertension - Diabetes

CHAPTER 4: CPG DOMAIN KNOWLEDGE ONTOLOGY (CPG-DKO)

4.1. Introduction

Knowledge artifacts available for decision making in a specific problem may come in different formats. These knowledge artifacts are represented in semantically heterogeneous ontologies with different classes, properties and instances. These ontologies cover different aspects of the domain knowledge and barely overlap. For instance, ontologies designed to capture the knowledge in a CPG and a CP can be very different in terms of coverage of concepts and structure. Moreover, ontologies developed to describe the same type of knowledge artifacts for a CDSS can be heterogeneous as well. Open nature of semantic web has led to development of several heterogeneous ontologies describing the same type of knowledge artifacts, such as CPG [20][30][104][105][106][107][108][133][134][135][136][137][138][140][145][155]. Ontologies that describe the same type of knowledge artifacts in a domain of interest (LKO) can differ in their coverage of the domain knowledge, points of views and levels of granularity. Thus, ontologies of a domain area can be semantically heterogeneous regardless of the fact they are representing knowledge about the same domain.

Semantic heterogeneity of ontologies describing the knowledge artifacts is a great hindrance to knowledge morphing and execution. It is not always a feasible option to find the related concepts and properties in order to merge several heterogeneous LKO. Moreover, reasoning on these ontologies in order to deliver decision support is not straightforward as it requires combining the results of reasoning on these ontologies into a unified decision.

A solution to address semantic heterogeneity, as it pertains to knowledge representation using ontologies, is to transform the different source ontologies to a common representation ontology. Note that due to semantic heterogeneity of these ontologies, none of them can be chosen as this common representation ontology. Therefore, an alternative option is to map and transform all these ontologies to a domain ontology that can act as the *background* ontology [146][147]. The background ontology that is a *comprehensive DKO* is expected to have the highest coverage of the domain and the most detailed granularity compared to the source ontologies describing the domain. The DKO can be used as an Interlingua between ontologies of the domain thus providing a mechanism to interrelate semantically heterogeneous ontologies [109][110][111][112][114]. We have developed a DKO for CPG called CPG-DKO that can be used as the common representation of CPG represented in heterogeneous ontologies. All the ontologically modelled CPG are mapped to CPG-DKO and their instances are transformed to it. Merging and execution of these CPG happen in this ontology. In section 4.2, we discuss how this ontology has been developed so that it can be used as a comprehensive DKO for CPG. We use terms comprehensive DKO and background ontology interchangeably in this thesis.

Mapping different ontologies to a DKO has the following benefits: **(a)** problem of semantic heterogeneity between ontologies is alleviated; **(b)** the internal structure of the background and the mapped ontologies along with the mappings can be used to derive new mappings between the mapped ontologies and **(c)** reasoning can be performed in case of inconsistency between the mapped ontologies as the reasoning is performed on the DKO and each of the mapped ontologies separately rather than on all of them at the same time.

4.2. CPG Domain Knowledge Ontology Engineering

In order to develop a comprehensive DKO for CPG we needed an ontology development methodology. Many of the ontology development methodologies are designed to organize the joint effort of a group of developers involved in development of large-scale ontologies [118][119][120][121][122] that have several resemblances to software development methodologies [123][124] that are utilized for large software engineering projects and do not match our needs for development of a small-scale ontology. We adopted the methodology introduced in [125] due to its simplicity and proven capability in development of small-scale ontologies [126][127][128][129]. This methodology is composed of the steps shown in Figure 4.1. These steps should be performed in a cyclic fashion until the ontology with the desired features has been developed.

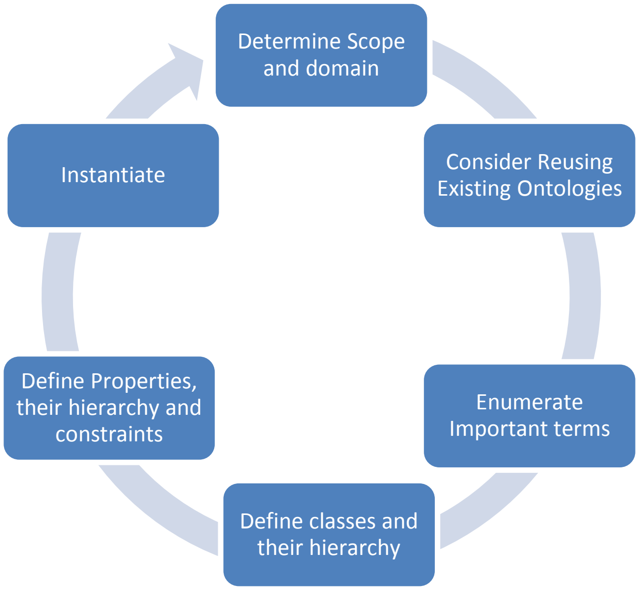

Determine Scope and domain
Consider Reusing Existing Ontologies
Enumerate Important terms
Define classes and their hierarchy
Define Properties, their hierarchy and constraints
Instantiate

Figure 4.1 Steps of the ontology development methodology proposed in [125]

In the rest of this section we discuss how each of the steps in Figure 4.1 are performed in order to develop a DKO for CPG.

Step 1. Determine the domain, scope and purpose of the ontology

In the first step of this methodology, the ontology developer should determine the scope and the intended use of the ontology. Our ontology will be used as a common representation language for CPG merged in our CPG merging framework. We focus on the workflow structure of CPG rather than their medical knowledge for two reasons: (1) Our area of expertise is computer science and (2) the elements of the medical knowledge in CPG act as input values to the workflow structure and affects its execution. Therefore, if we model the decision variables in the workflow structure, the necessary mappings between the medical knowledge elements and the workflow structure can be created. For instance, if we model comparison of numeric variables for decision making in our ontology, we can simply create the necessary mappings between these variables and the corresponding values in the medical knowledge to decide if a patient suffers from hypertension.

As we discussed previously, a comprehensive DKO is needed to cope with semantic heterogeneity. Differences in points of view, coverage and granularity levels are three limiting factors of semantic heterogeneity. Our assumption is that all CPG are developed from the point of view of the physician responsible for managing the medial condition in the patient; therefore, the fact that different ontologies might have different points of view does not concern us and is kept out of the equation. Thus, our CPG Comprehensive DKO should be able to alleviate the lack of enough overlap and different levels of granularity between CPG ontologies.

To deal with lack of enough overlap, we focused on making our ontology as comprehensive as possible in terms of covering the workflow patterns used in CPG ontologies. Therefore, the goal is to have a corresponding concept in the Comprehensive DKO for any existing workflow pattern in any of the CPG representation languages. Some examples of these workflow patterns are decisions, cycles and sequential constraints.

Difference in granularity is another hindrance to unifying the representation of the CPG ontologies. This is different from coverage in the sense that ontologies with different granularity can have the same coverage but describe the knowledge with different levels of detail. As an example, two ontologies may cover the concept of cyclic tasks but one of them may be able to express cycles in more details. For instance, the more detailed ontology can be able to define several exit points for a while loop whereas the other ontology can only support traditional while loops with only one exit point. To resolve this issue, not only we tried to include every workflow pattern in our ontology, but also all its variations. For instance, in case of decisions, we added all the variations of decisions such as if-then-else, switch, multiple-decision-option and argumentation rules to our ontology. We also expanded augmentation rules into three subclasses in order to be able to capture more than one variant of the argumentation rules. Moreover, we tried to avoid general concepts that are specialized using values of properties. Instead, we focused on creating a more detailed class hierarchy. We believe that this feature makes it easier to differentiate between different levels of details and make the necessary mappings between CPG ontologies and our Comprehensive DKO.

Step 2. Considering re-using existing ontologies

In this step, existing ontologies are checked for reusing possibilities. If an existing ontology serves our purpose there is no need to develop a new one. If no ontology exists that matches our needs reusing some parts of them might be an option. We reviewed the existing CPG representation languages [34][108][133][134][135][136][138][137][140][155] and their implementations specially the ones developed in the NICHE research group [20][30][104][105][106][107][145] due to their implementations in OWL. The only attempt that is made to create a Comprehensive DKO for CPG is reported in [114]. Wang et al. [114] has developed a comprehensive DKO called the "generalized guideline" for CPG representation languages. In order to develop their comprehensive DKO they performed an extensive survey [113] to find the common workflow patterns existing in different CPG representation languages. However, their approach has the following limitations: (1) A limited list of workflow structures as compared to the list of workflow elements listed in [101] are implemented; (2) They use of frames as opposed to OWL, where frames are less expressive than OWL ontologies; (3) Limited ability to share the mapping with OWL based ontologies [115]; (4) frames are now obsolete and most recent CPG representation languages do not use frames.

NICHE ontologies are in OWL and provide us with a wide range of workflow patterns that can be reused. However, since they are not as comprehensive as we need, we create our own ontology and reuse some parts of these NICHE ontologies. We did not come across any other comprehensive DKO for CPG implemented in OWL. As we describe our ontology, we discuss which parts have been reused from existing CPG ontologies.

Step 3: Enumerate important terms in the ontology

In this step of the ontology development methodology, a list of terms and concept that the developed ontology is about should be created. For instance, for a hypertension CPG ontology, the terms Diabetes_Types, Blood_Pressure and Therapy are among the terms that would be in this list. In this step, we focused on finding the workflow patterns that is reported in existing CPG representation languages [20][30][104][105][106][107][108][133][134][135][136] [137][138][140][145][155] and the surveys that review these languages in terms of their workflow representation capabilities [88][89][101][103][113]. We created a list of workflow patterns that have been used at least in one of the major CPG representation languages. The identified workflow patterns that we would like our ontology to capture are discussed in section 4.3.

Step 4 and 5: Implementation

As this methodology points out, implementing classes, properties and their hierarchies are intertwined tasks and cannot be performed in a sequential manner. In this step, we went through the workflow patterns identified in the last step and checked their existence in NICHE ontologies. If the workflow patterns exists in those ontologies, we simply reused the associated classes, properties and instances in our ontology. If reusing was not an option due to lack of covering the concept, new classes and properties were created. Upon creating classes and properties, their position in the hierarchy were determined as well. For instance, after creating the class *WhileLoop*, we assigned it as a subclass of the class *Cycle*. The developed ontology is in OWL-DL as it does not use the OWL-Full abilities such as treating ontology classes as instances or properties at the same time. This ontology is described in section 4.4.

Step 6: Instantiation

Instantiation of the ontology is the last step of the methodology. In this step, classes are instantiated and their associated properties are utilized to create relations between instances.

Each instantiation of our ontology is manually created and represents the workflow structure of a disease-specific CPG. We instantiated our ontology in order to represent the workflow structure of several real and imaginary CPG.

Cycling through steps of the ontology development methodology

The methodology utilized suggests that steps 1 to 5 should be repeatedly performed until a desired ontology has been developed. However, this methodology fails to describe how to use the instantiation results in order to improve the ontology when we go back to the first step in the development cycle. We adopted our own approach to fill this gap. We instantiated our ontology in order to represent the workflow structure of several real and imaginary CPG. If we came across a workflow pattern that could not be represented in our ontology and could not be reused from existing OWL ontologies, we would update the list created in step 3. The new workflow patterns in the list were added to the ontology during the next phase of the implementation. If a workflow pattern already existed in our ontology but was not properly represented, the necessary modifications were done during the next implementation phase. We repeated the steps of the ontology development twice. We did not perform any evaluations in terms of efficacy in a CPG merging framework at this stage of the development.

4.3. Common Workflow Patterns in CPG Representation Languages

To develop a comprehensive DKO that alleviates the lack of overlap and difference in granularity, we need to identify the workflow patterns and all their variants that have been utilized in existing CPG representation languages. In a seminal work by Mulyar et al. [101], Asbru, GLIF, EON and PRO*forma* CPG representation languages are reviewed in order to identify the presence of 43 classic workflow patterns. We follow their categorizations of workflow patterns and use their results in the specification of a detailed list of CPG workflow elements that are subsequently used to develop CPG-DKO. Note that CPG workflow patterns in [101] are categorized under 7 classes which have been previously identified by Russel et al. [102]. Table 4.1 shows the categorization of workflow patterns discussed in [102] and their description.

Table 4.1 Workflow pattern categories discussed in [101] and their description

Pattern Category	Pattern Description
Basic Control-flow	These patterns represent the basic elements of workflows such as sequential constraints, nesting, branches, synchronizations and conditions.
Advanced branching and synchronization	Patterns describing advanced branching and synchronization points.
Structural patterns	This category includes implicit termination and multiple entry and exit point cycles.
Multiple instances patterns	This category of patterns deal with concurrent execution of several instances of a specific type
State-based patterns	These patterns are able to describe execution scenarios based on executional states of tasks in the workflow.
Cancellation patterns	These patterns characterize the scenarios in which one or several tasks should be cancelled if certain criteria are met.
New patterns	Patterns that authors of [101] believe are not adequately captured in existing workflow representation languages

Albeit authors of [101] have provided a thorough comparison of CPG modelling languages from a workflow pattern perspective, they have not elaborated on different decision models that exist in these languages. These decision models that play an important role in controlling the flow of the executions of CPG are surveyed by Peleg et al. [88][103]. Based on our review, we identified the following workflow patterns that are present in CPG representation languages. We briefly explain the function of each of the identified workflow element.

- ***Basic control-flow patterns*:**

Sequence: This workflow pattern simply specifies the order in which tasks should be executed.

Preconditions: This pattern is used to define preconditions for tasks. These preconditions should be satisfied in order to be able to execute the conditional tasks. If the preconditions are not satisfied, the conditional task should be discarded.

Nesting of guideline: Not all the tasks in a CPG are atomic and they may be composed of several sub-steps. Nesting pattern is used to define sub-components for CPG and composite tasks.

Parallel Split: When execution of the CPG reaches this point, several paths that should be executed in parallel get activated.

Synchronization: Several parallel paths that have diverged before during execution of the CPG converge at this point to a single path of execution.

Exclusive choice: It is similar to a parallel split with the difference that exactly one of the branches will be selected for execution based on a selection mechanism.

Simple merge: It is similar to a synchronization point with the difference that completion of any of the incoming branches causes the next task after the merge to be enabled. Execution of the rest of the paths continues

Please note that we reused the implementation of Basic Control patterns and the rest of the implementation is our contribution.

- ***Advanced branching and synchronization*:**

Multi-choice: It is similar to a parallel split with the difference that not necessarily all the branches get activated. For instance, it is possible to activate only 2 execution paths out of 6 after the split.

Structured synchronizing merge: A multi-choice is usually followed by a structured synchronization merge. When all of the active paths of the multi choice are completed, the next task after the structured synchronization merge gets activated. This workflow pattern acts similar to synchronization point with the difference that it is only waiting for the completion of the paths that are previously activated by a multi-choice and not all of the paths.

Structured discriminator: The same as simple synchronization. The only difference is that the parallel paths that are synchronized should be emanating from a single parallel split and they might not be engaged in other synchronization points and splits.

- ***Structural patterns*:**

Arbitrary cycle: Many CPG may contain repetitive tasks which should be iterated over a predefined number of times or until a specific condition is satisfied. A cycle may have several entry and exit points.

Implicit termination: Implicit termination causes the execution to finish when there are no more tasks to be executed during execution of a CPG. We believe that this is a feature of the CPG execution engine rather than a workflow pattern. Nonetheless we have included this in our comparison.

- ***State-based patterns:***

Deferred choice: This pattern is similar to a branch step. The difference is that only one of the branches based on the locally available data and not the user decision is selected. Upon activation of one of the branches the rest are discarded.

Interleaved parallel routing: This pattern represents a set of partially ordered tasks.

Milestone: Consider three tasks named A, B, and C contained in a CPG. Activity A is only enabled if activity B has been executed and C has not been executed yet, i.e. A is not enabled before the execution of B and A is not enabled after the execution of C. This pattern is called milestone. We believe that this is just a special case of using executional states of tasks as preconditions.

- ***Cancellation Patterns***

Cancel activity: This pattern enables users to cancel a specific task in the workflow.

Cancel case: Certain combination of conditions leads to cancellation of a set of tasks and their associated subtasks.

- ***New Patterns***:

Structured loop: This is similar to the arbitrary loop with the difference that there is exactly one entry and exit point.

Transient triggers: Triggers are signals that can act as conditions and cause tasks to change their executional states. For instance, having a blood pressure value more than a predefined may trigger “prescribe hypertension medication”. These triggers can be generated by tasks in the CPG or may come from an external source. Transient triggers are lost if they are not acted upon instantly. For instance, if the high blood pressure trigger is generated and the task “prescribe hypertension medication” is satisfied, this task will get activated otherwise the trigger is lost and the task will not get activated later because of this trigger.

Persistent triggers On the contrary to transient triggers, persistent triggers are permanent and they do not need to be acted upon instantaneously. They will last until they are used by a task to be triggered.

Cancel multiple instance task: Execution of this pattern leads to cancelation of tasks which are of a specific type. For instance, execution of this pattern can cause cancellation of all of the blood works that are due to be executed. All the cancelled tasks will be marked as unsuccessfully completed.

Complete multiple instance task: Same as cancel multiple instance task with the difference that tasks should become completed instead of cancelled. All the completed tasks will be marked as successfully completed.

Cancelling discriminator: This pattern is very similar to the synchronization pattern. The difference is that when the first branch reaches the cancelling discriminator, execution of the rest of the branches stops whereas execution of the branches continues in case of a synchronization pattern.

Structured N-out-of-M join: This pattern is a synchronization point that passes the execution to the next task when n out of m parallel branches are completed. Execution of the rest of the branches continues when the task after the structured N-out-of-M join is activated.

Cancelling N-out-of-M join: Exactly the same as above with the difference that execution of the rest of the branches stops when n out of m branches are completed.

Local synchronizing merge: This pattern is similar to the structured synchronizing merge. However, the number of tasks that this synchronizing merge is waiting for, may come from a local data and does not necessarily come from a multi-choice.

Critical section: Two or more sub-plans of the CPG are considered critical sections in the following situation: if a sub-task of one of sub plans is under execution, no task can be activated in the rest of the sub-plans. Thus, once one of the critical section sub-plans starts, the other sub-plans cannot start.

Interleaved routing: This pattern represents a composite task which is composed of a set of tasks that should be executed only once in any desired order. No two sub-tasks can be executed at the same time. Once all the sub-tasks finished, the composite task is regarded as a completed task as well.

- ***Decision patterns:***

If-then-else: This is a commonly used decision mechanism that exists in most of the CPG representation languages. If a specific condition is satisfied the *then* part of the if-then-else is executed, otherwise the *else* part will be executed.

Switch: This pattern is a more general form of the if-then-else pattern. This pattern has several conditions with a task assigned to each one of them. If a condition is satisfied the associated task will be executed and the switch will be completed and as the result, the rest of the conditional tasks are discarded. If two or more conditions are satisfied at the same time, the task with more priority will be executed and the rest of the satisfied conditions will be ignored.

Argumentation rules: When several options are available at a decision point during the execution of a CPG, argumentation rules which can be “against” or “for” specific options can be used to make a dynamic decision based on the available information. Each one of these rules add or remove a value from the overall balance of each option (which is set to zero before execution). The overall balance of the rules which are against(-) and in favor(+) of the option is the value which will be evaluated in order to determine if the option should be selected or not. Three types of Argumentations exist depending on how the overall balances are used towards choosing options: **(1)** Type1: The option with the largest overall balance will be selected and the rest will be discarded, **(2)** Type2: All the options with the overall balance of greater than a specific threshold which is common for all of them will be selected and the rest will be discarded, **(3)** Type 3: Each option has its own threshold and all the options with the overall balance of more than their specific threshold will be executed. The CPG representation language used for authoring the argumentation rules should provide the user with a rule language.

Preference for options: In situations where there are several options available to the user it may be desirable to express different levels of preference for them. These preferences can be shown to the users in order to leave the decisions to them or they can be acted upon by an execution engine in order to automate the decision making process. This preference can be either numerical or symbolic which can be translated to a numeric-based preference.

We contributed to the results in [101][102][103] by reviewing GASTON and NICHE CPG representation ontologies and evaluating the presence of the abovementioned workflow patterns. NICHE CPG representation ontologies are several ontologies that have been successfully developed and utilized in our research group namely; Prostate Cancer Ontology [20], Chronic Heart Failure and Atrial Fibrillation [107] , Nursing Care Plans Ontology [145] and a number of general CPG modelling ontologies [30][104][105][106]. Moreover, we reviewed Asbru [108], EON [137][155], GLIF [133][140], PROforma [136][138] for presence of the Decision Patterns. In Table 4.2, we list the presence of the abovementioned workflow elements in the reviewed CPG representation languages. Workflow patterns listed in Table 4.2 are organized according to the categorization provided in Table 4.1.

Table 4.2 Representable workflow patterns in Asbru, EON, GLIF, PROforma, GASTON and CPG representation ontologies developed in NICHE research group. ± symbol means that the corresponding feature is partially supported.

Workflow pattern	**Asbru**	**EON**	**GLIF**	**PRO*forma***	**GASTON**	**NICHE[1]**
Basic Control-Flow						
1. Sequence	+	+	+	+	+	+
2. Preconditions	+	+	+	+	+	+
3. Nesting of guidelines	+	+	+	+	+	+
4. Parallel Split	+	+	+	+	+	+
5. Synchronization	+	+	+	+	+	+
6. Exclusive Choice	+	+	+	+	+	+
Advanced Branching and synchronization						
7. Multi choice	+	+	+	+	+	+
8. Structured synchronization merge	±	-	-	-	+	+

[1] NICHE represents all the CPG representation ontologies developed in NICHE research group

9. Structured discriminator	+	+	+	+	+	+
Structural Patterns						
10. Arbitrary cycle	-	+	+	-	-	+
11. Implicit Termination	+	+	+	+	+	+
State based patterns						
12. Deferred choice	+	-	+	+	-	+
13. Interleaved parallel routing	+	-	-	-	-	-
14. Milestone	-	-	-	+	-	-
Cancellation Patterns						
15. Cancel activity	+	+	+	+	-	+
16. Cancel case	+	-	±	+	-	+
New Patterns						
17. Structured loop	+	+	+	+	-	+
18. Transient trigger	-	-	-	+	-	-
19. Persistent trigger	-	-	+	+	-	-
20. Cancel multiple	+	-	+	+	-	-

instance activity						
21. Completed multiple instance activity	+	-	-	+	-	-
22. Cancelling discriminator	+	-	-	+	-	+
23. Structured N-out-of-M join	+	-	+	+	+	+
24. Cancelling N-out-of-M join	-	-	-	+	-	+
25. Local synchronizing merge	-	-	-	+	+	+
26. Critical section	+	-	+	-	-	-
27. Interleaved routing	+	-	+	-	-	-
Decision Mechanisms						
28. If-then-else	+	+	+	+	+	±
29. Switch	+	+	+	+	-	-
30. Argumentation rules	-	+	+	+	-	-
31. Preference for options	+	-	-	+	-	-

As we discussed previously, we need to find the coverage that our ontology should have in order to be used as a comprehensive DKO. This survey found the workflow patterns that CPG-DKO should be capable of representing. Not only these workflow patterns should be representable, but also all their variants in order to deal with difference in granularity. For instance, if we are able to represent cyclic tasks, all the possible variants are needed to be supported in order to successfully use the developed ontology as a comprehensive DKO.

4.4. OWL-DL Implementation of the CPG Domain Knowledge Ontology

We discussed the workflow patterns and their variants are needed to be modelled CPG-DKO. In this section, we discuss how each of the workflow patterns is represented in terms of OWL classes, properties and instances. Figure 4.2 (a) shows the class hierarchy of the CPG-DKO. In Figure 4.2 (b) class *Task* is expanded to show more details of this ontology.

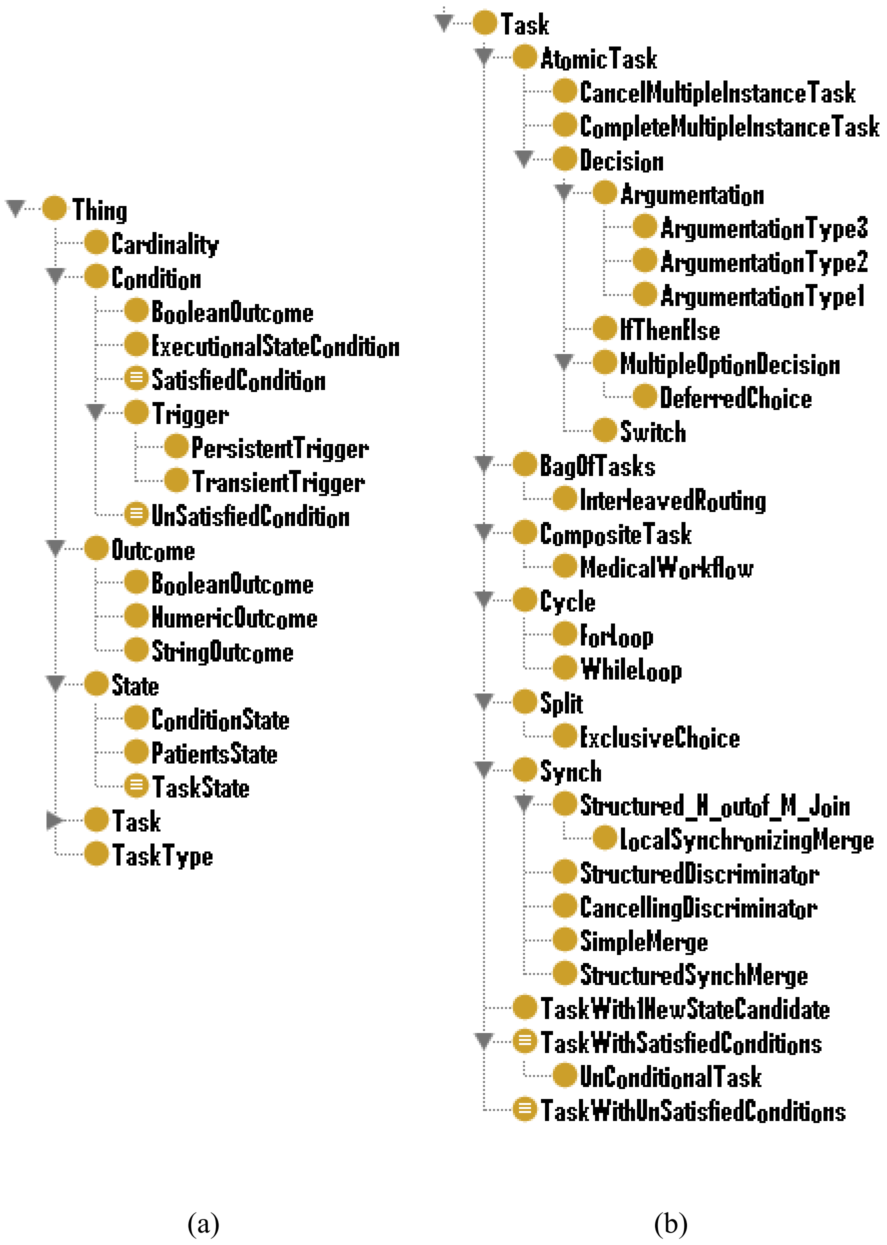


Figure 4.2 (a) Class hierarchy of CPG-DKO. Please note that class Task is not expanded. (b) Subclasses of the Task class.

In this section of the thesis, we go through some of the identified workflow patterns and discuss the classes used to represent those patterns, their associated properties and instances and their purposes. TURTLE syntax is used to write OWL triples of our ontology. In order to easily identify classes, properties and instances in the text, class names are italicized and their first letter are capitalized (e.g. *ClassNameExample*), property names are *italicized* (e.g. *propertyExample*) and instance names are underlined (e.g. instanceExample). Please note that the Sequence, Precondition, Multi-choice, Simple merge, Exclusive Choice, Synchronization, Parallel Split and Nesting of guidelines have been reused from the existing NICHE CPG ontologies and the rest of the implementation is our contribution.

Sequence

A simple object property can be easily used to assert the sequence in which tasks should be executed in a CPG execution engine. This property is called *hasNext*. Domain and range of this class are *Task* class. This class is used to model medical tasks in our ontology. As an example, assume two medical tasks called t1 and t2 exist in a CPG and we want to assert that task t2 should be executed after task t1. We simply use the *hasNext* property and assert the following triple in our ontology: ":t1 :hasNext :t2". This property helps the execution engine to activate the task t2 when the task t1 is finished. This property has been reused from existing NICHE ontologies.

Nesting of guideline

Tasks may be composed of several sub-tasks. An obvious example of this situation is the CPG itself that is composed of several sub-tasks that should be completed in order to consider the CPG as a completed task as well. *hasTask* property(inverse of *isTaskOf*) with the domain and range of *Task* is defined to assign subtasks to a composite task. *hasFirstTask* (inverse of *isFirstTaskOf*) is a sub property of *hasTask* that indicates the first subtask to be executed among all the subtasks. For instance, in order to capture cpg1 has subtasks t1 and t2 and t1 is the first task that should be executed among the subtasks, the following triples will be asserted:

```
:cpg1 a CPG;
:hasFirstTask :t1;
```

```
:hasTask :t2.
```

CPG is the class that represents the Clinical Practice Guidelines. More levels of nesting are possible by assigning composite tasks as tasks of other tasks. Tasks with no subtasks belong to the class *AtomicTask*. These properties are reused from existing NICHE ontologies.

Parallel Split

Split class represents this pattern. This class has the *hasBranch* property (inverse of *isBranchOf*) with the domain of *Split* and range of *Task*. This property points to the tasks that may be executed in parallel upon activation of the split point. The following example shows a split with 4 tasks that may be executed in parallel when the execution engine reaches that split point. We call this workflow structure also as the Branch structure.

```
:b1 a Split;
     :hasBranch :t1,:t2,:t3,:t4.
```

This part of the ontology is reused from existing NICHE ontologies.

Synchronization

Synch class represents this pattern. Instances of the *Synch* class indicate what tasks are synchronized using the property *isWaitingForTask* (inverse of *isWaitedBy*) with the domain of *Synch* and range of *Task*. *Cardinality* class, *hasCardinalityType* and *hasCardinalityValue* properties can be used to define how many of the parallel paths should complete in order to pass the execution to the next task after the *Synch* point. They are used in the same fashion that condition satisfaction criteria have been defined. The following example shows a synchronization point that is waiting for 4 parallel paths. Completion of minimum 2 out of 4 parallel paths leads to passing the execution to the next task after the *Synch* point.

```
:s1 a :Synch;
     :isWaitingForTask :t1,:t2,:t3,:t4;
     :hasCardinalityType :min;
     :hasNumericValueForCardinality “2”^^xsd:positiveInteger.
```

This part of the ontology is reused from existing NICHE ontologies.

Exclusive Choice

ExclusiveChoice class is a subclass of *Branch* class with the cardinality type of <u>max</u> and cardinality value of one. It is execution engine’s responsibility to perform the execution correctly by asking the user which paths he or she wants to execute. This class is defined as follows:

```
:ExclusiveChoice rdfs:subClassOf
[owl:intersectionOf(
     :Split
     [a owl:Restriction;
          owl:onProperty :hasCardinalityType;
          owl:hasValue :max
     ]
     [a owl:Restriction;
          owl:onProperty :hasCardinalityTypeValue;
          owl:hasValue “1”^^xsd:positiveInteger
     ]
)].
```

Simple merge

This pattern is a *Synch* point with the cardinality type of <u>min</u> and cardinality value of 1. In order to assert that, the *SimpleMerge* class is defined as follows:

```
:SimpleMerge rdfs:subClassOf
[owl:intersectionOf(
      :Synch
      [a owl:Restriction;
            owl:onProperty :hasCardinalityType;
            owl:hasValue :min
]
      [a owl:Restriction;
            owl:onProperty :hasCardinalityTypeValue;
            owl:hasValue "1"^^xsd:positiveInteger
]
)].
```

Multi-choice

MultipleOptionDecision class represents this pattern and *hasOption* property (inverse of *isOptionOf*) with the domain of *MultipleOptionDecision* and range of *BooleanOutcome* points to the outcomes that may lead to activation of their assigned branches. These outcomes are used as conditions for the first task of the paths. A selected option is an outcome that has happened as a result of execution the instance of the *MultipleOptionDecision*.

We again use the *Cardinality* class, *hasCardinalityType* property and *hasNumericValueForCardinality* as we used in preconditions pattern to indicate how many branches are allowed to be activated when the branch step has been reached. The following example shows a multi-choice pattern which has 4 possible options. The user or the execution engine can choose no more than two of the options:

```
:Choose_a_BP_Medication a :MultipleOptionDecision
:hasOption :Aldactone, :Dyrenium, :Lasix, :Lozol;
:hasCardinalityType :max;
:hasNumericValueForCardinality "2"^^xsd:positiveInteger.
```

```
:take_Aldactone :hasCondition :Aldactone.
:take_Dyrenium :hasCondition :Dyrenium.
:take_Lasix :hasCondition :Lasix.
:take_Lozol :hasCondition :Lozol.
```

When the execution engine reaches Choose_a_BP_Medication, it shows the outcomes Aldactone Dyrenium, Lasix, Lozol as the options to the user. Selection of any of the options leads to execution of the corresponding path.

Switch

Instances of the *Switch* class represent this pattern. *hasSwitchCondition* property (inverse of *isSwitchConditionOf*) with the domain of *Switch* and the range of *Condition* indicates what conditions lead to activation of the switch tasks. When a condition of the switch construct is satisfied, the property *hasAssociatedPath* with the domain of *Condition* and range of *Task* indicates what task should be executed as a result. Since only one task can be activated, if two or more conditions are satisfied, we use the transitive property *hasPriorityOverCondition* with the domain and range of *Condition* to indicate what conditions have more priority over others. The example below shows how a switch with three conditions can be modeled in our ontology.

```
:switch1 a :Switch;
    :hasSwitchCondition :sc1;
    :hasSwitchCondition :sc2;
    :hasSwitchCondition :sc3.

:sc1 :hasPriorityOverCondition :sc2.
:sc2 :hasPriorityOverCondition :sc3.

:sc1 :hasAssociatedPath :t1.
:sc2 :hasAssociatedPath :t2.
```

```
:sc3 :hasAssociatedPath :t3.
```

We did not include implementation of the rest of the workflow patterns due to space limitations.

4.5. Conclusion

Review of the existing CPG representation languages shows that they are able to represent a wide range of workflow patterns. However, besides the basic workflow patterns, there is not much of overlap between the supported workflow patterns. Moreover, the supported workflow patterns can be described with different detail levels. Hence, none of these languages can be used as a comprehensive domain knowledge ontology due to lack of (1) enough coverage of the domain knowledge and (2) expressivity to represent details of the covered knowledge. To address this problem, we developed a background CPG ontology based on our survey of the existence of workflow patterns in prominent CPG representation languages. In comparison to the existing CPG representation languages, CPG-KPO can represent the most comprehensive set of workflow patterns. This ontology is especially superior to NICHE CPG ontologies as they only support basic control flow patterns discussed in Table 4.1. Therefore, we believe this ontology can alleviate the lack of enough overlap between several CPG ontologies. CPG-DKO can represent not only all the identified workflow patterns but also all their identified variations. Therefore, this ontology can also alleviate the differences in granularity level of CPG ontologies.

A limitation of our research is that CPG-DKO bears several resemblances to NICHE CPG ontologies because of reusing their implementation of the basic control workflow patterns in CPG-DKO. Hence, CPG-DKO might be more effective when used as the background ontology of the NICHE CPG ontologies. Evaluations using other CPG ontologies can shed light on usefulness of CPG-DKO as the background ontology for other CPG ontologies. Results of this evaluation can be used to further improve this ontology.

CHAPTER 5: EXPRESSION ONTOLOGY

5.1. Introduction

We developed a CPG-DKO in the previous section of the thesis in order to be used as an Interlingua between several heterogeneous CPG ontologies. CPG deal with qualitative data—the data is either being measured by a device or is calculated using a set of patient data items. Clinical decisions are based on such qualitative analysis of the data. However, even though ontologically modelled CPG may contain mathematical, Boolean and string manipulation and comparison functions, CPG-DKO is unable to represent this aspect of CPG representation. This is a great hindrance to representation of different heterogeneous CPG ontologies in CPG-DKO. Moreover, to execute CPG, we need to have the ability to represent and perform mathematical functions on clinical data. The outputs of these functions guide the clinical decisions and the ensuing clinical workflows.

In a logical framework, such as an ontology, there are no natural mechanisms to handle quantitative data processing vis-à-vis mathematical functions. Therefore, there is a need to develop a formal representation scheme for the enacting mathematical functions in a semantic web framework. In this chapter, we present a dedicated expression ontology in OWL-DL called *Expression Ontology* that was developed to process quantitative clinical data to support the execution of a CPG. This ontology is in OWL-DL due to using constructs such as owl:hasValue that pushes the ontology out of OWL-lite to OWL-DL. Moreover, this ontology is not in OWL-Full as we did not use any of OWL-Full capabilities such as mixture of classes, properties and instances. The expression ontology works in tandem with the execution engine, which utilizes OWL and SWRL reasoning services for execution.

SWRL language has been used to implement the execution semantics of the expression ontology. SWRL offers a range of built-ins to execute expressions to perform mathematical and Boolean computations and to perform string manipulation. Using OWL and SWRL creates a unified representation and execution formalism for both CPG and expressions. Moreover, SWRL allows writing rules using OWL classes, properties and instances. This enables the definition of expressions that utilize the structure of the ontology that can be reasoned over. For instance, it can check the membership of an instance in a specific OWL class and use the result as the precondition for execution of the rule.

5.2. Expression Ontology Engineering

We used the same ontology development methodology used for development of CPG-DKO in section 4.2 to develop the expression ontology. The steps of this ontology development methodology are depicted in Figure 4.1. We discuss the details of each of these steps in the development of the expression ontology in the rest of this section.

Step 1. Determine the domain, scope and purpose of the ontology

We need an ontology that is able to represent numerical, string and Boolean manipulation and comparison functions and their combinations–i.e. expressions–and their effects in the decision making process of CPG execution. Moreover, the ontologically represented expressions should be interpreted according to the patients' data during the execution in order to deliver successful clinical decision support. For instance, ontologically represented BMI formula should be computed during execution when patient's data are available in order to make decisions regarding obesity. For execution time interpretation of the expressions, the execution semantics should be captured in a computer understandable format. To put it simply, we need an ontology for representation and execution of mathematical, Boolean and string manipulation and comparison in a semantic web based framework.

Step 2. Considering re-using existing ontologies

In the CPG modeling literature we note that most major CPG representation formalisms, such as GLIF, EON, GLIF, PROforma and GASTON, have a dedicated expression language that supports mathematical, comparison operators and logical rules to represent the decision criteria, preconditions, post conditions and decisional rules. GELLO [92] is an example of an expression language developed for GLIF. However, we believe that developing a new expression ontology in OWL is a superior solution to reusing these existing languages due to the following reasons:

- In order to use an existing expression language such as GELLO in our OWL-based CPG execution engine, an interface program should be written which transfers the data and from our CPG ontology to the GELLO executer and transfers the results back. We will be then facing the questions when and how a transfer should happen.

- Besides an interface program, there should also be a parser and an execution engine for that expression language that works in tandem with the CPG execution engine. However, using OWL and SWRL enables us to use the same OWL reasoner used for execution of CPG, to interpret expressions as well. Therefore, all the execution related tasks are performed in unified reasoning framework.

- We are using OWL ontologies to model CPG in a computer interpretable format. Therefore, the modelling of expressions in OWL creates a unified representation framework for modeling CPG and the related data processing operators. This increases the shareability of the computerized CPG as a lower number of softwares components are needed to modify, visualize and execute them.

Due to high levels of expressivity in the existing expression languages of CPG representation languages [34][88][108][133][134][135][136][138][137][140][155], our intent is not to create a new language that is more expressive or efficient compared to the existing ones. Rather, our objectives for developing a dedicated expression ontology for modelling and execution of CPG and their expressions in a unified representation and reasoning framework.

- **Step 3: Enumerate important terms in the ontology**

In this step of the ontology development methodology, we need to create a list of concepts that our ontology is going to represent. Review of the CPG representation language shows that these languages commonly support the following concepts: Function, Variable, Operator, Input and Output. A function represents the smallest computation entity. A function takes one or more inputs, perform an operation on them and produce an output. Inputs can be predefined values or variables and outputs can only be variables. Different expression languages support different types of variables and operators. Boolean, string and numerical variables are the most common supported variable types. Depending on the variables used in a function, different types of operators may be utilized for manipulation of the data or comparing them. Table 5.1 shows the operators associated with each type of variables.

Table 5.1 Common manipulation and comparison operators for Boolean, numerical and string variables in CPG expression languages

String Variables		**Numbers Variables**		**Boolean Values**
Manipulation Operators	**Comparison Operators**	**Manipulation Operators**	**Comparison Operators**	**Manipulation Operators**
Concatenation	=	+	=	And
	≠	-	<	Or
		/	>	Xor
		*	<=	Not
			>=	
			≠	

Table 5.1 shows the types of variables and their associated operators that our ontology needs to model. Please note that we are reusing the existing expression languages at the knowledge level by remodelling their concepts in our ontology rather than reusing them at the implementation level by reusing their actual implementations in their original format.

- **Step 4 and 5: Implementation**

We implemented this ontology in two steps: (1) We first implemented classes, properties and instances that represent the concepts identified in the previous step of the ontology development; (2) We then implemented the execution semantics in SWRL language. SWRL rules are written to enable an OWL reasoner with SWRL support to execute the expression represented in our expression ontology. The developed ontology in the first step is discussed in section 5.3 and the SWRL rules written to execute expressions are disused in section 5.4

- **Step 6: Instantiation**

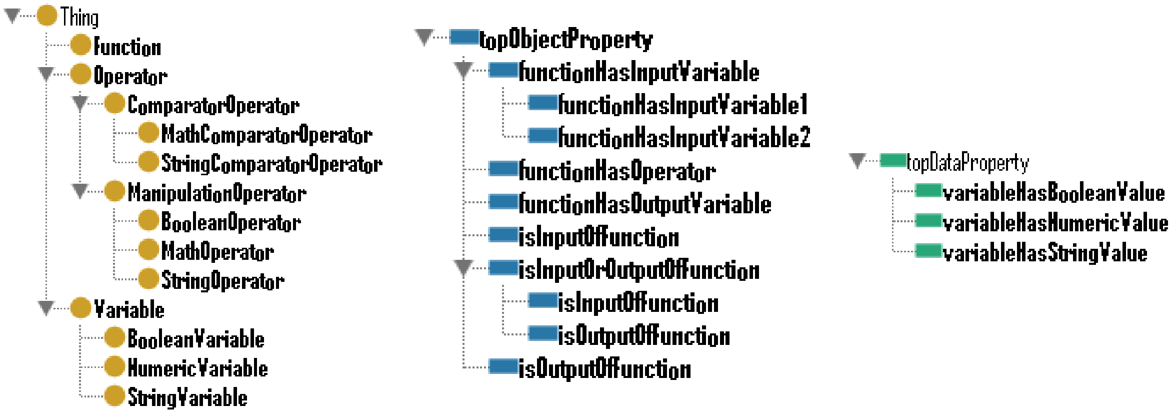
We instantiated the expression ontology in order to represent the expressions in several real and imaginary CPG.

- **Cycling through steps of the ontology development methodology**

For each of the modelled operators we created a function that utilized that operator. We then fed each function with at least two different input sets and validated the output. If any of the outputs were not correct, we would check the classes, properties and instances related and the SWRL rules associated to it. We went through the ontology development twice. After finishing the second cycle, we did not come across a miscalculation in any of the operators.

5.3. Expression Ontology in OWL-DL

In order to easily identify classes, properties and instances in the text, class names are *Italicized* and their first letter are Capitalized (e.g. *ClassNameExample*), property names are *italicized* (e.g. *propertyNameExample*) and instance names are underlined (e.g. instanceNameExample). Figure 5.1 shows the class hierarchy, data type and object properties of the ontology. The purpose of each of the classes and then its associated properties are described. Moreover, a practical example in each subsection has been discussed.



(a) Class Hierarchy (b) Object Properties (c) Data Type Properties

Figure 5.1 (a) Class hierarchy, (b) object properties and (c) data type properties of the Expression Ontology

5.3.1. Variables

Expression variables are represented by the *Variable* class. Boolean, string and numerical variables are represented by the following subclasses of the *Variable* class respectively: *BooleanVariable*, *StringVariable* and *NumericalVariable*. Table 5.2 shows the data type properties that can be used to assign values to these variables. All of these properties are sub properties of the *topDataProperty* data type property.

Table 5.2 Datatype properties that are used to assign values to variable in the Expression Ontology, their domains and ranges

Property	**Domain**	**Range**
variableHasBooleanValue	*BooleanVariable*	xsd:boolean
variableHasStringValue	*StringVariable*	xsd:string
variableHasNumericalValue	*NumericalVariable*	xsd:float
variableHasValue	*Variable*	

As an example, imagine that weights of patients are retrieved from an EMR during execution and compared to 84kg in order to make a decision regarding the dosage of the Synthroid drug. The following instances of the *NumericVariable* class represent these two variables:

```
:patientsWeight a :NumericVariable. # patient’s weight
:weightThreshold a :NumericVariable; # threshold
     :variableHasNumericalValue “84”^^xsd:float.
```

If we assume that EMR shows that the patient weighs 95kg the following triple is added to the ontology to reflect this fact:

```
:patientsWeight :variableHasNumericalValue “95”^^xsd:float.
```

5.3.2. Functions, Inputs and Outputs

As we discussed previously, a function is the smallest entity for manipulation or comparison of data. Class *Function* represents this concept. Each function accepts two inputs and an operator and generates an output. Table 5.3 shows the properties used to assign these items to a function.

Table 5.3 Properties used to assign the input and output variables and operator to functions in the Expression Ontology

Property	**Domain**	**Range**	**Inverse Property**
functionHasInputVariable1	*Function*	*Variable*	*isInputOfFunction*
functionHasInputVariable2	*Function*	*Variable*	*isInputOfFunction*
functionHasOperator	*Function*	*Operator*	*isOperatorOfFunction*
functionHasOutputVariable	*Function*	*Variable*	*isOutputOfFunction*

We continue our weight comparison example to depict how these properties are used to define a function that can be used in a real CPG. The following instantiation shows a function that accepts the variables patientsWeight and weightThreshold from our previous example:

```
:weight_comparison_function a :Function;
      :functionHasInputVariable1 :patientsWeight;
      :functionHasInputVariable2 :weightThreshold;
```

In order to make use of the comparison result for decision making, the output of the above function should be captured in a Boolean variable using property *functionHasOutputVariable*:

```
:weight_comparison_result a :BooleanVariable.
:weight_comparison_function a :Function;
      :functionHasInputVariable1 :patientsWeight;
      :functionHasInputVariable2 :weightThreshold;
      :functionHasOutputVariable :weight_comparison_result.
```

The value of the weight_comparison_results is assigned to it by the CPG execution engine during the execution using property *variableHasBooleanValue*. We yet have to define the type of comparison that this function performs on its input data. We discuss the concepts of operators in the next subsection.

5.3.3. Operators

Each function has exactly one operator indicated using the functional property *functionHasOperator* with the domain of *Function* and range of *Operator*. The *Operator* class represents different types of operations representable in our expression languages. Figure 5.1 shows the sub classes of the *Operator* class. Classes *BooleanOperator*, *MathOperator* and *StringOperator* represent the manipulation operators and classes *MathComparatorOperator* and *StringComparatorOperator* represent the comparator operators.

Operator of a function determines the type of output it has. For instance, if a function is using any of the comparator operators the outcome will be *BooleanVariable*. The following triples assert this fact:

```
[a owl:Restriction;
 owl:onProperty :functionHasOperator
 owl:someValuesFrom :CompratorOperator
]
rdfs:subClassOf
 [a owl:Restriction;
  owl:onProperty :functionHasOutput
  owl:allValuesFrom :BooleanVariable
 ].
```

Therefore, if we assign a numerical variable to output of a comparator function, expression ontology becomes inconsistent. This makes the ontology instantiation process less error-prone and easier to debug. Several other similar OWL triples have been added to expression ontology to assert the type of outputs that functions should have based on their operators. Continuing from our example in which we defined the function weight_comparison_function, we define the operator for this function:

```
:weight_comparison_function a :Function;
      :functionHasInputVariable1 :patientsWeight;
      :functionHasInputVariable2 :weightThreshold;
      :functionHasOutputVariable :weight_comparison_result;
      :functionHasOperator :gtMO.# greater than
```

In the above example, the value of the weight_comparison_result will be true if patientsWeight > weightThreshold and false otherwise.

5.3.4. Expression Ontology Instantiation Examples

To further familiarize the reader with our expression ontology, we go through two examples.

Example1. In this example, we create a function with two input variables synthroid_Pill_Dose and number_Of_pills_taken. This function multiplies these numbers and assigns the value to the variable total_TSH_Dosage. Function f1 is defined as follows:

```
:f1 :functionHasInputVariable1 :synthroid_Pill_Dose;
    :functionHasInputVariable1 :number_Of_pills_taken;
    :functionHasOutputVariable :total_TSH_Dosage;
    :functionHasOperator :multiplyMO.
```

Now if we assign values 3 and 6 to a and b using the following triples

```
:synthroid_Pill_Dose :variableHasNumericalValue “.075”^^xsd:float.
:number_Of_pills_taken :variableHasNumericalValue “3”^^xsd:int.
```

We want the following triple to be added to our ontology:

```
:total_TSH_Dosage :variableHasNumericalValue “.225”^^xsd:float.
```

As we will see later in section 5.4, this calculation is performed using SWRL built-ins.

Example2. In this example, we use the result from our previous example to decide if synthroid_Pill_Dose multiplied by number_Of_pills_taken is bigger than “.3”. If that is the case and the TSH is more than normal, patient should be advised to lose weight. Therefore, TSH_biggerthan_point_three that is output of the function f2, is reused as the precondition of advise_patient_to_lose_weight. Function f2 is defined as follows:

```
:f2 :functionHasInputVariable1 :total_TSH_Dosage;
    :functionHasInputVariable2 “.3”^^xsd:float;
    :functionHasOutputVariable :TSH_biggerthan_point_three;
    :functionHasOperator :gtMO.

:advise_patient_to_lose_weight a :Task;
      :hasCondition :TSH_biggerthan_point_three, :TSH_High;
      :hasCardinalityType :all.
```

5.4. Execution Semantics of Expression Ontology in SWRL

As we mentioned earlier, we need to define the execution semantics of the expression language in order to be able to calculate the results of expressions using an OWL reasoner. Thus, we write SWRL rules that enable dynamic calculation of results of functions as their inputs become available during the execution. Please note that these rules are only used by the OWL reasoner and a user who instantiates this ontology does not need to be aware of their existence. In order to describe the general structure of these SWRL rules, we discuss the rule written for handling addMO (+) mathematical operator in details:

```
functionHasInputVariable1(?f1,?iv1) ^
functionHasInputVariable2(?f1,?iv2) ^
functionHasOutputVariable(?f1,?ovar) ^
functionHasOperator(?f1,:addMO) ^
variableHasNumericValue(?iv1,?v1) ^
variableHasNumericValue(?iv2,?v2) ^
swrlb:add(?v1PlusV2,?v1,?v2)
-> variableHasNumericValue(?ovar,?v1PlusV2)
```

In order to describe the values assigned to SWRL variables in the above rule, we discuss the working of this rule for the following function instantiated in the expression ontology:

```
:func1 a :Function;
    :functionHasInputVariable1 :a;
    :functionHasInputVariable2 :b;
    :functionHasOutputVariable :c;
    :functionHasOperator :addMO.

:a :variableHasNumericValue “8”^^xsd:float.
:b :variableHasNumericValue “3”^^xsd:float.
```

The above example shows a function that adds the value of a and b expression variables and puts the result in expression variable c. We are assuming that values of variables a and b are 8 and 3 respectively at this specific moment of the execution.

In the remainder of this section, we discuss the general structure of the written SWRL rules and give examples from func1 that makes use of the addMO operator. Each rule is composed of four parts that have the following purposes:

1. **Identifying the input values of the functions when available and assign them to SWRL variable.**

The following SWRL axioms in the body of the rule find the input variables and the output variable from the expression ontology using properties *functionHasInputVariable1*, *functionHasInputVariable2* and *functionHasOutputVariable* and assign them to SWRL variables ?iv1, ?iv2 and ?ov respectively:

```
functionHasInputVariable1(?f1,?iv1) ^
functionHasInputVariable2(?f1,?iv2) ^
```

```
functionHasOutputVariable(?f1,?ov)
```

Table 5.4 shows the expression ontology elements that are assigned to SWRL variables ?iv, ?iv2, ?ov and ?f1 for execution of func1. Depending on the type of expression variables, different properties are used to access their values. If a variable is an instance of the *BooleanVariable*, *StringVariable* or *NumericVariable*, properties *variableHasBooleanValue*, *variableHasStringValue* or *variableHasNumericVariable* will be used to access their values respectively. Since addMO inputs are numerical variables, we use the property *variableHasNumericVariable* in this SWRL rule to find the values of the input variables:

```
variableHasNumericVariable(?iv1,?v1)^
variableHasNumericVariable(?iv2,?v2)^
```

As a result of execution of this part of the SWRL body in our example, values 8 and 3 are assigned to ?v1 and ?v2 SWRL variables. Please note that if the input variables have no values yet, no values are assigned to ?v1 and ?v2 and this rule will not be executed.

Table 5.4 Values assigned to SWRL variables during execution of a function with addMO operator

SWRL Variable	**Assigned Value from Expression Ontology**
?f1	func1
?iv1	a
?iv2	b
?ov	c
?v1	8
?v2	3

2. **Identifying the operator**

This part of the SWRL rules checks the operator of the function using property *functionHasOperator*. If the function's operator is what the SWRL rule is intended for, the SWRL rule will be executed. For instance, the following SWRL axiom acts as a precondition and lets the SWRL rule to be executed if the value of the property *functionHasOperator* is addMO:

```
functionHasOperator(?f1,:addMO)
```

If the operator of the function is not addMO, this SWRL rules does not apply to it.

3. **Calculating the results of the operation.**

This part of the rule performs the operation identified in the previous parts of the rule on the input values. Since the SWRL built-in that corresponds addMO is swrlb:add, the following axioms performs the addition operation on ?v1 and ?v2 and puts the result in SWRL variable ?v1PlusV2:

```
swrlb:add(?v1PlusV2,?v1,?v2)
```

As a result of execution of this part of the SWRL rule in our example, value 11 is assigned to SWRL variable ?v1PlusV2.

4. **Assigning the results to the output variable of the function.**

The SWRL variable ?v1PlusV2 represents the result of the addition and the SWRL variable ?ovar represents c that is the output variable of func1. We just need to assign?v1PlusV2 as the value of the property *variableHasNumericVariable* to ?ov using the following axiom in the head of the SWRL rule:

```
variableHasNumericValue(?ov,?v1PlusV2)
```

As the result of execution of the abovementioned rule in our example, 11 will be assigned to variable c as the value of the property *variableHasNumericValue*. Please note that this SWRL rule is not specific to func1 and can execute any instantiation of the *Function* class that uses addMO as its operator.

5.5. Conclusion

As we discussed previously, DKO should be covering every aspect of the domain knowledge in order to be used as the background ontology in our knowledge morphing framework. An aspect of CPG domain knowledge that was not covered in CPG-DKO was representation of the expressions. Expression ontology aims at filling this gap in order to provide CPG-DKO with more coverage of the CPG domain knowledge. This ontology allows a CPG encoder to define functions that use a wide range of useful operators to perform mathematical, string and logical operations, perform comparisons and use the result in the decision making process of our execution engine. Our expression ontology and its execution semantics along with our CPG ontology provides us with a unified formalism for both representation and execution of CPG.

We like to point out that this is the first attempt to represent and execute expressions in an ontology. Even though our ontology is not capable of representing very complex expressions such as sets, bags, temporal expressions, etc. it is a start for developing OWL ontologies for representation of more complex expressions.

A disadvantage of using OWL and SWRL for implementation of expressions is inability to execute the operators that are not supported by SWRL. Albeit it is possible to extend SWRL with new built-ins, it can harm the shareability of the computerized CPG as the extension code is reasoner-specific.

An improvement of this research will be a formal syntax for the our expression language. Defining formal syntax makes it possible to write programs that can parse and check the syntax for correctness. Another important improvement can be increasing the expressiveness of the language using more XML data types and SWRL built-ins. It is also possible to improve the expressiveness of the expression ontology using query languages such as SPARQLE [130].

CHAPTER 6: OWL-BASED KNOWLEDGE EXECUTION ENGINE

6.1. Introduction

In ontology-based DSS, an execution algorithm is utilized to perform reasoning on the input ontologies in order to derive the best decision according to current state of the problem and user's input. This execution algorithm defines (1) the details of the reasoning process, (2) the ontology elements that are involved in this process and (3) the frequency that this reasoning process is needed. We believe that a general knowledge execution algorithm is not possible to be developed as each domain area has its own subtleties that demands specialized reasoning algorithms for decision making. Moreover, we also believe that a general knowledge execution algorithm for a particular domain area is not feasible either as this algorithm operates directly on a specific ontology that can be quite different compared to other ontologies of that domain. For instance, a knowledge execution algorithm developed for instances of ontology O1 will not be able to operate on instances of ontology O2 due to difference in the class, properties and instances used for knowledge representation. Thus, not only we believe that each domain area needs a specialized execution algorithm but also every ontology in that domain area. Since the input of our knowledge morphing framework are ontologically modeled CPG in DKO, we develop a knowledge execution algorithm for CPG modeled in DKO. The module that runs this algorithm is called *CPG Execution Algorithm*.

Functionally speaking, a CPG execution engine interprets the procedural and decision logic inherent within the computerized CPG and in conjunction with patient data and physician's input. In other words, the CPG execution engine orchestrates the executional flow of the CPG—i.e. stipulating the ordering of clinical tasks, evaluating the satisfaction of criteria of decisions, constraints, conditions and responding to outcomes of clinical tasks—to provide patient-specific and disease-specific recommendations about care interventions and clinical decision making. The execution of CPG is one of the most challenging topics in health informatics in general and healthcare knowledge management in particular. CPG encapsulate highly complex elements in terms of domain knowledge, decision logic, clinical constraints, set of clinical actions and temporal relations; therefore the execution of a computerized CPG needs to monitor and handle such complexities in line with the specific patient data and the institution's operational conditions. In operation, a CPG execution engine, therefore, operates as (a) *interpreter* of the CPG knowledge represented in a specific health knowledge representation formalism; (b) *operator* of the CPG decision logic under different clinical situations; (c) *orchestrator* of the CPG process model to manage the CPG workflow; and (d) *monitor* of the clinical triggers and temporal constraints.

There are two main approaches to develop CPG execution engines: (i) A dedicated CPG execution engine approach stipulates the development of a specialized execution engines that are specific to a particular CPG representation formalism. In this regard, quite a few CPG representation formalisms, in particular GLIF [133] and EON [137], have specialized CPG execution engines that apply a graph parsing approach which treats the CPG as a graph and executes the CPG steps as a graph-based state transition problem; (ii) A transformation approach that transforms the original CPG representation scheme to an existing knowledge processing/inferencing method, such as Petri nets, etc, and then exploits existing execution engines supporting the standard knowledge processing method [140]. PROforma [136] and GASTON [134] execution engines are developed with this approach.

The emergence of Semantic Web technologies has offered new ways of developing knowledge-centric CPG-based CDSS. In a semantic web paradigm, the clinical knowledge and workflows encapsulated within a CPG can be modeled and computerized using a semantically-rich formalism—i.e. an ontology using the Web Ontology Language (OWL) [142]—and the ontologically-modeled CPG can be executed by reasoning over the CPG's execution logic captured using logical rules. The use of OWL ontologies as knowledge representation formalisms for representing CPG is quite profound and is used by several CPG computerization formalisms [106][133]0[145].

We argue that the existing CPG execution engines, both graph-based and transformation based approaches that use OWL-based ontologies to model the CPG, do not attempt to exploit the reasoning capabilities offered by OWL to provide a generic execution engine. Reasoning services offered by OWL can be readily applied to execute CPG that are modeled using OWL. The advantage of our approach is the realization of a native OWL-based environment for both the representation and execution of CPG such that both the form and function of a CPG is captured by representational structures. More so, this will help to realize a generic OWL-based CPG execution engine supporting OWL-based CPG models.

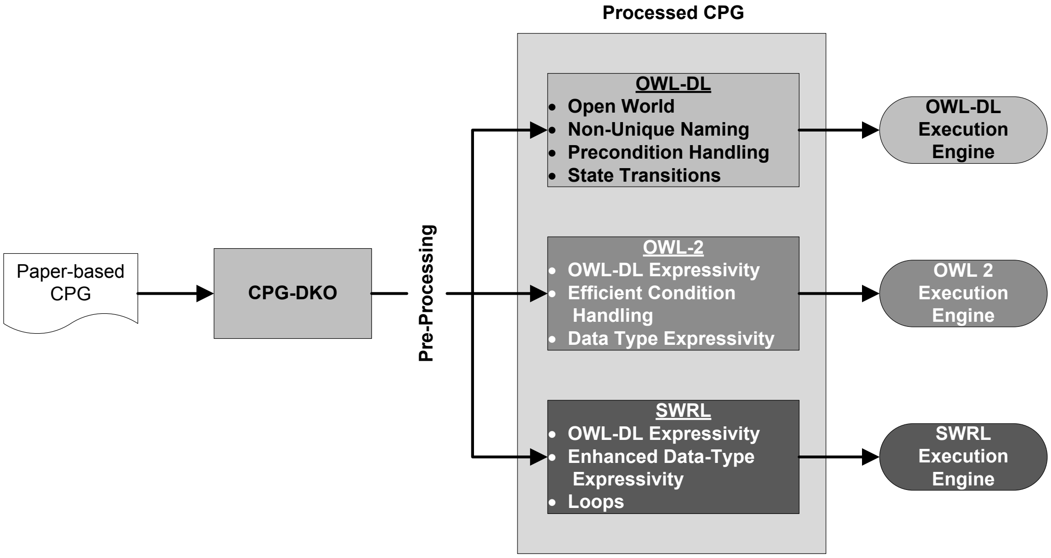


Figure 6.1 Schematic of our CPG modeling and execution approach.

In this chapter of the thesis we present a CPG modeling and execution framework (shown in Figure 6.1) that comprises two main elements:

a) *OWL based CPG ontology*:

for modeling a CPG in terms of an ontological model—the CPG ontology entails a rich representation of CPG based concepts, relations and axioms. The computerization of a CPG is achieved by instantiating it using CPG-DKO in chapter 4 of the thesis.

b) *OWL-based CPG execution engine*:

This engine is capable of executing a CPG, modeled using the CPG ontology, in terms of suggesting CPG-mediated recommendations in line with a clinical case, patient data and clinician decisions. To develop the CPG execution engine we investigated different variants of OWL, each offering a different degree of expressivity and executional capabilities. We have developed three variants of OWL-based CPG execution engines as follows:

- **OWL-DL based CPG execution engine:** CPG execution engine based on OWL-DL [142] (a sub-language of OWL) is the most basic execution engine that offers reasoning services that are sufficient for executing a range of simple CPG constructs, such as: (a) Handling of state transitions for the various CPG tasks being executed, (b) Ordering of medical tasks, (c) Evaluation of conditions of tasks, (d) Responding to outcomes of medical tasks, (e) Handling decisions, (f) Support for the following workflow structures: branch, synchronization, deferred choice, interleaved routing, multiple option decision, split, switch, triggers and while loops, (g) Handling nesting of procedures within CPG. However, this particular CPG execution engine has certain limitations that are inherited from the lack of expressiveness of OWL-DL such as: lack of data type expressivity, Qualified Cardinality Restrictions [151] (QCR) and relational expressivity. This renders the CPG execution engine unable to handle operations on data.

- **OWL 2 based CPG execution engine:** CPG execution engine utilizing OWL 2 RL [152] (a profile of OWL 2) offers additional expressivity to handle (a) user defined data ranges; (b) Some while loops; and (c) more efficient condition handling. OWL 2 [153] offers qualified cardinality restriction handling which we have employed in our OWL 2 based execution engine to check the satisfaction of conditions for the activation of tasks that demand certain conditions to be satisfied before being executed. This execution engine can also handle conditions which entail the satisfaction of user defined ranges to compare numeric values of datatype properties (e.g. *hasAge* property) with predefined values (e.g. 18). In this way, the lack of data type expressivity that was noted in OWL-DL has been partially addressed since OWL 2 still lacks advance data type expressivity capabilities, such as handling of variables and Mathematical/String manipulation functions that are necessary for the execution of other more complex CPG tasks.

- **OWL-DL + SWRL based CPG execution engine:** This engine offers advance data manipulation capabilities that are not available in the earlier CPG execution engines. SWRL extends the expressivity of OWL by including Horn-like rules to the OWL knowledge base, thus providing a range of built-in functions that enable: (a) the computation of mathematical functions (e.g. multiplication) by using the SWRL rules that have been written to support the operators in our expression ontology described in chapter 5 of the thesis, (b) comparison of values of data type properties and (c) the ability to support for the following workflow structures: executional state condition, force to state and for loop. This is the most advance CPG execution engine, whereby SWRL rules can be incorporated within the OWL-DL based execution engine to enhance the expressivity and reasoning of OWL-DL.

In this section, we demonstrate the use of OWL reasoning services to develop the abovementioned CPG execution engines. We will explain the evolution of the CPG execution engines, starting with OWL-DL based execution engine to OWL-DL+SWRL based execution engine, highlighting the rationale and the CPG executional capabilities for each execution engine.

6.2. OWL-Based Execution of Clinical Practice Guidelines

In this section, we present our OWL-based CPG execution engines. The OWL-DL based execution engine is the baseline execution engine which is capable of performing simple CPG executional tasks, whereas for more complex executional capabilities (described later) we use OWL 2 and SWRL to extend the functionalities of our CPG execution engines. For the abovementioned CPG execution engines our CPG execution approach is:

- **Ontology-based CPG Modeling:** We use CPG-DKO described in chapter 4 of the thesis to computerize a paper-based CPG in terms of an OWL-based model that captures the CPG's domain and functional concepts, workflow elements, decisional aspects and patient/medical data.

- **Adding State Transition Rules:** Tasks go through several states during application of a CPG to a patient in the real world. In order to simulate these state transitions in our CPG execution engines, the rules governing these state transitions should be carefully implemented to assure the correctness of the generated recommendations. These rules are added manually to the CPG ontology. These state transition rules are a part of CPG-DKO. Hence, they exist in any instantiation of the CPG ontology and it is not needed to add them again to the instantiations.

- **Pre-processing of CPG Model:** To extend the expressivity of the OWL-based CPG model—i.e. the CPG ontology—to handle high-level executional constructs we preprocess the computerized CPG whereby additional execution-specific constructs are added to the CPG ontology in order to prepare it for execution. Depending on the complexity of the CPG and the executional capabilities sought, different pre-processing steps are performed on the instantiated CPG ontology. This step is performed fully automatically.

- **Execution of CPG:** Our CPG execution strategy is to query the CPG for active tasks with respect to the CPG's workflow and upon completion of a task insert triples that indicate the completion of specific tasks, the data generated and the decisions made by the healthcare professionals. The entire execution is managed and monitored within the CPG ontology using an OWL reasoner.

In the following sections we described our OWL-DL, OWL 2 and SWRL based execution engines.

6.2.1. OWL-DL based CPG Execution Engine

Functionally speaking, a CPG execution engine interprets a task's state, responds to the actions associated with the task, and enacts changes to the task's stage in response to the outcome of task's actions. We have developed the following task state transition for this purpose.

6.2.1.1. State Transition Model in OWL-DL based Execution Engine

To execute an ontologically-modeled CPG using an OWL reasoner, we have developed executional semantics that determine: (a) the current state of a task described in the CPG, and (b) the transition of tasks from one state to another in response to operations/actions applied to a task. To control CPG execution, we have developed a CPG execution model that comprises 5 states as follows:

1. *inactive*: Tasks that are not ready for execution yet. All tasks are in this state before the execution starts.

2. *active*: Tasks that are ready for execution and either will proceed based on available data or will be acted upon by a healthcare professional.

3. *discarded*: Tasks that will not be considered active and cannot be executed. The only way for them to get activated is their repetition through a loop which is not supported in our OWL-DL execution engine.

4. *completed*: Tasks that their execution is completed normally.

5. *started*: Tasks go to this state when they are under execution. For instance, composite tasks that are under execution have this state.

Each one of these states is an instance of the class *TaskState.* To show that a task has one of these specific states, *hasTaskState* property with the domain of *Task* and range of *TaskState* is used. For instance, *":t1 hasTaskState :active*" means that *t1* is an active task in the CPG. The state of a task can be either inferred by the OWL reasoner or can be forced by the execution engine. Tasks that have states *active*, *inactive*, *completed*, *started* and *discarded* are instances of classes *ActiveTask*, *InactiveTask*, *CompletedTask*, *StartedTask* and *DiscardedTask* respectively by the following OWL construct and other similar ones.

```
[a owl:Restriction ;
 owl:onProperty :hasTaskState ;
 owl:hasValue :active]
 owl:equivalentClass :ActiveTask.
```

Instead of using "*[a owl:Restriction; owl:onProperty :hasTaskState; owl:hasValue :active]"* we can simply use *ActiveTask* in the ontology. Please note the sole purpose of this is to make the ontology less verbose and easier to read in the TURTLE syntax. Figure 6.2 shows possible state transitions during execution of a CPG.

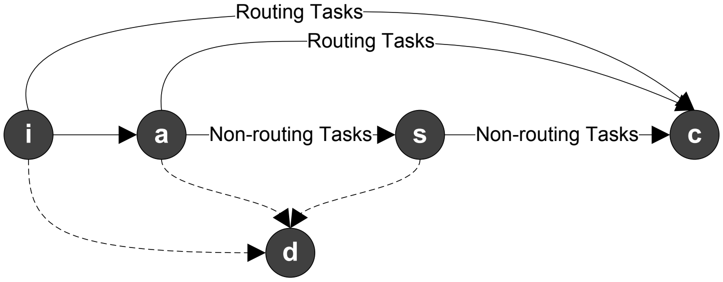


Figure 6.2 State transitions in OWL-DL based execution engine (i = *inactive*, a = *active*, s = *started*, c = *completed*, d = *discarded*). labels of the arcs show the kind of task that may go through the indicated state transition.

State transition rules are manually added to CPG-DKO. We discuss the state transition rules of the *Task* class instances as an example of these state transition rules. Please note that the property *hasTaskStateNewCandidate* property with the domain of *Task* and range of *TaskState* shows the candidate values for the new state of a task. It is execution engines responsibility to select one of the candidates and assign the state to the task.

- Inactive → Active

An inactive non-routing task which includes instances of the *AtomicTask*, *BagOfTasks*, *CompositeTask*, *Cycle*, *Decision* or *Inquiry* classes with satisfied condition criterion goes to active if it is preceded by a completed task or it is the first task of the started composite task or the first task of a bag of tasks. The following OWL axiom implement this state transition:

```
[ a owl:Class ;
 owl:intersectionOf (
     :InactiveTask
     :TaskWithSatisfiedCriterion
     [ a owl:Class ;
       owl:unionOf (
           :AtomicTask :BagOfTasks
             :CompositeTask :Cycle :Decision :Inquiry )]
     [ a owl:Class ;
      owl:unionOf (
     [ a owl:Restriction ;
           owl:onProperty :hasPrevious ;#inverse of hasNext
           owl:someValuesFrom :CompletedTask
     ]
     [ a owl:Restriction ;
        owl:onProperty :isTaskOf;
        owl:someValuesFrom [ a owl:Class ;
        owl:intersectionOf ( :BagOfTasks :StartedTask )]
      ])])]
rdfs:subClassOf
      [ a owl:Restriction ;
       owl:onProperty :hasTaskStateNewCandidate ;
       owl:hasValue :active].
```

Having defined the abovementioned OWL axioms, an OWL reasoner will infer that a task with the above criteria should get activated by asserting the value active for the property *hasTaskStateNewCandidate*.

- Active→ Started

It is execution engine's responsibility to change the state of a task from active to started when the user indicates through the interface that he is in the process of doing the task.

- Started→ Completed

Same as "active → started" state transition, this state transition is initiated by the user when he has completed an *AtomicTask* through the execution engine's interface. However, this state transition is inferred for instances of *CompositeTask* and *BagOfTasks* classes. These tasks will be regarded as completed when all the sub-tasks are completed or discarded.

- All states→ Discarded

An active, started or inactive task goes to the discarded state if it is a sub-task of a discarded step or all the preceding tasks are discarded:

```
[ a owl:Class ;
  owl:intersectionOf (
     [owl:unionOf(:ActiveTask :StartedTask :InactiveTask)]
        [owl:unionOf(
           [ a owl:Restriction ;
            owl:onProperty :isTaskOf ;
            owl:someValuesFrom :DiscardedTask
           ]
           [ a owl:Restriction ;
             owl:onProperty :hasPrevious ;
             owl:allValuesFrom :DiscardedTask
           ]
      )])]
rdfs:subClassOf
      [ a owl:Restriction ;
          owl:onProperty :hasTaskStateNewCandidate ;
          owl:hasValue :discarded
        ] .
```

- Completed or Discarded→ Inactive

Tasks become inactive to be re-executed in the cycles.

6.2.1.2. Pre-processing Phase

To execute an ontologically-modeled CPG using OWL reasoners, we perform a CPG transformation on instantiations of our CPG ontology whereby additional workflow elements—i.e. specific properties, classes, instances and necessary relationships—are augmented to the instantiated CPG ontology (as shown in Figure 6.1). Our transformation is performed automatically and no manual editing of the ontology is needed.

In contrast to adding the state transition rules which is performed manually on the CPG ontology only once, the pre-processing phase should be performed on instantiations of the CPG ontology. CPG transformation involves the following activities:

6.2.1.2.1. Handling the Open World Assumption

Most of the OWL reasoners adopt the open world assumption. This assumption indicates that absence of a statement does not mean it is false and no inference can be made from it.

For example, earlier we described a state transition rule which states that a task is a *CompletedTask* if ***all*** of its sub-tasks are completed. Suppose t1 is a composite task whose sub-tasks are t1-1 and t1-2 and they have the *completed* state. Since OWL adopts the open-world assumption an OWL reasoner will not infer that all of the sub-tasks of t1 are *CompletedTask*s because t1 might have more sub-tasks which are not included in this ontology. We solve this problem by adding the following triples:

```
:t1 a [a owl:Restriction;
    owl:onProperty :hasTask;
    owl:cardinality 2].
```

This triple set clearly indicates that t1 is a member of class of tasks who has exactly two different values for their *hasTask* property. The same process should be performed wherever that *owl:allValuesFrom* is used in our CPG execution engine. Conditions, condition combination points, synchronization points, and composite tasks makes use of *owl:allValuesFrom*.

6.2.1.2.2. Handling the Non-Unique Naming Assumption

This assumption indicates that two different names may refer to the same entity in an ontology. Continuing our previous example, since OWL adopts the non-unique naming assumption the reasoner considers the possibility that t1-1 and t1-2 are different names for the same task. Because of this possibility, completion of these two tasks is not enough for the reasoner to infer that their super-task is completed too. The following triples should be added during the pre-processing to make the reasoner to draw the desired inference.

```
  :t1-1 owl:differentFrom :t1-2.
```

The same process should be performed wherever that *owl:allValuesFrom* is used.

6.2.1.2.3. Handling Conditions

Checking conditions of tasks to determine their eligibility for execution is an important aspect of CPG execution. The condition satisfaction criteria can be "*all*", "*any*" or "*any k*". Criteria for "*all*" and "*any*" are easily handled by using *owl:allValuesFrom* and *owl:someValuesFrom* from OWL. However, to handle "*any k*" out of N we need Qualified Cardinality Restriction (QCR) expressivity which is not supported in OWL-DL. We developed a workaround to this problem by making additional intermediate nodes that go between conditions and tasks. Each of these intermediates nodes represents a combination of k out of N conditions. These intermediate nodes which are instances of class *ConditionCombinationPoint* are added during the pre-processing and the CPG modeler does not need to deal with them. The conditions that are represented by a combination point are indicated by the property *isWaitingForCondition*. Figure 6.3 illustrates the changes that we make for a task with 3 condition and criterion of "*any 2"* out of 3 conditions.

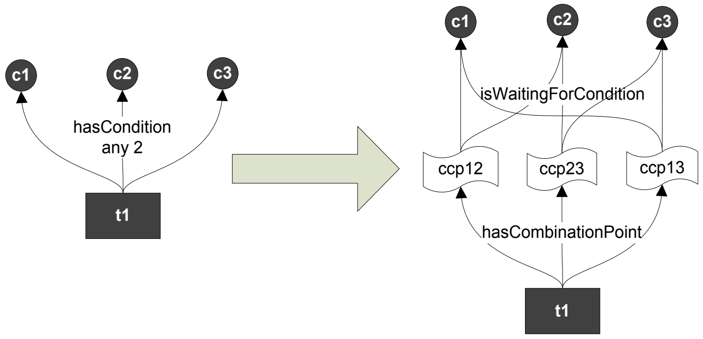


Figure 6.3 The transformation for handling condition satisfaction criteria in absence of QCR for conditional tasks. (*c1*, *c2* and c*3* conditions and *ccp12*, *ccp23* and *ccp12* are instances of *ConditionCombinationPoint*)

6.2.1.3. OWL-Based CPG Execution Algorithm

The CPG ontology pre-processing stage renders the CPG for execution using OWL reasoners. In this section we explain our CPG execution algorithm. This execution algorithm is the specialized version of the execution algorithm shown in Table 3.2. To understand how the execution algorithm works, we go through a small example CPG pictured in Figure 6.4. This small CPG is composed to two atomic tasks. As you can see both of the tasks t1 and t2 are in inactive state in the beginning of the execution. t1 which is the first task of this small CPG has a possible outcome which is the condition for t2. If o1 happens as the result of executing t1, t2 can be executed too otherwise it is discarded.

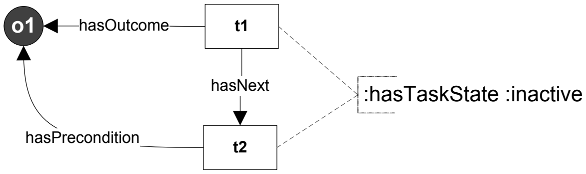


Figure 6.4 A small example CPG used for explaining the execution algorithm

The following steps show how execution engine works:

Step 1: Load the instantiated CPG ontology to the OWL reasoner and activate the first task in the CPG. Figure 6.5 shows the execution states in this step. In order to activate the first task, the previous value of the *hasTaskState* property is deleted and this triple is added to the instantiated ontology "*:t1:hasTaskState :active*".

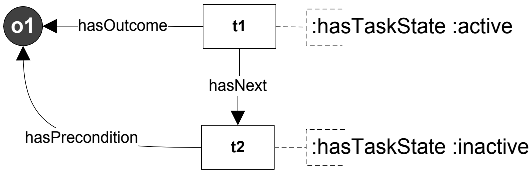


Figure 6.5 Example CPG in step 1 of the execution algorithm

Step 2: Query the ontology for the active tasks and show them to the user. In order to find the active tasks, a query will be performed on the values of the property *hasTaskState*. If the value of the *hasTaskState* property is active the task is returned in this query. The active tasks are shown to the user and he is asked to act upon active tasks. If the query does not return anything execution terminates. In our example, t1 is the only active task. We assume that this task is selected by the user for execution.

Step 3: Record the start of tasks by asserting triples into the CPG ontology. To record the start of task t1 by the user, triple "*:t1 :hasTaskState :started*" is inserted into the ontology. Prior to asserting the new state, the previous state is deleted from the ontology by deleting the triple "*:t1 :hasTaskState :active*". Figure 6.6 shows the execution states in this step.

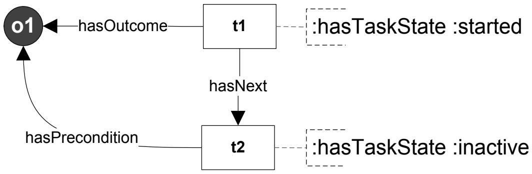


Figure 6.6 Example CPG in step 3 of the execution algorithm

Step 4: Record the completion of tasks and their outcomes. When the user has completed a task, it is time to record completion of that task by inserting the triple "*:t1:hasTaskState :completed*". In order to find the outcomes of the active tasks, the values of the property *hasOutcome* is queried for the completed task and shown as the outcomes. In our example, o1 is shown to the user as the possible outcome of t1 and he is asked if the outcome has actually happened or not. For instance, t1 can be measure_BP and its outcome can be high_BP. In order to show that high_BP is the result of execution of t1, the following triple is added to the instantiated ontology "*:high_BP :hasConditionState :happenedOutcome*". The *HappenedOutcome* is equivalent class of the *SatisfiedCondition* class. If the user indicates that the outcome has not happened, "*:o1 :hasOutcomeState notHappenedOutcome*" will be put inside the ontology. *NotHappenedOutcome* is equivalent class of the *UnSatisfiedCondition* class. The result of indicating that t1 is completed and o1 is a happened outcome is shown in Figure 6.7.

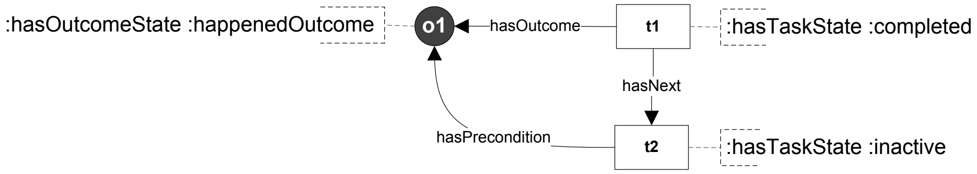


Figure 6.7 Example CPG in step 4 of the execution algorithm

Step 5: Use the execution semantics embedded within the CPG ontology in conjunction with the existing inferred/asserted values to calculate the candidate values for the new states of the tasks. This step is simply performing reasoning on the ontology after the modifications that have been performed on it in the previous steps. In our example, an OWL reasoner will use the current states of the tasks and the OWL triples mentioned in section 6.2.1.1 for implementation of the *inactive* → *active* state transition rule to infer that the task t2 has the value active for the property *hasTaskStateNewCandidate*. The result of this step is shown in Figure 6.8.

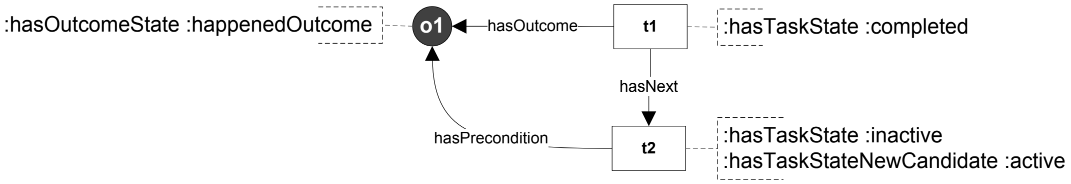


Figure 6.8 Example CPG in step 5 of the execution algorithm

Step 6: Apply the new states of tasks that are inferred in the previous step to them. The values of *hasTaskStateNewCandidate* property are queried and used as the new state of the tasks. For instance, if a task has the value active for this property, the values of the properties *hasTaskState* and *hasTaskStateNewCandidate* are deleted and active is assigned as the value of the property *hasTaskState.* The result of this step is activation of task t2. The result is shown in Figure 6.9.

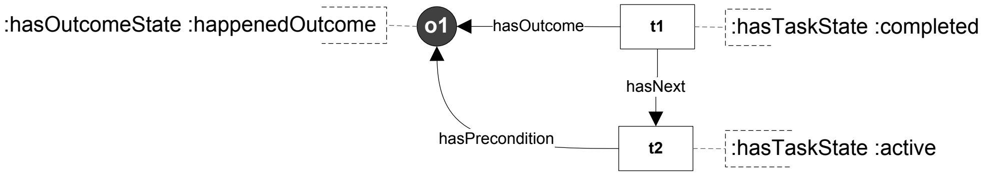


Figure 6.9 Example CPG in step 5 of the execution algorithm

Please note that a task may have several candidates for its new state coming from different transition rules. As an example, a state transition rule may indicate that a task should be discarded because its super task is discarded and another rule may indicate that the task should be active because its previous task is completed. If more than one value exists for the property *hasTaskStateNewCandidate*, a conflict resolution process should be applied because a task cannot be in two different states simultaneously. In our execution engine, we have given the following priorities to the states: discarded > completed > started > active > inactive. For instance, if a task has the values <u>completed</u> and <u>discarded</u> as the value the property *hasTaskStateNewCandidate*, the next state of the task will be the discarded. More complex state transition rules can be implemented if necessary.

Step 6: Go to step 2. In our example, the algorithm runs one more iteration for the execution of the task t2.

The abovementioned execution algorithm shows that the only elements from the CPG ontology that the CPG execution engine needs to deal with are the ones pertaining to tasks (*Task* Class), activation and completion of tasks (<u>started</u>, <u>active</u> instances and *hasTaskState*, *hasTaskStateNewCandidate* properties) and their outcomes (*hasOutcome, hasOutcomeState* properties and *happenedOutcome* and *nothappenedOutcome* instances) and the rest of the ontology (e.g. *:hasNext* property etc.) are used by the reasoner to execute branches, synchronization, state transitions, condition checking, etc.

We have used Jena library [160] to add triples to and query the ontology. We use Pellet [161] as the OWL reasoner. We have used TURTLE syntax [159] to show the OWL constructs that represent our ontologies in this section of the paper.

6.2.2. OWL 2 based CPG Execution Engine

OWL 2 is the new version of OWL that offers two enhancements that are relevant to CPG execution—i.e.(a) modeling of and reasoning on qualified cardinality restriction and (b) limited datatype expressivity that supports user defined ranges. We extended OWL-DL based execution engine by incorporating OWL 2 constructs to offer the following functionalities: **(a)** handling complex conditions **(b)** handling data type expressivity.

OWL 2 based execution algorithm works in the same way as our OWL-DL based execution algorithm. The only difference is in the pre-processing phase. In this phase, the step for handling conditions is performed differently in order to create a more efficient way of executing conditions. There is also an extra step for adding data type expressivity to the instantiated CPG ontology. Every other element of the execution engine such as state transition and execution algorithm are exactly the same as OWL-DL based execution algorithm.

6.2.2.1. Handling Conditions in OWL 2

This section shows how conditions are handled differently in the pre-processing phase of OWL 2 based execution engine. We use the QCR modeling and reasoning capability of OWL 2 to handle "*any k*" condition criteria more effectively. We go through an example to explain how QCR capability of OWL is used in the pre-processing phase for condition handling. Suppose that t is a task which has tree conditions p1, p2 and p3 and its condition satisfaction criterion is "*any 2*". To handle conditions of a task with satisfaction criterion of "*any k*" out of N we create six new OWL classes during the following 4 steps of the pre-processing.

(1) *Task_Requires_k_Satisfied_Conditions* is a class whose members are tasks that need at least *k* satisfied conditions in order to satisfy their criteria. During the pre-processing we add the following triples to the ontology.

```
:t a Task_Requires_2_Satisfied_Conditions.
```

(2) *Task_Has_Atleast_k_Satisfied_Conditions* is a class whose members are tasks that have at least *k* conditions satisfied at that point in time during execution regardless of the number of satisfied conditions that they actually need. For instance, a task which is waiting for *k* conditions and has two satisfied so far is a member of the following class which makes use of OWL 2 QCR capability.

```
Task_Has_atleast_2_Satisfied_Conditions
   [a owl:Restriction;
     owl:onProperty :hasCondition;
     owl:onClass :SatisfiedCondition;
     owl:minqualifiedCardinality 2].
```

To enable the CPG execution engine to perform reasoning on QCR, open world and non-unique naming assumptions should be taken care of.

(3) To determine whether a task has satisfied the requisite number of conditions (which is “any k”), we have created an anonymous class which takes the intersection of the two abovementioned classes– if a task is a member of the class intersection then it can be inferred that it has a satisfied condition criterion.

```
[a owl:Class; owl:intesectionOf(
    :Task_Requires_2_Satisfied_Conditions
    :Task_Has_Atleast_2_Satisfied_Conditions)
] rdfs:subClassOf :TaskWithSatisfiedCriterion.
```

(4) Three other classes are defined in a similar fashion to handle the case where the number of unsatisfied conditions is more than or equal to *N-k+1* (3-2+1 = 2). In this case, the

number of satisfied conditions is less than *k* and the condition criterion of the task is unsatisfied. The following OWL triples show these classes:

```
:t a Task_Requires_1_UnSatisfied_Conditions.
:Task_Has_atleast_1_UnSatisfied_Conditions
   [a owl:Restriction;  owl:onProperty
    :hasCondition; owl:onClass
    :UnSatisfiedCondition; owl:minqualifiedCardinality 1].
[a owl:Class; owl:intesectionOf(
    :Task_Requires_1_UnSatisfied_Conditions
    :Task_Has_atleast_1_UnSatisfied_Conditions
)
] rdfs:subClassOf :TaskWithUnSatisfiedCriterion.
```

6.2.2.2. Data Type Expressivity in OWL 2 Based Execution Engine

We do not necessarily need SWRL rules for mathematical computations and OWL 2 is able to support some level of data type expressivity. Our OWL 2 based execution engine is capable of evaluating conditions such as “patient older than N” that need to compare a numeric value against a data range. To achieve this functionality, we utilized user defined ranges in OWL 2 to perform arithmetic comparison on numeric values of data type properties and use the result in the decision makings in CPG. To evaluate a precondition as stated earlier, we create 5 new OWL classes during CPG pre-processing. We define a class *Condition_Should_be_Greater_Than_N* and set the condition (c1) as its instance during pre-processing:

```
:c1 a :Instance_Should_Have_hasNumericValue_GT_N.
```

Next, we define a class *Condition_Has_GT_N* that makes use of user defined data ranges in OWL 2 as follows:

```
:Condition_is_GreathaerThanEqual_N a owl:Class;
owl:equivalentClass
```

```
[a owl:Restriction ;
owl:onProperty :has_age  ;
owl:someValuesFrom
    [a rdfs:Datatype ;
     owl:onDatatype xsd:int ;
     owl:withRestrictions ( [ xsd:min N+1"^^xsd:int ] ) ]
].
```

The above construct ensures that during CPG execution when the value of the *has_age* property is greater than *N*, c1 will be regarded as an instance of *Condition_is_GreaterThan_N* by the OWL reasoner. If a data value should belong to a specific range and it belongs to a class of instances which have that specific range we can reason that the condition regarding the data value is satisfied. Finally, we create an anonymous class which is the intersection of *Condition_Should_be_Greater_Than_N* and *Condition_is_GreaterThan_N* and set it as a subclass of *SatisfiedCondition*:

```
 [ a owl:intersectionOf(
:Condition_Should_be_Greater_Than_N
:Condition_is_GreaterThan_N)
]
rdfs:subClassOf :SatisfiedCondition.
```

Conditions are considered as *UnsatisfiedPrecondition*s when the age is less than equal *N* in a similar fashion:

```
:Condition_is_LessThanEqual_N a owl:Class;
owl:equivalentClass
[a owl:Restriction ;
owl:onProperty :has_age  ;
owl:someValuesFrom
    [a rdfs:Datatype ;
     owl:onDatatype xsd:int ;
     owl:withRestrictions ( [ xsd:max N"^^xsd:int ] ) ]
].

[ a owl:intersectionOf(
:Condition_Should_be_Greater_Than_N
:Condition_is_LessThanEqual_N)
]
rdfs:subClassOf :SatisfiedCondition.
```

6.2.3. SWRL based CPG Execution Engine

OWL 2 has a limited support for mathematical functions and comparison of numerical values. For instance, it cannot compare values of two datatype properties or perform mathematical operations. To address these shortcomings, we have developed a CPG execution engine that leverages SWRL to enhance the functionalities of the OWL-DL based CPG execution engine. The advantage of using SWRL is that it offers a set of useful built-in functions. These built-in functions have been used to support mathematical, string and Boolean operators, executional state conditions and for-loops. As we described our expression ontology in chapter 5, we discussed the SWRL rules that have been written to support the mathematical, string and Boolean operators. Using an OWL reasoner that supports SWRL rules enables us to make use of all the capabilities of our expression ontology in the execution engine. The details of how For-loops and executional state condition are supported using SWRL rules are described in this section.

Pre-processing phase and the execution algorithm of the SWRL based CPG execution engine are exactly the same as OWL-DL. The difference is that a number of extra SWRL rules have been manually added to CPG-DKO. As a result, any instantiation of the CPG ontology will contain these rules and it is not needed to add them manually again. These rules are fired in the OWL reasoner that supports SWRL rules during execution in order to enhance the expressivity of OWL-DL. To account for decidability issues that may arise as a result of using SWRL we restricted the SWRL rules to DL-Safe ones [162]. We illustrate below the use of SWRL to address the following CPG execution issues:

6.2.3.1. State Transition Model in SWRL based Execution Engine

6.2.3.1.1. ForLoop

- Inactive→ Active

The same state transition rule described for activation of *Task* class instances is applied.

- Active→ Started

A *ForLoop* task can get activated through the execution engine by a user. The first inactive subtasks get activated when the *ForLoop* is in started state. Activation of the first task is the first step towards repetition of the subtasks of the for-loop. These tasks should be repeated until we reach the maximum number of allowed iterations. A simple solution would be adding 1 to the value of a datatype property when, let's say, the last task of the loop is completed. However, this rule is not DL-safe as it uses the new value of the data type property to fire the rule infinitely. We have implemented a DL-Safe solution by defining two data type properties and using *swrl:add* built-in function. *hasItrNum* that holds the iteration number and is set to 0 at the beginning. *hasItrNumCopy* holds a copy of the *hasItrNum*'s value from the previous iteration. The loop handling rules operate as follows:

(i) When the first task of a loops is completed, value of *itrNum* is copied to *itrNumCopy*:

```
Loop(?l1)^ StartedTask(?l1)^hasFirstTask(?l1,?ft)^
hasState(?ft,completed)^ hasItrNum(?l1,?n)  →
hasItrNumCopy(?l1,?n)
```

(ii) When the last task of the loop is completed 1 is added to *itrNumCopy* and the result is put in *itrNum*:

```
Loop(?l1) ^ Loop(?l1) ^ hasLastTask(?l1,?lt) ^
hasState(?lt,completed) ^ hasItrNumCopy(?l1,?n) ^
swrlb:add(?nPlus1,?n,1)→ hasItrNum (?l1,?nPlus1)
```

- Started→ Completed

This state transition happen when the value of *hasItrNum* reaches the maximum number of iterations:

```
Loop(?l1) hasMaxRepetitionNumber(?l1,?max) ^ hasItrNum(?l1,?n) ^
swrlb:greaterThanOrEqual(?n,?max)→
hasTaskStateNewCandidate(?l1,completed)
```

- All states→ Discarded

The same state transition rule described in section 6.2.1.1 will apply to this task.

6.2.3.1.2. Executional State Conditions

As we discussed in section 4.4 of the thesis, the state of tasks can be used as conditions for other tasks. Implementation of these conditions is done using SWRL rules. A *ExecutionalStateCondition* is a satisfied condition if the state of the task identified by the property *isChekingExecutionalStateOfTask* is equal to the value of the property *shouldHaveExecStateForSatisfaction*:

```
ExecutionalStateCondition(?c) ^ hasTaskState(?t, ?s1) ^
isChekingExecutionalStateOfTask(?c, ?t) ^
shouldHaveExecStateForSatisfaction(?c, ?s1) ->
conditionHasStateNew(?c, satisfied)
```

The property *conditionHasStateNew* will be used by the execution engine to find the new state of the executional state condition. A *ExecutionalStateCondition* is an unsatisfied condition if state of the task identified by the property *isChekingExecutionalStateOfTask* is not equal to the value of the property *shouldHaveExecStateForSatisfaction*:

```
ExecutionalStateCondition(?c) ^ UndefinedCondition(?c) ^
hasTaskState(?t, ?s2), isBiggerThanStateBO(?s2, ?s1)  ^
isChekingExecutionalStateOfTask(?c, ?t) ^
shouldHaveExecStateForSatisfaction(?c, ?s1) ->
conditionHasStateNew(?c, unsatisfied)
```

6.3. Conclusion

In this section, we developed three OWL based CPG execution engines based on OWL-DL, OWL 2 and SWRL. OWL-DL CPG execution engine was capable of execution of all the workflow patterns identified in section 4.3. OWL 2 qualified cardinality restrictions and user defined data ranges that are not available in OWL-DL were needed to handle conditions more effectively and perform numeric comparison respectively. In order to achieve more expressivity such as rules and mathematical calculations, the OWL-DL CPG execution engine was needed to be augmented with SWRL rules. Table 6.1 compares the executional capabilities of these three OWL based CPG execution engines.

Table 6.1 Capabilities of our CPG execution engines (±: somewhat supported, +: supported, -: not supported).

	OWL-DL	OWL 2	SWRL
Workflow Execution	+	+	+
QCR	-	+	+
Datatype Expressivity	-	±	+
Rules	-	-	+
Reasoning	+	+	+
Loops	-	±	+

Due to the fact that execution of a great majority of CPG needs datatype expressivity and execution of loops, OWL-DL + SWRL based CPG execution engine is the most practical solution.

The unique aspect of our research is that we presented a CPG execution framework that utilizes a single formalism—i.e. OWL—to both represent and execute CPG. Our approach has several advantages over the existing graph-based CPG execution engines:

(1) Ease of switching to new technologies: In our method, after CPG undergo the preprocessing phase they can be executed by any OWL reasoner and any API. For instance, it is easily possible to execute the CPG using OWL API and Hermit reasoner or any other OWL reasoner and API that will be developed in future even in different programming languages.

(2) Increased shareability and flexibility: To execute the preprocessed CPG there is a need for a very simple program that performs queries on and puts new triples into the ontology and there is no need to be familiar with all the elements of the CPG ontology or to know

any of the execution semantics at all. In this way, the CPG is untied from a special CPG execution engine. This increases shareability of the computerized CPG and also gives the flexibility to the developers to implements their desired behavior in the execution engine programs. For instance, the CPG execution engine developer can make the user to indicate the possible outcomes immediately after the completion of a task or give the user to option to enter the outcomes at a later point in time when the outcome are ready.

(3) Reusing existing reasoners instead of writing a new one: Developing a graph-parsing CPG execution engine involves writing a small reasoner which performs reasoning on the execution semantics of the used knowledge representation language. Using OWL enables us to use several existing reasoners instead of developing one.

Below we list disadvantages of our approach compared to graph-parsing approaches:

(1) Low efficiency: OWL reasoners are general-purpose reasoners whereas graph-parsing algorithms written for execution of a CPG language are problem-specific. These graph-parsing algorithms only perform the reasoning tasks that are necessary for CPG execution and avoid unnecessary reasoning activities such as classification that is performed in the OWL reasoners regardless of its usefulness in CPG execution.

(2) Lack of expressivity for temporal reasoning and abstraction: For instance, if it is needed to detect a temporal pattern such as "*existence of two gout attacks in the last three months that each lasted at least 2 days*" in a patient's health record, OWL and SWRL are not expressive enough for this task.

(3) Debugging ontologies are more difficult as opposed to programming languages. Therefore, OWL based CPG execution engines are more prone to have errors and those errors are more difficult to be detected and fixed.

CHAPTER 7: KNOWLEDGE MAPPING ONTOLOGY (KMO)

7.1. Introduction

As we discussed previously, knowledge artifacts used in a DSS may be represented in heterogeneous Local Knowledge Ontologies (LKO). Our solution is to map all LKO to a Domain Knowledge Ontology (DKO) in order to unify the representation of LKO for merging and execution. In order to merge and then execute LKO they all are needed to be mapped and have their instances to DKO. In a CPG merging framework, LKO contain ontologically modeled CPG and DKO is CPG-DKO. In order to merge and then simultaneously execute CPG to create therapy plan for a comorbid patient, these comorbidity CPG should be transformed to a unified representation as no execution engine is capable of executing heterogeneously represented ontologies. For instance, if we have a cough CPG modeled in GASTON and a Chronic Heart Failure CPG in COMET [107], there is no way of simultaneous execution of these CPG in a single execution engine. Our solution to this problem is to map and then transform all the participating CPG to the developed CPG-DKO in chapter 4 of the thesis.

We are interested in using semantic-based ontology mapping techniques as our assumption is that the CPG that are being merged are in OWL language which has well defined semantics and it is accompanied by several reasoning engines with practical reasoning algorithms. Anchoring two ontologies needs an expressive mapping representation language. This mapping will be used by the domain expert or the mapping discovery algorithm to find the initial mappings. As we reviewed the mapping representation languages and the papers that compare these languages in section 2.2, we came to the conclusion these languages suffer from lack of expressivity. They are mostly designed to be the output of the mapping discovery algorithms which are not very powerful and as a result are not capable of discovering complex mappings. Since ontologies designed to model CPG are very complex and make heavy use of OWL constructs, the existing ontology mapping representation languages are not expressive enough to capture the mappings between these ontologies for the anchoring step. We feel the need for a more expressive mapping representation language that can be used by a domain expert to manually map the CPG related to comorbidities to CPG-DKO.

Recently OWL (a sub-language of DL) has been accepted by the semantic web community as the standard language for representation of ontologies and several tools also have been developed to support manipulation, querying, visualization and reasoning on OWL ontologies. This makes us believe that description logic is the natural candidate to be used in semantic based ontology mappings. Careful review of the semantic based mapping literature reveals that DL has been widely used for debugging and deriving simple mappings (class equivalence, etc.) [54][55][56][57] but no attempt has been made to create more complex mappings. We believe that DL's potentials have not been carefully investigated for representation of the mappings and reasoning on them and further research is needed.

We have developed a new expressive ontology mapping representation language for two reasons: (1) Lack of expressivity in the existing ontology mapping representation languages; (2) Lack of formal semantics to use the mapping representation languages in semantic-based ontology mapping approaches. In section 2.3 of the thesis we reviewed the literature pertaining to ontology mapping representation languages and created a list of important features those mapping representation languages should have. Our mapping representation language which is in OWL-Full supports most of the identified requirements in that list. We call this ontology the Knowledge Mapping Ontology. We have also created a translation algorithm that translates the mapping to OWL-DL + SWRL in order to define the mapping semantics formally in such a way that reasoners can perform the mapping and instance transformation. Using OWL-Full for mapping representation and translating them to OWL-DL + SWRL makes the mappings more expressive and less verbose—i.e. less OWL triples are needed for representation of the mappings.

Our approach has three unique aspects: (1) Ontologies and the mapping are modeled in a unified format. This increases the shareability of the mappings; (2) As we will see in the comparison section our mapping representation algorithm is one of the most expressive existing ontology mapping languages which support a wide range of mapping patterns; (3) As we discussed no description logic based approach which formally defines the mapping semantics exists that can be used for mapping and instance transformation. Therefore, our approach is the first approach that fully defines the semantics of our mapping in such a way that OWL reasoners can handle the mapping and transformation. We believe that these three features make our approach an advancement in the field of semantic-based ontology mapping.

7.2. Ontology Mapping Process

In order to map to ontologies successfully and then transform instances of the source ontology to the target ontology using our approach several steps are needed to be taken:

1. Importing: The two mapped ontologies are imported by our mapping ontology so that the instances, properties and classes of the mapped ontologies are available in the mapping ontology.

2. Mapping (Anchoring): The two ontologies are mapped by manually instantiating our mapping ontology by a domain expert. The instantiation of the mapping ontology is performed by creating semantic relations between the instances of the mapped ontology using the ontology mapping classes, instances and properties. In the remainder of this section the prefixes “source:”, “target:” and “mo:” represent source, target and the mapping ontologies respectively. In the following example, the mapping ontology expresses that classes *Human* from the source ontology is a subclass of the *Person* in the target ontology:

```
mo:m a mo:ClassMapping;
     mo:hasSourceClass source:Human;
     mo:hasTargetClass target:Person;
```

```
    mo:hasRelation mo:subClassRelation.
```

Please note that the “mo:”, “source:” and “target:” are the namespaces for the mapping ontology, the source and the target ontologies respectively. Therefore, *ClassMapping* OWL class, subClassRelation instance and *hasSourceClass*, *hasTargetClass* and *hasRelation* object properties belongs to the mapping ontology. *Human* and *Person* classes belong to the source and target ontologies respectively. Therefore, the output of the mapping process is the two mapped ontology and an instantiation of the mapping ontology. Our mapping ontology is described in section 7.3 of the thesis.

Since CPG demand high level expressivity to be computerized, the designed CPG representation languages are very expressive in terms of workflow constructs and the medical knowledge. The more expressive two ontologies are the more complex the mappings between them. Due to the complexity of the mappings that exist between CPG representation languages, we believe that automatic mapping discovery algorithms are not capable of performing an acceptable job in finding the mappings between the CPG that are being merged and CPG-DKO. Therefore, we do not make use of automatic discovering algorithms and perform the anchoring step manually.

3. Translation to OWL-DL + SWRL: We transform the instantiation of the mapping ontology to a combination of OWL-DL + SWRL in order to formally define the mapping semantics in a computer understandable format. Formal semantics of the mapping are defined in OWL-DL + SWRL because: (1) Our mapping ontology is in OWL-Full because we do not respect the class-instance-property separation that is needed to keep the ontologies in OWL-DL. For instance, in our previous example *Human* and *Person* OWL classes are treated as instances by being used as the values of the object properties *hasSourceClass* and *hasTargetClass*. OWL-Full is undecidable and it is not possible to perform reasoning on it for mapping and instance transformation purposes. Therefore, we translate it to OWL-DL which is a decidable specie of OWL to make the reasoning possible; (2) As we will discuss later OWL-DL is not expressive enough for our mapping purposes and our translation algorithm will add the necessary SWRL rules automatically to prepare the mappings for reasoning. To avoid the possible undecidability as the result of using SWRL rules, only DL-Safe rules [162] are added in the translation process. Our translation algorithm has been discussed in section 7.4. As an example, the mapping instantiation described above will be translated from OWL-Full to the following OWL-DL triple. Please note that SWRL rules are not needed for this mapping.

```
source:Human owl:subClassOf target:Person.
```

4. Reasoning: We use OWL reasoners to perform reasoning on the translated mapping for discovering new mappings and performing instance transformation. As an example of instance transformation, if john is an instance of the class *source:Human* the reasoning will infer that john is also an instance of the class *target:Person*. As an instance for discovering new mappings, if the class *source*:*MaleHuman* is a subclass of the *source*:*Human* class the reasoner will also infer that the *source*:*MaleHuman* class is a sub-class of the *taget:Person*. As you can see, this inference is new and has not been expressed in the instantiation of the mapping ontology. We have used pellet as our reasoner since it supports both OWL and SWRL.

7.3. Knowledge Mapping Ontology (KMO)

In this section of the thesis we describe our OWL-Full mapping ontology. While designing this ontology, we included all the aspects listed in section 2.3.1 of the thesis in which we reviewed the existing representation languages and the requirements identified in the literature. While our goal is to meet all of the identified requirements, we have specially focused on expressivity needs of the language.

7.3.1. Mappings and Relations

Class *Mapping* is used to map classes, instances and properties between ontologies. The attributes of the mappings are assigned to a mapping using the following properties which all have the class *Mapping* as their domain: (1) *mappingHasSource*: This property indicates the source of the mapping. Since any element of ontology can be used as the source of a mapping the range of this property is *owl:Thing*; (2) *mappingHasTarget*: this property the target of the mapping and the range of this property is *owl:Thing*; (3) *mappingHasFunction*: Several functions may be utilized to indicate how data transformation and condition evaluation should be performed in a mapping. This property with the range *Function* indicates what functions are in a mapping. The *Function* class is elaborated in section 5.3.2; (4) *hasCondition*: same as tasks in CPG-DKO, mappings can be conditional. This property with the range *Condition* shows the conditions that should be satisfied in order to execute the corresponding mapping.

We have identified three types of mapping in the literature and created three subclasses of the *Mapping* class in order to model them in our ontology: (1) *RelationalMapping*, (2) *TransformationMapping* and (3) *ValueTransferMapping*.

(1) Relational mapping: These mappings create a relation between two or more elements of the source and the target ontology. As a simple example, a relational mapping may express that a class in the source ontology is a subclass of a class in the target ontology. Depending if the source and the target ontologies are treated as instances, properties or classes one of the class instance, property or class mapping relations may be used respectively. The property *mappingHasRelation* with the domain *RelationalMapping* and the range *MappingRelation* is used to define the relation in a relational mapping. *MappingRelation* Class and its subclasses and their instances can be seen in Table 7.1.

Table 7.1 MappingRelation class, its subclasses and their instances

Subclasses of MappingRelation	**Instances**
ClassRelation	classRelationEquivalentClass classRelationDisjointWith classRelationSubClass classRelationSuperClass
PropertyRelation	propertyRelationEquivalentProperty propertyRelationDisjointWith propertyRelationSubProperty propertyRelationSuperProperty
InstanceRelation	instanceIsMemberOf instanceDifferentFrom instanceSameAs

The classRelationEquivalentClass, classRelationDisjointWith, classRelationSubClass, classRelationSuperClass instances of the *ClassRelation* can be used to indicate that the source class of the mapping is equivalent to, disjoint with, sub class of or super-class of the target class. The propertyRelationEquivalentProperty, propertyRelationDisjointWith, propertyRelationSubProperty, propertyRelationSuperProperty instances of the *PropertyRelation* class can be used to indicate that the source property of the mapping is equivalent to, disjoint with, sub property of or super-property of the target property.

Our mapping language is also capable of expressing relations between instances of the source or the target ontology. instanceDifferentFrom and instanceSameAs instances of the *InstanceRelation* class express that the source and the target ontologies are different or the same instances respectively. instanceIsMemberOf relation means that a specific instance in the source ontology is an instance of a specific class in the target ontology.

As an example the following instantiation of KMO expresses that the class *ECG_finding* in the source ontology is a subclass of the *Diagnostic_test_result* in the target ontology.

```
mo:m1 a :RelationalMapping;
     mo:mappingHasSource source:ECG_finding;
     mo:mappingHasTarget target:Diagnostic_test_result;
     mo:mappingHasRelation mo:classRelationSubClass.
```

(2) Transformation mapping: In this mapping, a transformation process is utilized in order to transform the sources of the mapping to the targets of the mapping rather than just by establishing a mapping relation between them. This mapping is represented by *TransformationMapping* class. The sub-category of the transformation mapping we have come across is property to class and class to property transformation. The class *PropertyToClassTransformation* and *ClassToPropertyTransformation* represents these mappings.

- ***PropertyToClassTransformation*:**

In this relation, two instances and the property that is connecting them is transformed to a class that has those two instances as values of two specific properties. For instance, ":patient1 :is_Taking_Medication :esmolol_Brevibloc" is transformed to an instance of the class Patient_Medication_Entry in the target ontology in the following format:

```
:patient1_Medication_Entry1 a : Medication_Entry;
:medication_Name :esmolol_Brevibloc;
:patiet_ID :patient1.
```

As you can see the two instances patient1 and esmolol_Brevibloc that are connected by the property *is_Taking_Medication* are transformed to an instance of the *Medication_Entry* class and are connect to it by the properties *patiet_ID* and *medication_Name* properties in the target ontology.

The following properties with the domain *PropertyToClassTransformation* are used to define the details of property to class mapping: (1) *propertyToClassTranHasSourceProperty*: The value of this property which is also a property indicates the source property that is being transformed to a class in the target ontology. Range of this property is *owl:ObjectProperty*; (2) *propertyToClassTransformationHasTargetClass* property is used to indicate what class the property is being transformed to; (3)(4) *propertyToClassTransHasFirstTargetProperty* and *propertyToClassTransHasSecondTargetProperty* indicate the properties in the target ontology which have the target class as their domain and the subject and object of the transformed source property as their values.

- ***ClassToPropertyTransformation*:**

The other type of transformation mapping is the opposite of the property to class transformation. In *ClassToPropertyTransformation* mapping, an instance of a specific class, which is connected to two other instances using two specific object properties are transformed to relation between those two instances using a specific property in the target ontology. As an example the instance of the *Medication_Entry* class (patient1_Medication_Entry1) that is connected to instances esmolol_Brevibloc and patient1 using properties *medication_Name* and *patient_ID* is transformed to ":patient1 :is_Taking_Medication :esmolol_Brevibloc" as a result of a *classToPropertyMapping*.

```
:patient1_Medication_Entry1 a :Medication_Entry;
 :medication_Name :esmolol_Brevibloc;
 :patient_ID :patient1.
```

Four properties with the domain of *ClassToPropertyTransformation* are used to specify the details of the mapping. *classToPropertyTransformationHasSourceClass* with the range *owl:Class* shows the source class. *classToPropertyTransformationHasSourceFirstProperty* and *classToPropertyTransformationHasSourceSecondProperty* with the range *owl:ObjectProperty* show the first and the second source properties respectively and *classToPropertyTransformationHasTargetProperty* with the same range shows the target property. To create a class to property transformation mapping, the mapping ontology should be instantiated in the following way:

```
:m a : ClassToPropertyTransformation;
 :propertyToClassTranHasTargetProperty :is_Taking_Medication;
 :propertyToClassTransformationHasSourceClass :Medication_Entry;
 :propertyToClassTransHasFirstSourceProperty : patient_ID;
 :propertyToClassTransHasSecondSourceProperty :medication_Name.
```

(3) Value transfer mapping: In this mapping, no relation exists or no transformation on the structure of the ontology element is needed to happen. These mappings represent a value transfer from the source to the target ontology. This mapping is represented by the *ValueTransferMapping* class. Several functions can be used to manipulate the transferred value. No special property is needed to be defined for this class. The only difference with the relational mapping is that no relation exists and the mapping is making use of a data variable which does not have a value for *variableHasValue* property. These mapping will be translated to SWRL rules for execution. An example of this mapping will be computing the Body Mass Index of a person and transferring it to the target ontology and assign it as the value of the property *hasBMI*:

```
source:john source:has_Weight_KG "82.2"^^xsd:float;
         source:has_Height_CM "187.0"^^xsd:float.
```

will be transferred to

```
source:john target:has_BMI "23.5"^^xsd:float.
```

In order to show how this transformation is executed in our mapping ontology we first need to explain the concepts of variables, operators and functions. We will review an example in section 7.4.4.3 after covering the necessary background.

7.3.2. Variables

Having variables in a mapping representation language can increase the expressiveness significantly by representing a fragment of the ontology rather than a specific class, property or instances. However, variables cannot be defined in most of the mapping representation languages. For instance, a class variable may represent all the patients who have at least two or more symptoms of a specific disease. In the expression ontology in chapter 5 of the ontology we defined a *Variable* class with three subclasses *NumericVariable*, *StringVariable* and *BooleanVariables* which can be used for data values. We also need variables For instance, classes and properties. These variables are represented by *ClassVariable*, *InstanceDataVariable* and *PropertyVariable* classes respectively in our mapping ontology. An instance of the *ClassVariable* represents a class variable. Class variables are basically a set of instances that may belong to a specific class and can have a restriction on a specific property. For instance, a class variable may represent Atrial Fibrillation patients who have at least 2 risk factors of bleeding. Therefore, our class variable is representing instances that belong to the class *AF_Patient* and have two different values for their *has_Bleeding_Risk_Factor* property. Restrictions can be put on the value of a property, type of the value of the property or the number of different values that the property can have. Class variables can be used as an input or output of a function or source or target of a mapping.

In order to indicate what class the instances represented by the class variable should belong to, the property *classVariableHasClass* with the domain of *ClassVariable* and range of *owl:Class* has been defined. For instance, the following example shows a class of instances which is equal to the *AF_Patient* class.

```
:cv1 a :ClassVariable;
   :classVariableHasClass :AF_Patient;
```

The class represented by <u>cv1</u> is an equivalent to the *Student* class because the instances belonging to the class represented by <u>cv1</u> are all members of the Student class and vice versa. However, if we put a restrictions on the number of different values that instances of this class variable can have on the property *hasSummerCourse*, cv1 will be representing a subclass of the *Student* class. Please note that use of *classVariableHasClass* property is not obligatory. Therefore, if no value is assigned by this property, the instances that belong to this class variable can belong to any class as long as they satisfy the property constraints.

Regardless of whether a value exists for the *classVariableHasClass* property or not we can use the property *classVariableHasProperty* with the domain of *ClassVariable* and range of *owl:ObjectProperty* to indicate what property is restricted in this class variable. Please note that making use of this property is optional and it is only used to indicate what property the restriction is put on.

To indicate that an instance variable belongs to a specific class, the property *classPropertyHasValueRestrictionHasInstanceVariable* with the domain of *ClassVariable* and range of *InstanceVariable* is used. This property makes the restriction that the instance variable represents an instance which belongs to the class represented by the corresponding *ClassVariable*. For instance, the following example shows an instance variable (patientInstVar) which belongs to the class *Student*.

```
:cv1 a :ClassVariable;
  :classPropertyHasValueRestrictionHasInstanceVariable
        :patientInstVar;
  :classVariableHasClass :Patient.
```

To put a restriction on the properties of class variables one of the two subclasses of the class variable described in sections 7.3.2.1 and 7.3.2.2 should be used.

Previously, we explained that *variableHasNumericValue*, *variableHasBooleanValue* and *variableHasStringValue* properties are used to assign values to data variables. We use *variableHasClassValue*, *variableHasInstanceValue*, *variableHasPropertyValue* to assign values to instances of the *ClassVariable*, *InstanceVariable* and *PropertyVariable* respectively. Table 7.2 shows these properties which are sub properties *variableHasValue* property along with their domain and ranges.

Table 7.2 Properties used to assign values to ClassVariable, InstanceVariable and PropertyVariable instances along with domains and ranges.

Property	**Domain**	**Range**
variableHasClassValue	*ClassVariable*	*owl:Class*
variableHasInstanceValue	*InstanceVariable*	*owl:Thing*
VariableHasPropertyValue	*PropertyVariable*	*owl:ObjectProperty*

When a value is assigned to variables using one of the abovementioned properties, it will be replaced by that value wherever it is used. For instance, if the value of a cv1 is

```
:cv1 :variableHasClassValue
   [a owl:Restriction;
    owl:onProperty :has_Bleeding_Risk_Factor;
    owl:hasValue :currently_Smokes].
```

cv1 will be replaced by this value during translation to OWL and SWRL axioms. This translation process is described in section 7.4.1.

7.3.2.1. ClassPropertyHasValueRestriction class

ClassPropertyHasValueRestriction is a subclass of the *ClassVariable*. In this subclass of the *VaribleClass* a restriction is put on the value of the restricted property. To restrict a datatype property to a specific value or an object property to a specific instance, properties *classPropertyHasValueRestrictionHasValue* and *classPropertyHasValueRestrictionHasInstance* are used respectively.

7.3.2.2. ClassPropertyQualifiedCardinalityRestriction class

ClassPropertyQualifiedCardinalityRestriction is another subclass of the *ClassVariable* which can be used to put a restriction on either type or the number of values that a specific property can have in a class variable. *hasCardinalityType* with the range of *Cardinality* shows the type of cardinality restriction on the restricted property. The instances of the *Cardinality* class are any, all, min, and max. *variableHasNumericValueForCardinality* data type property indicates what the cardinality number is and *classPropertyQCROnClass* with the range *owl:Class* property indicates what class the values of the restricted property should belong to. For instance the following example shows an existential (any) restriction on the type of the values of the *warfarin_is_contraindicated_due_to* property:

```
:cv1 a :ClassPropertyHasValueRestriction;
    :classVariableHasProperty :warfarin_is_contraindicated_due_to;
    :classPropertyQCROnClass :Contraindication_to_warfarin
    :hasCardinalityType :any.
```

This class variable represents instances that have at least one contraindication to Warfarin. This class variable can, for instance, be mapped to class *Patients_whith_no_Warfatin_Tolerance*.

As another instance, the following example shows a class of instances that has at least two values for the *warfarin_is_contraindicated_due_to* property (a cardinality restriction):

```
:cv1 a :ClassPropertyHasValueRestriction;
:classVariableHasProperty :warfarin_is_contraindicated_due_to;
:variableHasNumericValueForCardinality “2”^^xsd:int;
:hasCardinalityType :min.// or any
```

This class variable can be mapped to *Patient_With_Two_Constraindications_to_Warfarin*. Besides the number of values of a property, the type of those values can be restricted in order to create a Qualified Cardinality Restriction.

7.3.3. Functions and Operators

We have augmented the expression ontology that we developed in chapter 5 of the thesis to be used in our mapping ontology. *Function* class is an entity that can be used for computation in our mapping methodology. Each function accepts an operator, one or two inputs and generates an output. We have expanded the operator class of the expression ontology by several new operators to manipulate classes, properties and instances. These new classes of operators are *SetOperator*, *PropertyOperator*, *ConvertOperator*, *ClassComparatorOperator* and *CreateOperator*. Object property *functionHasOperator* with the domain *Function* and range of *Operator* can be used to assign an operator to a function. Regardless of the operator type, the output of the function is assigned to it by the *functionHasOutputVariable* with the domain of *Function* and range of *Variable*. Depending on the type of the operator used in a function, it can be categorized as on the subclasses of the *Function* Class:

(1) CreateFunction:

This function which makes use of one of the instances of the *CreateOperatorClass* is used for creating new elements in the target ontology during the mapping. Depending on the type of create operator the necessary input variables are assigned to each type of the function using special properties. We will discuss these properties when we review the instances of the *CreateOperator* class in the remainder of this section.

(2) ConvertFunction:

These functions make use *ConvertOperator* instances as their operators to create new instances in the target ontology. How the input of these functions are assigned to them are discussed when each instance of the *ConvertOperator* is reviewed in the remainder of this section

(3) ExpressionFunction:

All other operators are used by instances of the *ExpressionFunction* Class. We call them expression functions because they all have the same style of accepting inputs as the functions in our expression ontology discussed previously. In this section of the thesis, we expand the set of operators defined in chapter 5 and introduce a new set of operators which are used for ontology mapping purposes. Properties *functionHasInputVariable1*and *functionHasInputVariable2* which have *Function* class as their domain and *Variable* as their range are used to assign two input variables to these functions. For instance, in order to define a function which multiplies two input variables aVar and bVar and puts the result in variable cVar our mapping ontology will be instantiated in the following way:

```
mo:divideFunc1 a mo:ExpressionFunction;
mo:functionHasOperator mo:multiplyMO;
      mo:functionHasInputVariable1 mo:aVar;
      mo:functionHasInputVariable2 mo:bVar;
      mo:functionHasOutputVariable mo:cVar.
```

Please note that multiplyMO operator was introduced in the expression ontology. The instances of the new operators, the type of the function that will be using them and output type of those functions are listed in Table 7.3.

Table 7.3 Instances of the *SetOperator*, *PropertyOperator*, *ConvertOperator*, *ClassComparatorOperator* and *CreateOperator* classes, the functions that makes use of these variable and their output types

Operator Class	**Instance**	**Function**	**Function Output**
SetOperator	unionSO intersectionSO	*ExpressionFunction*	*ClassVariable*
PropertyOperator	chainPO	*ExpressionFunction*	*PropertyVariable*
ConvertOperator	convertToInstanceTO convertToClassTO convertToPropertyTO	*ConvertFunction*	*InstanceVariable* *ClassVariable* *PropertyVariable*
ClassComparatorOperator	equivalanceCO subclassCO superClassCO disjointCO complementCO	*ExpressionFunction*	*BooleanVariable*
CreateOperator	classCreateOperator instanceCreateOperator propertyCreateOperator	*CreateFunction*	*ClassVariable* *InstanceVariable* *PropertyVariable*

In Appnedix D, we review the new introduced operators in KMO, their purposes and the details of using them in a function along with the properties used for assigning inputs to them:

7.3.4. Meta Data

Euzenat and his colleagues [164] have created a comprehensive list of annotation properties for ontology mappings. We use their list of annotation properties and make some minor modifications to support modeling meta-data in our mapping ontology. Table 7.4 shows the properties that have been created for capturing meta-data, their ranges and purposes. The domain of all these properties is the class *Mapping*.

Table 7.4 Properties used for Meta data modeling in KMO

Property	Range	Purpose
hasCreator	xsd:string	Indicates the person who is responsible for creating the mapping
hasDate	xsd:date	Shows the creation date of the mapping
hasPurpose	xsd:string	Shows the purpose of the mapping
createdByAlgorithm	xsd:string	Shows what algorithm has been used to create the mapping
timeSpentToCreate	xsd:duration	Shows the times that has been taken to create the mapping
hasParameters	xsd:string	A textual description of the parameters passed to mapping discovery algorithm
hasConfidence	xsd:float	This property shows the confidence measure of the mapping that belongs to [0,1]
hasDescription	xsd:string	Any other details can be added using this property

7.3.5. Relations between Mappings

MAFRA is the only mapping representation language that allows the user to establish a set relations between the mappings. As we saw previously in chapter 2, the authors of MAFRA identified four types of relations: Specialization, Abstraction, Composition and Alternatives. We have added the necessary constructs to our mapping ontology to capture these relations.

mappingIsSpecializedFormOf object property with the domain and range of *Mapping* is used to represent specialization relation between mappings. Before starting the translation we go through all of the specialization relationships and copy all of the values of the mapping related properties from the general mapping and assign them to the specialized mapping.

To indicate the abstract mappings the data type property *mappingIsAbstract* with the domain *Mapping* and range of xsd:Boolean is used. If a mapping is abstract it will not be translated and won't participate in the instance translation.

Composition of the mappings is also possible by using the target of a mapping as the source of another mapping. Moreover, a mapping can be composed of several other sub-mappings indicated by the *mappingHasSubMapping* with the domain and rang of Mapping. Conditions are only applied to the super-mappings the sub-mappings are unconditional.

Alternative relation can be easily captured using conditions and no special construct is necessary for handling it.

7.4. Translation of Mappings to OWL and SWRL

A reason to map ontologies is to transform instances of the source ontology to the target ontology. There are two options in order to automatically achieve the instance transformation task: (1) Write a computer program which understands the mapping semantics and parses the mappings and loads the ontologies to perform the instance transformation; (2) Define formal semantics for the mapping language and perform the instance transformation using a reasoner for the language in which the formal semantics are defined. We choose the second approach in order to transform the instances between the ontologies that are mapped using our mapping ontology.

We believe that Semantic Web is expressive enough for instance transformation but it has not been fully utilized for this task. We transform the instantiation of the mapping ontology to a combination of OWL (either OWL-DL or OWL 2 RL) and SWRL. Using SWRL rules does not harm the decidability, soundness and completeness of the reasoning process. This mixture of OWL and SWRL then fed to an OWL reasoner that supports both OWL and SWRL. In order to translate instantiation of our mapping ontology we have two general steps (a) Translate the class, instance, property and data variables in each mapping (steps 1-7 of the Table 7.5) to OWL-Full; (b) Translate each mapping to OWL-DL + SWRL in order to be fed to an OWL reasoner (step 8 of the Table 7.5). Table 7.5 shows the pseudocode for translation of each mapping in our methodology.

Table 7.5 The pseudocode for translation of KMO instantiations to OWL + SWRL

```
1. Put all the variables of the current mapping that are not output of functions (Except for Instance and Data variables) in list1. Put all the output variables of the current mapping (Except for Boolean variables) in list 2. Put all Boolean output variables of the current mapping in list 3.
2. Translate the variables in list 1 until no further translation is possible. (Section 7.4.1)
3. Translate the variables in list 2 until no further translation is possible. (Section 7.4.1.3)
4. If list1 and list2 are empty go to 5 else go to 2.
5. Translate all the Boolean variables in list 3. (Section 7.4.2)
6. Process conditions. (Section 7.4.3)
7. Prepare the translated mapping for reasoning according to the translated variables. (section 7.4.4)
```

List 1 and 2 are repeatedly swept for variables to be translated until both of the lists are empty as the input variables should be first translated in order to translate the output variables based on them. However, the input variables may be dependent on the translation of the output variables as well. For example, an instance variable may belong to a class which is the output of a set function. In order to translate that instance variable, the class variable that it belongs to should be translated in list 2.

Boolean functions are treated separately since they are not supported by either OWL or SWRL directly. Mappings with Boolean functions are first translated into a single mapping rule without considering the Boolean functions in it. Then, considering the Boolean functions a single mapping rule may be transformed to several other SWRL rules in order to consider the effect of Boolean functions on the mapping.

The reason that we do not put instance and data variables in list 1 is that all of them are created in class variable *ClassPropertyHasValueRestriction* using properties *classPropertyHasValueRestrictionHasValue* and *classPropertyHasValueRestrictionHasInstance*. Therefore, if we take care of instances of the *ClassPropertyHasValueRestriction* in lists 1 and 2, all data and instance variables are taken care of as well. During the translation, variables are replaced by their values that are assigned to them using *variableHasClassValue*, *variableHasPropertyValue*, *variableHasInstanceValue*, *variableHasBooleanValue*, v*ariableHasNumericValue* and *variableHasStringValue* properties.

7.4.1. Translation of None-Output Variables

In this section, we explain how class, instance and property variables that are not output of functions are translated to OWL or SWRL axioms. Variables that are output of functions are translated during the translation of operators in section 7.4.1.3. Our algorithm for translation of these variables work in the following way:

(1) Remove each variable from the list1 once. If any of the values of the properties *classVariableHasProperty*, *classPropertyQCROnClass*, *classVariableHasClass* is a variable that exists in either list1 or list2 it means that variable is not translate yet and it should be put back in the list1 otherwise it is translated according to the algorithm explained in this section.

(2) If a variable was translated in step 1 go to step 1 else go to next step of the main algorithm.

How variables are translated in step 1 of the abovementioned algorithm is explained in the remainder of this section. Depending on the properties that are used for defining a class variable they may be translated to either OWL or SWRL axioms. If any of the following conditions are satisfied we will be using SWRL axioms (section 7.4.1.2) otherwise they are translated to OWL axioms (section 7.4.1.1).

(1) If the instance of the *ClassVariable* has a value for the property *classPropertyHasValueRestrictionHasInstanceVariable*. In this case, an instance variable is being defined.

(2) If value of the property *classPropertyHasValueRestrictionHasValue* is a variable which does not have a value for the property *variableHasValue* (super property of the *variableHasNumericValue*, *variableHasBooleanValue* and *variableHasStringValue*). In this case, a value variable is being defined.

7.4.1.1. Translation of Non-output Variables to OWL Axioms

In order to translate the *ClassVariable* class to OWL axioms our transformation algorithm uses the correspondences listed in Table 7.6 to translate instantiations of the mapping ontology to corresponding OWL-DL or OWL 2 axioms. When we arrive at an instantiation of the mapping ontology which belongs to cell on the left hand side of this table, it is translated to the corresponding OWL axiom mentioned in the cell on the right.

Table 7.6 Correspondence table between instantiations of our mapping ontology and OWL axioms. ?p, ?v and ?c represent properties, data values and classes or variables representing them respectively.

Mapping Ontology	**Equivalent OWL-DL, OWL 2 Construct**
classVariableHasProperty ?p classPropertyQCROnClass ?c hasCardinalityType any	[a owl:Restriction; owl:onProperty ?p owl:someValuesFrom ?c]
classVariableHasProperty ?p classPropertyQCROnClass ?c hasCardinalityType any	[a owl:Restriction; owl:onProperty ?p owl:allValuesFrom ?c]
classVariableHasProperty ?p hasCardinalityType any variableHasNumericValueForCardinality ?v	[a owl:Restriction; owl:onProperty ?p owl:cardinality ?v]
classVariableHasProperty ?p hasCardinalityType min variableHasNumericValueForCardinality ?v	[a owl:Restriction; owl:onProperty ?p owl:minCardinality ?v]
classVariableHasProperty ?p hasCardinalityType max variableHasNumericValueForCardinality	[a owl:Restriction; owl:onProperty ?p owl:maxCardinality ?v]
classPropertyQCROnClass ?c classVariableHasProperty ?p hasCardinalityType min variableHasNumericValueForCardinality ?v	[a owl:Restriction; owl:onClass ?c owl:onProperty ?p owl:minQualifiedCardinality ?v]
classPropertyQCROnClass ?c classVariableHasProperty ?p hasCardinalityType max variableHasNumericValueForCardinality ?v	[a owl:Restriction; owl:onClass ?c owl:onProperty ?p owl:maxQualifiedCardinality ?v]
classVariableHasProperty ?p variableHasNumericValueForCardinality ?v	[a owl:Restriction; owl:onProperty ?p owl:hasValue ?v]

In the table above we did not assign a value to the property *classVariableHasClass*. If any of the examples in the first column of the Table 7.6 has a value for this property, the value mentioned in the second column of the table should be intersected with the value of the property *classVariableHasClass*.

For instance, if we assign a value to this property in the example in the first row of the Table 7.6 the value in the row 1:

```
:cv1 a :ClassVariable;
:classVariableHasClass ?ic;
:classVariableHasProperty ?p;
:classPropertyQCROnClass ?c;
:hasCardinalityType any;
```

row 1 column 2 will be:

```
[a owl:Class; owl:intersectionOf
( ?ic
 [a owl:Restriction;
    owl:onProperty ?p;
    owl:someValuesFrom ?c]
   )
]
```

The same restriction is created for all of the examples in Table 7.6 if there is a value for the property *classVariableHasClass*.

As we mentioned previously, during translation to assign the real value of the class variable to it the property *variableClassHasClassValue* is used. For instance, the class variable in the above example has an assigned value using the following triples:

```
:cv1 :variableClassHasClassValue
[a owl:Class; owl:intersectionOf
```

```
 (?ic
  [a owl:Restriction;
  owl:onProperty ?p;
  owl:someValuesFrom ?c]
 )
]
```

7.4.1.2. Translation of Non-output Variables to SWRL Axioms

As we described earlier if the variable class that is being translated has a value for the property *classPropertyHasValueRestrictionHasInstanceVariable* or value of the property *classPropertyHasValueRestrictionHasValue* is a variable which does not have a value for the property *variableHasValue* it will be translated to SWRL axioms that will be used in SWRL rules.

To explain how we translate these variables to SWRL axioms we will explain the result of each step of our translation algorithm on the following example:

```
:cv1 a :ClassVariable;
 :classPropertyHasValueRestrictionHasInstanceVariable :personVar;
 :classVariableHasClass :Student;
 :classVariableHasProperty :hasWeight;
 :classVariabfleHasValue :weightVar.
```

Step1:

Find the value of the property *classPropertyHasValueRestrictionHasInstanceVariable* and create a SWRL variable with the name of the value of this property + “SWRLVar”.

```
:personVarSWRLVar a swrl:Variable.
```

Then create a SWRL axiom which shows that the created variable belongs the class represented by the property *classVariableHasClass*:

```
[a swrl:ClassAtom ;
 swrl:argument1 :personVarSWRLVar;
 swrl:classPredicate source:Person
]
```

If the value of the *classVariableHasClass* is an *owl:Class*, it will be used in the OWL axiom. If this value is a class variable, the value of the property *classVariableHasClass* will be used in the created OWL axioms.

Step2:

If the value of the *classPropertyHasValueRestrictionHasValue* is a variable that does not have a value for the property *variableHasValue* a SWRL variable is created with the name of the value of this property + "SWRLVar":

```
: weightVarSWRLVar a swrl:Variable.
```

Step 3:

Another axiom is created which shows that the SWRL variables created in the first and second steps are connected using the property indicated by the *classVariableHasProperty* which is *hasWeight*:

```
  [a swrl:DatavaluedPropertyAtom ;
   swrl:argument1 :personVarSWRLVar;
   swrl:argument2 :weightVarSWRLVar;
   swrl:propertyPredicate :hasWeight
  ]
```

Please note that since *hasWeight* is a data type property *swrl*:*DatavaluedPropertyAtom* should be used according to SWRL syntax. If we were creating SWRL axioms for a class variable that puts restrictions on an object property such as *hasFather*, we would use swrl:*IndividualPropertyAtom*:

```
[a swrl:IndividualPropertyAtom;
 swrl:argument1 :personVar1SWRLVar;
 swrl:argument2 :fatherVar2SWRLVar;
 swrl:propertyPredicate :hasFather
]
```

If the class variable belongs to the source of the mapping, these SWRL axioms are added to the body of the SWRL rules otherwise they are added to the head of the SWRL rule.

7.4.1.3. Translation of Non-Boolean Output variables

In this section we describe how the non-Boolean output variables are translated to OWL and SWRL rules. Output variables of functions are translated according the operator used in corresponding function. To translate output variables, the function producing it, the first and second input variable, the output variable and operator of the function should be retrieved from the ontology. Depending on the function if it is *ExpressionFunction*, *ConvertFunction* or *CreateFunction* the inputs of them should be retrieved using different properties. Our algorithm for translation of these variables works in the following way:

(1) Remove each variable once from the list2. If any the input variables that are needed to compute the output variable is in the list1 or list 2 put the variable back to the list2 otherwise proceed with the translation of output variable according to the used operator (explained in sections 7.4.1.4, 7.4.1.5, 7.4.1.6 and 7.4.1.7).

(2) If a variable was translated in step 1 go to step 1 else go to next step of the main algorithm.

7.4.1.4. Output Variables of Functions with Convert Operators

Convert functions are translated using Jena API functions. If the operator is convertToClassTO or convertToPropertyTO, the first input is converted to a class or a property respectively. If the operator is convertToInstanceTO, the first input is converted to an instance which belongs to the class that is indicated by input 2. Please note that the original element remains intact and it is the function's output that holds the new converted values. Table 7.7 shows the Jena API functions that have been used to execute convert operators. This output variables can be used to define other variables or be used in other functions.

Table 7.7 Jena functions have been used to implement the convert operators

Operator	**Jena API function**
convertToClassTO	OntModel.createClass(String)
convertToInstanceTO	OntModel.createIndividual(String)
convertToPropertyTO	OntModel.createProperty(String)

For instance, if we have a function with the following specification:

```
mo:f1 a :Function;
   mo:functionHasInputVariable1 source:airbus;
   mo:functionHasInputVariable2 source:AirPlance;
   mo:functionHasOutputVariable mo:fout;
   mo:functionHasOperator mo:convertToClassTO.
```

The result of translation of fout variable will be adding the following triples to the instantiation of the mapping ontology:

```
target:Airbus a owl:Class.
   mo:fout mo:classVariableHasClassValue target:Airbus.
```

As you can see, the *target:Airbus* class has been used as a value of the property *classVariableHasClassValue*. Therefore, this triple is in OWL-Full which is undecidable. These variables will be translated to OWL-DL + SWRL during the translation of the mapping that uses the *mo:fout* variable. This process is described in 7.4.4.

7.4.1.5. Output Variables of Functions with Set Operators

Several operators have been defined to perform set operations on classes and class variables. The true values of the input variables (indicated by the properties in Table 7.2) will be put in a java RDFList then the corresponding Jena function is used to create the necessary OWL axioms. If the operator is intersectionSO, unionSO or intersectionSO the Jena functions OntModel.createIntersectionClass(String, RDFList), OntModel.createUnionClass(String, RDFList) or OntModel.createComplementClass(String, Resource) will be used respectively to create values for the output variables. The created class will be set as the value of the property *variableHasClassValue* of the output variable. For instance, the following instantiation of the mapping ontology:

```
:func1 a :Function;
:functionHasInputVariable1 :Male;
:functionHasInputVariable2 :Parent;
:functionHasOperator :intersectionSO;
:functionHasOutputVariable :func1OutVar.
```

will be translated to the following triples:

```
:func1OutVar :variableHasClassValue
[a owl:class;
 owl:intersectionOf( :Parent :Male)
].
```

Now this class variable is ready to be used in other class variables or as an input of functions. Please note that these triples will not be used for reasoning because they are in OWL-Full and the final stage will transform these triples to OWL-DL + SWRL. This process is described in 7.4.4.

7.4.1.6. Output Variables of Functions with Mathematic Operators

Output variables of functions that make use of instances of *StringOperator*, *MathOperators*, *CompratorOperator* and *BooleanOperator* will be translated to SWRL axioms that will be added to the body of the SWRL rule representing a mapping. These SWRL rules will be used by a reasoner to perform the instance transformation. In order to create SWRL axioms first input, second input and output variables and the operator should be retrieved from the instantiation of the mapping ontology. Each of the variables defined in our mapping ontology will be represented by a SWRL variable with the name: name of the variable + "SWRLVar". Then the corresponding SWRL operator is found to be used in the SWRL rule. Finally a SWRL built-in atom is created based on the inputs, the operator and the output. For instance the following example

```
:f1 a :Function;
:functionHasInputVariable1 :input1;
:functionHasInputVariable2 :input2;
:functionHasOutputVariable :output
:functionHasOperator :divideMO
```

is translated to

```
:outputSWRLVar a swrl:Variable.
:input1SWRLVar a swrl:Variable.
:input2SWRLVar a swrl:Variable.
[a swrl:BuiltinAtom ;
swrl:arguments (:outputSWRLVar :input1SWRLVar :input2SWRLVar);
swrl:builtin swrlb:divide]
```

Depending on the operator, a different swrl:builtin may be used. For instance, for the multiplyMO operator we will use swrlb:multiply to create the corresponding SWRL axiom.

7.4.1.7. Output Variables of InstanceFromPropertyCreateFunction

All create functions can be easily implemented using Jena API. However, the *InstanceFromPropertyCreateFunction* needs special attention for implementation. In order to understand how this function is translated to OWL and SWRL rules, we go through the following example:

```
:icf1 a :InstanceFromPropertyCreateFunction;
    :functionHasInputVariable1 :a;
    :functionHasInputVariable1 :b;
    :functionHasOutputVariable :c;
    :functionPropertyToCreateInstnace source:is_Taking_Medication.
```

Upon reaching an *InstanceFromPropertyCreateFunction* output the following steps will be taken:

(1) A SWRL variable is created by adding the “SWRLVar” the name of each of the input variables and the output variable:

```
mo:aSWRLVar a swrl:Variable.
mo:bSWRLVar a swrl:Variable.
mo:cSWRLVar a swrl:Variable.
```

(2) Every two instance that are connected using the property indicated by *functionPropertyToCreateInstnace* are found and an instance with the name of the subject + _property_ + object is created. In our example, if there is a triple “source:patient1 source:is_Taking_Medication source:Aspirin” the following triple is created in the target ontology “target: patient1_ is_Taking_Medication_Aspirin”.

(3) Property *hasName* with the domain owl:Thing and range xsd:string is used to assign the names to the created instance and the instances that have been used for creating it. In our example the created instance is patient1 is_Taking_Medication_Aspirin and instances that have been used to create it are patient1 and Aspirin:

```
:patient1 :hasName “patient1”^^xsd:string.
```

```
:Aspirin :hasName “Aspirin”^^xsd:string.
:patient1 is_Taking_Medication_Aspirin :hasName
        “patient1_is_Taking_Medication_Aspirin”^^xsd:string.
```

Please note that SWRL cannot use the name of the instances in the reasoning process. This is why we have added the names as property values to make them available to the SWRL rules. These names will be used to identify the created instances in the SWRL rules.

(4) A SWRL rule which makes use of the assigned name to identify the created instance in the target ontology is made. The following rule identifies the created instance in the target ontology based on the instances that are used to create it (instances that are assigned to aSWRLVar and bSWRLVar), by checking the values of the *hasName* property for SWRL variables aSWRLVar, bSWRLVar and cSWRLVar.

```
mo:hasName(?aSWRLVar,?v1name) ^ mo:hasName(?bSWRLVar,?v2name) ^
swrlb:stringConcat(?sv1,?v1name,”_is_Taking_Medication_”) ^
swrlb:stringConcat(?sv2,sv1,v2name) ^
mo:hasName(?cSWRLVar,?sv2) →
target:patient_ID(?cSWRLVar,?aSWRLVar) ^
target:medication_?Name(?cSWRLVar,?bSWRLVar)
```

In the example above, for the instances patient1 and Aspirin and the property *is_Taking_Medication*, the created instance patient1_is_Taking_Medication_Aspirin will be assigned to the variable cSWRLVar. Now this variable can be used in the head of the SWRL rule for ontology mapping to assert that

```
target:patient1_is_Taking_Medication_Aspirin
      target:patient_ID source:patient1;
      target:mediation_Name source:jane.
```

7.4.2. Translation of Boolean Output Variables

When both list1 and list2 are empty, we start translation of output of the Boolean operators. Unfortunately SWRL does not provide the built-in for the orBO, andBO and xorBO operators; therefore we need to take care of them separately. In order to handle Boolean orBO, andBO and xorBO operators we need to break the existing SWRL rule into two several other SWRL rules.

We first create a SWRL variable for each of the Boolean variables in the function by adding the name "SWRLVar" to the end of each of those variables. For instance, the variable for the variable mo:a the following variable is created:

```
mo:aSWRLVar a swrl:Variable.
```

After creation of the necessary SWRL variables, we iterate through all the possible values of the Boolean variables that are not output of functions and compute the values of the Boolean output variables in list3 (using the SWRL rules described in section 5.4 for implementation of the expression functions). As we iterate through the values, we create a copy of the existing SWRL rule created for the current mapping and add a SWRL axioms that represents the current value of the all of the Boolean values. Therefore, if we have n non-output Boolean variables, we create 2^n different set of SWRL axioms.

7.4.3. Handling Conditions

Mappings can be conditional and have complex condition satisfaction criteria. The same properties and classes used in precondition section of CPG-DKO in section 4.4 of the thesis are used to indicate conditions for conditional mappings. Classes used for handling conditions are *Condition*, *SatisfiedCondition*, *UnSatisfiedCondition* and *Cardinality* and the properties used for this purpose are *hasCondition*, *hasCardinalityType* and *hasCardinalityValue*.

As we saw previously, mappings with variables that have no asserted values for the property *variableHasValue* will be translated to SWRL rules because they are representing a fragment of the ontology rather than a specific element of it. For instance, in the following example, *ageVar* represents all the numeric values of the property *hasAge* in the source ontology.

```
mo:ageVar a mo:NumericVariable.
mo:cv1 a :ClassVariable;
 mo:classPropertyHasValueRestrictionHasInstanceVariable
:personVar;
 mo:classVariableHasProperty source:hasAge
 mo:classVariabfleHasValue mo:ageVar.
```

Having preconditions will lead to using variables that need SWRL rules. Therefore, whenever a mapping is conditional, it is represented by SWRL rules. In order to allow only the mappings that have satisfied condition criteria to be used in instance transformation, we review the SWRL rules that have been created in section 7.4.2 and see if the condition satisfaction criteria is satisfied in them by checking the values of Boolean variables. If the condition satisfaction criteria is met, the SWRL rule is kept otherwise it is removed from the list of SWRL rules representing the mapping. In this step, usually the size of the set of SWRL rules representing a mapping is significantly reduced.

Continuing our previous example in section 7.4.2, we mentioned that 8 SWRL rules will be created to handle the Boolean operators in that mapping. Suppose that both isEligibleForFreeFluShotVar and isDiabeticAndHasHFOutVar are conditions of the mapping and the condition satisfaction criterion is all. We go through all of the SWRL rules created for that mapping and review the Boolean variables and remove the SWRL rule when either of the variables' value equals to "false"^^xsd:Boolean. Therefore, of those 8 created SWRL rules the only ones are kept that have both of the following axioms in the head of the rule:

```
[ a swrl:BuiltinAtom ;
swrl:arguments (isEligibleForFreeFluShotVarSWRLVar
"true"^^xsd:boolean) ;
swrl:builtin swrlb:equal
]
[ a swrl:BuiltinAtom ;
swrl:arguments (isDiabeticAndHasHFOutVarSWRLVar "true"^^xsd:boolean) ;
swrl:builtin swrlb:equal
]
```

Other condition satisfaction criteria such as any, any k out of n are handled in the same way.

7.4.4. Translation of Instance of the Mapping Class

The value transfer mappings are already in SWRL and ready for execution. However, Relational and transformation mappings are in OWL-Full because of not respecting the class-instance separation that is needed in OWL-DL. In this section we describe how these two mappings are translated to OWL-DL + SWRL for reasoning.

7.4.4.1. Translation of Relational Mapping

Relational mappings can only have one source and one target with a value for the property *mappingHasRelation*. In order to be able to translate these mapping to OWL-DL for instance transformation we need to write a triple in the following form:

(Real value of the source variable, the corresponding OWL relation, real value of the target variable). For instance, imagine the following mapping and the following translated source and target variables:

```
:m1 a :Mapping;
      :mappingHasRelation :classRelationEquivalentClass
      :MappingHasSource :out1;
      :MappingHasTarget :out2;

:out1 :variableHasClassValue
```

```
      [a owl:Class;
           owl:intersectionOf (source:Parent source:Male)
      ]

:out2 :variableHasClassValue
      [a owl:Class; owl:intersectionOf(
           target:Person
           [a owl:Class; owl:ComplementOf(target:Mother)]
        )
      ]
```

This is in OWL-Full because OWL classes have been used as instances and cannot be used for reasoners instance transformation. This mapping is translated to the following OWL-DL axiom:

```
 [a owl:Class;
   owl:intersectionOf (source:Parent source:Male)
]
owl:equivalentClass
[a owl:Class; owl:intersectionOf(
      target:Person
      [a owl:Class; owl:ComplementOf(target:Mother)]
  )
]
```

7.4.4.2. Translation of Transformation Mappings

After translation of all the variables it is possible to translate transformation mappings as well. The first type of transformation mapping is *ClassToPropertyTransformation*. These mappings can be easily translated using SWRL rules.

To translate these mappings, the true value of the properties *classToPropertyTransformationHasSourceClass*, *classToPropertyTransformationHasSourceFirstProperty*, *classToPropertyTransformationHasSourceSecondProperty* and *classToPropertyTransformationHasTargetProperty* are found and then a SWRL rule which makes use of these values is written to represent the mapping. If we assume the values of these properties are *sourceClass*, sourceProp1, sourceProp2 and *targetProp* respectively, the mapping will be represented by the following SWRL rule:

```
source:sourceClass(?x) ^ source:sourceProp1(?x,?x1) ^
source:sourceProp2(?x,?x2) → target:targetProp(?x1,?x2)
```

Another type of transformation mapping is the *PropertyToClassTransformationMapping*. To translate this mapping to OWL, the value of the properties *propertyToClassTransformationHasSourceProperty* , *propertyToClassTransformationHasTargetClass*, *propertyToClassTransformationHasFirstTargetProperty* and *propertyToClassTransformationHasSecondTargetProperty* are found. Assume that they have the following values *sourceProp*, *TargetClass*, *firstTargetProp* and *secondTargetProp*. For every two instance that are connected using *sourceProp*, an instance created in the target ontology with the name: name of the subject +_ +name of the *sourceProp* + _+ name of the object. This instance is set as an instance of the *TargetClass*. Then subject and objects of the *sourceProp* are connected to this created instance using *firstTargetProp* and *secondTargetProp* properties respectively.For instance if we have the following mapping:

```
:m a :PropertyToClassTransformationMapping
:propertyToClassTransformationHasSourceProperty source:marriedTo;
  :propertyToClassTransformationHasTargetClass target:Marriage;
  :propertyToClassTransformationHasFirstTargetProperty
target:hasPartner1;
  :propertyToClassTransformationHasSecondTargetProperty
                                   target:hasPartner2.
```

and we have the following triple “source:john source:marriedTo source:jane” in the source ontology, the outcome of the abovementioned procedure will be

```
target:john_jane_marriedTo_PropToClassTrans a target:Marriage;
      target:hasPartner1 :john;
      target:hasPartner1 :jane.
```

7.4.4.3. Translation of Value Transfer Mappings

As the variables are translated, SWRL rules representing these mappings are built as well. Therefore, these mappings that are represented by SWRL rules are ready to be used by reasoners to perform instance transformation. These mappings can have several sources and targets with no value for *mappingHasRelation*. Since we had not covered the concept of variables and functions in section 7.3.1, we discuss an example of the value transfer mapping in this section. We would like to create a mapping which makes use of the height and the weight of a person in the source ontology and assign the BMI of the person to him or her using the *hasBMI* from the target ontology. For instance from having this:

```
source:john source:hasWeightKG "82.2"^^xsd:float;
          source:hasHeightCM "187.0"^^xsd:float.
```

We would like to have the following triple after the mapping in the target ontology:

```
source:john target:hasBMI_KG_CM "23.5"^^xsd:float
```

We first need to create the variables that represent the weight, height and BMI of a person and the person itself:

```
mo:weightVar a mo:NumericVariable.
mo:HeightVar a mo:NumericVariable.
mo:BMIVar a mo:NumericVariable.
mo:personVar a mo:InstanceVariable.
```

```
mo:cv1 a mo:ClassPropertyHasValueRestriction ;
     mo:classVariableHasClass source:Person ;
     mo:classVariableHasProperty source:hasWeight ;
     mo:classPropertyHasValueRestrictionHasInstanceVariable
        mo:personVar;
     mo:classPropertyHasValueRestrictionHasValue mo:weightVar

mo:cv2 a :ClassPropertyHasValueRestriction ;
     mo:classVariableHasClass source:Person ;
     mo:classVariableHasProperty source:hasHeight ;
     mo:classPropertyHasValueRestrictionHasInstanceVariable
      :personVar;
     mo:classPropertyHasValueRestrictionHasValue :HeightVar.
```

Then we need functions that make use of weightVar and HeightVar to compute the BMIVar according to the following formula: weightVar/(HeightVar)^2:

```
:mathfunc1  a :MathFunction;
    :functionHasInputVariable1 :weightVar;
    :functionHasInputVariable2 :HeightVar;
    :functionHasOutputVariable :mathfunc1Out;
    :functionHasOperator :divideMO.

:mathfunc2  a :MathFunction;
    :functionHasInputVariable1 :mathfunc1Out;
    :functionHasInputVariable2 :HeightVar;
    :functionHasOutputVariable :BMIVar;
    :functionHasOperator :divideMO.
```

Then we need to assign the value of the BMIVar to the instance variable personVar using the *target:hasBMI* property:

```
mo:cv3 a mo:ClassPropertyHasValueRestriction ;
     mo:classVariableHasClass target:Person ;
     mo:classVariableHasProperty target:hasBMI ;
     mo:classPropertyHasValueRestrictionHasInstanceVariable
mo:personVar;
     mo:classPropertyHasValueRestrictionHasValue :BMIVar .
```

And finally we need to create a transformation mapping which makes use of the created class variables and mathematical functions for the mapping:

```
:m12 a :ValueTransferMapping;
    :mappingHasSource :cv1;
    :mappingHasSource :cv2;
    :mappingHasFunction :mathfunc3;
    :mappingHasFunction :mathfunc4;
    :mappingHasTarget :cv3.
```

This mapping is translated to the following SWRL rule to be executed in Pellet reasoner:

```
:SWRLRule1
      a       swrl:Imp ;
  swrl:body (
      [ a       swrl:ClassAtom ;
       swrl:argument1 :personVarSWRLVar;
       swrl:classPredicate source:Person
      ]
      [ a       swrl:ClassAtom ;
      swrl:argument1 : personVarSWRLVar ;
       swrl:classPredicate source:Person
      ]
      [ a       swrl:DatavaluedPropertyAtom ;
       swrl:argument1 :sourceInstSWRLVar ;
       swrl:argument2 :weightVarSWRLVar ;
       swrl:propertyPredicate  source:hasWeight
      ]

      [ a       swrl:DatavaluedPropertyAtom ;
       swrl:argument1 :sourceInstSWRLVar ;
       swrl:argument2 :HeightVarSWRLVar ;
       swrl:propertyPredicate source:hasHeight
      ]
      [ a       swrl:BuiltinAtom ; swrl:arguments
       (:mathfunc1OutSWRLVar  :weightVarSWRLVar :HeightVarSWRLVar)
;
       swrl:builtin swrlb:divide
      ]
      [ a       swrl:BuiltinAtom ;
       swrl:arguments(:BMIVarSWRLVar :mathfunc1OutSWRLVar
:HeightVarSWRLVar);
       swrl:builtin swrlb:divide
      ]
) ;
  swrl:head (
      [ a       swrl:ClassAtom ;
      swrl:argument1 : personVarSWRLVar ;
       swrl:classPredicate target:Person
      ]
      [ a       swrl:DatavaluedPropertyAtom ;
       swrl:argument1 :personVarSWRLVar;
       swrl:argument2 :BMIVarSWRLVar ;
       swrl:propertyPredicate target:hasBMI
      ]) .
```

7.5. Conclusion

In this section, we introduced our ontology mapping framework that is composed of (1) an ontology for representation of mappings and (2) an algorithm for translation of the mappings to OWL-DL + SWRL. Our ontology can map any two ontologies in OWL. Our ontology mapping framework has the following unique features compared to the exiting mapping representation languages:

- **High Expressivity:**

KMO is of high expressivity for representation of complex mappings. This ontology is superior to the existing ontology mapping representation languages in terms of representation of predefined mapping patterns, conditions, expressions and structural modification operators. Two of the closest mapping representation languages in terms of expressiveness are discussed in [165] and [166]. However the language in [165] does not have an expression language. The language provided in [166] is more expressive than the one presented in [165] but still suffers from lack of several important expressivity features such as structural modification operators.

- **Translation of mappings to OWL + SWRL:**

We define the formal semantics of the ontology mappings in KMO by translating them from OWL-Full to OWL-DL + SWRL using our translation algorithm. As opposed to using OWL + SWRL directly to map two ontologies, our approach has the following benefits:

i. The expressivity of KMO being OWL-Full—i.e. using properties and classes as instances—makes the ontology mappings more readable and less verbose.

ii. It enables us to support conditional mappings and complex condition satisfaction criteria, Meta modelling, Boolean operators and converting ontology elements and creating new ones which are not directly supported by either OWL or SWRL.

iii. SWRL rules are difficult to write and can become undecidable if not written correctly. In our translation algorithm, DL-Safe SWRL rules are generated automatically thus relieving the user about decidability concerns.

There are other mapping frameworks such as [164] and [166] that can translate the mappings to either SWRL or OWL but not a combination of them. However, we saw in this chapter that a combination of both languages is necessary to effectively map two OWL ontologies. Unfortunately, no explanations or details of the translation process in these frameworks are provided.

- **Formal semantics:**

We define the formal semantics of our mapping representation ontology by translating it to OWL-DL + SWRL. An OWL reasoner can use the formal semantics to discover new mappings between the mapped ontologies and to transform instances from the source ontology to the target ontology automatically.

Our review of the mapping representation languages in Chapter 2 showed that only a few of the mapping representation languages such as OWL [142], C-OWL [58], SWRL [154] and Euzenat1 [166] have formal semantics. OWL, C-OWL, SWRL are not expressive enough for complex mappings. Authors of [166] have defined the semantics of their mapping language in first order logic and have assumed they can replace all the expressions by first-order formulas. There are two issues with that (1) First order logic can be undecidable and no explanations have been provided in this regard; (2) No practical details have been provided that can help with an implementation of a semantic based ontology mapping approach. The tool that is developed to work with this language does not perform reasoning on it and uses Java for instance transformation.

Our ontology mapping approach has the following limitations:

- If inconsistencies exist between the mapped ontologies or the mappings themselves, instance transformation or mapping discovery is not possible.

- It is only possible to map OWL ontologies.

- Worst case complexity of reasoning on OWL-DL ontologies is NEXPTIME-complete [152]. This means that reasoning on OWL-DL ontologies can be computationally expensive. Therefore, when two large ontologies are mapped, the instance transformation process can be time consuming.

We also suggest the following potential future works in order to improve this research:

- Defining a formal syntax for the mapping representation language so that mappings can be checked for syntactical correctness.

- Extracting contextualized sub-ontologies: it will be beneficial to extract the contextualized sub-ontologies that are relevant to the context and to perform reasoning on them and the mappings as opposed to using the whole ontology. This will reduce the reasoning time and avoid any potential inconsistencies between the parts that are not extracted.

CHAPTER 8: ONTOLOGY MERGING

8.1. Introduction

Thus far, we have developed the modules that are needed to map several LKO to DKO and have all their instances transformed to it. Moreover, our knowledge execution module is capable of delivering decision support based on concurrent and independent execution of several knowledge artifacts that are transformed to DKO. There are still two components to be developed in order to dynamically merge the transformed LKO represented in DKO: (1) A language in order to represent the morphing constructs—i.e. merging constraints—between the knowledge artifacts and (2) a knowledge morphing execution engine that is capable of modifying the decisions made in each of the knowledge artifacts according to the semantics of the merging constraints in order to provide an improved and conflict-free decision support for the problem at hand. Since these components are domain-specific, we develop these two components for our *CPG Merging Framework*.

The purpose of our CPG Merging Framework is to provide decision support for treatment of comorbidities. The treatment of comorbid diseases—i.e. the simultaneous presentation of multiple medical conditions in a patient—is a challenge, as it demands the careful coordination between the disease-specific therapeutic plans of the comorbid diseases. CPG are usually focused towards a specialized disease, where they present evidence-based recommendations about the management of a single disease. Although, some CPG allude to the presence of comorbid conditions but they do not stipulate specialized clinical pathways on the management of comorbid conditions [107].

The execution of CPG, in the realm of a computerized DSS, is a challenging exercise since CPG encapsulate highly complex elements in terms of domain knowledge, decision logic, clinical constraints, set of clinical actions and temporal relations. Therefore, execution of a computerized CPG needs to monitor and handle such complexities in line with the specific patient data and the institution’s operational conditions. To handle comorbidities in a clinical decision support environment is further challenging as it demands the coordination of two disease-specific CPG. Here, the challenge is not just to optimize the overall care process in terms of avoiding duplication of tasks, but more importantly ensuring patient safety such that actions recommended by one CPG do not cause harm to the patient's comorbid condition. Therefore, the concurrent execution of two independent CPG to handle comorbidities is not a viable solution. We argue that to handle comorbid conditions, it is important to establish a relationship between the comorbidity CPG so that the overall care process is coordinated. We argue that to handle comorbid conditions in a clinical decision support environment, one approach is to systematically merge the independent CPG of the comorbid conditions leading to the generation of a mutually consistent comorbid CPG. Merging multiple CPG is best defined by Abidi [107] as: "*Merging multiple CPG is the process of merging the knowledge which is encapsulated in them into a unified CPG that can be used to treat the patients for their comorbidities while the integrity and pragmatics of the medical knowledge is kept intact*". Merging CPG to handle comorbid conditions can have several benefits: (a) Timely and cost effective treatment of patients by avoiding unnecessary duplication of tasks, visits and medical test and reusing existing test results; (b) Identifying potential adverse interactions and preventing patients from treatments that may compromise their safety; (c) Standardization of care across multiple institutions for comorbid patients.

Several frameworks have been developed to model paper-based CPG in a semantically explicit and computationally executable formalism in order to be used in CDSS. PROforma [136][138], GLIF [133][140], GASTON [134][135], Asbru [108], EON [137][155], SAGE [34] and COMET [107] are a few to name. However, only few of the existing approaches have the ability to merge multiple CPG in order to handle comorbid situations [24][26][30][99][107]. In our work, we utilize semantic web technologies to address the merging of multiple CPG in order to deliver guideline-mediated decision support via a CDSS. Our CPG merging framework has been discussed in the next subsection.

8.2. CPG Merging Framework

We are pursuing the problem of dynamic merging of two CPG to handle comorbid conditions by utilizing our knowledge morphing framework. Our solution approach is to: (a) model the paper-based CPG in terms of CPG-DKO (b) establish semantically-explicit merging constraints that need to be satisfied to safely merge multiple CPG; and (c) merge multiple ontologically-modeled CPG by executing the pre-specified morphing constraints. These steps are performed using the following components:

- **CPG Domain Knowledge Ontology (CPG-DKO):**

The purpose of a CPG domain ontology is to computerize each individual CPG using a standard and semantically-explicit representation formalism. We use our CPG-DKO described in chapter 4 for this purpose. Each CPG is as an instantiation of this ontology.

- **CPG Knowledge morPhing Ontology (CPG-KPO):**

Morphing constructs in our CPG merging framework represent the CPG merging constraints. A morphing construct describes a constraint that needs to be respected during parallel execution of CPG. It may also describe the modifications that should be performed on the reasoned decisions in individual comorbidity CPG in order to avoid violation of the represented constraint. Two CPG can merge at the action/task level subject to the satisfaction of a priori defined conditions for the morphing constraints.

During our review of the related literature to CPG merging we noticed that the only merging constraints that are considered in the existing CPG merging frameworks are aligning common tasks between CPG and identifying the conflicting tasks. However, we believe that merging disease-specific CPG in clinical decision support is more complicated than merely finding the common tasks and conflicts and more complicated criteria can be taken into consideration. To create a more comprehensive list of merging constraints, we discussed with the physicians in our research group, reviewed CPG merging literature [20][21][22][24][25][26][29][30][107] and got inspired from classical AI planning literature [183]. For instance, a merge constraint in our list may express that two inter-CPG tasks (a) are identical, (b) should be executed simultaneously, (c) are conflicting, (d) can reuse each other's results, (e) have a temporal or sequential constraint between them, (f) can be combined to a new task and (g) their execution schedule depends on operational constraints for their simultaneous execution.

To represent the CPG merging constraints in a computer understandable format, we have developed an OWL-DL ontology referred to as *CPG Knowledge morPhing Ontology* (CPG-KPO). This ontology is instantiated manually with the help of a domain expert familiar with the comorbidities and their merging criteria. Therefore, the source of knowledge encapsulated in an instantiation of this ontology can be physicians' personal experience participating in the ontology instantiation or their interpretation of evidence based CPG that contain knowledge regarding comorbidities. We have also defined the formal semantics of this ontology in terms of OWL and SWRL. Using formal semantics, decision reconciliation can be performed automatically using our OWL-based CPG execution engine.

- **Comorbidity CPG Execution Engine:**

The Knowledge Morphing Execution Engine of our CPG merging framework is called the *Comorbidity CPG Execution Engine.* This engine takes two sets of inputs: (1) Two or more instantiation of CPG-DKO that represent disease-specific CPG and (2) an instantiation of CPG-KPO that represents the morphing constraints. The output of this merge execution engine is recommendations generated based on the dynamically merged CPG during their concurrent execution. To implement this engine, we made modifications to our OWL based CPG execution engine in order to enable it to take into account the semantics of the CPG-KPO. Considering semantics of CPG-KPO enables this engine to detect merging constraint violations and perform modifications on the generated therapy plans in each of the CPG in order to avoid constraint violation.

Even though it is possible to capture the morphing constraints between CPG computerized in different ontologies (LKO) using CPG-KPO, dynamic merging of these merged CPG will be difficult as it involves execution of all the CPG representation languages. However, execution engines including ours are not capable of executing more than one CPG representation language. Hence, our merging execution engine solely operates on CPG represented in CPG-DKO

8.3. CPG Knowledge morPhing Ontology (CPG-KPO)

In this section, we describe CPG-KPO as it is central to our CPG merging framework. We use OWL to model the morphing constraints because: (1) Our CPG ontology is modeled in OWL language; using OWL creates a unified representation format for both CPG representation and merging; (2) Since the merging constraints can be rather complex, OWL provides reasonable expressiveness to represent these complex morphing constraints.

In the discussion, the CPG-KPO's properties are listed in *italics* and classes are both *italicized* and Capitalized and instances are <u>underlined</u>. CPG-KPO can be instantiated in order to represent the merging constraints between two ontologically modeled CPG. This ontology is composed of two parts called the *independent part* and the *shared part*. The shared part that is shared between CPG-KPO and CPG-DKO represents the basic concepts of the domain knowledge that are used in definition of morphing constraints. These ontology elements are the classes *Task* and its subclasses that represent different states of this task which are *ActiveTask*, *InactiveTask*, *CompletedTask*, *StartedTask* and *DiscardedTask* classes. We also added the class *PendingTask* to shared part. This class is used for execution purposes. Moreover, class *Condition* and its subclasses which are *SatisfiedCondition* and *UnSatisfiedCondition* classes belong to this part of the ontology.

The independent part of this ontology is instantiated to represent the merging constraints between instances of the *Task* class in two or more CPG. No relations are created between classes or properties of CPG ontologies and CPG-DKO. Therefore, we can say that merging happens at the instance level and classes and properties of these CPG ontologies are not involved in the merge. The morphing constraints are represented as a set of constraints that need to be respected in order to merge two tasks from different CPG during the concurrent execution of two or more CPG. The *Constraint* class represents the merging constraints—the constraints are further categorized in to four types represented by the following subclasses of the *Constraints* class:

1. *WorkflowConstraint*: This class represents constraints that affect how the workflow structures of the CPG are interpreted during concurrent execution with other CPG.

2. *OperationalConstraint*: These constraints represent the operational constraints that affect merging several CPG in a specific institution. The intent is to account for the fact that the handling of comorbidities is potentially different at different institutions. Therefore, these constraints represent the institutional regulations regarding the merging of CPG.

3. *TemporalConstraint*: This class represents the temporal constraints between the inter-CPG tasks. For instance, a temporal constraint may indicate that a specific task from the CHF CPG should not be executed at least ten days after another specific task in the AF CPG.

4. *MedicalConstraint*: This class represents constraints pertaining to the potential reuse of the results of medical tasks.

The tasks that are merged by a constraints are indicated by the properties *hasTask1* and *hasTask2* with the domain of *Constraint* and range of *Task*. We call these tasks task1 and task2 respectively for all of the constraints in the rest of this section. The *Task* class represents all possible clinical tasks in CPG. For instance, a blood test will be indicated by an instance of the *Task* class. Figure 8.1 shows the class hierarchy of the CPG-KPO.

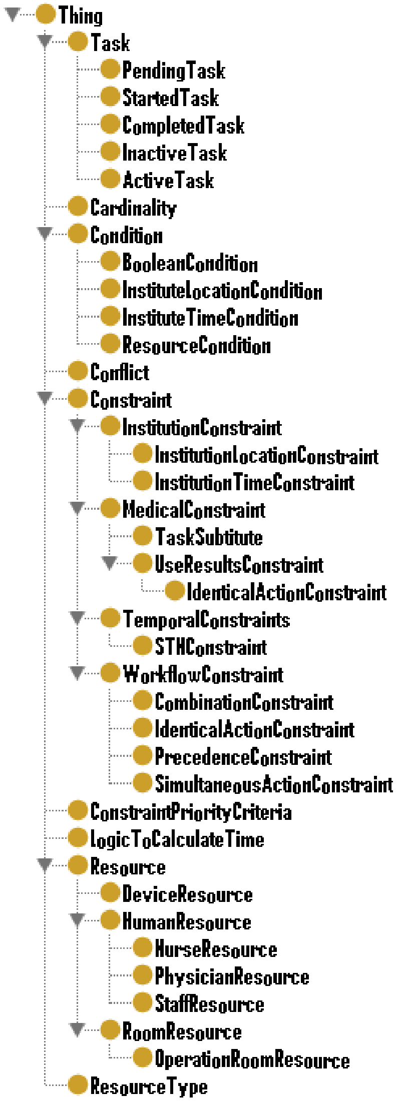


Figure 8.1 Class Hierarchy of CPG-KPO

8.3.1. Merging Constraints

In this section, we describe in detail the workflow, operational, temporal and medical merging constraints. As we describe these concepts, we give simple imaginary examples in order to explain the purpose and meaning of each of the introduced concepts. We also go through several real CPG merging examples and discuss how CPG-KPO can represent the intended merging constraints.

8.3.1.1. Workflow Constraints

The workflow constraint class has four subclasses: *IdenticalActionConstraint*, *PrecedenceConstraint*, *SimultaneousActionConstraint* and *CombinationConstraint*. *IdenticalActionConstraint* indicates that task1 and task2, each belonging to a different CPG, are identical and hence a single instance of this task needs to be executed rather than executing both these identical tasks. This constraint aims to avoid duplication of tasks in order to minimize resource usage and costs. For instance, if two CPG are both recommending a CT-Scan then this constraint will order a single CT-scan and re-use its result. Since the two similar tasks may not necessarily co-occur in both CPG, the CPG reaching the common task first will execute the task and the other CPG will re-use the results when it reaches the common task, provided the results of the common task are still clinically and temporally valid. Data type properties *hasValidityPeriodT1* and *hasValidityPeriodT2* determine how long after the execution of a common task its results are clinically valid. For instance, suppose task1 and task2 are identical and task1 in CPG1 gets executed. When the CPG execution engine reaches the task2 in CPG2, it compares the value of the *hasValidityPeriodT1* with the amount of time that has been passed since the execution of task1. If the elapsed time is within the validity period, then task2 will not be executed and CPG2 will take the results of task1. The following example shows that **CHF**: Pre-treatment_electrolytes_assessment_and_correction and **AF**:Pre-treatment_electrolytes_assessment_and_correction from CHF and AF CPG respectively are identical and the results of these tasks are valid for 10 days after execution:

```
:c1 a :IdenticalActionConstraint;
 :hasValidityPeriodT1 “10”^^xsd:int; #days
 :hasValidityPeriodT2 “10”^^xsd:int; #days
 :hasTask1 CHF:Pretreatment_electrolytes_assessment_correction;
 :hasTask2 AF:Pretreatment_electrolytes_assessment_correction.
```

PrecedenceConstraint represents the situation where task1 should be executed before task2. An instance of this class creates a sequential constraint between the tasks of the merged

CPG. The rationale for defining such a constraint is to ensure the ordering of certain critical tasks, for instance to avoid adverse interactions between medical actions. The following example shows that task2 from CPG2 should be executed after task1 from CPG1:

```
:c1 a :IdenticalActionConstraint;
  :hasTask1 cpg1:tak1;
  :hasTask2 cpg2:tak2.
```

SimultaneousActionConstraint class is used when task1 and task2 from different CPG should be executed simultaneously. For instance, the task “take Warfarin” in the AF CPG and the task “take beta blockers” in CHF CPG should be executed at the same time under certain conditions. Therefore, during the execution of these two CPG if CHF CPG reaches chf:takeBetaBlocker first but the AF CPG has not yet reached af:takeWarfarin, due to this constraint the execution engine will delay the execution of chf:takeBetaBlocker until the AF CPG catches up. Delaying the chf:takeBetaBlocker should be performed under two circumstances: (1) The delay does not harm the treatment process and (2) The lagging CPG (in this example the AF CPG) is expected to catch up within an acceptable delay. Property *task1CanWaitFor* with the domain of *Constraint* shows how long task1 can be safely delayed for the lagging CPG to catch up. Data type property *takesTimeToReachT2* with the domain of *Constraint* shows how long it is going to take for the lagging CPG to reach task2. Calculating the time that it takes to reach task2 from the current position in the CPG can be based on the average, shortest or longest time that it has taken to reach task2 in previous executions. The object property *hasLogicToCalculateTheTimeToReachT2* with the domain of *Constraint* and range *LogicToCalculateTime* defines the method used for calculation of the value of the property *takesTimeToReachT2*. Instances of the class *LogicToCalculateTime* are shortest, average and longest. Another example of this constraint can be delaying a blood test requested by a CPG and doing it with a blood test requested by another CPG so that unnecessary visits can be avoided. The following example shows simultaneous action constraint between CPG1:bloodWork1 and CPG2:bloodWork2.

The logic to calculate the catching up time of task2 is the average time that has been taken in previous executions in order to go from the current point of execution to task2 in CPG2. We assume this average time is 10 days:

```
:c1 a :SimultaneousActionConstraint;
  :hasTask1 cpg1:tak1;
  :hasTask2 cpg2:tak2;
  :hasLogicToCalculateTheTimeToReachT2 :average;
  :task1CanWaitFor "10"^^xsd:int.
```

Please note that the value of the *hasLogicToCalculateTheTimeToReachT2* property can be calculated more accurately as the number of execution of this CPG increases.

The *CombinationConstraint* class represents the case where two tasks are combined and transformed to a new task. If such a combination of two tasks is feasible, both task1 and task2 will be combined and replaced by the value of the property *hasMergeOutcomeTask* with the domain of *CombinationConstraint* and range of *Task*. This constraint is used to prevent adverse interactions or create more time and cost effective treatments. For instance, two procedures can be replaced by a cheaper and more effective procedure if possible. *SimultaneousActionConstraint* properties *task1CanWaitFor* and *takesTimeToReachT2* are used to define an acceptable time frame for approving the combination of tasks. For instance, if one of the tasks is required to be executed instantaneously while the other CPG is still behind, this merging constraint will be ignored. The following instantiation of the CPG-KPO shows a combination constraint between task1 and task2. This constraint expresses that the outcome of combining these two tasks is task3. Moreover, if task1 becomes active, it can wait up to 10 days until task2 becomes active as well so it can be combined with task1. The shortest recorded execution time between current active point and the task 2 is used as the time that CPG2 needs for activation of task2:

```
:c1 a :CombinationConstraint;
  :hasTask1 cpg1:tak1;
  :hasTask2 cpg2:tak2;
  :hasMergeOutcomeTask cpg1-cpg2:task3;
  :hasLogicToCalculateTheTimeToReachT2 :shortest;
  :task1CanWaitFor "10"^^xsd:int.
```

8.3.1.2. Operational Constraints

OperationalConstraint represent the institution-specific conditions that govern the merging of two tasks across different CPG. For instance, a *SimultaneousActionConstraint* may be ignored if the operational constraints do not allow it.

Data type properties *canBeExecutedAtTheSameTime*, *shouldBeExecutedAtTheSameTime*, *canBeExecutedAtTheSameLocation*, *shouldBeExecutedAtTheSameLocation* with the domain of *OperationalConstraint* and range of xsd:Boolean express these constraints. In order to express that two tasks should or can happen at the same time or location the value "true"^^xsd:Boolean is assigned as the values of one of these properties. For instance, assigning "true"^^xsd:Boolean as the value of the *shouldBeExecutedAtTheSameTime* property means that task1 and task2 **should** be executed at the same **time** in the institution that the merge is happening. Assigning "false"^^xsd:Boolean to these properties has the opposite meaning. For instance, assigning "false"^^xsd:Boolean as the value of the property *shouldBeExecutedAtTheSameLocation* property means that task1 and task2 **should not** be executed at the same location in this institution. For instance, the following constraint expresses that task1 and task2 should be executed at the same location in QEII hospital:

```
:c1_QEII a :OperationalConstraint;
  :shouldBeExecutedAtTheSameLocation "true"^^xsd:Boolean;
  :hasTask1 cpg1:tak1;
  :hasTask2 cpg2:tak2.
```

As another example, the following constraint expresses that task1 and task2 cannot be executed at the same time in QEII hospital:

```
:c1_QEII a :OperationalConstraint;
  :canBeExecutedAtTheSameTime “false”^^xsd:Boolean;
  :hasTask1 cpg1:tak1;
  :hasTask2 cpg2:tak2.
```

8.3.1.3. Temporal Constraints

A merging criterion can be defining temporal constraints between the tasks of the CPG. *TemporalConstraint* is another subclass of the *Constraint* class that represents this concept. A commonly used temporal constraint is called Simple Temporal Network (STN). This temporal constraint defines a temporal window between two tasks during which the second task should be executed after the first task. *STNConstraint* models this temporal constraint. *stnConstraintHasTime1* and *stnConstraintHasTime2* properties with the domain of *STNConstraint* define the lower and upper bounds of the temporal window. Defining more temporal constraints is left as subject of future work. The following example shows that the task2 in cpg2 should be executed between 6 days to 10 days after task1 in cpg1:

```
:c1 a :STNConstraint;
  :stnConstraintHasTime1 “6”^^xsd:int;
  :stnConstraintHasTime2 “10”^^xsd:int;
  :hasTask1 cpg1:tak1;
  :hasTask2 cpg2:tak2.
```

8.3.1.4. Medical Constraints

MedicalConstraint deals with reusing test results and avoiding conflicts and adverse interactions. Its subclasses are *TaskSubstitute* and *UseResultsConstraint*. *TaskSubstitute* constraint indicates that in case there is a conflict between two tasks that appear in two CPG that need to be merged, then either of the conflicting tasks should be replaced with an alternative task to facilitate the merging of the two CPG. If such a situation arises, during CPG execution, task1 is replaced with the value of the property *t1ToBeReplacedWith* with the domain *TaskSubstitute* and range *Task*.

Another medical merging constraint is reusing a specific result of a task from another CPG. This constraint is represented by the *UseResultsConstraint* class. This constraint indicates that task1 should reuse the results of task2 if the result validity period indicated by *hasValidityPeriodT2* has not been yet expired. *useOutcomeOfT2* property with the domain of *UseResultsConstraint* and range of *Outcome* indicates which outcome of task2 is reused by task1. The *Outcome* class represents the medical outcomes in our CPG ontology. The following example shows an instance of the *UseResultsConstraint*. This merging constraint expresses that the value of TSH acquired in CPG2 using task bloodtest2 can be re-used as the outcome of the bloodTest1 in CPG1:

```
:useResultConst1 a :UseResultsConstraint;
    :hasTask1 cpg1:bloodTest1;
    :hasTask2 cpg2:bloodTest2;
    :useOutcomeOfT2 cpg2:TSHValue.
```

bloodTest1 in CPG1 can be skipped if it is only carried out to find the value of TSH.

In our framework, we offer the provision of conditional merging constraints where it is possible to define the satisfaction of *n* out of *m* constraints to pursue CPG merging. The property *hasCondition* with the domain of *Constraint* and range of *Condition* assigns one or more conditions to a constraint. If a condition is satisfied, it belongs to the class *SatisfiedCondition* and an unsatisfied condition belongs to the class *UnsatisfiedCondition*. Subclasses of the condition class are *BooleanCondition*, *ResourceCondition*. Boolean conditions solely check the presence of a fact in patient's data. For instance, having a high blood pressure can be a Boolean condition. Therefore, Boolean conditions' purpose and working are exactly the same as *BooleanCondition* class in CPG-DKO.

ResourceCondition checks the availability of the needed resources for CPG merging. The *Resource* class represents the care resources (human, equipment, space, etc.). Object property *conditionNeedsResource* with the domain of *ResourceCondition* and range of *ResourceType* defines the type of resource that is needed for the satisfaction of the condition. Data type property *numberOfNeededResource* with the domain of *ResourceCondition* indicates the number of resources that is needed for satisfaction of a resource condition. For instance, the following example shows a condition that is checking the existence of two free nurses for a simultaneous action constraint:

```
:rc1 a :ResourceCondition;
  :conditionNeedsResource :nurseResource;
  :numberOfNeededResource “2”^^xsd:int.

:c1 a :SimultaneousActionConstraint;
  :hasTask1 cpg1:task1
  :hasTask2 cpg2:task2;
  :hasCondition :rc1.
```

Object property *hasCardinalityType* with the domain of *Constraint* and range of *Cardinality* assigns the cardinality of the condition satisfaction criterion. Instances of the *Cardinality* class are min, max, exactly, any, all. The data type property *hasCardinalityValue* with the domain of *Constraint* and range of xsd:int defines the cardinality value. For instance, the following instantiation of the CPG-KPO shows a workflow constraint with 3 Boolean conditions from two different CPG and the condition satisfaction criterion of minimum 2 out of 3:

```
:c1 a :WorkflowConstraint;
  :hasCondition cpg1:lowTSH,cpg2:Systolic_GreaterThan_119,
                cpg2:Infant_patient;
  :hasCardinalityType :min;
  :hasCardinalityValue “2”^^xsd:int.

cpg1:lowTSH a :BooleanCondition.
cpg2:Systolic_GreaterThan_119 a :BooleanCondition.
cpg2:Infant_patient a :BooleanCondition.
```

If no values for the property *hasCardinalityType* and *hasCardinalityValue* are entered, the condition satisfaction criteria of "any" is used as the default.

8.3.2. Conflicts

Some constraints might be conflicting such that they both cannot be respected during the merge. Class *Conflict* represents this concept in the ontology. Object property *hasConstraint* with the domain of *Conflict* and range of *Constraint* indicates the conflicting constraints. For instance, the following instantiation of this class expresses that simultaneous constraint simConst and precedence constraint precConst are conflicting and only one of them should be respected during the merge:

```
:conf1 a :Conflict;
      :hasConstraint :simConst;
      :hasConstraint :PrecConst.
```

If any of the constraints is already respected during execution, the other one will be discarded. If two constraints are about to be respected simultaneously, the constraint with more priority will be respected and the other constraint is ignored. Property *hasHighestPriority* property with the domain of *Conflict* and range of *Constraint* determines which constraint should be given priority. For instance, we can express that simConst has more priority in the above example in the following fashion:

```
:conf1 a :Conflict;
      :hasHighestPriority :simConst.
```

If no priority information has been given, the decision on the priority of the constraints can be made based on the time or the cost that can be saved by respecting each of the constraints. The object property *hasPriorityDecisionCriterion* with the domain of *Conflict* and range of *PriorityDecisionCriterion* determines whether cost or time is the basis of defining priority. Instances of the class *PriorityDecisionCriterion* are *timeCriterion* and *costCriterion*. This property can be used to decide about the priority of the conflicting constraints automatically during execution. The following example shows that priority between constraints simConst and precConst should be decided based on the time saving achievable via each of the merges:

```
:conf1 a :Conflict;
      :hasConstraint :simConst;
      :hasConstraint :PrecConst;
:hasPriorityDecisionCriterion :timeCriterion.
```

Please note that more instances can be added to *PriorityDecisionCriterion* class in order to represent a wider set of criteria for priority decision making.

8.3.2.1. Conflict Detection Rules

Several constraints may be conflicting because they may contain inconsistent pieces of knowledge on how two CPG should be merged. For instance, if constraint 1 expresses that task1 should be executed after task2 and constraint 2 is expressing that task2 should be executed after task1, constraints 1 and 2 are conflicting. Identifying conflicts can be a tedious task if performed manually. In order to automate the process of detecting the conflicting constraints, we have identified the possible conflicts that different types of constraint might have with each other. A SWRL rule that can automatically detect conflicts is written for each of the possible identified conflicts is written. Below, we give an example of the SWRL rules written to detect conflicts between *SimultaneousActionConstraint* and *PrecedenceConstraint* instances:

```
PrecedenceConstraint(?pc), SimultaneousActionConstraint(?sac),
hasTask1(?pc, ?t1), hasTask1(?sac, ?t1), hasTask2(?pc, ?t2),
hasTask2(?sac, ?t2) -> constraintIsInConflictWith(?sac, ?pc)
```

8.3.3. Real World Examples

In this section, we see examples of CPG merging for CHF-AF, Transient Ischemic Attack-Duodenal Ulcer, Osteoarthritis-Hypertension, Diabetes-Osteoarthritis and Diabetes-

Hypertension comorbidities. In each example, we describe the CPG merging constraints and explain how CPG-KPO has been instantiated to represent those criteria.

8.3.3.1. CPG Merging Example1: CHF-AF

In this section, we see how an example of *IdenticalActionConstraint* is handled in our CPG merging ontology. We work with two independent CPG, one for Chronic Heart Failure (CHF) and the other for Atrial Fibrillation (AF) that were originally computerized in [107]. In order to merge these two CPG we first transformed the instantiations of these ontologies to CPG-DKO using the ontology mapping technique previously discussed in this thesis. We then instantiated the CPG-KPO in order to capture the morphing constraints between the two transformed CPG. These two CPG share several identical tasks. We have created several *IdenticalActionConstraint*s in order to capture these identical tasks between the two CPG. An example of these tasks is pre-treatment_electrolyte_assessment_&_correction that might be executed in both of the CPG for comorbid patients. Let us assume that the results of this step are valid for 10 days. Our CPG-KPO has been instantiated as follows to capture this merging constraint:

```
:const1 a :IdenticalActionConstraint;
:hasTask1 chf:CHF-AF1 # pre-treatment electrolyte assessment &
correction
 :hasTask2 af:CHF-AF1 # pre-treatment electrolyte assessment &
correction
  :hasValidityPeriodT1 "10"xsd:int;
  :hasValidityPeriodT2 "10"xsd:int.
```

Figure 8.2 shows the instantiation of the CPG-KPO connect AF and CHF CPG.

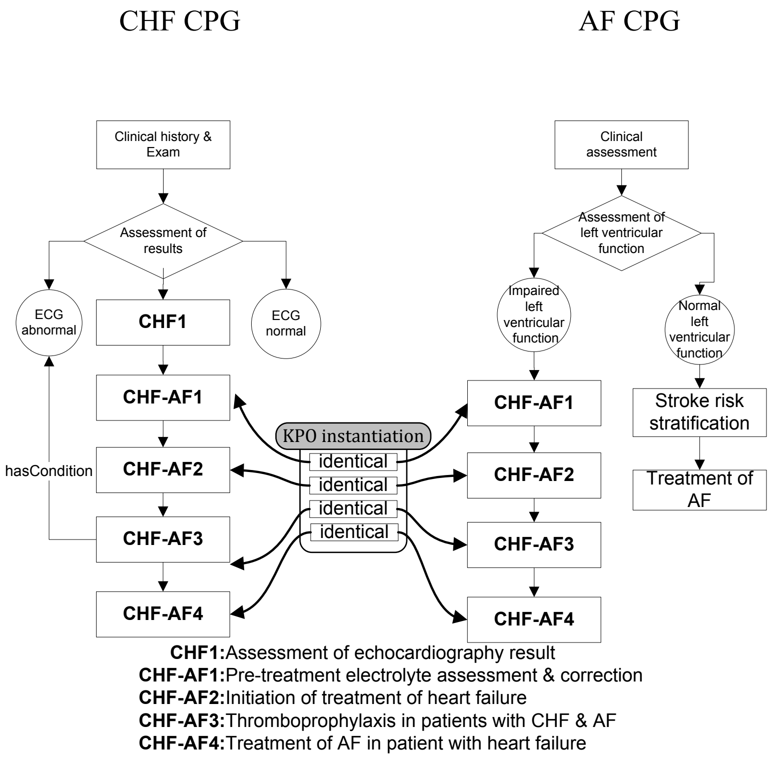

CHF CPG
AF CPG
Clinical history & Exam
Assessment of results
ECG abnormal
CHF1
ECG normal
CHF-AF1
CHF-AF2
CHF-AF3
CHF-AF4
hasCondition
KPO instantiation
identical
identical
identical
identical
Clinical assessment
Assessment of left ventricular function
Impaired left ventricular function
Normal left ventricular function
Stroke risk stratification
Treatment of AF
CHF-AF1
CHF-AF2
CHF-AF3
CHF-AF4
CHF1:Assessment of echocardiography result
CHF-AF1:Pre-treatment electrolyte assessment & correction
CHF-AF2:Initiation of treatment of heart failure
CHF-AF3:Thromboprophylaxis in patients with CHF & AF
CHF-AF4:Treatment of AF in patient with heart failure

Figure 8.2 The instantiation of the CPG-KPO and the parts of the CHF and AF CPG that participate in the merge [107]

8.3.3.2. CPG Merging Example2: Transient Ischemic Attack - Duodenal Ulcer

Transient Ischemic Attack - Duodenal Ulcer comorbidity has been used in [25] as an example for the proposed pre-execution merging algorithm. We computerized these CPG in CPG-DKO based on the provided flowchart in [25] and merged them using CPG-KPO. The only defined merging constraint for this comorbidity is the conflict between the task tia:A (Give Aspirin) from transient ischemic attack and the task dc:AS (Stop taking aspirin if used) from the duodenal ulcer. This constraint is modeled in our CPG-KPO by creating *TaskSubstituteInCaseOfConflict* constraint:

```
:const1  a  :TaskSubstituteInCaseOfConflict;
 :hasTask1 tia:A ;#Give aspirin
 :hasTask2 dc:SA; # Stop taking aspirin if used;
 :t1ToBeReplacedWith tia-dc:DGA;# DO_NOT_GIVE_ASPIRIN. # DGA
```

In the above example, the task tia-dc:DGA ("*do not give aspirin*") will replace the task tia:GA ("*give aspirin*") in the case of a conflict. If no conflict happens, both of the CPG will continue their normal execution flow. Figure 8.2 shows these two CPG and the instantiation of the CPG-KPO that merges them. Please note that if TIA

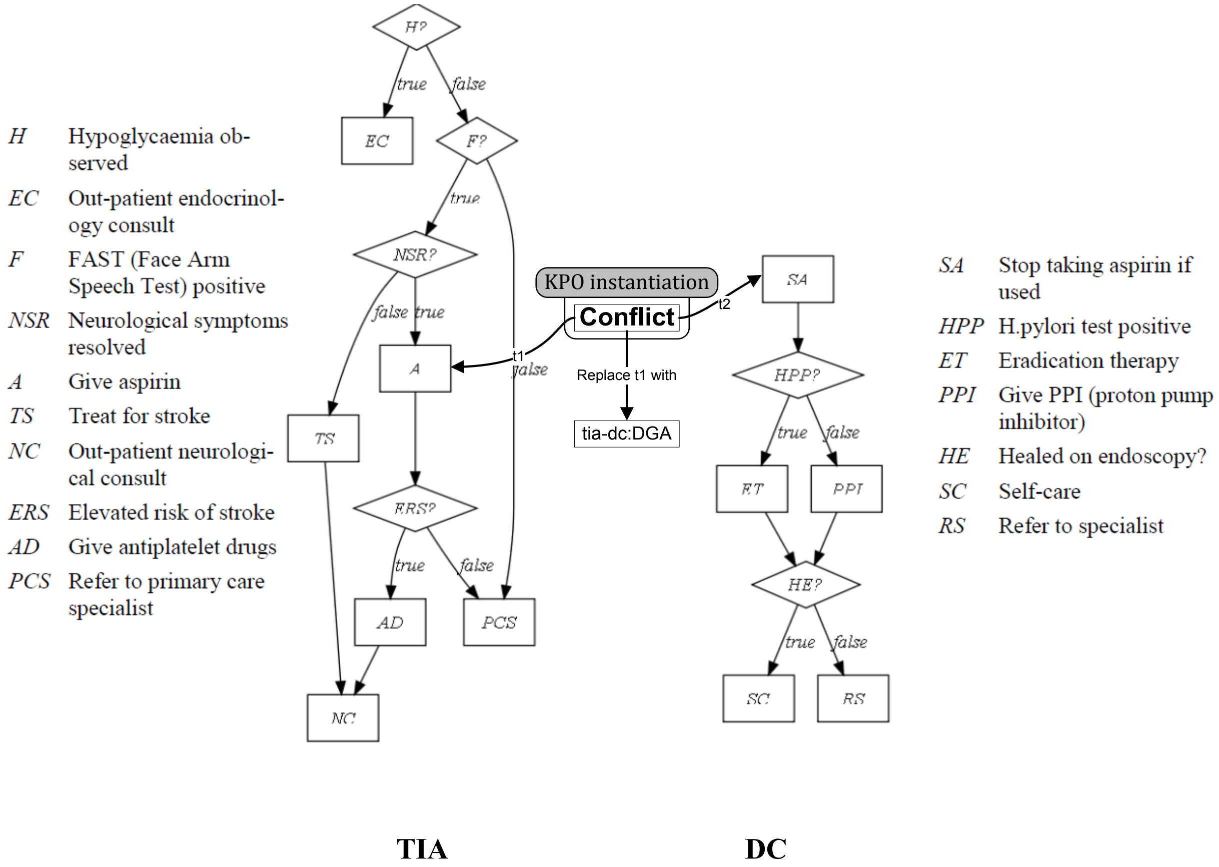

H Hypoglycaemia observed
EC Out-patient endocrinology consult
F FAST (Face Arm Speech Test) positive
NSR Neurological symptoms resolved
A Give aspirin
TS Treat for stroke
NC Out-patient neurological consult
ERS Elevated risk of stroke
AD Give antiplatelet drugs
PCS Refer to primary care specialist
H?
true
false
EC
F?
true
NSR?
false
true
A
TS
false
ERS?
true
false
AD
PCS
NC
KPO instantiation
Conflict
t1
t2
Replace t1 with
tia-dc:DGA
SA
HPP?
true
false
ET
PPI
HE?
true
false
SC
RS
SA Stop taking aspirin if used
HPP H.pylori test positive
ET Eradication therapy
PPI Give PPI (proton pump inhibitor)
HE Healed on endoscopy?
SC Self-care
RS Refer to specialist
TIA
DC

Figure 8.3 The instantiation of the CPG-KPO and the parts of the TIA and DC CPG that participate in the merge [25].

8.3.3.3. CPG Merging Example3: Osteoarthritis-Hypertension – Diabetes

Osteoarthritis, hypertension and diabetes are three diseases that frequently co-occur in comorbid patients. Review of the treatments of these diseases reveals several possible pair-wise conflicts between them. We give three examples of the *TaskSubstituteInCaseOfConflict* in the pair-wise comorbidity of these diseases.

- **Osteoarthritis and Hypertension:**

We will see an example of the *TaskSubstituteInCaseOfConflict* in this section. During the treatment of a patient for osteoarthritis, non-steroidal anti-inflammatory drugs (NSAD) such as aspirin, ibuprofen, and naproxen may be prescribed for the patient. However, these drugs aggravate the patient's hypertension by increasing the blood pressure. The solution is to replace any of these drugs with alternatives such as acetaminophen, tramadol or narcotic analgesics. We reviewed the osteoarthritis treatment algorithm published by National Health Services Tayside of Scotland [169] and computerized it in CPG-DKO. We found three cases of prescribing NSAD drugs in the osteoarthritis treatment algorithm. These three tasks are: (1) add_Ibuprofen_1_point_2g_day, (2) Substitute_Naproxen_Instead_Of_Ibuprofen and (3) inquiry_about_possibility_of_continuing_Ibuprofen_after_6_month. We have created the following three *TaskSubstituteInCaseOfConflict* constraints in order to replace all these tasks with the task RNSAD (replace NSAD with Acetaminophen Tramadol Narcotic analgesics). ht: and os: represent the namespace for the hypertension and the osteoarthritis ontologies respectively.

```
:conflict1 a  :TaskSubstituteInCaseOfConflict;
    :hasTask1 os:AB; #add Ibuprofen 1 point 2g day;
    :hasTask2 ht:hypertension_CPG;
    :t1ToBeReplacedWith os-ht:RNSAD;
```

```
:conflict2 a  :TaskSubstituteInCaseOfConflict;
    :hasTask1 os:Substitute_Naproxen_Instead_Of_Ibuprofen;
    :hasTask2 ht:hypertension_CPG;
    :t1ToBeReplacedWith os-ht:RNSAD.

:conflict3 a  :TaskSubstituteInCaseOfConflict;
    :hasTask1 os:os1;
    :hasTask2 ht:hypertension_CPG;
    :t1ToBeReplacedWith os-ht:RNSAD.


#os-ht:RNSAD = Replace NSAD with Acetaminophen Tramadol Narcotic
narcotic analgesics.

#os:os1 = inquiry about possibility of continuing Ibuprofen after
6 month;
```

Figure 8.4 shows the parts of the osteoarthritis and hypertension pathways that participate in the CPG merging and the instantiation of the CPG-KPO.

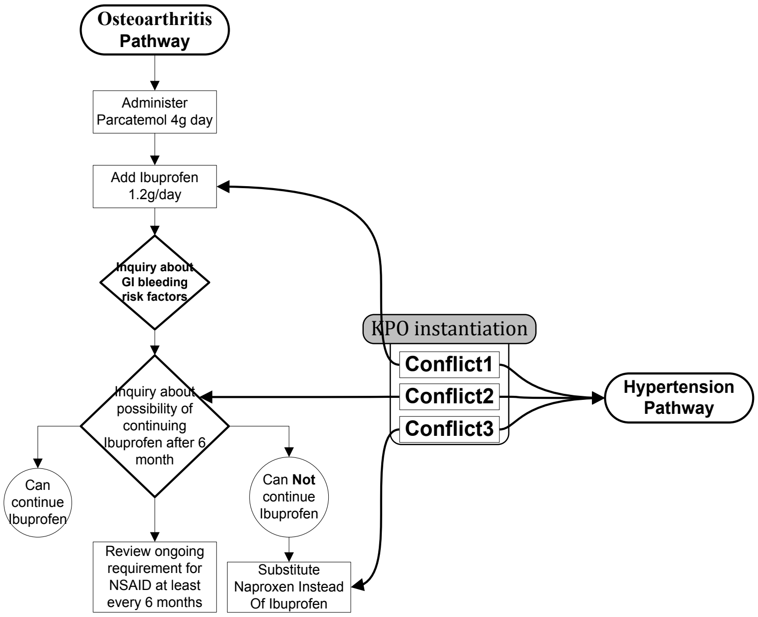

Osteoarthritis Pathway
Administer Parcatemol 4g day
Add Ibuprofen 1.2g/day
Inquiry about GI bleeding risk factors
Inquiry about possibility of continuing Ibuprofen after 6 month
Can continue Ibuprofen
Can Not continue Ibuprofen
Review ongoing requirement for NSAID at least every 6 months
Substitute Naproxen Instead Of Ibuprofen
KPO instantiation
Conflict1
Conflict2
Conflict3
Hypertension Pathway

Figure 8.4 The instantiation of the CPG-KPO and the parts of the osteoarthritis and hypertension pathways that participate in the merge.

- **Diabetes and Hypertension,**

We will see an example of the *TaskSubstituteInCaseOfConflict* in this section. Domain expert believes that prescribing high doses of diuretics for treatment of hypertension can aggravate diabetes by increasing levels of blood sugar. We reviewed and computerized the clinical pathways for hypertension [170] and diabetes [171] published by National Institute of Health and Clinical Excellence of UK. We identified two cases of prescribing diuretics in the diabetes pathway: (1) step_3_ACE_inhibitor_or_low_cost_angiotensin_II_receptor_blocker_PLUS_calcium_channel_blocker_PLUS_thiazide_like_diuretic and (2) step_4_ACE_inhibitor_or_low_cost_angiotensin_II_receptor_blocker_PLUS_calcium_channel_blocker_PLUS_thiazide_like_diuretic_PLUS_consider_further_diuretic_or_alpha_or_beta_blocker_IF_HT_PERSISTENT. These two tasks are replaced with the task reduce_high_dose_Of_thiazide_like_diuretic_to_lower_doses in the diabetes pathway if the patient is also being treated for hypertension. The following instantiation of the CPG-KPO expresses these two merging constraints:

```
:Hypertension_Diabetes_1  a  :TaskSubstituteInCaseOfConflict;
    :hasTask1 ht:step3; #Hypertension
    :hasTask2 db:diabetes_CPG;
    :t1ToBeReplacedWith ht-db:ComorbidityTask1.

:Hypertension_Diabetes_2  a  :TaskSubstituteInCaseOfConflict;
    :hasTask1 ht:step4; #Hypertension
    :hasTask2 db:diabetes_CPG;
    :t1ToBeReplacedWith:ht-db:ComorbidityTask1.

##-----------------------------------------
##Ht:step3 = step 3 ACE inhibitor or low cost angiotensin II
receptor blocker PLUS calcium channel blocker PLUS thiazide like
diuretic;”

##ht-db:ComorbidityTask1 = reduce high dose of thiazide like
diuretic to lower doses.

##ht:Step4 = step 4 ACE inhibitor or low cost angiotensin II
receptor blocker PLUS calcium channel blocker PLUS thiazide like
diuretic PLUS consider further diuretic or alpha or beta blocker
IF HT PERSISTENT
```

- **Diabetes and Osteoarthritis:**

We will see an example of the *TaskSubstituteInCaseOfConflict* in this section. Domain expert believes that prescribing high doses of NSAD in both diabetes and osteoarthritis will increase the risk of bleeding in patients with osteoarthritis. The solution is to replace these NSAD drugs with alternatives such as acetaminophen, tramadol or narcotic analgesics. We reviewed and computerized the hypertension pathway [170] published by National Institute of Health and Clinical Excellence of UK and osteoarthritis treatment algorithm [169] published by National Health Services Tayside of Scotland and computerized them in CPG-DKO. We identified one case of prescribing NSAD drugs in diabetes and two cases of prescribing NSAD drugs in the osteoarthritis pathway. The following two constraints are expressing that the NASD prescribing tasks in diabetes pathway are replaced with the task replace_NSAD_in_Osteoarthritis_with_Acetaminophen_Tramadol_Narcotic_narcotic_analgesics if the osteoarthritis pathway is prescribing NSAD as well. The following instantiation of the CPG-KPO represents these CPG merging constraints:

```
:diabetes_Osteoarthritis_1 a :TaskSubstituteInCaseOfConflict;
  :hasTask1 ht:offer_low-dose_75_mg_daily_aspirin; #Dibates
  :hasTask2 os:add_Ibuprofen_1_point_2g_day; #Osteoarthritis
  :t1ToBeReplacedWith db-os:ComorbidityTask1.

:diabetes_Osteoarthritis_2  a TaskSubstituteInCaseOfConflict;
 :hasTask1 db:offer_low-dose_75_mg_daily_aspirin; #Dibates
 :hasTask2 :osSubstitute_Naproxen_Instead_Of_Ibuprofen;
 :t1ToBeReplacedWith db-os:ComorbidityTask1

# db-os:ComorbidityTask1 = "replace NSAD in Osteoarthritis with
Acetaminophen Tramadol Narcotic narcotic analgesics"
```

8.3.3.4. CPG Merging Example4: CHF-AF

The last example features a *SimultaneousActionConstraint* from CHF-AF comorbidity. During the treatment of the CHF patients, angiotensin-converting-enzyme inhibitors (ACEI) and beta blockers (BB) may be prescribed simultaneously represented by the task initiate_treatment_with_beta_blockers_in_addition_to_ACEI. Moreover, AF patients may need to take Warfarin represented by the task anticoagulation_with_Warfarin to avoid formation of blood clots. Domain experts believe that simultaneous prescription of these drugs has better results in CHF-AF comorbid patients. We have made two assumptions in this example in order to illustrate the capability of CPG-KPO in expressing complex conditions: (1) prescription of the ACEI and BB in CHF patients can be delayed for at most 7 days if the stage of their CHF disease is either one of the stages A, B or C of the American College of Cardiology/American Heart Association working group stages [172]; (2) This task can be delayed for 24 hours if the patient is in stage D of the American heart association stages. The following instantiation of the CPG-KPO represents these two CPG merging constraints:

```
:simacconst1  a :SimultaneousActionConstraint;
  :hasTask1 chf:Initiate_BB_and_ACEI;
  :hasTask2 af:anticoagulation_with_Warfarin;
  :task1CanWait "7"^^xsd:int. #7 days
  :hasCondition af:AFStageA, af:AFStageB, af:AFStageC;
  :hasCardinalityType :any.


:simacconst2  a :SimultaneousActionConstraint;
  :hasTask1 chf:Initiate_BB_and_ACEI;
  :hasTask2 af:anticoagulation_with_Warfarin;
  :task1CanWait "1"^^xsd:int; #one day
  :hasCondition chf:AFStageD;
  :hasCardinalityType :any.
```

```
#chf:Initiate_BB_and_ACEI = Initiate treatment with beta blockers
in addition to ACEI;
```

On the other hand, if we assume that prescription of Warfarin for AF patients can be delayed for 3 days if patient is in one of the Paroxysmal AF or Persistent AF stages, this constraint can be represented by the following instantiation of the CPG-KPO :

```
:simacconst3  a :SimultaneousActionConstraint;
  :hasTask1 af:anticoagulation_with_Warfarin;
  :hasTask2 chf:Initiate_BB_and_ACEI;
  :task1CanWait "3"^^xsd:int; #three days
  :hasCondition af:paroxysmal_AF, af:persistent AF ;
  :hasCondition chf:AFStageD;
  :hasCardinalityType :any.

#chf:Initiate_BB_and_ACEI = Initiate treatment with beta blockers
in addition to ACEI;
```

Figure 8.5 shows the abovementioned three merging constraints between the AF and CHF CPG and the parts of the CPG that participate in the merging. It also shows the instantiation of the CPG-KPO for CPG merging.

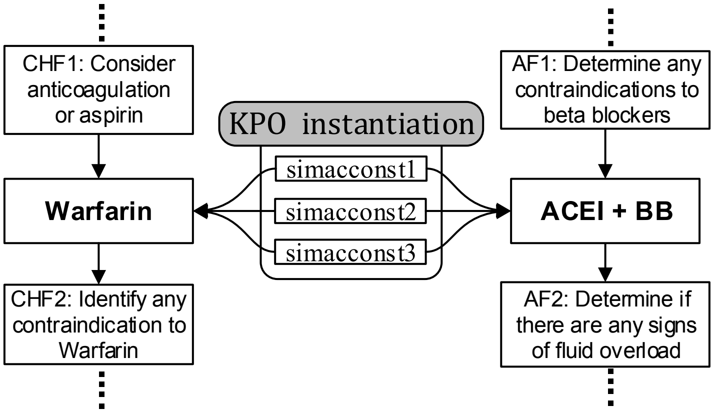

CHF1: Consider anticoagulation or aspirin
Warfarin
CHF2: Identify any contraindication to Warfarin
KPO instantiation
simacconst1
simacconst2
simacconst3
AF1: Determine any contraindications to beta blockers
ACEI + BB
AF2: Determine if there are any signs of fluid overload

Figure 8.5 The instantiation of the CPG-KPO and the parts of the CHF and AF CPG that participate in the merge.

8.4. CPG Knowledge Morphing Execution Engine

All of the existing execution engines including our OWL-based CPG execution engine function based on the concept of the state transition systems and are designed to execute a single CPG rather than several CPG concurrently. A state transition system is a graph in which the nodes are the states and the arcs represent the transitions between the states. The goal of creating a state transformation system in CPG execution engines is to simulate the tasks' states in the real world and how they change their state as a result of treatments or changes in the patients' state.

Similar to tasks, merging constraints considered by physicians go through several states in the real world while patients are treated. Moreover, state transitions in constraints may cause state transitions in tasks too. These state transitions in tasks are actually the modifications that are needed to be performed on medical actions recommended in each of the CPG in order to avoid duplications and conflicts. For instance, the activation of a constraint that deals with a conflict may lead to discarding the troublesome task. Hence, in order to successfully execute several CPG related to a comorbidity we need to model (a) state transition system of medical tasks in the CPG ontology; (b) state transition system of merging constraint in the CPG-KPO and (c) the effect of state transitions in constraints on state transitions in tasks. We augment our OWL-based CPG execution engine by adding two sets of OWL axioms and SWRL rules to the CPG execution engine in order to model items b and c from the list above, as explained in sections 8.4.1 and 8.4.2 respectively. Please note that our execution engine is already capable of handling item a.

8.4.1. Constraint State Transition

During the CPG execution, constraints go through the *InactiveConstraint*, *activeConstraint*, *StartedConstraint*, *DiscardedConstraint* and *CompletedConstraint* states. Figure 8.6 shows the possible state transitions for constraints during execution.

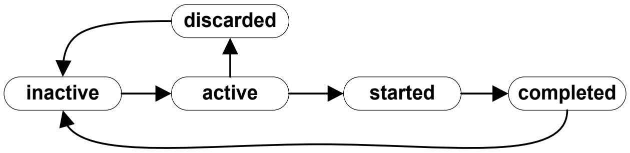


Figure 8.6 Constraints' state transitions

Instances of classes *ActiveConstraint*, *InactiveConstraint*, *CompletedConstraint*, *StartedConstraint* and *DiscardedConstraint* have the following values for the property *hasConstraintState* with the domain of *Constraint* and range of *ConstraintState*: active, inactive, completed, sarted and discarded respectively by the following OWL construct and other similar ones.

```
[a owl:Restriction ;
 owl:onProperty :hasConstraintState ;
 owl:hasValue :active]
 owl:equivalentClass :ActiveConstraint.
```

Please note that the default name space ":" represents CPG-KPO in this section of the thesis. In the rest of this section, we describe what scenarios trigger the state transitions shown in Figure 8.6 and what OWL triples and SWRL rules have been written to support the execution of them in our OWL based CPG execution engine for *SimultaneousActionConstraint*. Please note that the inferred values of the property *constraintHasStateNew* with the domain of *Constraint* and range of *ConstraintState* by the reasoner shows the new states of the constraints that should be assigned to them by the execution engine.

- ***InactiveConstraint → ActiveConstraint*:**

All of the constraints start from the inactive state. As long as the natural flow of execution of the CPG in the execution engine respects a constraint, it does not leave this state. A constraint goes to the active state from the inactive state when (a) the constraint is about to be violated and (b) the conditions of the constraint is satisfied. Since each constraint may be violated by a different combination of CPG tasks states, they all have their own specific rules for activation. Please note that a task goes to the *StartedTask* in the CPG execution engine when it is being executed. *SimultaneousActionConstraint* is violated if one the merged tasks is started while the other one is still inactive. Therefore, this constraint becomes active if (a) it is inactive, (b) its conditions are satisfied and (c) the first merged task is started, and the second merged task is still inactive and the started task has enough time to wait for the inactive task to catch up for simultaneous execution. This means the value of *task1CanWaitFor* is greater than *takesTimeToReachT2*. The following SWRL rule represents this state transition rule:

```
SimultaneousActionConstraint(?const) ^
hasConstraintState(?const,:InactiveConstraint)^
ConstraintWithSatisfiedCondition(?const) ^ hasTask1(?const,?t1) ^
hasTaskState(?t1,:StartedTask) ^ hasTask2(?const,?t2) ^
hasTaskState(?t2,:StartedTask) ^
takesTimeToReachT2(?const,?timeToT2) ^
task1CanWait(?const,?canWaitt1) ^
```

```
swrlb:greaterThan(?canWaitt1,?timeToT2) →
constraintHasStateNew(?const,:ActiveConstraint)
```

- ***ActiveConstraint* → *StartedConstraint*:**

An active constraint can potentially change the execution states of the tasks in the merged CPG in order to stop the execution engine from violating it, if (1) all the constraints with more priority are either inactive or discarded or (2) all the constraints with less priority are in inactive, active or discarded state. As we will see in the next state transition rule, the constraints with less priority in the active state will be discarded upon activation of a constraint with more priority. *constrainthasLessPriorityThan* which is the inverse of *constraintHasPriorityOver* with the domain and range of *Constraint* can be used to express the priority of conflicting constraint in case that both of them are active or started. Both of these properties are sub properties of the *constraintIsInConflictWith* property described in section 8.3.2.1.

- ***ActiveConstraint* → *DiscardedState*:**

An active constraint will be discarded if (1) Any of the conflicting constraint regardless of its priority has already been started or completed or (2) A constraint with more priority is active. A discarded constraint will not be able to have an effect on the states of the tasks in the CPG in order to avoid its violation.

- *StartedConstraint* → *CompletedConstraint*:

If all of the involved tasks in a constraint are completed or discarded the constraint is considered a completed constraint as it will not be able to have any further effects on the state of the tasks. As we described in the previous state transition rule, in the rest of the execution, the conflicting constraints with a completed constraint will be discarded.

- *CompletedConstraint* and *DiscardedConstraint* → *InactiveConstraint*:

If all of the tasks merged by a constraint become *InactiveTask* to be re-executed in a loop, the constraint goes to the *InactiveConstraint* state so that the tasks can be re-merged.

8.4.2. Effect of Constraint State Transitions on Tasks' States

In our OWL based CPG execution engine, tasks go through states *InactiveTask*, *ActiveTask*, *StartedTask*, *CompletedTask* and *DiscardedTask* during execution. We also add a *PendingTask* state to this engine to support execution of constraints that is only used by the constraints. As we discussed previously, each type of constraint that goes to the *StartedConstraint* state is about to be violated and has its own special way of avoiding its violation by triggering state transitions in tasks. These state transitions in tasks are modification of the medical actions reasoned in individual CPG. We explain these rules for *SimultaneousActionConstraint* class. When a *SimultaneousActionConstraint* is started, it will force the task ahead to go to the pending state if (1) the constraint is in started state and (2) the behind task is in one of inactive, pending or active states and (3) the behind task is going to catch-up in an acceptable time frame for the task in front (value of the property *hasTimeToReachT2* < value of the property *task1CanWait*):

```
IPATask(?t2), SimultaneousActionConstraint(?const),
StartedConstraint(?const), hasTaskState(?t1, started),
hasTask1(?const, ?t1), hasTask2(?const, ?t2),
hasTimeToReachT2(?const, ?timeToReacht2), task1CanWait(?const,
?canWait1), swrlb:greaterThan(?canWait1, ?timeToReacht2) ->
hasTaskStateNewCandidate(?t1, pending)
:IPATask a [owl:unionOf(:InactiveTask :ActiveTask :PendingTask)].
```

This pending task will go the started state when the behind task is started as well so they both can be executed at the same time and make the execution engine respect the constraint.

```
StartedConstraint(?const),SimultaneousActionConstraint(?const),
hasTask1(?const, ?t1), hasTask2(?const, ?t2),
PendingTask(?t1), StartedTask(?t2), hasTimeToReachT2(?const,
?timeToReacht2), task1CanWait(?const, ?canWait1),
swrlb:greaterThan(?canWait1, ?timeToReacht2) ->
hasTaskStateNewCandidate(?t1, started)
```

All other constraints have similar rules that make the necessary state transitions the task state transition to force execution engine to respect them.

8.4.3. CPG Merging Execution Algorithm

In order to merge several CPG, we feed the computerized CPG, the instantiation of the CPG-KPO and the constraints' state transition rules and their effects of tasks' state transition rules to our OWL-based CPG execution engine. The SWRL rules and the OWL axioms written for modeling the constraints' state transition models and their effects on the executional states of the tasks are used by our OWL-DL + SWRL based execution engine to generate therapy plans for several CPG modeled in CPG-DKO. Table 8.1 shows the execution algorithm from section 6.2.1 of the thesis with the new steps added to handle the merging constraints:

Table 8.1 CPG merging execution algorithm

Step 1: Load the instantiated CPG ontology to the OWL reasoner and activate the first task in the CPG.

Step 2: Query the ontology for the active tasks and show them to the user.

Step 3: Wait for the user to start one or more active tasks. Record the start of tasks by asserting triples into the CPG ontology.

Step 4: Wait for the user to complete one or more started tasks. Record the completion of tasks and their outcomes.

Step 5: Perform reasoning on the ontology to calculate the candidate values for the new states of the tasks and **the new states of the constraints**. The new states of the constraints are the values of the *constraintHasStateNew* property.

Step 6: Apply the new states of tasks and **constraints** that are inferred in the previous step to them. If a task or constraint state change has happened go to step 5 otherwise go to step 2.

As you can see, this CPG merging execution algorithm is the same as the execution algorithm described in section 6.2.1 with the difference that (1) step 5 finds the new states of constraints in addition to the new states of the tasks and the effect of constraints states on the state transitions of the tasks and (2) step 6 assigns the new states to constraints besides assigning the new states of the tasks. If a constraint has a value for the property *constraintHasStateNew*, this value is used as the new state of the constraint by assigning it as the value of the property *hasConstraintState.* The previous value of this property and the value of the *constraintHasStateNew* are removed from the ontology in the process. For instance, if a constraint has the value active for *constraintHasStateNew* property, the values of the properties *hasConstraintState* and *constraintHasStateNew* are deleted and active is assigned as the value of the property *hasConstraintState.* Please note that modification of the medical actions in individual CPG happens in the step 6 of the abovementioned algorithm by assigning new states to them.

8.5. Conclusion

In this chapter, we identified a new set of potentially useful CPG merging constraints. These new merging constraints can capture complex workflow, temporal and operational constraints between CPG tasks and enable an execution engine to merge two or more CPG dynamically. We also developed an OWL-DL ontology called CPG-KPO in order to represent these new merging constraints as well as the previously identified CPG merging constraints in the related literature. CPG-KPO is the first expressive ontological approach to represent CPG merging constraints. Table 8.2 summarizes the purpose of each of the constraints described in this chapter in the CPG merging application.

Table 8.2 Aspects of CPG merging that can be captured by each of the merging constraints (-+ means partial coverage of the merging aspect)

	Avoiding Duplication	**Temporal and sequential constraint**	**Avoiding conflicts**	**Representing operational constraints**
Identical Action Constraint	+			
Precedence Constraint		+	+	+
Combination Constraint			+	
Simultaneous Constraint	-+	+	-+	-+
Operational Location Constraint				+
Operational Time Constraint		+		+
Temporal Constraints		+	+	+

Another novel aspect of our research is to define the formal semantics of the merging constraints in OWL and SWRL. We modified the CPG execution algorithm in Table 3.1 so that it can utilize the CPG-DKO and CPG-KPO formal semantics in an OWL reasoner and merge CPG during execution. On the other hand, in the existing CPG merging frameworks, CPG are modified and merged pre-execution and there is no flexibility during CPG execution. As an example, in pre-execution merge, if two tasks are identified as identical and merged before execution, there will be only one execution of the these tasks during execution even though it might not be possible to merge them due the temporal constraints. However, our framework decides if it is possible to merge these two tasks according to the flow of the execution in CPG on the fly. Therefore, our framework is the first execution-time CPG merging framework.

In the existing CPG merging frameworks, merged CPG are modified and can no longer be used for single disease decision support. On the contrary, our approach purports a clear separation between the CPG and the morphing constraints. In this way, one ontological representation of CPG can be used in single disease CDSS or be merged by an instantiation of the CPG-KPO in a CPG merging framework.

A limitation of our work is lack of evaluations for medical validity of (1) the CPG merging constraints and (2) the generated recommendations for comorbidities using our CPG merging framework. Moreover, we had only a few real examples to work with. Therefore, we evaluated the CPG-KPO and the CPG merging engine using few real examples and a large number of imaginary test cases that do not necessarily reflect the complex nature of real CPG merging. Furthermore, CPG-KPO should be instantiated manually, which can be a labour intensive task. Currently, a constraint in CPG-KPO can only merge two tasks. For instance, to indicate that m tasks should be executed simultaneously, total of $m \times (m-1)/2$ constraints should be created. As a future work, we can define constraints that can merge more than two tasks. This improvement makes CPG-KPO instantiations smaller and easier to create while merging more than two CPG. An important aspect that should always be taken into consideration while treating comorbidies is drug interactions. We believe that our framework needs a drug interaction knowledge source that can capture drug-action and dug-drug interactions. Moreover, the knowledge encapsulated in this knowledge source should be used in the execution-time CPG merging engine to improve the generate recommendations for comorbid patients.

CHAPTER 9: EVALUATION

In this section, we evaluate our knowledge morphing framework for the task of CPG merging. To evaluate our framework, we evaluate ontologies and algorithms developed in our problem specific knowledge morphing framework. We have developed two problem-specific ontologies CPG-DKO (see chapter 4) and CPG-KPO (see section 8.3) and a problem-independent ontology KMO (see chapter 7) in our knowledge morphing framework. We have also developed four algorithms in our framework namely; 1. CPG-DKO and CPG-KPO preprocessing algorithm (see section 6.2.1.2), 2. KMO to OWL+SWRL translation (see section 7.4), 3. CPG Execution algorithm (see section 6.2.1.3) and 4. CPG merging execution algorithm (see section 8.4.3). In this section of the thesis, we evaluate ontologies one by one and then we evaluate the output of their associated algorithms for real world and imaginary input sets.

Ontologies of our framework are used for both knowledge representation and reasoning in OWL reasoners in order to accomplish the necessary steps for merging CPG. To use an ontology for reasoning, it should be *consistent*. An ontology can be regarded as logically consistent if it is satisfiable, which means that it does not contain contradictory information [173]. We have used pellet reasoner to verify the consistency all of the designed ontologies. To use an ontology for knowledge representation, it should be *syntactically correct*. Syntactical correctness can be evaluated using OWL parsers. Both Pellet and protégé have OWL parsers that can be used for syntactical correctness evaluation. Computational complexity of the reasoning process on an ontology is another aspect that plays a prime role in usability of ontologies. We check the species of ontologies of our framework, before, during and after being used in our algorithms. Since OWL-Full is not decidable, we need our ontologies to belong to either OWL-lite or OWL-DL. Moreover, an ontology should be *complete* in order to be successfully utilized for knowledge sharing and semantic interoperability. An ontology is complete "if and only if all the knowledge that is supposed to be in the ontology is explicitly stated in it, or can be inferred" [173].

Approaches reported in the literature for evaluating the completeness of ontologies usually belong to one of the following four categories [174]:

i. Methods based on comparing the ontology to a golden standard ontology:

To evaluate an ontology in this method, a number of metrics are extracted from the ontology and compared to those of a golden standard ontology. However, there are no golden standard ontologies for any of the ontologies that we have developed in this thesis. The only ontology that has a number of counterpart ontologies in OWL is CPG-DKO. We extract and compare a number of metrics proposed in [178] in order to get a sense of quality of our ontology compared to the existing ontologies along these metrics. However, this comparison is far from an ideal comparison because these ontologies are used for modelling both medical knowledge and workflow structure of the CPG whereas our ontology is only capable of modelling the workflow structure of CPG. For the ontologies that there are no equivalent OWL ontologies, we merely report the extracted metrics and discuss the interesting aspects of them. These metrics are reported in section 9.4 of the thesis.

ii. Methods based on using the ontology in an application and evaluating the results:

Ontologies are used in several ontology-driven applications as the knowledge source. The quality of the outcome of these applications can be used for evaluation of both completeness of the ontology and correctness of the application. Therefore, we can use this evaluation method to evaluate both our ontologies and their associated algorithms.

iii. Methods based on existing body of literature describing what an ontology related to a specific domain should be able to capture:

Albeit the golden standard ontology does not always exist, there might a body of literature on the necessary features of these ontologies. Features of the evaluated ontology can be compared against the list of the desired features defined by the related literature to measure the completeness of the ontology. This body of literature exists for KMO and CPG-DKO. However, almost no research has been performed in order to identify a list of necessary features for CPG-KPO. Therefore, we only use this approach to evaluate the completeness of CPG-DKO and KMO.

iv. Methods based on evaluation by domain experts

Domain experts familiar with knowledge management, ontologies and semantic web technologies can be the judge of completeness of an ontology. They can point out to the strength and weak points of an ontology. We use this approach for evaluation of the CPG-KPO.

9.1. Evaluation of CPG-DKO and the OWL-Based CPG Execution Engine

We used OWL reasoners to evaluate the CPG-DKO for consistency. Pellet reasoner shows that our ontology is consistent without any instantiations. We also checked the ontology after instantiation for modeling any of the executed CPG. Our evaluations show that our ontology remains consistent after pre-processing and during and after execution. Moreover, in all instantiations of the CPG, the CPG-DKO ontology remained in the OWL specie that was used for execution. For instance, if an ontology was executed by the OWL-DL execution the instantiation of the CPG-DKO remained in OWL-DL in all steps of the execution. This ensures decidability and efficiency of the reasoning.

In order to evaluate the completeness of CPG-DKO we (1) compare the features of CPG-DKO with the necessary feature of these languages identified in the literature in section 9.1.1. The representation needs of this ontology for representing the workflow structure of the CPG is our focus. This comparison shows that our domain knowledge ontology is one the most expressive ontologies of the CPG domain area.

We also executed several computerized CPG using our OWL-based CPG execution engine in order to use the outcome for evaluation of both the engine and CPG-DKO. We monitored and recorded the state transitions of tasks during execution. We evaluate these state transitions in chapter 9. We also used the generate recommendations by our execution engine and other execution engines for the same CPG in section 9.1.3. In order to evaluate the generated recommendations for one of the CPG that was not executable using any other execution engine we asked the domain expert to comment on them. We discuss this in section 9.1.4.

We developed several patient scenarios for each of the executed computerized CPG to be used for evaluation. These imaginary patient scenarios are created in order to evaluate the capability of the CPG execution engine in execution of all of the existing paths in a CPG. To make sure that every single path is covered in the CPG, we have created enough number of patient scenarios by creating different combinations of values used in the decision making point of the CPG. Therefore, each patient scenario is actually a sequence of the values for the decision variables. For instance, imagine we are designing imaginary patient scenarios for the small CPG in Figure 9.1.

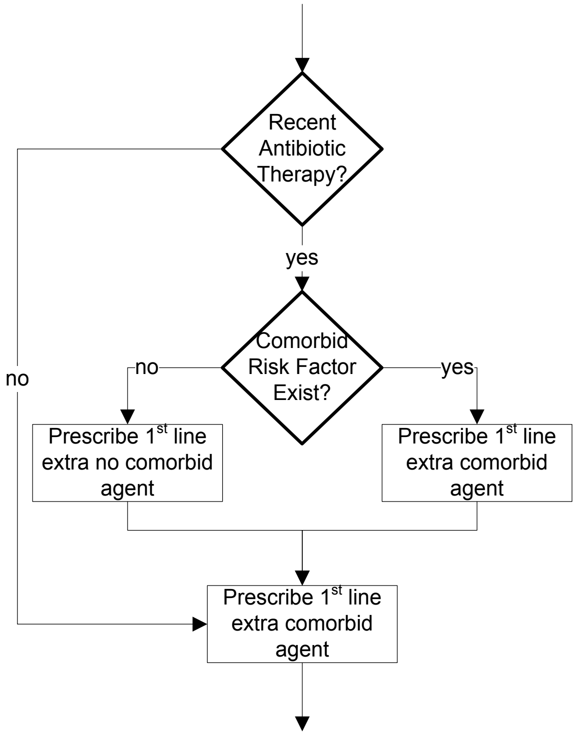

Recent Antibiotic Therapy?
yes
no
Comorbid Risk Factor Exist?
no
yes
Prescribe 1st line extra no comorbid agent
Prescribe 1st line extra comorbid agent
Prescribe 1st line extra comorbid agent

Figure 9.1 A part of the CAP CPG workflow

Table 9.1 shows the scenarios used to cause the execution flow to go to all of the possible paths of this part of the CPG.

Table 9.1 Two examples of the patient scenario for the CAP CPG ontology

	Variables' Values	
	Recent Antibiotic Therapy?	Comorbid Risk Factor Exist?
Scenario 1	Yes	Yes
Scenario 2	Yes	No

As you can see in the above table the only value that we consider for the output variable of the question “Recent Antibiotic Therapy” in our imaginary patient scenarios is “yes” because answering no will cause the execution engine to skip the execution path that contains the tasks “Prescribe 1st line extra No comorbid Agent” and “Prescribe 1st Line Extra Comorbid Agent”. If we assume a CPG contains n Boolean variables used in its decision making process we do not necessarily need to make 2^n different patient scenarios for 2 reasons: (1) Several different combinations of values of variable may lead to execution of the same paths in the CPG, (2) A combination may activate several paths of the CPG. Therefore, the number of different combinations in order to go through all the paths of the CPG is far less than 2^n.

For evaluation of the expression ontology, we created an example function for each of the representable operators, fed the function with two different input sets, and evaluated the output. In all cases. the functions were producing the correct outputs for all the inputs.

9.1.1. Evaluation by Comparison with the Features Identified in the Related Literature

An approach for completeness evaluation of an ontology is to compare its main features with the identified requirements in the related literature. Since our focus is to capture the workflow structure and the medical knowledge that participates in the workflow interpretation of CPG, we compared the support for the workflow patterns in CPG-DKO and the commonly used CPG representation languages.

As we described in section 4.3, Mulyar et al. [101] reviewed Asbru, GLIF, EON and PRO*forma* CPG representation languages in order to identify the presence of 43 classic workflow patterns. Workflow patterns in [101] are categorized under 7 classes which have been previously identified by Russel et al. [102]. As we saw previously in Table 4.1, Russel’s categorization has the following classes: (1) Basic control-flow patterns, (2) Advanced branching and synchronization, (3) Structural patterns, (4) Multiple instances patterns, (5) State-based patterns, (6) Cancellation patterns and (7) New patterns. Albeit the authors of [101] have provided a thorough comparison CPG modeling languages from a workflow pattern perspective, they have not elaborated on different decision models that exist in these languages. These decision models that play an important role in controlling the flow of the executions of CPG are surveyed by Peleg et al. [88][103]. Other than Asbru, EON, GLIF and PRO*forma* that are already reviewed in [88][101][103], we also reviewed the CPG ontologies which are developed by members of the NICHE research group and GASTON and used the results in the comparison. Table 9.2 compares the workflow patterns supported by CPG-DKO against the patterns supported by the existing CPG representation languages:

Table 9.2 Comparison of support for workflow patterns in Asbru, EON, GLIF, PRO*forma*, GASTON and CPG representation languages that are developed by NICHE research group members and CPG-DKO. +, - and ± mean full support, no support and partial support respectively.

Workflow pattern	**Asbru**	**EON**	**GLIF**	**PRO*forma***	**GASTON**	**NICHE[2]**	**CPG-DKO**
Basic Control-Flow							
1. Sequence	+	+	+	+	+	+	+
2. Preconditions	+	+	+	+	+	+	+
3. Nesting of guidelines	+	+	+	+	+	+	+
4. Parallel Split	+	+	+	+	+	+	+
5. Synchronization	+	+	+	+	+	+	+
6. Exclusive Choice	+	+	+	+	+	+	+
Advanced Branching and synchronization	**Asbru**	**EON**	**GLIF**	**PRO*forma***	**GASTON**	**NICHE**	**CPG-DKO**
7. Multi choice	+	+	+	+	+	+	+
8. Structured synchronization merge	±	-	-	-	+	+	+
9. Structured discriminator	+	+	+	+	+	+	+

[2] NICHE represents all the CPG representation ontologies developed in NICHE research group

Structural Patterns	**Asbru**	**EON**	**GLIF**	**PRO*forma***	**GASTON**	**NICHE**	**CPG-DKO**
10. Arbitrary cycle	-	+	+	-	-	+	+
11. Implicit Termination	+	+	+	+	+	+	+
State based patterns	**Asbru**	**EON**	**GLIF**	**PRO*forma***	**GASTON**	**NICHE**	**CPG-DKO**
12. Deferred choice	+	-	+	+	-	+	+
13. Interleaved parallel routing	+	-	-	-	-	-	+
14. Milestone	-	-	-	+	-	-	+
Cancellation Patterns	**Asbru**	**EON**	**GLIF**	**PRO*forma***	**GASTON**	**NICHE**	**CPG-DKO**
15. Cancel activity	+	+	+	+	-	+	+
16. Cancel case	+	-	±	+	-	+	+
New Patterns	**Asbru**	**EON**	**GLIF**	**PRO*forma***	**GASTON**	**NICHE**	**CPG-DKO**
17. Structured loop	+	+	+	+	-	+	+
18. Transient trigger	-	-	-	+	-	-	+
19. Persistent trigger	-	-	+	+	-	-	+
20. Cancel multiple instance activity	+	-	+	+	-	-	+
21. Completed multiple	+	-	-	+	-	-	+

instance activity							
22. Cancelling discriminator	+	-	-	+	-	+	+
23. Structured N-out-of-M join	+	-	+	+	+	+	+
24. Cancelling N-out-of-M join	-	-	-	+	-	+	+
25. Local synchronizing merge	-	-	-	+	+	+	+
26. Critical section	+	-	+	-	-	-	+
27. Interleaved routing	+	-	+	-	-	-	+
Decision Mechanisms	**Asbru**	**EON**	**GLIF**	**PROforma**	**GASTON**	**NICHE**	**CPG-DKO**
28. If-then-else	+	+	+	+	+	±	+
29. Switch	+	+	+	+	-	-	+
30. Argumentation rules	-	+	+	+	-	-	+
31. Preference for options	+	-	-	+	-	-	+

As we can see in Table 9.2, our CPG ontology is capable of modeling every important workflow patterns or decision mechanism that is supported by any of the commonly used CPG representation languages. In order to obtain a quantitative measure for comparison, we counted the *workflow score* (number of supported workflow patterns) and compared them in Table 9.3. + counts as one and ± counts as a 0.5 for calculating the workflow score. This table shows that based on the support for the workflow patterns we can rank the CPG representation languages follows: CPG-DKO > PROforma > Asbru > GLIF >NICHE > EON > GASTON.

Table 9.3 Workflow score of CPG-DKO, Asbru, EON, GLIF and NICHE ontologies

	CPG-DKO	**Asbru**	**EON**	**GLIF**	**PRO*forma***	**GASTON**	**NICHE[3]**
Workflow Score	32	24.5	16	22.5	27	14	21.5

9.1.2. Monitoring the Execution of the Existing CPG

Another way for evaluation of the CPG execution engine and CPG-DKO is to monitor the execution of different patient scenarios and compare them to the intended executional behavior dictated by the execution semantics. For instance, execution semantics dictate that if all the conditions of a task are unsatisfied, that task needs to be discarded. If during the execution of a patient scenario this expected state transition does not happen we can conclude that either one of CPG-DKO or the CPG execution engine is faulty. Therefore, we monitor the state transitions in execution of several CPG and compare them with the desired state transitions dictated by the execution semantics. We go through some examples of these scenarios and discuss how each of the state transitions recorded during each step of the execution is consistent with the intended execution semantics. In our examples of execution which have been used for evaluation, if the number of tasks is less than 10, we use tables to describe several execution scenarios by listing the states of the tasks at each

[3] NICHE represents all the CPG representation ontologies developed in NICHE research group

step of the execution. If the number of tasks is more than that and as a result all the tasks cannot be put in a single table we annotate the CPG diagram with the states and their associated time steps. We use a logical clock to keep track of time: Logical time is zero at the beginning and increases when a user either starts a task (step 3 of execution algorithm in section 6.2.1.3) or completes a task (step 4 of execution algorithm in section 6.2.1.3) and the reasoning happens in between two logical times. Each logical time is considered as a step.

We monitored the execution of the CAP, CHF-AF and Nursing CPG ontologies that were transformed to instances of CPG-DKO during execution of manually crated patient scenarios that cover all the paths of the CPG. We review examples of execution of these CPG in this section in order to show how we evaluated the OWL-DL, OWL-2 and SWRL-based CPG execution engines.

Scenario 1. CHF pathway entry point #5 Initiation of treatment of HF

We choose our first example from the CHF-AF ontology as it is one of the most detailed and expressive ontology representation language and the instantiations of it make use of complex workflow structures. CHF-AF ontology is composed of several independent parts. The Chronic Heart Failure (CHF) treatment algorithm corresponds to the therapies outlined in Canadian CPG [175]. The treatment schedule (Figure 9.2) depicts temporal and spatial relationships between various prescriptions and the activities related to prescriptions such as checking for the drug indications and contraindications. Once a particular drug is prescribed, it has to be up-titrated based on a specific schedule. The drug up-titration schemas along with dosage information for medications are nested within each prescription and are developed as separate algorithms (not shown in Figure 9.2). The algorithm comprises of four decision steps each enabling the clinician to evaluate the presence or absence of any contraindication to a particular medication to ensure that the therapy can be safely followed. The computerized version of this CPG and the figure used in this scenario are prepared in [107]. We have mapped the CPG ontology in [107] and transformed its instances to CPG-DKO for execution. Figure 9.2 shows an execution scenario that has been used for evaluation. Each task and condition has been annotated with its state at each specific point in time. A task or a condition retains its state until a new state has been applied to it.

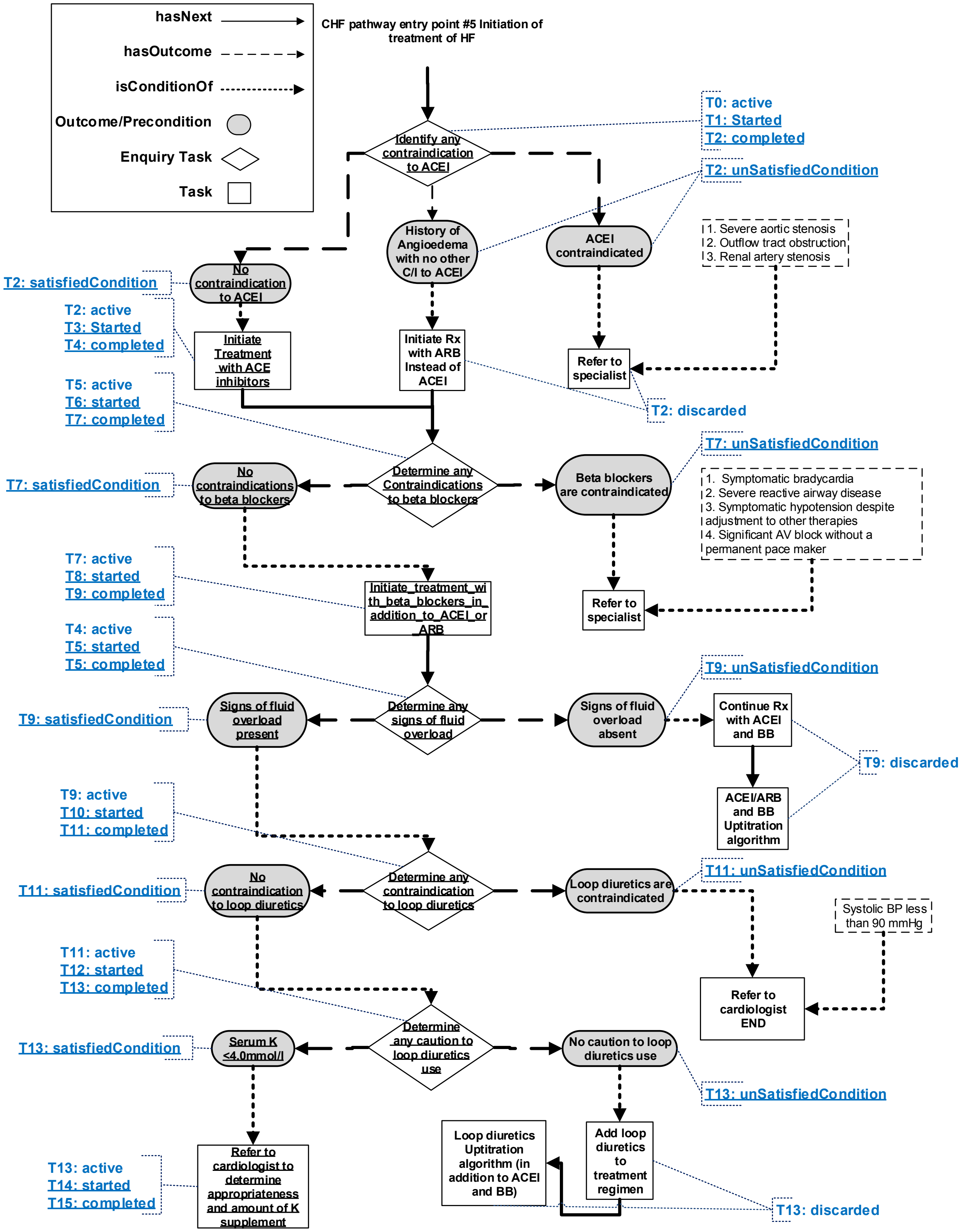

hasNext
hasOutcome
isConditionOf
Outcome/Precondition
Enquiry Task
Task
CHF pathway entry point #5 Initiation of treatment of HF
Identify any contraindication to ACEI
T0: active
T1: Started
T2: completed
T2: unSatisfiedCondition
History of Angioedema with no other C/I to ACEI
ACEI contraindicated
1. Severe aortic stenosis
2. Outflow tract obstruction
3. Renal artery stenosis
No contraindication to ACEI
T2: satisfiedCondition
T2: active
T3: Started
T4: completed
Initiate Treatment with ACE inhibitors
Initiate Rx with ARB Instead of ACEI
Refer to specialist
T2: discarded
T5: active
T6: started
T7: completed
T7: unSatisfiedCondition
No contraindications to beta blockers
Determine any Contraindications to beta blockers
Beta blockers are contraindicated
T7: satisfiedCondition
1. Symptomatic bradycardia
2. Severe reactive airway disease
3. Symptomatic hypotension despite adjustment to other therapies
4. Significant AV block without a permanent pace maker
T7: active
T8: started
T9: completed
Initiate_treatment_with_beta_blockers_in_addition_to_ACEI_or_ARB
Refer to specialist
T4: active
T5: started
T5: completed
T9: unSatisfiedCondition
Signs of fluid overload present
Determine any signs of fluid overload
Signs of fluid overload absent
Continue Rx with ACEI and BB
T9: satisfiedCondition
T9: discarded
ACEI/ARB and BB Uptitration algorithm
T9: active
T10: started
T11: completed
T11: unSatisfiedCondition
No contraindication to loop diuretics
Determine any contraindication to loop diuretics
Loop diuretics are contraindicated
T11: satisfiedCondition
Systolic BP less than 90 mmHg
T11: active
T12: started
T13: completed
Refer to cardiologist END
Serum K <4.0mmol/l
Determine any caution to loop diuretics use
No caution to loop diuretics use
T13: satisfiedCondition
T13: unSatisfiedCondition
Loop diuretics Uptitration algorithm (in addition to ACEI and BB)
Add loop diuretics to treatment regimen
T13: active
T14: started
T15: completed
Refer to cardiologist to determine appropriateness and amount of K supplement
T13: discarded

Figure 9.2 Execution of "CHF pathway entry point #5 Initiation of treatment of HF" [107]. Each task and outcome/condition of the CPG is annotated with the new values of the *hasTaskState* and *hasConditionState* properties and their corresponding logical times. Underlined annotations indicate the triples that are entered by the execution engine in the ontology and the rest of the triples are inferred by the OWL reasoner at the annotated time. Executed tasks and satisfied conditions/happened outcomes are underlined.

In this example, we show how we evaluated the working of our OWL-DL execution engine using a patient scenario for of the "CHF pathway entry point #5 Initiation of treatment of HF" computerized in [107]. This CPG will be used for evaluation of the OWL-DL based CPG execution engine because it does not contain any forms of datatype expressivity or loops. The CPG is composed of a series of sequential decisions that should be made by the clinician. Some tasks have conditions that are not investigated or generated in this particular CPG and they may come from other CPG or from an electronic medical record. The below descriptions provide a detailed account of state transitions in some of the more interesting steps of the execution:

T0: The only active task at the beginning is "*Identify any contraindication to ACEI*". This task is found by the CPG execution engine (in our case we use Jena functions to query the ontology) and shown on the screen. User is asked to execute this task as the first sub-task of the CPG.

T1: User decides that he is going to execute the task "Identify any contraindication to ACEI". Therefore, this task goes to the started state.

T2: When the user has completed "*Identify any contraindication to ACEI*" task, the execution engine checks the possible outcomes (values that this task has for *hasOutcome* property). Three outcomes are listed for this task in the ontology and they are shown to the user. When an outcome ("ACEI is not contraindicated") is selected by the user, the execution engine adds the values satisfiedCondition/happenedOutcome for the property *conditionHasState* of the selected outcome. This will cause the OWL reasoner to infer that the task "Initiate Treatment with ACE inhibitors" is active. Other outcomes ("*History of*

Angioedema with no other C/I to ACEI" and "*ACEI contraindicated*") that are not selected will have the value notHappenedOutcome/unSatisfiedCondition for the *hasConditionState* property. Moreover, the reasoner will infer that the tasks "Initiate Rx with ARB Instead of ACEI" and "*Refer to specialist*" are discarded because their conditions are unsatisfied. After handling the outcomes, the execution engine inserts a triple (*hasTaskState Completed*) for the executed task to let the OWL reasoner know that the task is accomplished by the user.

The rest of the decisions are handled in the same way during the execution. In this example we evaluated the capabilities of our OWL-DL based execution engine in handling state transitions, task orderings, outcomes specification, checking conditions and decisions. As we monitored the state transitions, all the state transitions where concordant to the execution semantics and we did not come across any inconsistencies.

Scenario 2. CHF Pathway Entry point # 2 Assessment of Initial Tests Results

The Canadian CPG [175] for the diagnosis of heart failure involves: (i) performing tests to rule out heart failure as diagnosis; (ii) if it cannot be ruled out then the next step is to determine any reduction in the left ventricular systolic function through echocardiography. Figure 9.3 depicting the clinical algorithm comprising rules to evaluate initial clinical assessments and tests to rule out heart failure as diagnosis. We have mapped the CPG ontology in [107] and transformed its instances to CPG-DKO for execution. The interesting aspect of this CPG is that it starts with a branch which is followed by three tasks that can be executed in parallel by the health care professionals. These tasks have outcomes which are conditions for two other tasks. Decision steps of the guideline are: (a) "Assess BNP", (b) "Assess X-Ray" (c) "Assess ECG" and (d) "Calculate Cumulative Boston Criteria Score". Boston Criteria [107] is a point score system for diagnosis of heart failure. It assigns scores to symptoms, signs and chest X-ray findings. Since heart failure signs and symptoms are non-organ specific, it is difficult to reach any diagnostic conclusion purely based on signs and symptoms. As a result, a scoring scheme such as Boston Criteria is helpful for deriving a more objective picture based on clinical and radiological findings. Boston criteria has three possible options: 1. Score is less than 4 points– means heart failure is unlikely 2. Score is between 5 and 7 points– means heart failure is possible 3. Score is more than 8–12 points means heart failure is definite. These decisions have outcomes which are conditions for two other tasks. The task "Order Echocardiography" can get activated by satisfaction of any of its 5 conditions. The CPG also indicates that even if one of the conditions is satisfied, the task should not be acted upon until all of the BNP, ECG and X-Ray assessments are finished first. To enforce this requirement, a synchronization point which is waiting for the completion of all of the assessment tasks is put before "Order Echocardiography".

Figure 9.3 shows an execution scenario that has been used for evaluation. Each task and condition has been annotated with its current state at each specific point in time. A task or a condition retains its state until a new state has been applied to it.

CHF Pathway Entry point # 2
Assessment of Initial
Tests Results

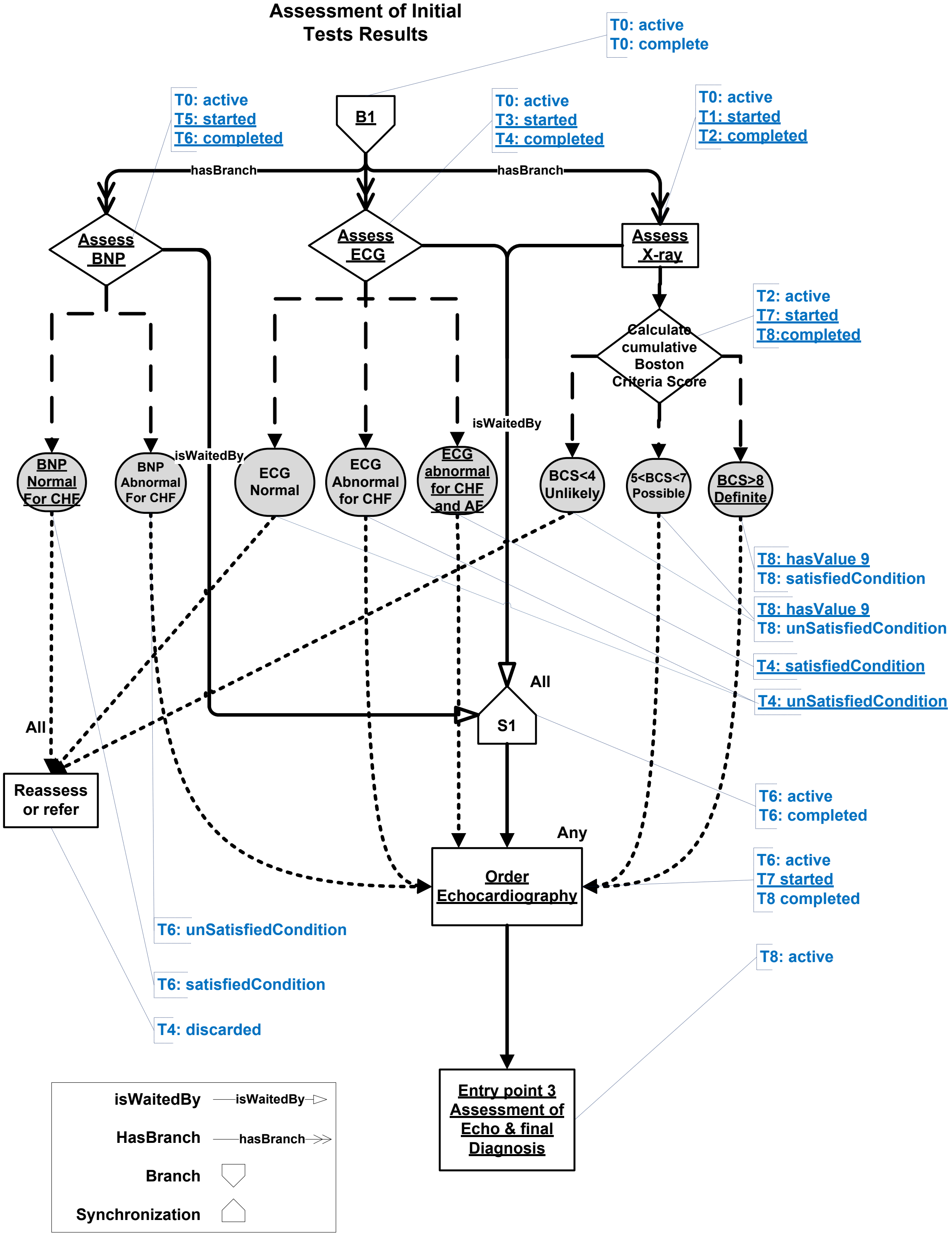


Figure 9.3 Execution of "CHF Pathway Entry point # 2 Assessment of Initial Tests Results" [107] CPG. Each task and outcome/condition of the guideline is annotated with the new values of the *hasTaskState* and *hasConditionState* properties and their corresponding logical times. Underlined annotations indicate the triples that are entered by the execution engine in the ontology and the rest of the triples are inferred by the OWL reasoner at the annotated time. Executed tasks and satisfied conditions/happened outcomes are underlined.

OWL-DL based execution engine is not capable of executing this CPG as it needs numerical comparisons. Since we are comparing the Boston criteria with a predefined set of numbers (they are known before the execution) OWL-2 based execution is enough and we do not expressivity of the SWRL. The below descriptions provide a detailed account of state transitions in some of the more interesting steps of the execution:

T0: Initial activation of the branch step "B1" leads to activation of all of its branches. There is no specific order in which "*Assess BNP*", "*Assess ECG*" and "*Assess X-Ray*" should be executed so it is left to the user to decide. Since this is a routing task and has made the intended effect on the state transition it goes directly to the completed state.

T4: In this step, the user indicates that the outcome "*ECG Normal*" and "*ECG Abnormal for CHF*" are both unsatisfied conditions. However, these two are conditions of the task "Reassess or refer" with the condition satisfaction criterion of all. Therefore, this task goes to the discarded state from inactive state.

T6: In this step, all the tasks that the synchronization task s1 is waiting for are completed. Therefore, this task goes to the completed state directly in order to unify the execution flow and direct it to the next task. Therefore, the task "*Order Echocardiography*" becomes active.

T8: After execution of "Calculate cumulative Boston Criteria Score" the execution engine asks the user to enter a value as the outcome and uses this value (9) to enter the triple "*hasValue 9*" for the corresponding outcomes. Our OWL 2 based inference engine uses the user defined ranges defined in the pre-processing phase to infer that "*BCS<4 Unlikely*" is unsatisfied and the task “Reassess or refer” that is waiting for “all” of its conditions is discarded. It also infers that “*BCS>8 Definite*” has state satisfiedCondition.

In this scenario, we evaluated the capabilities of our OWL-2 based execution engine in handling state transitions, task orderings, outcomes specification, checking conditions, decisions, branches, synchronization and data type expressivity. As we monitored the state transitions, all the state transitions where concordant to the execution semantics and we did not come across any inconsistencies.

Scenario 3. CHF entry point #4-Pre-treatment electrolyte assessment and correction

According Canadian guideline [175] for CHF, it is absolutely imperative to assess some preliminary parameters such as: serum creatinine, potassium and sodium and correct them if possible before CHF therapy with Angiotensin-Converting Enzyme Inhibitor (ACEI) and Beta Blocker (BB) can be safely administered. The purpose of the CPG (Figure 9.4) in this scenario is to guide a family physician, in a stepwise fashion, to evaluate these parameters and take appropriate actions in accordance to the outcome assessments. The interesting aspects of this CPG are: (i) two while loops and (ii) numeric comparisons. OWL 2 execution engine can execute this guideline and there is no need to use SWRL rules here as no mathematical function is needed to handle while loops and the guideline does not need to compare two numbers that are entered during execution.

We have mapped the CPG ontology in [107] and transformed its instances to our CPG-DKO for execution.

Figure 9.4 that is taken from [107] shows an execution scenario that has been used for evaluation. Each task and condition has been annotated with its current state at each specific point in time. A task or a condition retains its state until a new state has been applied to it.

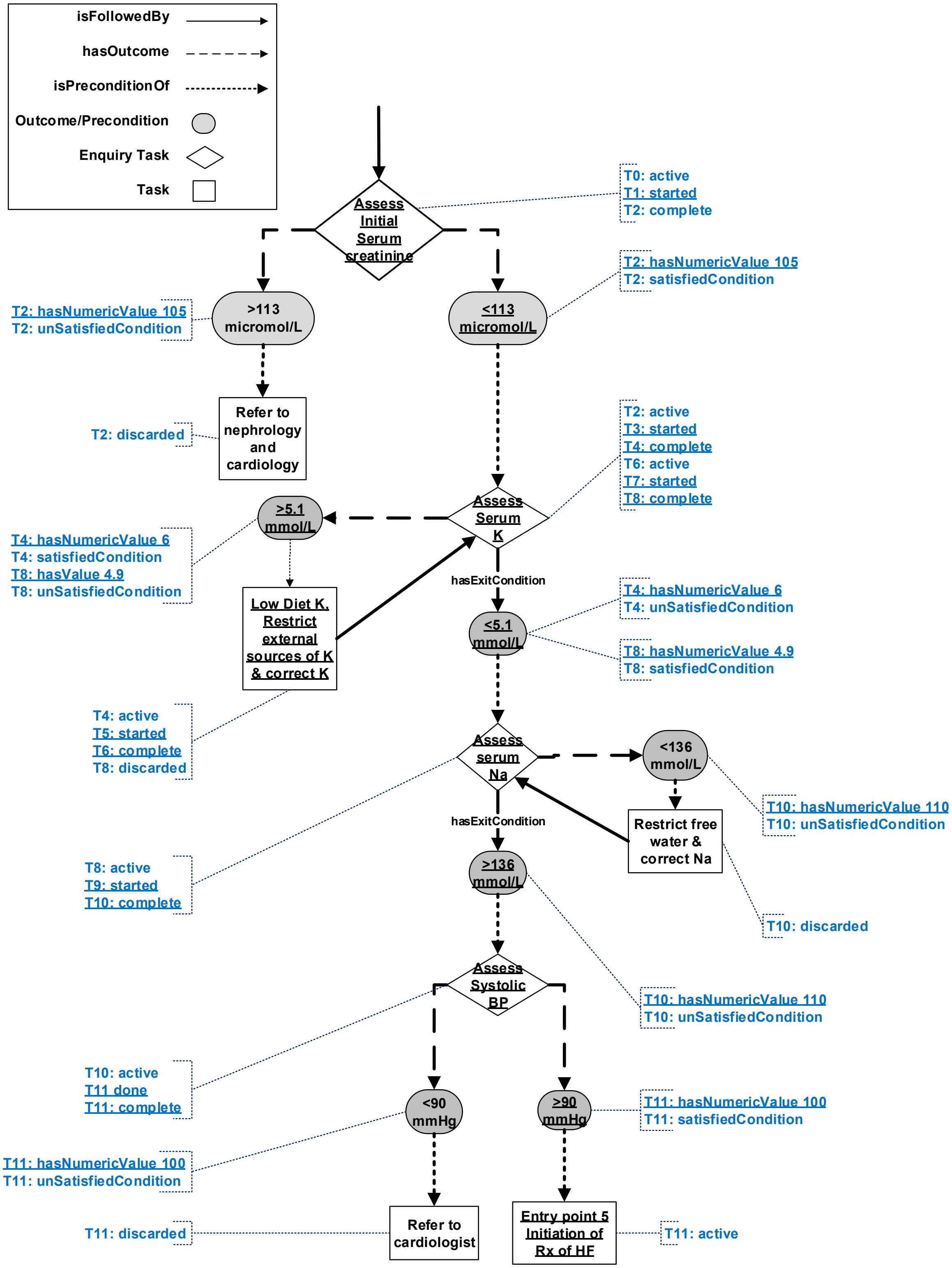

isFollowedBy
hasOutcome
isPreconditionOf
Outcome/Precondition
Enquiry Task
Task
Assess Initial Serum creatinine
T0: active
T1: started
T2: complete
T2: hasNumericValue 105
T2: satisfiedCondition
T2: hasNumericValue 105
T2: unSatisfiedCondition
>113 micromol/L
<113 micromol/L
T2: discarded
Refer to nephrology and cardiology
T2: active
T3: started
T4: complete
T6: active
T7: started
T8: complete
>5.1 mmol/L
Assess Serum K
T4: hasNumericValue 6
T4: satisfiedCondition
T8: hasValue 4.9
T8: unSatisfiedCondition
hasExitCondition
Low Diet K, Restrict external sources of K & correct K
<5.1 mmol/L
T4: hasNumericValue 6
T4: unSatisfiedCondition
T8: hasNumericValue 4.9
T8: satisfiedCondition
T4: active
T5: started
T6: complete
T8: discarded
Assess serum Na
<136 mmol/L
Restrict free water & correct Na
T10: hasNumericValue 110
T10: unSatisfiedCondition
hasExitCondition
T8: active
T9: started
T10: complete
>136 mmol/L
T10: discarded
Assess Systolic BP
T10: hasNumericValue 110
T10: unSatisfiedCondition
T10: active
T11 done
T11: complete
<90 mmHg
>90 mmHg
T11: hasNumericValue 100
T11: satisfiedCondition
T11: hasNumericValue 100
T11: unSatisfiedCondition
T11: discarded
Refer to cardiologist
Entry point 5 Initiation of Rx of HF
T11: active

Figure 9.4 Execution of "CHF entry point #4-Pre-treatment electrolyte assessment and correction" CPG [107]. Each task and outcome/condition of the guideline is annotated with the new values of the hasTaskState and hasConditionState properties and their corresponding logical times. Underlined annotations indicate the triples that are entered by the execution engine in the ontology and the rest of the triples are inferred by the OWL reasoner at the annotated time. Executed tasks and satisfied conditions/happened outcomes are underlined.

In this example, we show how we evaluated the working of our OWL 2 based execution engine using a patient scenario for of the "CHF entry point #4-Pre-treatment electrolyte assessment and correction" computerized in [107].The CPG is composed of a series of inquiry tasks that are used to decide about the continuation of the while-loop tasks. The below descriptions provide a detailed account of state transitions in some of the more interesting steps of the execution:

T2: Upon completion of the task "*Assess Initial Serum creatinine*" by the user, the execution engine checks the outcome of the task. User is asked to enter a value for the serum creatinine. We assume that in this patient scenario the value 105 micromol/L is entered for this outcome. As a result of this outcome, the OWL reasoner will infer that the condition *"<113 mmol/L*" is satisfied and the task "*Assess Serum Na"* is active. Moreover, the reasoner will infer that the condition *">113 mmol/L*" is unsatisfied and the task "*Refer to nephrology and cardiology*" is discarded. Now the only active task is "*Assess serum K*" which is the second task of the loop.

T4: When user has finished the task "*Assess serum K*" user is asked to enter a numeric value as the outcome of this task. We assume that in our patient scenario this value is 6 mmol/L. Therefore, the OWL reasoner will infer that condition "*>5.1 mmol/L*" is satisfied, "*Low Diet K, Restrict external sources of K & correct K*" is activated and condition "*<5.1mmol/L*" has state "*unSatisfiedCondition*". Now "*Low Diet K, Restrict external sources of K & correct K*" which is the first task of the loop becomes activated. This task is shown to the user to be acted upon.

T6: When the user is done with the task "*Low Diet K, Restrict external sources of K & correct K*" the first task of the loop "*Assess serum K*" is activated again and is ready for execution and evaluation of its outcome.

T8: This time, users enter the value 4.9 mmol as the outcome of the task "*Assess serum K*" and the OWL reasoner infers that exit condition "*<5.1 mmol/L*" is satisfied. Therefore the next task after the loop which is "*assess serum NA*" becomes activated.

The rest of the execution is performed in a similar fashion. Please note that we do not go to the second loop and just skip it in this scenario. This example has been discussed to show how our OWL 2 based execution engine is evaluated for handling state transitions, task orderings, outcome specification, checking conditions, decisions, while loops and some level of data type expressivity. As we monitored the state transitions, all the state transitions where concordant to the execution semantics and we did not come across any inconsistencies.

Scenario 4. Adjustment of Oral Furosemide Dosing for Changes in Dry Weight

Figure 9.5 depicts a hypothetical case of diuretic dose adjustment. Diuretics such as Furosemide are administered for control of congestion and fluid retention. Once optimal fluid balance has been restored, the patients are usually maintained at an effective yet lowest possible dose. During the diuretic therapy, patients are instructed to record their weight daily. If there is an increase in weight by more than by 1 kg over a week, then the diuretic dose has to be adjusted. This algorithm (Figure 9.5) compares patient's weights over period of a week to recommend appropriate dose adjustment. This guideline can only be executed using our SWRL based execution engine because of the need to perform mathematical comparisons between patient's data that are not available during the pre-processing. To execute the above mentioned CPG, a function with the operator subtractMO which has the outcomes weight1 and weight2 as its inputs and the weight difference between the two measurements as its output. The output of this math function is the input of two other functions that compare the weight difference with the value 1 kg. The outputs these functions with the operators greaterThanEqualMO and lessThanMO are used as the conditions of the tasks that adjust the dose of the Furosemide. Figure 9.5 shows an execution scenario that has been used for evaluation. Each task and condition has been annotated with its current state at each specific point in time. A task or a condition retains its state until a new state has been applied to it.

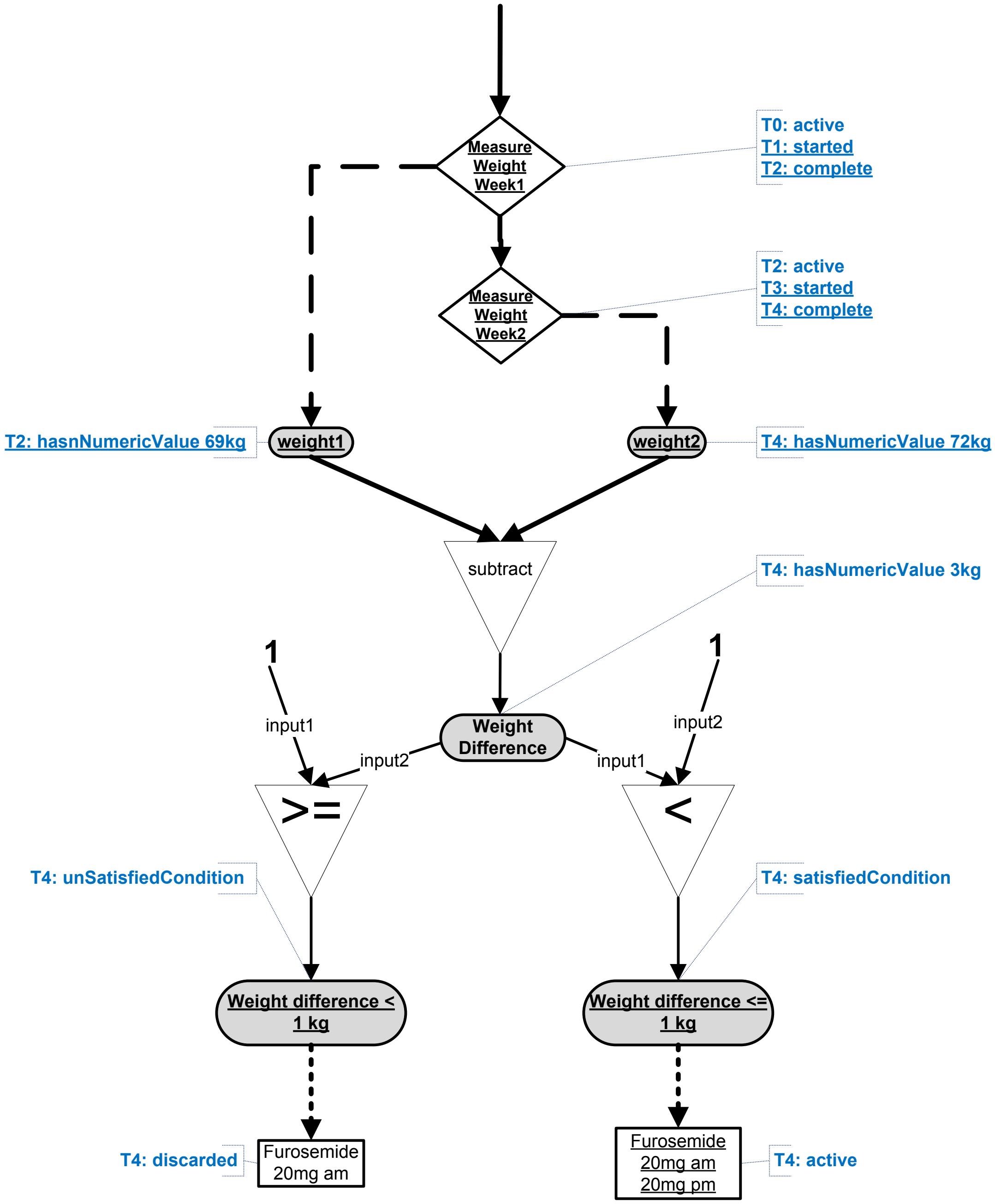

Measure Weight Week1
T0: active
T1: started
T2: complete
Measure Weight Week2
T2: active
T3: started
T4: complete
T2: hasnNumericValue 69kg
weight1
weight2
T4: hasNumericValue 72kg
subtract
T4: hasNumericValue 3kg
1
input1
input2
Weight Difference
input1
1
input2
>=
<
T4: unSatisfiedCondition
T4: satisfiedCondition
Weight difference < 1 kg
Weight difference <= 1 kg
T4: discarded
Furosemide 20mg am
Furosemide 20mg am 20mg pm
T4: active

Figure 9.5 Execution of "Adjustment of Oral Furosemide Dosing for Changes in Dry Weight". Each task and outcome/condition of the guideline is annotated with the new values of the hasTaskState and hasConditionState properties and their corresponding logical times. Underlined annotations indicate the triples that are entered by the execution engine in the ontology and the rest of the triples are inferred by the OWL reasoner at the annotated time. Executed tasks and satisfied conditions/happened outcomes are underlined.

In order to see how this CPG is used for evaluating our SWRL-based CPG execution engine, we go through a scenario and discuss the state transitions. T0 to T3 are executed the same as the previous scenarios discussed in this section. Below we describe the state transitions in step T4:

T4: When both of the weight measurement tasks are done, both operand of the subtract function are available for the corresponding SWRL rule to be executed by the OWL reasoning engine. After the execution of the rules, the corresponding value is assigned to the subtract function outcome. In our scenario, patient's weight has changed from 69kg to 72kg. Therefore the output of this function has the numeric value 3. Comparison of the value 3 with 1 in the two math functions will cause the OWL reasoner to infer that the conditions "*Weight difference < 1 kg*" and "*Weight difference >= 1 kg*" belong to *UnSatisfiedCondition* and *SatisfiedCondition* classes respectively.

This example scenario has been discussed in order to evaluate the capability of our SWRL based execution engine in handling state transitions, task orderings, outcomes specification, checking preconditions, decisions, higher level of data type expressivity compared to OWL 2 and OWL-DL. More data type expressivity includes mathematical computation and comparison of numbers which are not known before the execution. As we monitored the state transitions, all the state transitions where concordant to the execution semantics and we did not come across any inconsistencies.

9.1.3. Evaluation by Comparison of Execution Results

An ontology can be evaluated in an application that makes use of the ontology. As we discussed earlier we successfully mapped and transferred Nursing CPG, AF-CHF CPG and CAP ontologies to CPG-DKO. These CPG ontologies have total of 9 instantiations each representing a disease-specific CPG. Both Nursing CPG and AF-CHF CPG ontologies have proprietary execution engines developed in the NICHE group. The nursing CPG execution engine is developed by the author of the thesis in collaboration with the nursing CPG developer and encoder in [145]. CHF-AF CPG ontology has also a web based execution engine described in [107]. Both of these CPG execution engines adopt a graph parsing approach for execution. We can compare the generated therapy plans by these execution engines and our OWL-based execution engine for evaluation.

In order to compare the results of the CPG execution engines of the CHF-AF and nursing CPG ontologies and our OWL-based CPG execution engine that executes the CPG-DKO, we used the designed patient scenarios discussed in the beginning of the section 9.1. We executed each of the disease-specific CPG simultaneously in their original format with their own proprietary execution engine and in CPG-DKO ontology with our OWL-based CPG execution engine. In all of the 9 cases, the generated recommendations perfectly match each other and we do not find any difference between them as long as we follow the same execution strategy. Execution engines can execute a CPG based on a Breadth First Search or a Depth First Search strategy or a strategy that is dictated by the user. Our limitations in this evaluation are (1) We do not go through all the possible scenarios; (2) our scenarios are not necessarily medically valid.

9.1.4. Evaluation by Modeling and Executing New CPG

In order to perform more evaluations on the both CPG-DKO and the CPG execution engine, we have computerized Osteoarthritis treatment algorithm published by National Health Services Tayside of Scotland [169] and clinical pathways for hypertension [170] and diabetes [171] published by National Institute of Health and Clinical Excellence of UK.

Therefore, instead of performing instance transformation on existing computerized CPG, we have computerized these pathways from scratch. Since hypertension and diabetes two clinical pathways are very detailed and can have up to 5 levels of nesting, we have only gone two levels down and left the rest of the details for the future work. These pathways are not very complicated in terms of the workflow construct and are mainly computerized in order to evaluate our CPG merging methodology discussed later. However, execution of them can be used as an evaluation for capabilities of our CPG execution engine for execution of basic workflow constructs. In all cases, our CPG execution engine can successfully follow the CPG workflow constructs and execute the decisions successfully. We go through an example scenario in order to show how our execution engine successfully interprets the workflow structure of the CPG. Figure 9.6 shows the Osteoarthritis treatment algorithm published by National Health Services Tayside of Scotland [169] used for evaluation.

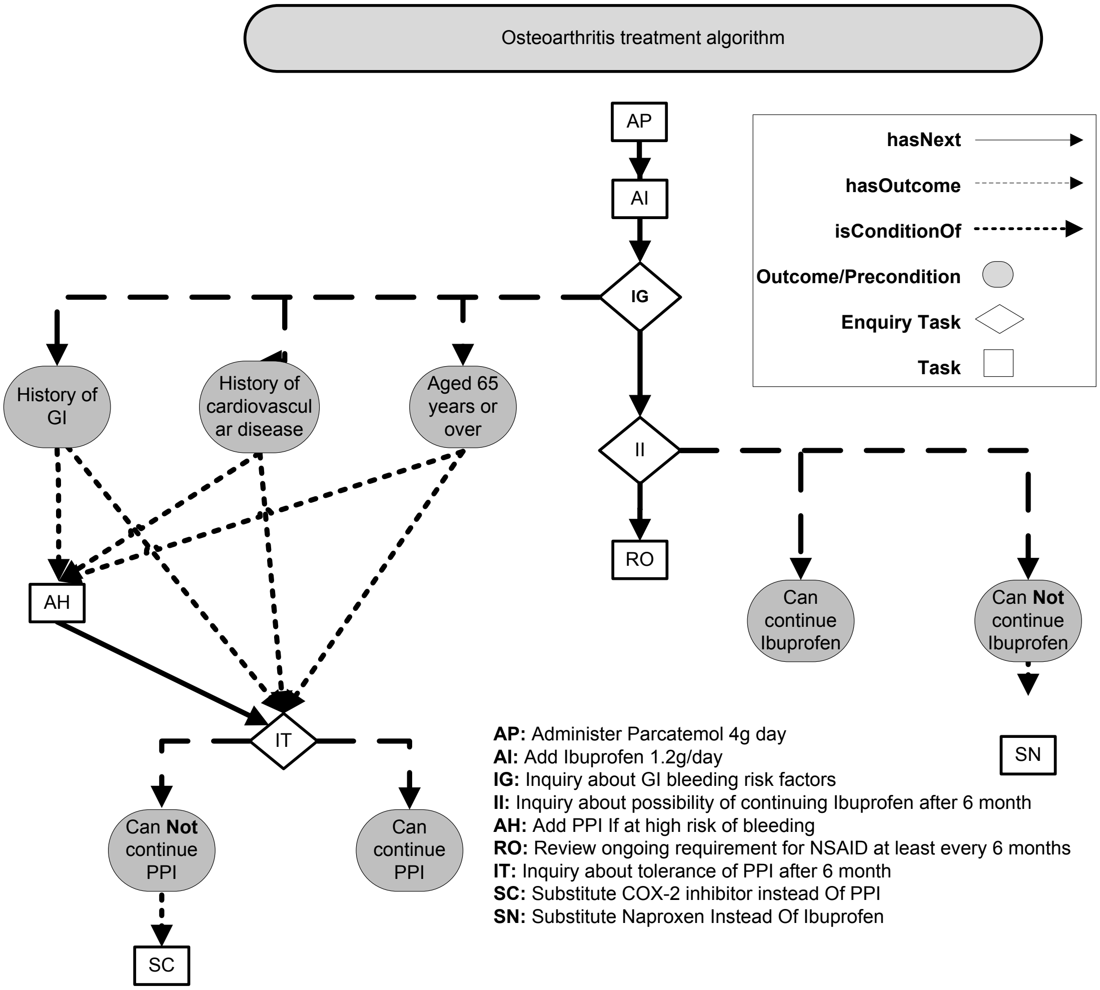

Osteoarthritis treatment algorithm
AP
AI
IG
II
RO
hasNext
hasOutcome
isConditionOf
Outcome/Precondition
Enquiry Task
Task
History of GI
History of cardiovascular disease
Aged 65 years or over
AH
IT
Can continue Ibuprofen
Can **Not** continue Ibuprofen
SN
Can **Not** continue PPI
Can continue PPI
SC
AP: Administer Parcatemol 4g day
AI: Add Ibuprofen 1.2g/day
IG: Inquiry about GI bleeding risk factors
II: Inquiry about possibility of continuing Ibuprofen after 6 month
AH: Add PPI If at high risk of bleeding
RO: Review ongoing requirement for NSAID at least every 6 months
IT: Inquiry about tolerance of PPI after 6 month
SC: Substitute COX-2 inhibitor instead Of PPI
SN: Substitute Naproxen Instead Of Ibuprofen

Figure 9.6 The Osteoarthritis treatment algorithm

As we described earlier, in order to evaluate the correctness of the generated recommendations we created several patient scenarios that cover all the execution paths in the CPG. We ran the execution engine with these imaginary scenarios and recorded the state transition of all the tasks in the CPG. Reviewing the state transitions of the tasks during execution can be used to evaluate if the workflow constructs are interpreted as they are supposed to. Successful interpretation of the workflow constructs indicates that CPG-DKO is correctly modeling the intended CPG and the execution engine is correctly executing the computerized CPG. In the following, we see an example of the recorded state transitions for an imaginary patient scenario:

T0: We start reviewing the state transitions of our scenario at this point of time. In this step the first task of the algorithm AP is already started by the user.

T1: In this time step, user indicates that he has finished the task AP. Therefore, this task goes to the completed state. Completion of this task will cause the activation of the next inactive task which is AI. This task becomes active according to the state transition rules and it is shown to the user to be acted upon.

T2: User starts execution of the task AI. Therefore, AI goes to the started task until the user indicates that he has completed this task.

T3: User indicates that the task *AI* is completed and the task goes to the completed state. Completion of this task causes activation of the next inactive task which is *IG*. This task is activated by the reasoner and shows to the user by the execution engine.

T4: User starts execution of the task *I*G. This task goes to the started state.

T5: Since IG is completed by the user it goes to the completed state. This task has three outcomes which are shown to the user after completion of IG: (1) “*History of GI*”, (2) “*History of Cardiovascular disease*” and (3) “*Age 65 or over*”. User can select zero or more of these outcomes. We assume that patient is younger than 65 and does not have a history of the GI or cardiovascular diseases. Therefore, none of these options are selected as the outcome of the task GI. These outcomes are conditions of the tasks AH and IT. Since these conditions are not satisfied both of these tasks are discarded. Since IT is discarded, its outcomes which are “*cannot continue PPI*” and “can’t continue PPI” can never be selected. As a result, the task SC will never have satisfied conditions. Therefore this task is discarded as well according the state transition rules.

T6 is similar to the states T1.

T7: Task II is completed in this state. This will cause the inactive task RO to get activated. After completion of II the outcomes of this task which are "*cannot continue ibuprofen*" and "*can't continue ibuprofen*" are shown to the user. If we assume that the patient is not capable of taking ibuprofen for some medical conditions, the second outcome will be selected. After selection of this outcome the conditions of the task SN will be satisfied and this task will become activated. Now, both RO and SN tasks are active.

T8: User opts to execute both of the active tasks simultaneously. Both tasks RO and SN go to the started state.

T9: User finishes execution of the tasks RO and SN and both of these tasks go to the completed state.

These state transitions have been reported in Table 9.4.

Table 9.4 State transition of tasks during execution of the CPG in Figure 9.6. New states are bolded and the state transitions performed by the user are underlined. (A = Active, I = Inactive, S = Started, D = Discarded, C = Completed)

	AP	**AI**	**IG**	**II**	**AH**	**RO**	**IT**	**SC**	**SN**	**Selected Outcome**
T0	**S**	I	I	I	I	I	I	I	I	
T1	**C**	**A**	I	I	I	I	I	I	I	
T2	C	**S**	I	I	I	I	I	I	I	
T3	C	**C**	**A**	I	I	I	I	I	I	
T4	C	C	**S**	I	I	I	I	I	I	
T5	C	C	**C**	**A**	**D**	I	**D**	**D**	I	None
T6	C	C	C	**S**	D	I	D	D	I	
T7	C	C	C	**C**	D	**A**	D	D	**A**	Cannot continue Ibuprofen
T8	C	C	C	C	D	**S**	D	D	**S**	
T9	C	C	C	C	D	**C**	D	D	**C**	

As you can see in the table above, our execution engine interprets the tasks' sequence, outcomes and conditions in the exact way that our execution semantics and the workflow structure of this CPG is representing.

9.2. Evaluation of the Ontology Mapping Module

For evaluation purposes, our mapping ontology is compared with the existing representation languages in section 9.2.1 in terms of the features identified in the related literature. These features have been reported in section 2.3.1 of the thesis. To evaluate the ability of the mapping languages in mapping complex ontologies and the translation algorithm in preparing the mappings for instance transformation we used them for mapping and transformation of several CPG ontologies to CPG-DKO. The results of this evaluation are discussed in section 9.2.2.

9.2.1. Evaluation of the Knowledge Mapping Ontology (KMO)

We used OWL reasoners to evaluate KMO and its instantiations for consistency and syntactical correctness. Pellet reasoner shows that our ontology is consistent. We also checked the consistency of the ontology after translation to OWL-DL + SWRL. Our evaluations show that our ontology remains consistent after translation. Moreover, the output of this function is always in OWL-DL + SWRL. This ensures decidability and efficiency of the reasoning algorithm.

In order to evaluate our mapping ontology for completeness we compare our ontology with the popular ontology mapping methodologies C-OWL [58], MARFA [62], SEKT [165], Yuangui1 [74], Euzenat1 [166], OWL [142] and SWRL [154] for representation and execution of mappings. This comparison includes support for mappings patterns, operators, condition, constraints, relations between mappings and several other aspects.

9.2.1.1. Support for mapping patterns

In this section, the support for a list of mapping patterns identified in [165] will be compared. The result of comparison of mapping patterns for classes, attributes and instance are listed in Table 9.5, Table 9.6 and Table 9.7 respectively.

Table 9.5 Comparison of KMO with the existing ontology mapping representation languages in terms of supporting class mapping patterns

Class Pattern	**MARFA**	**OWL**	**SWRL**	**C-OWL**	**SEKT**	**Yuangui1**	**Euzenat1**	**KMO**
Class Equivalence	+	+	-	+	+	+	+	+
Class Subsumption	+	+	-	+	+	+	+	+
Class by attribute mapping	+	+	+	-	-+	+	+	+
Class mapping by axiom	-	-	+	-	-+	-	+	+
Class join mapping	-	-	-	-	+	-	-	+
Class to attribute mapping	+	+	+	-	-+	-	+	+
Class Relation mapping	+	+	+	-	+	-	+	+

Table 9.6 Comparison of KMO with the existing ontology mapping representation languages in terms of supporting relation and attribute patterns

Attribute Pattern	**MARFA**	**OWL**	**SWRL**	**C-OWL**	**SEKT**	**Yuangui1**	**Euzenat1**	**KMO**
Property Equivalence	+	+	-	-	+	+	+	+
Super Sub Property	+	+	-	-	+	-	+	+
Mapping by axiom	-	-	-	-	-+	-	+	+
Attribute value to attribute value mapping	-	-	+	-	-+	-	+	+
Attribute value to class mapping	-	+	+	-	-+	-	-	+
Relation Equivalence	-	+	-+	-	+	-	+	+
Sub-super attribute value mapping	-	+	-	-	-+	-	-	+

Table 9.7 Comparison of KMO with the existing ontology mapping representation languages in terms of supporting instance mapping patterns

Instance Pattern	**MARFA**	**OWL**	**SWRL**	**C-OWL**	**SEKT**	**Yuangui1**	**Euzenat1**	**KMO**
Equivalence	-	+	+	-	+	-	-	+
Difference	-	+	+	-	-	-	-	+
Equivalent relation instance mapping	-	+	+	-	-+	-	-	+

Our language has a more comprehensive set of supported mapping patterns compared to the existing ontology mapping representation languages. All these aspects of the mapping can be greatly beneficial when complex mappings are needed for expressive ontologies.

9.2.1.2. Support for Mapping Operators

Operators play an important role in manipulation and preparation of ontology elements for ontology mapping. The more operators can be expressed and executed, the more expressive the mapping representation language is. Comparison of our language with the existing languages can be seen in Figure 9.8.

Table 9.8 Comparison of KMO with the existing ontology mapping representation languages in terms of support for instance transformation and ontology mapping operators

Operator Name	**MARFA**	**OWL**	**SWRL**	**C-OWL**	**SEKT**	**Yuangui1**	**Euzenat1**	**KMO**
Class Union	-	+	+	-	+	+	+	+
Class Intersection	-	+	+	-	+	-	+	+
Class Complement	-	+	-	-	+	-	+	+
Negation	-	-	-	-	+	-	+	+
Inverse	-	+	+	-	+	-	+	+
Transitive	-	+	+	-	+	-	+	+
Symmetric	-	+	+	-	+	-	+	+
Property Chain	-	+	+	-	-	-	-	+
Math Operator	-	-	+	-	-	-	+	+
String Operator	-+	-	+	-	-	-	+	+
Boolean Operator	-	-	+	-	-	-	-	+
Comparator Operator	-	-+	+	-	-	-	+	+
Convert Operators	+	-	-	-	-	-	-	+

Instance transformation, which is the main reason for us to map ontologies, makes heavy use of these operators. Supporting these operators make our language a very powerful tool for instance transformation.

9.2.1.3. Support for Conditions and Constraints

An important expressivity feature is to assign conditions, condition satisfaction criteria and constraints to mappings. A major advantage of our mapping representation language is the ability to define very complex conditions and condition satisfaction criteria that no other existing language is capable of expressing and executing. The result of comparison of our CPG representation language with the existing CPG representation languages is listed in Table 9.9.

Table 9.9 Comparison of KMO with the existing ontology mapping representation languages in terms of supporting conditions, constraints and various condition satisfaction criteria

Class Conditions	**MARFA**	**OWL**	**SWRL**	**C-OWL**	**SEKT**	**Yuangui1**	**Euzenat1**	**MKO**
Property HasValue	+	+	+	-	+	-	+	+
Property Cardinality Restriction	-	+	-	-	-	-	-	+
Property Qualified Cardinality Restriction	-	+	-	-	-+	-	-	+
Self Restriction	-	+	-	-	-	-	-	+
Attribute Condition	**MARFA**	**OWL**	**SWRL**	**C-OWL**	**SEKT**	**Yuangui1**	**Euzenat1**	**KMO**
HasValue	+	+	+	-	- +	-	+	+
Cardinality Restriction	-	+	-	-	-	-	-	+
Qualified Cardinality Restriction	-	+	-	-	-+	-	-	+
Condition Satisfaction Criteria For mappings	**MARFA**	**OWL**	**SWRL**	**C-OWL**	**SEKT**	**Yuangui1**	**Euzenat1**	**KMO**
any	-	-	-+	-	-	-	-	+
all	-	-	-+	-	-	-	-	+
any k	-	-	-	-	-	-	-	+

min	-	-	-	-	-	-	-	+
max	-	-	-	-	-	-	-	+

Table above shows that our methodology has made improvements in expressing mapping conditions and the condition satisfaction criteria.

9.2.1.4. Relation between Mappings

Another aspect that has been widely ignored by researchers in the field of ontology mapping is the possible relations between the mappings. As you can see in Table 9.10, MARFA and our language are the only languages that provide the user a set of possible relations between the mappings.

Table 9.10 Comparison of KMO with the existing ontology mapping representation languages based on the supporting relations between the mappings

Mapping Relation	MARFA	OWL	SWRL	C-OWL	SEKT	Yuangui1	Euzenat1	KMO
Abstraction	+	-	-	-	-	-	-	+
Specialization	+	-	-	-	-	-	-	+
Composition	+	-	-	-	-	+	-	+
Alternatives	+	-	-	-	-	-	-	+

Ability to express these relations can facilitate the mapping process by reusing the existing mappings in the current instantiation of the mapping ontology or based on other similar mappings problems.

9.2.1.5. Other Aspects of the Mappings

The two other important expressivity features are existence of an expression language for defining the condition and the constraints and the support for variables in that expression language. Supporting variables is extremely important in instance transformation as it can be used to transfer values between the source and target ontologies. Other than expressivity there are several important factors that should be taken into the consideration. All of these features are compared in Table 9.11.

Table 9.11 Comparison of KMO with the existing ontology mapping representation languages based on existence of tools, capturing meta-data, reasoning in presence of inconsistencies, formal semantics, expression languages and variables

Feature	MARFA	OWL	SWRL	C-OWL	SEKT	Yuangui1	Euzenat1	KMO
Existence of tools	-+	+	+	-	-+	-	-+	+
Capturing Meta-data	-	-	-	-	-	-	+	+
Reasoning in presence of inconsistencies	-	-	-	+	-	-	-	-+
Formal Semantics	-	+	+	+	-+	-	-+	+
An Expression Language	-	-	+	-	-	-	-+	+
Support For Variables	-	-	+	-	-	-+	-	+

These features are important as they improve shareability, expressivity and usability in semantic-based ontology mapping approaches. As the above table shows, all these features are supported by our mapping representation ontology.

9.2.2. Evaluation of KMO and the Translation to OWL+SWRL Algorithm

In order to evaluate KMO and the translation algorithm, we first need to instantiate KMO, feed those instantiations to the translation algorithm, and evaluate the output. As we discussed earlier, we instantiated KMO to map CHF-AF, CAP and Nursing CPG ontologies to CPG-DKO. Then we used our translation algorithm to translate the mappings to OWL-DL + SWRL in order to perform instance transformation. We then used the pellet reasoner to perform reasoning on the mapped CPG ontologies and the translated mappings in order to transfer instance of the source CPG ontologies to CPG-DKO. KMO and the translation algorithm can be evaluated in two ways:

(1) The output of the translation algorithm should have proper syntax, be in OWL-DL + SWRL and consistent. We used pellet to check the syntax and consistency of the output. In all three cases, the output is syntactically correct and consistent. Moreover, Protégé shows that the specie of the output is always OWL-DL.

(2) After the reasoning step of the semantic based ontology mapping, the transformed instances should represent the same knowledge as the source ontology. A way of doing this is to give the ontologies to the domain expert and ask them to compare the medical knowledge and the workflow structure represented by the both ontologies. However, comparing two ontologies needs extensive knowledge of the domain, ontologies and the semantics of the source and the target ontologies. Domain expert should go through each instance in the source ontology and its equivalent in the target ontology and compare their relations. There are two disadvantages to this approach: 1. This can be excessively time consuming to be done manually for large ontologies like the ones that we are dealing with. 2. An instance, class or property in the source ontology may not necessarily have an equivalent in the target ontology or vice versa due to different modeling approaches. Therefore, we believe the best way to validate the correctness of the transformed instances is to utilize them in applications that accept them as their inputs and evaluate the results of those applications. We use CPG execution engines as the evaluating application. In order to confirm that the transformed instances represent the same CPG ontology, we executed the transformed CPG using our own OWL-based execution engine and compared the results with the output of the execution of these CPG in their original format by their own proprietary execution engines. Both Nursing CPG and AF-CHF CPG ontologies have

designated CPG execution engines developed in the NICHE group. Comparison of the execution results using the scenarios designed to evaluate the execution engines show that the same output is generated in all scenarios for all the CPG.

Since the CAP ontology did not have an execution engine for comparison, we asked the person who had instantiated the ontology to use our engine to evaluate the transformed CAP ontology. In the beginning, the expert found several inconsistencies between their ontology and generated output of our CPG execution engine. Our investigations showed that these errors happen due to spelling errors and missing relations in the ontology. Fixing those errors made the final output of our OWL-based CPG execution engine consistent to what was expected form the CAP ontology according to the provided flowchart.

9.3. Evaluation of the CPG Merging Framework

Our merging approach has two components: (1) CPG-KPO and (2) CPG Merging Execution Engine. We seek experts' opinion in order to evaluate KMO as there is no golden standard ontology developed for this purpose and not enough research has been performed on requirements of CPG merging representation languages in order to be used for evaluation. We also evaluate both CPG-KPO and merge execution engine by monitoring the execution of 6 real and 10 imaginary CPG merging examples. We recorded the state transitions in order to validate that the execution of the merged CPG is performed as it is dictated by the execution semantics of the tasks and constraints.

9.3.1. Evaluation of the CPG-KPO

We used domain experts' opinion in order to evaluate this ontology. The evaluation process started with a 20-minutes presentation that discussed the concepts represented in the CPG-KPO, their significance and rationale in CPG merging. Then seven health informaticians, physicians and computer scientists who are experts in medical ontologies were asked to answer a questionnaire designed for evaluation of CPG-KPO. This questionnaire asked participants' opinions regarding ease of use, clarity, expressiveness, usefulness of CPG-

KPO concepts and participants' overall satisfaction. They were given 30 minutes to fill the questionnaires. During that time, participants were allowed to ask questions but not to share opinions with each other. In the rest of this section, we review each of the questions in the questionnaire and participants' responses to it:

9.3.1.1. Ease of Use

Table 9.12 shows the table used to evaluate CPG-KPO for "ease of use" by domain experts. This table shows the criteria used for evaluation and the possible answers for each of these criteria. The number in each of the cells shows the percentage that participants have chosen the corresponding answer.

Table 9.12 Criteria used for evaluating ease of use of CPG-KPO for CPG merging. The number in each table cell represents the percentage that each answer has been selected

	Strongly disagree	**Moderately disagree**	**Neither agree or disagree**	**Moderately agree**	**Strongly agree**
I find it easy to instantiate the CPG-KPO in order to merge CGP	0%	0%	43%	43%	14%
Names of the CPG-KPO elements are meaningful	0%	0%	14%	57%	29%
Purpose of CPG-KPO elements are clear	0%	0%	14%	29%	57%
Concept class hierarchy is appropriately defined	0%	0%	29%	57%	14%

Table 9.12 shows that the participants find CPG-KPO moderately easy to use for CPG merging.

9.3.1.2. Clarity

Table 9.13 shows the table used to evaluate CPG-KPO for "clarity" by domain experts. This table shows the criteria used for evaluation and the possible answers for each of these criteria. The number in each cell shows the percentage that participants have chosen the corresponding answer.

Table 9.13 Criteria used for evaluation of clarity of CPG-KPO concepts to participants. The number in each table cell represents the percentage that each answer has been selected

	Not Clear at all	**Not clear**	**Somewhat clear**	**Clear**	**Very Clear**
Constraints	0%	0%	0%	57%	43%
Conditions	0%	0%	0%	43%	57%
Constraint Priorities	0%	0%	14%	29%	57%
Constraint Conflicts	0%	0%	43%	28.5%	28.5%

Table 9.13 shows that the majority of the concepts are clear or very clear to participants. The concept that is less clear to participants is Constraint Conflicts.

9.3.1.3. Expressiveness

We also asked the participants to evaluate the expressiveness of our ontology for representing CPG merging aspects listed in Table 9.14. The number in each cell shows the percentage that participants have chosen the corresponding answer.

Table 9.14 Aspects of CPG merging evaluated for expressiveness by participants of our survey. The number in each of the table cells represents the percentage that each option has been selected

	Not Expressive at all	**Not expressive**	**Somewhat expressive**	**Expressive**	**Very expressive**
Modelling Workflow constraints	0%	0%	14%	43%	43%
Modelling Intuitional constraints	0%	0%	29%	29%	43%
Modelling Temporal constraints	0%	0%	29%	29%	43%
Modelling Medical Constraints	0%	14%	14%	58%	14%
Modelling Conflicts between Constraints	0%	14%	57%	0%	29%
Modelling Conditions	0%	0%	29%	0%	71%

Table 9.14 shows that our ontology is perceived as an expressive ontology for the task of CPG merging by participants. This table also shows that participants believe that the least expressive aspect of CPG-KPO is the ability to model conflicts between constraints. We believe that this is due to the fact that it was the least clear concept in CPG-KPO to participants. Participants also find the ontology very expressive for representation of conditions. This concept was among the clearest concepts to the participants.

9.3.1.4. Usefulness

We asked participants to evaluate the usefulness of concepts of CPG-KPO in representation of the CPG merging. The criteria used for usefulness evaluation are listed in Table 9.15. The number in each cell represents the percentage that the corresponding answer has been selected by our participants.

Table 9.15 Concepts of CPG-KPO evaluated for usefulness by participants of our survey. The number in each of the table cells represents the percentage that each option has been selected

	Not useful at all	**Not useful**	**Somewhat useful**	**Useful**	**Very useful**
Workflow Constraints	0%	0%	14%	14%	71%
Temporal Constraints	0%	0%	0%	29%	71%
Medical Constraints	0%	0%	14%	29%	57%
Operational Constraints	0%	0%	14%	43%	43%
Condition	0%	0%	0%	43%	57%
Conflict	0%	0%	14%	29%	57%
Conflict Detection Rules	0%	0%	0%	29%	71%

Table 9.15 shows that most of the participants in our study find the concepts representable in our ontology useful or very useful for CPG merging. Even though our ontology does not support many temporal constraints, this feature is identified as the most useful feature. This is an indication that more research on temporal constraint for CPG merging should be performed and our ontology can benefit from improvements in this regard.

9.3.1.5. Overall Satisfaction

Participants also expressed their overall satisfaction with CPG-KPO. Table 9.16 shows the table used to evaluate the overall satisfaction of participants with CPG-KPO. The number in each cell represents the percentage that the corresponding answer has been selected by our participants.

Table 9.16 Table used to evaluate the overall satisfaction of participants with CPG-KPO. The number in each of the table cells represents the percentage that each option has been selected

	Not satisfied at all	**Not satisfied**	**Somewhat satisfied**	**Satisfied**	**Very satisfied**
Overall Satisfaction	0%	0%	0%	83%	17%

Table 9.16 shows that all of the participants are either satisfied or very satisfied overly with CPG-KPO. One of the participants did not respond to this question.

9.3.1.6. Open-Ended Questions

Besides close-ended questions, we asked the participants to answer a number of open-ended questions as well. These questions were not answered by the all of the participants however. We discuss these questions and the answers given to them by the participants.

1. Are there any improvements that you would like to see in CPG-KPO?

The following list summarizes the improvements suggested by the participants:

- Modelling evidence conflict
- Modelling evolution of CPG and automatic updating of CPG-KPO for new versions of CPG
- Using temporal conditions for tasks

- Defining constraints between more than two tasks.

2. What is the most useful feature of CPG-KPO in regard with CPG merging?

There is a consensus among the participants that all constraints are very important. The constraint that models task conflicts is the most mentioned constraint.

3. What is the biggest hindrance to using CPG-KPO for merging real computerized CPG?

Two of the participants expressed since it is common to have more than two concurrent conditions in elderly patients, it is crucial to improve how CPG-KPO merges three or more CPG. No specific details were provided. Another participant believes that lack of the ability to model CPG evolution is the biggest hindrance to CPG merging.

9.3.2. Evaluation of the Merge Execution Engine

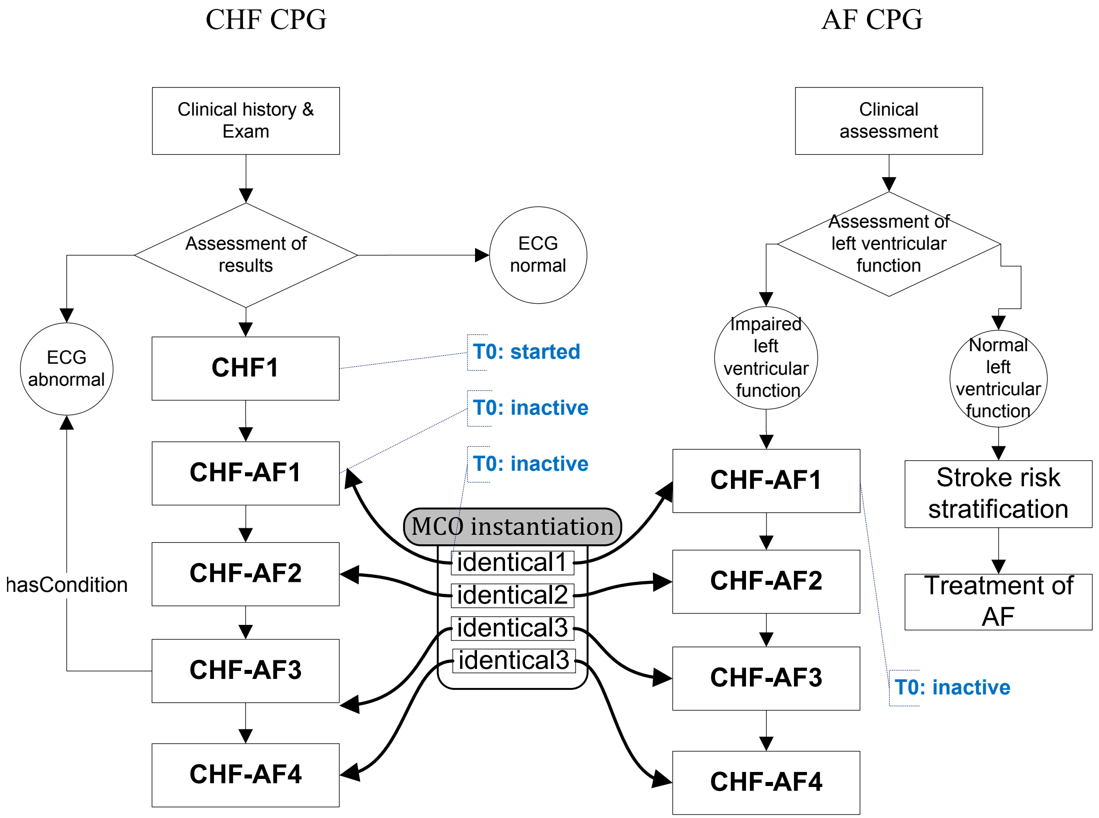
As we saw previously, we instantiated CPG-KPO in section 8.3.3 in order to capture the merging constraints between CPG in 6 real comorbidities. We use those real comorbidity cases to evaluate our CPG-KPO and merge execution engine. We have also created 10 imaginary examples of the merged CPG in order to evaluate the capability of CPG-KPO in modeling complex conditions, conflicts and the constraints that we have not found in our real examples. For each of the real or imaginary comorbidities, we have created at least four imaginary patients scenarios and executed the merged CPG using our merge execution engine.

During execution, we have recorded all the state transitions and compared them to the desired state transitions defined by the execution semantics of the tasks and constraints. Please note that the evaluation of the medical correctness of the generated outcome is not the objective of this comparison and we are only evaluating the capabilities of the execution engine from a workflow point of view. In all 16 examples our merge execution engine can successfully model the constraints' state transitions and their effects on tasks' state transition systems. We go through 4 real example scenarios in order to show how we evaluate the capabilities of merge execution engine in dynamically merge several CPG during execution. We use tables to describe execution scenarios by listing the states of the tasks and constraints at each step of the execution. We also use a logical clock to keep track of time: Logical time is zero at the beginning and increases when a user either starts a task (step 3 of the merging algorithm) or completes a task (step 4 of the merging algorithm) and the reasoning happens in between two logical times. Each logical time is considered as a step of the execution and it is broken into several sub-steps in order to show the sequence in which the reasoning happens in the OWL reasoner.

9.3.2.1. CPG Merging Scenario 1: CHF-AF

CHF-AF comorbidity CPG ontology [107] gives us several opportunities to evaluate our merge execution engine. In this section, we show how we have evaluated the capability of our execution engine in handling *IdenticalActionConstraint* using CHF-AF comorbidity CPG. In order to merge CHF and AF CPG, we first transformed the instantiations of these ontologies to CPG-DKO using our ontology mapping methodology described in chapter 7 of the thesis. We then instantiated the CPG-KPO in order to capture the morphing constraints between the two transformed CPG.

We have created several *IdenticalActionConstraint*s in order to capture the identical tasks between the CHF and AF CPG. An example of these identical tasks is "*pre-treatment electrolyte assessment & correction"(*CHF-AF1) which might be executed in both of the CPG for comorbid patients. The constraint merging these two tasks is called indetical1 in the instantiation of the CPG-KPO. Figure 8.2 shows the AF and CHF CPG flowcharts and the instantiation of the CPG-KPO that is used to merge these two CPG. In order to see how we evaluate the capability of the merge execution engine in handling the constraints we go through a part of one of the imaginary patient scenarios. We have annotated the tasks and constraints participating in the merge with their states in the first time step that we start to discuss the scenario.



CHF1: Assessment of echocardiography result
CHF-AF1: Pre-treatment electrolyte assessment & correction
CHF-AF2: Initiation of treatment of heart failure
CHF-AF3: Thromboprophylaxis in patients with CHF & AF
CHF-AF4: Treatment of AF in patient with heart failure

Figure 9.7 The instantiation of the CPG-KPO and the parts of the CHF and AF CPG that participate in the merge. Important tasks and constraints are annotated with their initial states in our example scenario.

We have recorded the state transition of the constraints and the tasks during this scenario and compared them with the expected state transitions dictated by the execution semantics. The states of the tasks and constraints during our scenario are given in Table 9.2 for each logical time.

Table 9.17 State transition of tasks and constraints during execution of the merged CPG in Figure 8.2. New states are bolded and the state transitions performed by the user are underlined.

		CHF			AF	Constraints
		CHF1	**CHF-AF1**	**CHF-AF2**	**CHF-AF1**	**Identical1**
T0		Started	Inactive	Inactive	Completed	Inactive
T1	1	**Completed**	Inactive	Inactive	Completed	Inactive
	2	Completed	**Active**	Inactive	Completed	Inactive
	3	Completed	Active	Inactive	Completed	**Active**
	4	Completed	Active	Inactive	Completed	**Started**
	5	Completed	**Completed**	Inactive	Completed	Started
	6	Completed	Completed	**Active**	Completed	**Completed**

The below descriptions provide a detailed account of some of the more interesting state transitions and their significance in dynamically merging comorbidity CPG during one of our merging scenarios:

T0: Our scenario starts from this step. Imagine that the task chf:CHF1 is in the started state and the common task CHF-AF1 (“*pre-treatment electrolyte assessment & correction*”) is already executed in the AF CPG three days ago. This task is still inactive in the CHF CPG because its previous task chf:CHF1 is still under execution.

T1: Several state transitions are inferred by the OWL reasoner in this step. The description of these state transitions are sorted based on the sequence they are inferred in the OWL reasoner: T1-1: In this sub-step, the user indicates that he has completed the task chf:CHF1. T1-2: According to tasks’ state transition rules, the task chf:CHF-AF1 becomes active. T1-3: Since the constraint indetical1 indicates that tasks chf:CHF-AF1 and af:CHF-AF1 are identical and should not be executed twice, execution of the chf:CHF-AF1 will be violating this constraints. According to the constraints' state transition rules, identical1 will go to the active state because the results of the previous execution are still valid. These results are still valid because only three days have been passed after execution of this common task and the validity period of those results are 10 days according to identical1. T1-4: Since no constraint is conflicting with identical1, it goes to the started state in order to avoid its violation. T1-5: The identical action constraint identical1 will cause the task chf:CHF-AF1 to go to completed state according to the rules written to implement the effect of constraints on tasks' state transitions in section 8.4.2. T1-6: According to constraints' state transition rules, identical1 will go to the completed state because the tasks merged by it are in the completed state as well.

The rest of the state transitions happen in a very similar fashion. All the discussed state transitions are in accordance with what we have implemented in our state transition systems for both tasks and constraints. Therefore, this imaginary scenario and other manually designed scenarios for evaluation show that our merging execution engine can successfully merge several comorbidity CPG form a workflow point of view.

9.3.2.2. CPG Merging Scenario 2: Transient Ischemic Attack-Duodenal Ulcer

In order to evaluate the capability of our CPG execution engine in handling *TaskSubstituteInCaseOfConflict* we have used the example described in [25]. In this example the transient ischemic attack (TIA) and duodenal ulcer (DC) CPG are merged. We modeled the workflow structure of these CPG in CPG-DKO based on the description and the flowcharts provided in [25]. Then we merged these two CPG by instantiating CPG-KPO. The only existing merge constraint between these two CPG is the conflict between the tasks tia:A ("*Give Aspirin*") from transient ischemic attack and the task dc:AS ("*Stop taking aspirin if used*") from the duodenal ulcer. In this conflict, the task tia-dc:DGA ("*do not give aspirin*") will replace the task tia:GA ("*give aspirin*") in the case of a conflict. Figure 9.8 shows these two CPG and the instantiation of the CPG-KPO that merges them.

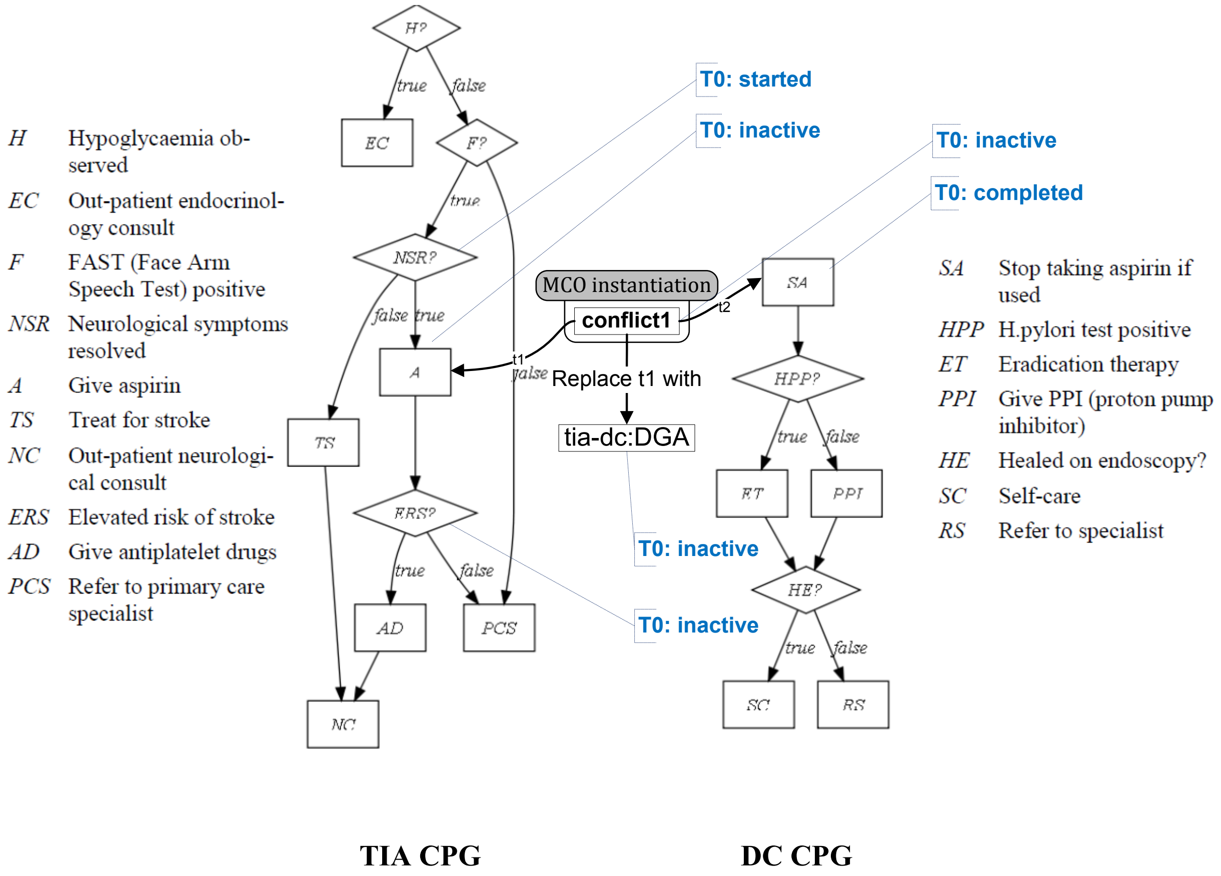


Figure 9.8 The instantiation of the CPG-KPO and the parts of the TIA and DC CPG that participate in the merge. The graphical representation of CPG are taken from [25]. Important tasks and constraints are annotated with their initial states in our example scenario.

We used four different scenarios in order to evaluate the correctness of the merging from a workflow point of view. During merging, we recorded the state transitions in both tasks and constraints and compared them with the transitions dictated by the execution semantics. The below descriptions provide a detailed account of some of the more interesting state transitions and their significance in dynamically merging comorbidity CPG during one of our merging scenarios:

T0: Our scenario starts in this step. Imagine that the SA CPG has been previously executed when the execution engine activates the task tia:A. This means that the task dc:SA is already in the completed state and the task tia:A is in active state.

T1: Several state transitions are inferred by the OWL reasoner in this step. The description of these state transitions are sorted based on the sequence they are inferred in the OWL reasoner: T1-1: tia:NSR task is completed by the user. Therefore, this task goes to the completed state. T1-2: Since the task tia:NSR is completed, the next inactive task which is tia:A becomes activated. T1-3: Execution of this task is in conflict with the task dc:SA according to conflict1. Therefore, this constraint becomes active in order to avoid its violation. T1-4: Since no conflicting constraints exist, this constraint will go the started state. T1-5: since conflict1 is started, it will cause the conflicting task tia:A to become discarded and the substitute comorbidity task tia-dc:DGA ("*don't give aspirin*") to become active. In this way, the conflicting task has been replaced by an alternative task. T1-6: When the merged tasks (tia:A and dc:SA) are both completed or discarded the constraint merging them (conflict1) will be regarded as a completed constraint.

T2: The substitute task tia-dc:DGA has been chosen for execution by the user in this step.

The states of the tasks and constraints are given in Table 9.18 for each logical time.

Table 9.18 State transition of tasks and constraints during execution of the merged CPG in Figure 8.3. New states are bolded and the state transitions performed by the user are underlined.

		TIA			TIA-DC	DC	Constraints
		NSR	**A**	**ERS**	**DGA**	**SA**	**const1**
T0		Started	inactive	Inactive	inactive	Completed	inactive
T1	1	**Completed**	inactive	Inactive	inactive	Completed	inactive
	2	Completed	**active**	Inactive	inactive	Completed	inactive
	3	Completed	active	Inactive	inactive	Completed	**active**
	4	Completed	active	Inactive	inactive	Completed	**Started**
	5	Completed	**discarded**	Inactive	**active**	Completed	Started
	6	Completed	discarded	Inactive	active	Completed	**completed**
T2		Completed	discarded	Inactive	**Started**	Completed	Started

All the discussed state transitions are in accordance with what we have implemented in our state transition systems for both tasks and constraints. Therefore, this imaginary scenario and other manually designed scenarios for evaluation show that our merging execution engine can successfully merge several comorbidity CPG form a workflow point of view.

9.3.2.3. CPG Merging Scenario 3: Osteoarthritis-Hypertension

Osteoarthritis and hypertension are two diseases that may co-occur in patients. In order to use these diseases for evaluation of our merging execution engine we computerized Osteoarthritis treatment algorithm [169] published by National Health Services Tayside of Scotland and clinical pathways of diabetes [171] published by National Institute of Health and Clinical Excellence of UK in CPG-DKO.

During the treatment of Osteoarthritis-Hypertension patient for osteoarthritis, non-steroidal anti-inflammatory drugs (NSAD) such as aspirin, ibuprofen, and naproxen may be prescribed. However, these drugs aggravate the hypertension patients by increasing the blood pressure. According to the domain expert, the solution is to replace any of these drugs with alternatives such as acetaminophen, tramadol or narcotic analgesics. These conflicts can be modeled using *TaskSubstituteInCaseOfConflict* constraints. As we discussed in section 8.3.3.3 we found three cases of prescribing NSAD drugs in the osteoarthritis treatment algorithm. These three tasks are: (1) "*add Ibuprofen 1 point 2g day*", (2) "*substitute Naproxen Instead Of Ibuprofen*" and (3) "*inquiry about possibility of continuing Ibuprofen after 6 month*". Figure 9.9 shows the parts of the osteoarthritis and hypertension pathways that participate in the CPG merging and the instantiation of the CPG-KPO that merges these two CPG.

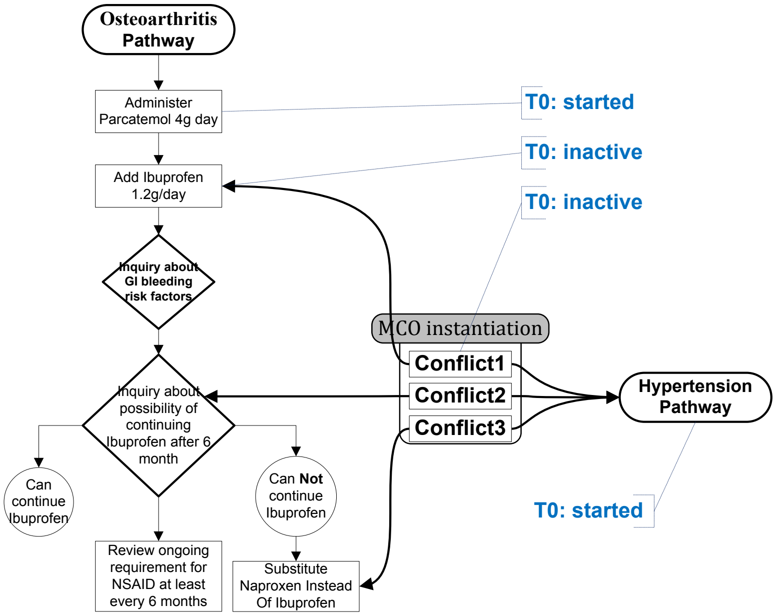

Osteoarthritis Pathway
Administer Parcatemol 4g day
Add Ibuprofen 1.2g/day
Inquiry about GI bleeding risk factors
Inquiry about possibility of continuing Ibuprofen after 6 month
Can continue Ibuprofen
Can Not continue Ibuprofen
Review ongoing requirement for NSAID at least every 6 months
Substitute Naproxen Instead Of Ibuprofen
T0: started
T0: inactive
T0: inactive
MCO instantiation
Conflict1
Conflict2
Conflict3
Hypertension Pathway
T0: started

Figure 9.9 The instantiation of the CPG-KPO and the parts of the osteoarthritis and hypertension pathways that participate in the merge. Important tasks and constraints are annotated with their initial states in our example scenario.

We used four different scenarios in order to evaluate the correctness of the merging from a workflow point of view. During merging, we recorded the state transitions in both tasks and constraints and compared them with the transitions dictated by the execution semantics. The below descriptions provide a detailed account of some of the more interesting state transitions and their significance in dynamically merging comorbidity CPG during one of our merging scenarios:

T0: Our scenario starts from this step. In this step, the hypertension pathway is under execution (in the started state) for the patient and the task os:AP ("*administer parcatemol*") is in the started state.

T1: Several state transitions are inferred by the OWL reasoner in this step. The description of these state transitions are sorted based on the sequence they are inferred in the OWL reasoner: T1-1: os:AP task is completed by the user. Therefore, this task goes to the completed state. T1-2: Since the task os:AP is completed, the next inactive task which is os:A becomes activated. T1-3: Execution of this task is in conflict with the task ht:hypertension_CPG which is already in the started state according to the conflict1 constraint. Therefore, the constraint conflict1 becomes active in order to avoid its violation. T1-4: Since no conflicting constraints exist, this constraint will go the started state. T1-5: since conflict1 is started, it will cause the conflicting task os:A to become discarded and the substitute task os-ht:RNSAD ("*don't give aspirin*") to become active. In this way, the conflicting task has been replaced by an alternative solution. T1-6: When the merged tasks (os:A and ht:hypertension_CPG) are both completed or discarded the constraint merging them will also be regarded as a completed constraint.

The states of the tasks and constraints are given in Table 9.19 for each logical time.

Table 9.19 State transition of tasks and constraints during execution of the merged CPG in Figure 8.4. New states are bolded and the state transitions performed by the user are underlined. (A = Active, I = Inactive, S = Started, D = Discarded, C = Completed)

		Osteoarthritis		**Osteoarthritis-Hypertension**	**Hypertension**	**Constraints**
		AP	**A**	**RNSAD**	**Hypertension CPG**	**conflict1**
T0		S	I	I	S	I
T1	1	**C**	I	I	S	I
	2	C	**A**	I	S	I
	3	C	A	I	S	**A**
	4	C	A	I	S	**S**
	5	C	**D**	**A**	S	S
	6	C	D	A	S	**C**
T2		C	D	**S**	S	C
T3		C	D	**C**	S	C

All the discussed state transitions are in accordance with what we have implemented in our state transition systems for both tasks and constraints. Therefore, this imaginary scenario and other manually designed scenarios for evaluation show that our merging execution engine can successfully merge several comorbidity CPG form a workflow point of view.

9.3.2.4. CPG Merging Scenario 4: CHF-AF

CHF-AF comorbidity can also be used to evaluate the execution of the *SimultanousActionConstraint* in our merging execution engine. During treatment of the CHF patients, angiotensin-converting-enzyme inhibitors (ACEI) and beta blockers (BB) may be prescribed for the patients represented by the task "*initiate treatment with beta* *blockers in addition to ACEI*'. Moreover, AF patients may need to take Warfarin represented by the task "*anticoagulation with Warfarin*" to avoid formation of blood clots. Domain experts believe that simultaneous prescription of these drugs has better overall results in treatment of CHF-AF comorbid patients. As we saw in the example of section 8.3.3.4 we created three *SimultaneousActionConstraint* called simacconst1, simacconst2 and simacconst3. For more details about these constraints, refer to section 8.3.3.4. Figure 9.10 shows the abovementioned three merging constraints between the AF and CHF CPG and the parts of the CPG that participate in the merging.

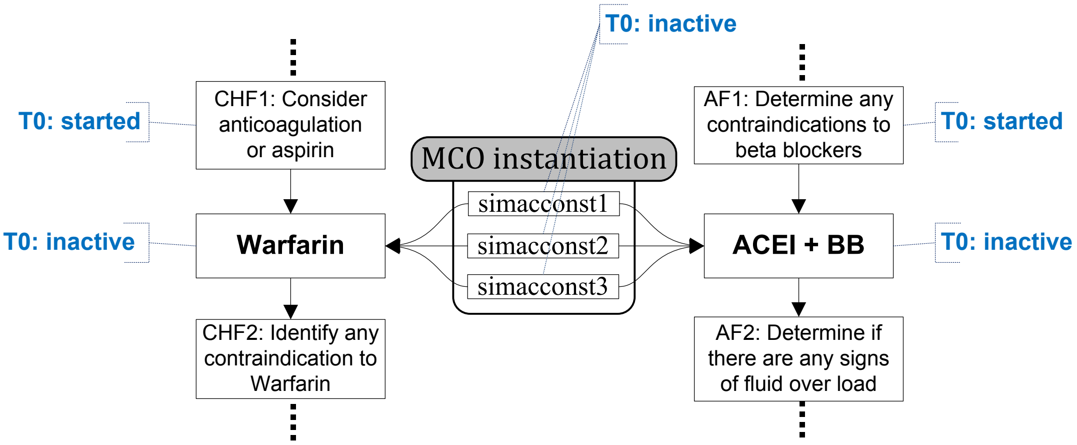

T0: inactive
T0: started
CHF1: Consider anticoagulation or aspirin
MCO instantiation
AF1: Determine any contraindications to beta blockers
T0: started
T0: inactive
Warfarin
simacconst1
simacconst2
simacconst3
ACEI + BB
T0: inactive
CHF2: Identify any contraindication to Warfarin
AF2: Determine if there are any signs of fluid over load

Figure 9.10 The instantiation of the CPG-KPO and the parts of the CHF and AF CPG that participate in the merge. Important tasks and constraints are annotated with their initial states in our example scenario.

We used four different scenarios in order to evaluate the correctness of the merging from a workflow point of view. During merging, we recorded the state transitions in both tasks and constraints and compared them with the transitions dictated by the execution semantics. The below description provide a detailed account of the state transitions and their significance in dynamically merging comorbidity CPG during one of our merging scenarios:

T0: Our scenario starts in this time step. Both tasks chf:CHF1 and af:AF1 are started and the af:ACEI+BB and chf:Warfarin are inactive.

T1: Several state transitions are inferred by the OWL reasoner in this step. The description of these state transitions are sorted based on the sequence they are inferred in the OWL reasoner: T1-1: In this step, the user indicates that the task chf:CHF1 is completed. T1-2: The task after chf:CHF1 which is chf:Warfarin gets activated according to the tasks' state transition rules. T1-3: Since af:ACEI+BB is still inactive, execution of the task chf:Warfarin will violate the constraint simacconst1, simacconst2 and simacconst3. If we assume that the simacconst1 has satisfied conditions and the task af:ACEI+BB can catch up in an acceptable time frame for the task chf:Warfarin, this constraint will go from inactive to active state. T1-4: This constraint has no conflicting constraints therefore it will go to the started state according to the constraint state transition rules in section 8.4.1. T1-5: Since the simacconst1 is started, it will cause the task chf:Warfarin to go to the pending state so that the task af:ACEI+BB can catch up later.

T2: In this step, user indicates that he is completed the task af:AF1. According to tasks' state transition rules this will lead to activation of the task af:ACEI+BB.

T3: T3-1: In this step, user indicates that he has started the execution of the task af:ACEI+BB. T3-2: Since the tasks af:ACEI+BB and chf:Warfarin are supposed to be executed simultaneously, the task chf:Warfarin also goes to the started state from the pending state. This state transition is the effect of the constraint simacconst1 according to the rules described in section 8.4.2. In this way, both of the tasks go under execution simultaneously.

T4: T4-1: In this step user indicates that both tasks af:ACEI+BB and chf:Warfarin are completed. T4-2: Since all the tasks merged by simacconst1 are completed, this constraint will be regarded as completed as well.

This scenario is summarized in Table 9.20.

Table 9.20 State transition of tasks and constraints during execution of the merged CPG in Figure 8.5. New states are bolded and the state transitions performed by the user are underlined. (A = Active, I = Inactive, S = Started, D = Discarded, C = Completed, P = Pending)

		CHF		AF		Constraints		
		CHF1	Warfarin	AF1	ACEI+BB	Simacconst1	Simacconst2	Simacconst3
T0		S	I	S	I	I	I	I
T1	1	**C**	I	S	I	I	I	I
	2	C	**A**	S	I	I	I	I
	3	C	A	S	I	**A**	I	I
	4	C	A	S	I	**S**	I	I
	5	C	**P**	S	I	S	I	I
T2	1	C	P	**C**	I	S	I	I
	2	C	P	C	**A**	S	I	I
T3	1	C	P	C	**S**	S	I	I
	2	C	**S**	C	S	S	I	I
T4	1	C	**C**	C	**C**	S	I	I
	2	C	C	C	C	**C**	I	I

All the discussed state transitions are in accordance with what we have implemented in our state transition systems for both tasks and constraints. Therefore, this imaginary scenario and other manually designed scenarios for evaluation show that our merging execution engine can successfully merge several comorbidity CPG form a workflow point of view.

9.4. Ontology Metrics

As we discussed previously, an approach to evaluate ontologies is to extract a number of metrics from them and compare them to those of a golden standard ontology. There have been numerous efforts for extracting metrics from ontologies in order to find the "best" ontology that best suits the need of the application at hand [177] among a number of ontologies. One of the most popular set of ontology metrics have been proposed by Tartir et al. [178]. They categorize their proposed metrics as Schema and Knowledge metrics. Schema metrics represent the features of the ontology design and knowledge metrics describe how data is represented in terms of instances of the ontology classes and properties. We briefly discuss these metrics below:

1. Schema metrics

- **Relation Richness:** This metric is the ratio of the number of non-subsumption relations divided by the total number of relations between classes of the ontology. Higher values of this metric indicate higher diversity of relations between classes and more expressivity as the result in the ontology.

- **Attribute Richness:** This metric is defined as the ratio of the number of properties (data type and object property) divided by the total number of classes in the ontology. Ontologies with higher values for this metric are more likely to be able to capture the domain knowledge in more detail.

- **Inheritance Relation:** This metric is defined as the ratio of the number of subsumption relations between classes of the ontology divided by the total number of classes. Class hierarchy of ontologies with high values for this metric are likely to have more width than depth. Therefore, these ontologies are more horizontal and more likely to have better coverage of the domain knowledge as opposed to ontologies with lower values for this metric. Class hierarchy of ontologies with lower values for this metric are more vertical and detailed in a specific part of the domain area.

2. Knowledge metrics

- **Class Richness:** This metric is defined as the number of classes with asserted instances divided by the total number of classes. Very low values of this metric can be indicative of unnecessary and redundant classes.

- **Class Importance:** Importance of a class is defined as the number of asserted and inferred instances of that class divided by the total number of instances in the ontology. This metric can be used to identify the key concepts of the domain knowledge in the ontology.

- **Relationship Richness:** This class-specific metric is defined as the ratio of the number of properties of a class and properties of its asserted and inferred super classes that have been used, divided by the total number of those properties. An ontology that contains classes with low values for this feature might contain redundant and unnecessary properties.

We have implemented a Java program for calculation of the above-mentioned metrics using Jena library [160]. In this section of the thesis, we extract these metrics from our ontologies. Since no golden standard OWL ontology exists for any of the ontologies that we have developed in our thesis, we merely report these metrics and discuss them. The only ontology that has a number of OWL counterparts is CPG-DKO ontology. We compare the schema metrics and the class richness of these ontologies and discuss them in the next section.

9.4.1. Ontology Metrics of CPG-DKO

9.4.1.1. Schema Metrics and Class Richness of CPG-DKO

Since no golden standard ontologies exist for representation of CPG, we compare CPG-DKO with the OWL ontologies available to us that have been used for computerization of CPG: CHF-AF [107], CAP [30] and NCPG [145]. While we compare the metrics, please bear in mind that these ontologies are used for modelling both workflow structure and medical knowledge of CPG whereas CPG-DKO is only used for modelling the workflow structure of CPG. Schema metrics and class richness of CPG-DKO, CHF-AF, CAP Ontology and NCPG Ontology are listed in Table 9.21.

Table 9.21 Schema metrics and class richness metric of CPG-DKO, NCPGO

	# of Classes	**#of Properties**	**# of Instances**	**Relationship Richness**	**Attribute Richness**	**Inheritance Richness**	**Class Richness**
CPG-DKO	42	75	1250	0.567	1.786	1.524	0.571
NCPG	98	144	372	0.0	1.469	0.490	0.347
CHF-AF	108	39	555	0.0	0.361	2.361	0.759
CAP	50	123	229	0.015	2.4	1.24	0.6

In the rest of this section, we compare the extracted metrics in Table 9.21.

1. Number of classes, properties and instances

As you can see in the Table 9.21, our ontology contains less properties and classes. This is due to the fact that our ontology is only capable of representing the workflow structure of CPG while others can be used to model the medical knowledge as well. Even considering this fact, other ontologies are not significantly superior to our ontologies in terms of these metrics. Our ontology contains way more instances compared to the rest of the ontologies because we have mapped and transformed instances of the compared ontologies to CPG-DKO.

2. Relationship richness

As you can see in Table 9.21, CPG-DKO is significantly superior to the rest of the CPG representation ontologies in terms of relationship richness. This shows that our ontology, other than subsumption relation, contains several other relations between classes such as *disjointWith*, *equivalentClass*, *complementOf*, etc. Our ontology has a greater value for this metric because we have defined the formal semantics in the ontology using these class relations. Therefore, we can conclude that the emphasis of the rest of the ontologies is on defining the class hierarchy rather than formally capturing the formal semantics of the domain knowledge.

3. Attribute richness

Table 9.21 shows that our ontology has a higher attribute richness compared to two out of three CPG ontologies. Review of CAP ontology that has a higher value for this metric reveals that this ontology contains 64 data type properties that are mostly used for capturing Meta-data and drug administration scheduling and dosages. However, we found that only four out of those 64 data-types might be actually used for modelling the workflow structure of the CPG. Not considering the irrelevant datatype properties can lessen this metric for CAP otology.

4. Inheritance richness

Table 9.21 shows that our ontology has a higher inheritance richness compared to two of the compared ontologies. Review of the CHF-AF ontology that has a higher value than CPG-DKO for this metric shows that several classes have been defined to represent the medical knowledge regarding Chronic Heart Failure and Atrial Fibrillation treatments and drugs. These classes are very detailed and as a result contribute in the high inheritance richness of this ontology. Moreover, classes that are related to the workflow structure of the CPG are categorized into several subclasses that are of no executional value and are interpreted the same in the execution engine. For instance, the class *Decision_option* in CFH-AF ontology has 12 subclasses that are all treated similarly during execution.

5. Class richness

Our ontology has a lower value for this metric compared to two of the CPG ontologies. We believe that this is because we have created several class hierarchy levels that are used for execution and not instantiation necessarily. For instance, class *Cycle* that has been repeatedly used in OWL triples pertaining to formal semantics is not instantiated. In order to define a cycle in CPG-DKO one of its subclasses such as *ForLoop* should be instantiated. Moreover, many of the classes related to execution such as *ActiveTask* or *SatisfiedCondition* classes will be instantiated during the execution.

9.4.1.2. Knowledge Metrics of CPG-DKO

In this section, we review the class connectivity, class importance and relation richness of CPG-DKO classes. Table 9.22, Table 9.23 and Table 9.24 show the top ten classes based on class connectivity, class importance and relation richness respectively.

Table 9.22 Top ten classes in CPG-DKO based on class connectivity

Class	**Connectivity**
Task	1350
CompositeTask	590
AtomicTask	363
MedicalWorkflow	285
Split	68
Cycle	68
WhileLoop	52
ActiveTask	51
WhileLoopMultipleExit	36
Synch	16

Table 9.22 shows the class with most connections is the *Task* class that represents medical tasks.

Table 9.23 Top ten classes in CPG-DKO based on class importance

Class	**Importance**
Task	0.635
Outcome	0.267
Condition	0.267
AtomicTask	0.103
CompositeTask	0.071
MedicalWorkflow	0.028
State	0.025
TaskState	0.017
Split	0.014
Cardinality	0.011

Table 9.23 shows the class with the most connections is the *Task* class.

Table 9.24 Top ten classes in CPG-DKO based on relation richness

Class	**Relation Richness**
AtomicTask	1.0
Synch	0.75
MultipleOptionDecision	0.571
Decision	0.5
IfThenElse	0.5
Task	0.45
Split	0.333
WhileLoop	0.333
ForLoop	0.333
WhileLoopMultipleExit	0.333

Table 9.24 shows that the only class that makes use of all its associated properties is the *AtomicTask* class. The reason that some classes do not have value 1.0 for this metric is (1) Lack of enough instantiations that makes use of all the properties and (2) existence of subclasses that make use of their super class' execution semantics but do not use some of their properties. For instance, *WhileLoopMultipleExit* makes use of the execution semantics of *WhileLoop* but uses its own proprietary properties for defining the termination condition.

9.4.2. Ontology Metrics of KMO

In this section of the thesis, we report the ontology metrics of KMO.

Table 9.25 shows the number of ontology elements, relationship richness, attribute richness, inheritance richness and class richness of KMO.

Table 9.25 Schema metrics and class richness metric of KMO

	# of Classes	#of Properties	# of Instances	Relationship Richness	Attribute Richness	Inheritance Richness	Class Richness
KMO	53	60	341	0.010	1.132	3.641	0.623

As you can see in Table 9.25, the relationship richness is a very low number because most of the semantics of the mappings are represented in terms of SWRL rules that have not been taken into consideration by the used metrics. Table 9.26, Table 9.27 and Table 9.28 show top ten classes of KMO based on class connectivity, class importance and relation richness respectively.

Table 9.26 Top ten classes in KMO based on class connectivity

Class	Connectivity
Mapping	617
TransformationMapping	429
ClassVariable	422
ClassPropertyHasValueRestriction	413
RelationalMapping	290
Variable	281
ExpressionFunction	225
Function	225
CreateFunction	117
InstanceFromPropertyCreateFunction	112

Table 9.26 shows that the most connected class of the ontology is the *Mapping* class based on the existing instances in the ontology.

Table 9.27 Top ten classes in KMO based on class importance

Class	**Importance**
Variable	0.354
ClassVariable	0.246
ClassPropertyHasValueRestriction	0.225
Mapping	0.105
Operator	0.085
MathOperator	0.079
Function	0.067
ExpressionFunction	0.067
TransformationMapping	0.058
RelationalMapping	0.058

Even though the *Mapping* class is the most connected class in the ontology, Table 9.27 shows that the *Variable* class comprises more than 1/3 of the total instances. This is because several variables may be involved in a single mapping.

Table 9.28 Top ten classes in KMO based on relation richness

Class	Relation Richness
ClassPropertyHasValueRestriction	1.0
NumericVariable	1.0
ClassPropertyQualifiedCardinalityRestriction	1.0
RelationalMapping	1.0
Function	1.0
ClassToPropertyTransformationMapping	1.0
PropertyCreateFunction	1.0
PropertyToClassTransformationMapping	0.8
ClassVariable	0.75
Mapping	0.5

Table 9.28 shows that the relation richness is 1.0 for 6 out of ten top classes in KMO. This is an indication that ontology is not likely to contain redundant and useless properties.

9.4.3. Ontology Metrics of CPG-KPO

In this section of the thesis, we report the ontology metrics of KMO. Table 9.29 shows the number of ontology elements, relationship richness, attribute richness, inheritance richness and class richness of CPG-KPO ontology.

Table 9.29 Schema metrics and class richness metric of CPG-KPO

	# of Classes	#of Properties	# of Instances	Relationship Richness	Attribute Richness	Inheritance Richness	Class Richness
CPG-KPO	61	66	158	0.096	1.082	1.852	0.459

Since this ontology has formal semantics, one may expect to see a high value for relation richness. However, as we can see in Table 9.29, this metric is very close to zero indicating that a great majority of relations between ontology classes are *owl:subClassOf*. This is because that most of the formal semantics of this language is defined in SWRL rather than OWL. Table 9.30, Table 9.31 and Table 9.32 show top ten classes of CPG-KPO based on class connectivity, class importance and relation richness respectively.

Table 9.30 Top ten classes in CPG-KPO based on class connectivity

Class	**Connectivity**
Constraint	338
Function	192
MedicalConstraint	158
WorkflowConstraint	144
TaskSubstituteInCaseOfConflict	81
UseResultsConstraint	77
SimultaneousActionConstraint	69
IdenticalActionConstraint	53
ConstraintWithSatisfiedCondition	44
PrecedenceConstraint	12
MergingConstraint	10

As one could expect, Table 9.31 shows that instances of the *Constraint* class has the most number of connections to instances of other classes.

Table 9.31 Top ten classes in CPG-KPO based on class importance

Class	**Importance**
Constraint	0.387
Task	0.265
WorkflowConstraint	0.120
MedicalConstraint	0.094
SimultaneousActionConstraint	0.069
TaskSubtituteInCaseOfConflict	0.056
State	0.051
Cardinality	0.038
ResourceType	0.038
UseResultsConstraint	0.038

Table 9.31 shows the most instantiated classes of the ontology is the *Constraint* class. As you can see, the *Task* class that is shared between CPG-KPO and CPG-DKO is the second most instantiated classes after the *Constraint* class. This is because each constraint is defined between two instances of the *Task* class; however, since several constraints are defined between the same two classes, constraint class has more instances.

Table 9.32 Top ten classes in CPG-KPO based on relation richness

Class	Relation Richness
Constraint	1.0
MedicalConstraint	1.0
PrecedenceConstraint	0.667
UseResultsConstraint	0.625
TaskSubtituteInCaseOfConflict	0.5
SimultaneousActionConstraint	0.445
OperationalConstraint	0.4
InstituteTimeCondition	0.375
Task	0.143
MergingConstraint	0.1

Review of Table 9.32 shows that some of the ontology classes have low values for the relation richness metric. We believe that this does not necessarily mean that the ontology contains redundant properties. Creating more instantiations and performing evaluations using real patient data will shed light on usefulness of the unused properties.

9.4.4. Ontology Metrics of Expression Ontology

In this section of the thesis, we report the ontology metrics of the Expression Ontology. Table 9.33 shows the number of ontology elements, relationship richness, attribute richness, inheritance richness and class richness of Expression Ontology.

Table 9.33 Schema metrics and class richness metric of the Expression Ontology

	# of Classes	#of Properties	# of Instances	Relationship Richness	Attribute Richness	Inheritance Richness	Class Richness
CPG-KPO	21	16	55	0.15	0.762	0.810	0.429

As you can see in Table 9.30, similar to CPG-KPO, this ontology has a close to zero value for the relationship richness metric. The reason is defining the formal semantics using SWRL rules rather than OWL triples. The only class with a non-zero value for class connectivity is the *Function* class. The value of class connectivity is 121 for this class. This does not mean that other classes are not instantiated. This means that they all have been used as the object of the OWL triples in the created instantiations. Table 9.34 shows the top ten classes of the Expression Ontology based on class importance and Table 9.35 shows the most instantiated classes in the Expression Ontology.

Table 9.34 Top ten classes in EO based on class importance

Class	Importance
Variable	0.436
Operator	0.291
Function	0.181
ComparatorOperator	0.127
NumericVariable	0.109
MathComparatorOperator	0.091
BooleanOperator	0.073
MathOperator	0.073
StringVariable	0.073
BooleanVariable	0.036

Table 9.34 shows the most instantiated classes in the Expression Ontology. As one may expect, *Variable*, *Operator* and *Function* classes are on top of the list.

Table 9.35 Classes with non-zero relation richness in Expression Ontology

Class	Relation Richness
StringVariable	1.0
BooleanVariable	1.0
NumericVariable	1.0
Function	0.8

As you can see in Table 9.35, most of the properties defined in the Expression Ontology have been used by the instances of this ontology. This means that this ontology is less likely to contain useless properties.

9.5. Conclusion

In this section, we evaluated our knowledge morphing framework as a CPG merging framework. We evaluated (1) CPG-DKO, the Expression Ontology and the OWL-based CPG execution engine; (2) KMO and the algorithm that translates its instantiations to OWL + SWRL; and (3) CPG-KPO and the CPG merging execution engine. Below we conclude the evaluation of these components of our solution.

- **CPG-DKO and OWL-based execution engine:**

Comparison of CPG-DKO with popular CPG computerization languages showed that CPG-DKO provides the most comprehensive set of workflow patterns expressible among all these languages. This shows that CPG-KPO can be potentially used as the comprehensive domain knowledge ontology for procedural aspect of CPG. We also extracted a number of metrics proposed in [178] from CPG-DKO and compared them with those of three other CPG ontologies. Our comparisons show that our ontology is more expressive than these ontologies and it is not likely to contain redundant classes and properties. A limitation of this evaluation is not taking into consideration some other procedural aspects of CPG such as [4]: Specification of goals/intention, Representing effects of actions and reasoning with them, Expression language capabilities and Patient information model.

We computerized three clinical pathways and transformed instantiations of three CPG ontologies to CPG-KPO. In all cases, execution results were consistent with the provided flowchart. Moreover, execution results of Nursing Clinical Practice Guidelines in our OWL CPG execution engine were consistent with their execution results in their proprietary developed execution engine in [145]. As a result, we conclude that (1) CPG-DKO is capable of representation of a wide range of CPG and (2) CPG execution engine can interpret the workflow structure of the computerized CPG correctly. As a future work, it will be interesting to use real patient scenarios to generate recommendations and evaluate them for medical validity with the help of domain experts. Moreover, execution of more complex CPG that for instance contain loops with complex continuation conditions will further evaluate CPG-DKO and the CPG execution engine.

- **KMO and the Translation Algorithm:**

Comparison of KMO with the existing ontology mapping representation languages reveals that it is more expressive in terms of representation of mappings patterns, variables, meta-modelling constructs, structural modification and data manipulation operators and conditions. We performed more evaluations by mapping several CPG ontologies to CPG-DKO and execution of the transformed CPG. In all cases, the execution results are consistent with the workflow diagram and the results of the original execution engines of those CPG. This shows that mappings are represented correctly and instance transformation has been successful in these test cases. Moreover, ontology quality metrics show that this ontology is not likely to contain unnecessary classes and properties. To improve this evaluation, ontologies from different domains such as aviation can be useful in order to show that our ontology mapping module is multipurpose.

- **CPG-KPO and the merge execution engine:**

Participants of our survey found CPG-KPO moderately easy to use, expressive and useful for the task of CPG merging. They also found the purpose of the concepts in CPG-KPO clear. Ontology quality metrics show that this ontology is not likely to contain unnecessary classes and properties. We also used several imaginary examples and a number of real examples from Chronic Heart Failure-Atrial Fibrillation, Transient Ischemic Attack - Duodenal Ulcer, Diabetes - Hypertension - Osteoarthritis comorbidities. In all cases, execution results are consistent with the provided flowcharts and the merging constraints for imaginary patient scenarios. Therefore, our evaluations show the correctness of the generated recommendation from a workflow point of view. However, medical evaluation of the generated recommendations with real patient scenarios will shed light on efficacy and usefulness of our CPG merging framework. It is also interesting to find more real CPG merging examples and evaluate the efficacy of the CPG merging framework in representation of the merging constraints and execution-time CPG merging.

CHAPTER 10: CONCLUSION

Domain experts may use the knowledge encapsulated in several knowledge artifacts to make decisions. This is due to the fact that a single knowledge source does not cover all the domain knowledge needed for efficient decision making. Thus, the required knowledge may be dispersed among several different knowledge sources. In the same way that domain experts may integrate the knowledge in several knowledge sources for decision making, computerized DSS can benefit from using several computerized knowledge sources [97]. The solution for decision making using several ontologies is knowledge morphing whereby several ontologies are merged to create a holistic view of the domain knowledge. This holistic view can be used by knowledge execution algorithm to provide improved and conflict-free decision making [12]. Knowledge morphing has been described by Abidi as "*the intelligent and autonomous fusion/integration of contextually, conceptually and functionally related knowledge objects that may exist in different representation modalities and formalisms, in order to establish a comprehensive, multi-faceted and networked view of all knowledge pertaining to a domain-specific problem*" [97].

The challenge towards merging and execution of the knowledge in several knowledge artifacts in ontology-based DSS is heterogeneity of these ontologies. These ontologies called Local Knowledge Ontologies (LKO) in this thesis are different in their vocabulary, points of view, coverage of the domain knowledge and level of granularity. In this thesis, we developed a semantic web based knowledge morphing framework in order to merge and execute the knowledge modeled in several heterogeneous OWL ontologies based on pre-defined constraints that govern the merging of the ontologies. Therefore, the input of our framework is several heterogeneous LKO in OWL representing different knowledge sources of a specific domain and the output is decision support for the problem at hand based on the dynamically merged knowledge of those knowledge artifacts. Our framework has three general steps namely; Ontology Mapping, Ontology Merging and Ontology Execution. In the rest of this section, we review the challenges in each of these steps, our solution and contributions. Moreover, we discuss the limitations of our work and some potential directions for future work.

10.1. Ontology Mapping: Challenges, Solution, and Contributions

Ontology mapping involves the conceptual mapping and instance transformation of all the heterogeneous LKO to a comprehensive DKO.

The challenges to ontology mapping originate from the reality that existing ontology mapping languages suffer from lack of expressivity in representation of the mappings that can be used for effective instances transformation. Moreover, most of these languages do not define the formal semantics of their mapping representation language. Semantics of a mapping language that describes the formal definition of the mappings can be used by a reasoner in a semantic based ontology mapping approach for improving the existing mappings and performing automatic instance transformation between source and target ontologies. Another challenge is finding/creating of an expressive comprehensive DKO that any other ontology of that domain can be mapped to. Creation of this ontology is particularly challenging, as it should have the highest coverage of the domain knowledge and the highest level of expressivity for representation of the details in the covered knowledge compared to any other existing domain knowledge ontology.

In this thesis, to address the abovementioned ontology mapping challenges we developed an OWL-Full ontology called Knowledge Mapping Ontology (KMO) to represent mappings between two ontologies. An instantiation of this ontology represents mappings between a source and a target ontology. After mapping two ontologies, KMO instantiations are translated to OWL-DL + SWRL in order to be used in a reasoner for discovering new mappings and performing automatic instance transformation.

Moreover, we proposed and implemented a background ontology called Domain Knowledge Ontology (DKO) to alleviate the lack of overlap between ontologies and to account for the different levels of granularity in the source ontologies.

In terms of contributions, a comparison of our mapping ontology with the existing ontology mapping representation languages shows that our language is more expressive in terms of support for mappings patterns, variables, meta-modelling constructs, structural modification and data manipulation operators and conditions. We have achieved this by transforming mappings from OWL-Full to OWL-DL + SWRL instead of direct use of OWL-DL + SWRL. Using OWL-Full also makes the mappings less verbose—i.e. a smaller number of triples are used for representation of the mappings. Another important contribution of this thesis is defining the formal semantics of the mapping by translating the mappings to OWL-DL that is a subset of the DL. Therefore, we can say that our framework is the most expressive DL-based ontology mapping framework with formal semantics.

In order to evaluate the efficacy of our knowledge mapping framework, we used KMO to map three CPG ontologies to CPG-DKO and transform their instances to it. The evaluation results of this mapping are discussed in section **9.2.1**. We also extracted a number of ontology metrics proposed in **[178]** in order to evaluate the quality KMO. These metrics showed that KMO is not likely to contain redundant classes and properties.

- Limitations and Future Work

Generation of a comprehensive domain knowledge ontology can be time consuming and challenging specially in dynamic environments. Moreover, instantiation of the KMO is a labour intensive task that reduces usability of our knowledge mapping framework. An interesting improvement to our work can be automation of these two processes. The automatically generated comprehensive domain knowledge ontology and the mappings can be reviewed by domain experts for further extension and enhancement.

Our knowledge mapping framework cannot transform instances between inconsistent ontologies due to incapacity of OWL reasoners in reasoning on inconsistent ontologies. Instance transformation between ontologies can also be computationally expensive because the worst case complexity of reasoning on OWL-DL ontologies is NEXPTIME-complete [152]. Extracting sub-contextualized ontologies [12] will improve our framework by reducing the risk of inconsistencies and speeding up the instance transformation process by reasoning on the extracted sub-ontologies.

10.2. Ontology Merging Challenges, Solution, and Contributions

In ontology merging, two or more LKO (transformed to DKO) are merged by defining the merging constraints. In the context of decision making, these constraints describe how the decision made according to each of those LKO should be modified in order to improve a set of predefined metrics such as overall duration or cost of the decision making process. We refer to this process as the decision reconciliation process. For ontology merging we encountered three research challenges as follows:

i. Identifying the merging constraints between knowledge artifacts of the specific domain that the DSS is developed for.

ii. Developing a computer understandable language for representation of the merging constraints

iii. Defining formal semantics that can be used by a reasoner for automatic decision reconciliation.

In this thesis we identified and represented merging constraints using an ontology called *Knowledge morPhing Ontology (KPO)*. These merging constraints represent the logic for decision reconciliation according the current state of the problem. We also proposed to define the formal semantics of the merging constraints to be represented in OWL + SWRL. Formal semantics is used in the knowledge execution step to perform automatic decision reconciliation

In our research, we recognized the fact that the merging constraints between knowledge artifacts may not be representable in the existing local ontologies of a domain. Therefore, the contribution of this thesis is the definition of the merging constraints in a separate OWL ontology. Defining the merging constraints in a separate ontology as opposed to modifying the ontological representation of knowledge artifacts in order to hardcode the merging constraints offered the following benefits:

i. Since the ontological representation of the knowledge artifacts are kept intact, they can be used in both traditional DSS and knowledge morphing frameworks. Therefore, only one version of the knowledge artifact serves both purposes.

ii. It makes possible to perform decision reconciliation dynamically during the knowledge execution step. This is superior to pre-execution morphing that is not able to modify the decisions according to the current state of the problem.

In addition, this thesis proposed a range of morphing constructs and represented their semantics in OWL. This increased the shareability of the merging constraints as they are all represented in a unified representation formalism.

- Limitations and Future Work

A potential limitation of our work is that all of the constraints defined in the KPO are problem specific and may not be applicable to other applications. However, the principle of identifying merging constraints and the formalism for representing merging constraint can be readily extended to other applications.

10.3. Ontology Execution Challenges, Solution, and Contributions

We performed knowledge execution by performing the following tasks: (1) Modelling the state of the problem in DKO; (2). Performing reasoning on DKO to infer the decision corresponding state of the problem, and (3) Representation of the inferred decision..

The challenge of ontology execution is the implementation of execution semantics. Execution semantics can be either hardcoded in terms of computer programs or can be defined as a formal knowledge representation formalism. In either scenario, it is challenging to define semantics of the DKO in such a way that reasoning process infers the decision corresponding to each state of the problem. Since we are using semantic web technologies for both representation and reasoning, we are interested in defining formal semantics in OWL and SWRL. However, defining formal semantics in OWL can be challenging due to decidability and performance issues, open-world and non-unique naming assumptions, lack of support for qualified cardinality restrictions and data type expressivity.

In this thesis we developed a domain specific DKO and its formal semantics in OWL-DL + SWRL. Therefore, several complex OWL axioms and SWRL rules are manually added to DKO in order to define domain-specific semantics. Moreover, in order to automate the time-consuming and error-prone tasks of handling non-unique and open world assumptions and lack of data-expressivity and qualified cardinality restriction, we proposed to preprocess an instantiation of DKO before execution. The lacking features are simulated by creating several triples added to the ontology using the preprocessing algorithms. Depending on the needed expressivity of DKO instantiations, they might be in OWL-DL or OWL 2 RL or OWL-DL + SWRL after preprocessing.

We used the algorithm in Table 3.1 to perform knowledge execution. This algorithm is able to execute the knowledge encapsulated in several knowledge artifacts represented in instantiations of DKO in parallel. We made minor modifications to the algorithm in Table 3.1 in order to perform dynamic decision reconciliation during knowledge execution. The modified algorithm can be seen in Table 8.1. This new algorithm modifies the decisions made based on each of the knowledge sources during their concurrent execution according to the semantics of the merging constraints.

Performing knowledge execution in a reasoner using formal semantics of the DKO instead of utilizing proprietary developed reasoners in a programming language has the following benefits:

i. Ease of switching to new technologies: In our method, after DKO instantiations undergo the preprocessing phase, they can be executed by any OWL reasoner and any API.

ii. Increased shareability and flexibility: During the preprocessing all the necessary execution semantics are added to the CPG in the form of DL and it is ready for execution. Thus, to execute the preprocessed CPG there is a need for a very simple program that performs queries on and puts new triples into the ontology.

iii. Reusing existing reasoners instead of writing a new one: Existing OWL reasoners can be utilized instead of developing a proprietary reasoner that performs reasoning on the execution semantics of the knowledge representation language.

Existing ontology based knowledge morphing frameworks propose to modify the ontological representation of knowledge encapsulated in the LKO in order to perform decision reconciliation [12][97]. Therefore, decision reconciliation is performed before knowledge execution at the modelling level. Our knowledge morphing framework is unique in the sense that it is the first framework that proposes to use an ontology for representation of merging constraints and performing decision reconciliation during execution. Our dynamic knowledge morphing approach has been observed to be better for the following reasons:

i. In pre-execution morphing, every possible state of the problem should be evaluated for a knowledge morphing scenario before the execution. However, a problem may have a huge problem state that makes this process extremely time consuming or even impossible. In our approach, merging scenario is decided dynamically upon according the current state of the problem.

ii. Most of the frameworks [12][97] make assumptions about the state of the problem in order to reduce it to a more manageable one. However, these assumptions may not hold true during knowledge execution hence rendering the generate recommendations irrelevant. Since we proposed to make the decisions dynamically, no irrelevant assumptions are made before the execution.

- Limitations and Future Work

Worst-case complexity of reasoning on OWL-DL and OWL 2 RL ontologies are NEXPTIME-complete and co-NP-complete [152] respectively. Therefore, execution of several large ontologies along with their merging constraints can be too slow especially for time-critical applications such as DSS in emergency rooms. Ontology patterns that make the reasoning process slower can be avoided by optimizing our ontologies. For instance, Pellint [182] that is an ontology repairing and optimization tool can be used for this purpose.

10.4. Knowledge Morphing Framework Application: CPG Merging

Several CPG computerization languages and their execution engines have been used for developing CDSS for assisting healthcare professionals with diagnosis, treatment and follow-up of patients with only one medical condition [34][108][133][134][135][136][137][138][140][155]. However, it is very common to have several concurrent medical conditions—i.e. comorbidities— in a patient. It is not a feasible solution to execute several CPG independently in order to provide decision support to comorbid patients due to possibility of unnecessary visits, duplication of tasks and adverse interactions that may exist between medical actions of different CPG. CPG merging research area aims at integrating several disease-specific CPG to provide decision support for comorbid patients while it reduce duration and cost of care, unnecessary visits, duplications and adverse interactions. We have used our knowledge morphing framework in order to provide a solution to the CPG merging problem. We faced three general research challenges in this research: 1. Computerization of CPG, 2. Merging CPG and 3. CPG Execution. We discuss each of these challenges, our solution to these challenges, our contributions and results, and limitations and future work.

10.4.1. Computerizing of CPG

To computerize CPG, every element of the CPG participating in the decision making and its relations with other elements should be found. Then these elements and their relations should be represented in terms of ontology classes, properties and instances. This is challenging because:

i. Paper-based CPG can be quite complex, lengthy and difficult to understand.

ii. Usually, there is no 1-to-1 mapping between elements of the paper-based CPG and the CPG ontology. Therefore, to computerize a CPG concept, several instances of the CPG classes might be needed to be created.

iii. CPG ontology of choice may not be able to represent some of the elements of the paper-based CPG.

Standardizing CPG formats by mapping and transforming them to a unified representation will enable us to reuse existing computerized CPG in other CPG ontologies and avoid the abovementioned challenges. However, existing CPG ontologies do not cover the whole CPG domain knowledge, have different vocabularies and different levels of expressivity for details of the domain knowledge. Therefore, none of the existing CPG ontologies can be used as this standardized format.

Our review of the ontology mapping representation languages shows that they suffer from lack of expressivity in representation of complex mappings and formal semantics for automatic instance transformation between the mapped ontologies. Due to the complex nature of CPG, mapping CPG ontologies using the existing ontology mapping representation languages is not possible.

To develop a standardized format for representation of CPG, we surveyed the identified workflow patterns in CPG representation languages and developed an ontology called CPG-KPO capable of representing these patterns and all their variants.

We also surveyed the literature that identify the desired features of the ontology mapping representation languages. We developed an OWL-Full ontology called KPO that can be instantiated to represent mappings between two OWL ontologies. We translated the mappings from OWL-Full to OWL-DL + SWRL so that an OWL reasoner can perform reasoning on the mapped ontologies and the mappings to perform instance transformation.

In terns of contributions, in this thesis we compared CPG-DKO with popular CPG representation languages in section 9.1. Our comparison showed the effectivenes of CPG-DKO in terms of expressivity for representation of workflow patterns. Computerization and mapping of total of 12 disease-specific CPG shows the usefulness of CPG-DKO as a CPG domain knowledge. Moreover, Expression Ontology developed with CPG-DKO is the first ontological representation of mathematical expressions. In contrast to the existing CPG representation languages that use a different expression language, this ontology enables us to represent both CPG and their expressions in a unified knowledge representation formalism. We also extracted a number of ontology metrics proposed in [178] in order to evaluate the quality of CPG-DKO. These metrics showed that CPG-DKO is more expressive compared to three existing CPG ontologies. Moreover, these metrics showed that this ontology is not likely to contain redundant classes and properties. .

We mapped several CPG ontologies to CPG-DKO and transforming their instances. Since in all cases the execution results are consistent with original computerized CPG, we can conclude that (1) mappings between CPG-DKO and other CPG ontologies are represented correctly and (2) instance transformation is performed successfully.

- Limitations and Future Work

The computerized CPG used for evaluation of our CPG merging framework are not complex and do not use most of the identified workflow patterns of CPG representation languages. Therefore, we could not evaluate the possibility of transforming complex CPG to CPG-DKO using real examples. Moreover, since we have reused implementation of basic control flow patterns of the CPG from NICHE CPG ontologies in CPG-DKO, this ontology bears several resemblances to them. Hence, CPG-DKO might be more effective when used as the background ontology of the NICHE CPG ontologies. Unfortunately, we did not have access to other CPG ontologies in OWL to perform more evaluations. Results of such an evaluation can be used to improve both CPG-DKO and KMO. Since worst case complexity of reasoning on OWL-DL is NEXPTIME-complete [152], instance transformation using KMO be computationally expensive.

10.4.1.1. CPG Merging

The challenge in CPG merging is to identify how CPG should be modified so they are safely merged with the rest of the CPG pertaining to a comorbidity. Computerized CPG can either be modified before or during execution. The challenge in pre-execution merging is to design an algorithm that merges several CPG by modification of them and connecting the related tasks while patients safety is not compromised. However, our investigations show that pre-execution merge is not a practical solution due to making several assumption regarding the execution flow of the CPG that may not necessarily hold true during execution hence rendering the generate recommendations irrelevant or potentially dangerous. An alternate approach is to merge different CPG during their concurrent execution so that decisions regarding CPG merging are based on the evolving patients' status. . To achieve this, the merging constraints should be captured in a computer understandable format that can be used by a CPG merging execution engine.

Our solution entailed the development of an ontology called CPG-KPO to represent the merging constraints between computerized CPG pertaining to a comorbidity. These merging constraints describe how the medical actions recommended in individual CPG should be modified or delayed so that a safe and improved therapy plan for comorbid patients are generated. To identify the constraints that should be included in our ontology, we discussed with the physicians in our research group, reviewed CPG merging literature [20][21][22][24][25][26][29][30][107] and got inspired from classical AI planning literature [183]. We also considered a number of new constraints as potentially useful constraints and included them in CPG-KPO.

We created a list of CPG merging constraints that can be used by other researchers interested in computerized CPG merging topic. CPG-KPO is the first and the most expressive ontological approach for representation of CPG merging constraints. We used this ontology to represent the merging constraints between ontologically modeled CPG that had already been merged in CPG merging literature [25][107]. In all cases, CPG-DKO is expressive enough to capture the merging constraints between their transformed versions in CPG-DKO. We also asked a physician to find the merging constraints between CPG of Osteoarthritis [169], Hypertension [170] and Diabetes [171] diseases represented in CPG-DKO. Our ontology was able to represent those merging constraints and their execution logic. We also asked the opinion of total of seven computer scientists, health informaticians and physicians about CPG-KPO. Participants in the survey found CPG-DKO moderately easy to use, very clear, expressive and very useful for CPG merging. The most suggested enhancement for CPG-DKO is to improve how more than two CPG are merged.

We also extracted a number of ontology metrics proposed in [178] in order to evaluate the quality of CPG-KPO as an ontology. These metrics show that our ontology is not likely to contain redundant classes and properties.

- Limitations and Future Work

We could not find examples for some of the suggested merging constraints. Usefulness of these constraints in real CPG merging scenarios has yet to be evaluated. Therefore, it will be useful to collaborate with domain experts involved in treatment of comorbid patients to find more real CPG merging examples.

A constraint in CPG-KPO can only merge two tasks. Therefore, to indicate that *m* tasks should be executed simultaneously, total of $m\times(m-1)/2$ constraints should be created. As a future work, we can define constraints that can merge more than two tasks. This improvement makes KPO instantiations smaller and easier to create while merging more than two knowledge sources. Moreover, most of the constraints defined in the KPO are problem specific and may not be useful in other applications. Depending on the problem, these constraints may need to be revised and improved.

10.4.1.2. CPG Execution

A CPG execution engine should interpret the procedural and the decision logic inherent within the computerized CPG and in conjunction with patient data and physician’s input. In other words, the CPG execution engine should orchestrates the executional flow of the CPG—i.e. stipulating the ordering of clinical tasks, evaluating the satisfaction of criteria of decisions, constraints, conditions and responding to outcomes of clinical tasks—to provide patient-specific and disease-specific recommendations about care interventions and clinical decision-making. This becomes more challenging when CPG merging constraints should also be taken into consideration while concurrently executing several CPG. We used OWL reasoning services for CPG execution, whilst solving the following research challenges:

i. Lack of an expression language to support data type expressivity needed for knowledge execution

ii. Decidability and performance issues

iii. Non-unique naming and open world assumptions

Our solution for CPG execution was to define the formal semantic of CPG-DKO in OWL-DL + SWRL. This enabled us to execute CPG by encoding the current state of the problem in CPG-DKO and performing reasoning on them. We preprocess instantiations of CPG-DKO before execution in order to deal with the non-unique naming and the open world assumptions and addresses the lack of qualified cardinality restriction, property chaining and data type expressivity in OWL.

We also defined the formal semantics of the merging constraints in OWL-DL + SWRL as well. The formal semantics of the CPG-DKO and CPG-KPO work in tandem in order to merge CPG during execution.

The main innovation of our OWL based CPG execution and merging engines was to define the execution semantics in OWL and SWRL in order to execute CPG using OWL reasoning services rather than implementing a graph parsing algorithm in a programming language. This approach of CPG execution is superior to the existing graph-parsing based CPG execution algorithms due to (1) Ease of switching to new semantic web technologies, (2) Increased shareability and flexibility and (3) Reusing existing reasoners instead of writing a new one. Our evaluations also showed that our OWL + SWRL based CPG execution engine is the engine with the most executional capabilities. We successfully executed 12 disease-specific CPG that were either mapped to CPG-DKO or computerized in CPG-DKO.

Our CPG merging execution engine is the first execution-time CPG merging framework. This engine is superior to existing CPG merging frameworks that all perform CPG merging before execution because:

i. Merging is performed in response to the current state as opposed to a priori assumptions about the outcomes of decisions, values of variables and CPG starting times.

ii. Our approach keeps the ontologically modeled CPG intact because it makes a clear separation between the CPG and the morphing constraints. This increases the shareability and facilitates the maintenance of both mergings and the computerized CPG.

- Limitation and Future Work

We have not evaluated the medical validity of the output of the CPG execution engines and the CPG merging engine. Therefore, a limitation of this work is lack of medical evaluation.

The worst-case complexity of reasoning on OWL-DL is NEXPTIME-complete [152]. Therefore, adding a new CPG to the set of CPG that are being merged can significantly increase the reasoning time. Currently, a constraint in CPG-KPO can only merge two tasks. Therefore, to indicate that *m* tasks should be executed simultaneously, $m \times (m-1)/2$ constraints should be created in total. Therefore, size of the instantiation of CPG-KPO can grow in a polynomial fashion as new CPG are added which can further slowdown the reasoning process. Therefore, we can conclude that our framework may not be scalable to comorbidities with a large number of disease-specific CPG.

10.4.2. CPG Merging Future Work

We believe that our research pertaining to CPG merging can be improved in the following ways:

1. Making use of standard vocabularies such as SNOMED CT [185] for automation of the mapping and the merging processes.

2. Implementation of interfaces with standard electronic health records protocols such as HL7 [184]. This is crucial for facilitation of evaluation with real patients' data.

3. Implementation of a multi-agent version of CPG merging framework to facilitate the geographically-dispersed nature of patient care especially for comorbidies.

4. Medical evaluation of the therapy plans generated based on CPG modeled in CPG-DKO using real patient data.

5. Medical evaluation of the therapy plans generated based on several computerized CPG by physicians who are involved in patient-care with the same comorbidities using real patient data.

6. Improvement of the CPG merging constraints representable in CPG-DKO by consulting domain experts involved in treatment of comorbid patients.

10.5. Knowledge Morphing Future Work

In this section of the thesis, we list general future research directions that can improve our knowledge morphing framework:

1. Using domain-specific standard vocabularies in order to automate the mapping and the morphing processes. For instance, as we suggested earlier, SNOMED CT [185] can be used in our CPG merging framework for this purpose.

2. Evaluation of the computational complexity of reasoning on our ontologies: Even though theoretical work has shown that worst-case complexity of reasoning on OWL-DL and OWL 2 RL ontologies are NEXPTIME-complete and co-NP-complete respectively [152], it is hard to predict the actual complexity of a specific OWL ontology. The approach proposed by Kang et al. [181] uses machine learning techniques to classify ontologies based on the complexity of reasoning on them. This approach and similar approaches can show how scalable our framework is.

3. Improvement of reasoning speed on our ontologies: Pellint [182] which is a tool for repairing modelling constructs with low reasoning performance on OWL ontologies can be used towards this goal. This tool however does not detect performance issues related to SWRL rules and their combination with OWL ontologies. Further research is needed in this regard.

4. Detecting and resolving inconsistencies: Several knowledge sources used for decision making may contain inconsistent pieces of knowledge [12]. Since OWL reasoners are not able to perform reasoning on inconsistent ontologies, inconsistencies between the morphed ontologies should be detected and resolved.

5. Extracting contextualized sub-ontologies: This process extracts the part of the ontology that is relevant to the decision making problem at hand and leaves the rest out [12]. This extraction can be beneficial in two ways: (1) it reduce the size of the final morphed ontology. A smaller ontology can be reasoned over more effectively; (2) it avoids inconsistencies between the relevant extracted part of an ontology with the non-extracted parts of other ontologies.

6. We used our ontology mapping framework to map several CPG ontologies. More evaluations using ontologies of different domains can prove the efficacy of our mapping approach in mapping ontologies pertaining to any decision making problem.

7. Using our knowledge morphing framework for different applications: It helps us to evaluate the efficacy of our framework as a general knowledge morphing framework. Multi-agent decision making [186], AI plan merging [183] and drug interaction systems [187] can be some candidate applications.

BIBLOGRAPHY

[1] R. A. Greenes, *Clinical decision support: the road ahead*. Academic Press, San Diego, 2011.

[2] D. Fensel, *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer, Berlin, 2001.

[3] E. Blomqvist, "The use of Semantic Web technologies for decision support–a survey," *Sem. Web,* Vol. 5, No. 2, 2014. – accepted for publication.

[4] M. Peleg, "Computer-interpretable clinical guidelines: A methodological review," *J. Biomed. Inform.*, vol. 46, no. 4, pp. 744 – 763, 2013.

[5] G. M. Marakas, *Decision support systems in the 21st century*, vol. 134. Prentice Hall, New Jersey, 2003.

[6] M. Choraś, R. Kozik, A. Flizikowski, R. Renk, and W. Hołubowicz, "Ontology-based decision support for security management in heterogeneous networks," in *Emerging Intelligent Computing Technology and Applications. With Aspects of Artificial Intelligence*, Springer, 2009, pp. 920–927.

[7] Y. A. Zagorulko and G. Zagorulko, "Ontology-Based Approach to Development of the Decision Support System for Oil-and-Gas Production Enterprise," in *9th International Conference on Software Methodologies, Tools and Techniques (SoMeT)*, 2010, pp. 457–466.

[8] P. Casanovas, N. Casellas, and Vallb'e Joan-Josep, "An Ontology-Based Decision Support System for Judges," in Proceedings of the 2009 conference on Law, Ontologies and the Semantic Web: Channelling the Legal Information Flood, 2009, pp. 165–175.

[9] T. Pangjitt and T. Sunetnanta, "A model of ontology driven case-based reasoning for electronic issue management systems," in *Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on*, 2011, pp. 87–90.

[10] A. Schatten, S. Biffl, and A. M. Tjoa, "Closing the gap: from nescience to knowledge management," in *Proceedings of Euromicro Conference* 2003, pp. 327–335.

[11] M. Fortin, G. Bravo, C. Hudon, A. Vanasse and L. Lapointe, "Prevalence of multimorbidity among adults seen in family practice," *Ann. Fam. Med.,* vol. 3, pp. 223-228, May-Jun, 2005.

[12] S. Hussain, K-MORPH: A Semantic Web Based Knowledge Representation and Context-Driven Morphing Framework,” in *Advances in Artificial Intelligence*, vol. 5549, Y. Gao and N. Japkowicz, Eds. Springer Berlin Heidelberg, 2009, pp. 279–282.

[13] A. Hameed, A. Preece, and D. Sleeman, “Ontology reconciliation,” in *Handbook on ontologies*, Springer, 2004, pp. 231–250.

[14] P. Hitzler, M. Krötzsch, M. Ehrig, and Y. Sure, “What is ontology merging?-a categorytheoretic perspective using pushouts,” in *Proc. First International Workshop on Contexts and Ontologies: Theory, Practice and Applications*, 2005.

[15] B. C. Van Fraassen, *Formal semantics and logic*. Macmillan New York, 1971.

[16] “Semantic Web,” http://www.w3.org/standards/semanticweb/

[17] F. Baader and U. Sattler, “Tableau algorithms for description logics,” in *Automated Reasoning with Analytic Tableaux and Related Methods*, Springer, 2000, pp. 1–18.

[18] J. M. Grimshaw and I. T. Russell, “Effect of clinical guidelines on medical practice: a systematic review of rigorous evaluations,” *Lancet*, vol. 342, no. 8883, pp. 1317–1322, 1993.

[19] J. Fox and S. K. Das, "The RED knowledge representation language," in *Safe and Sound : Artificial Intelligence in Hazardous Applications*Menlo Park, California.: AAAI Press/MIT Press, 2000, pp. 191-206.

[20] S. Abidi, S. Abidi, L. Butler and S. Hussain, "Operationalizing prostate cancer clinical pathways: An ontological model to computerize, merge and execute institution-specific clinical pathways," in *Knowledge Management for Health Care Procedures*, D. Riaño, Ed. Springer Berlin / Heidelberg, 2009, pp. 1-12.

[21] S. R. Abidi and S. S. R. Abidi, "Towards the merging of multiple clinical protocols and guidelines via ontology-driven modeling," in *Artificial Intelligence in Medicine*, C. Combi, Y. Shahar and A. Abu-Hanna, Eds. Springer Berlin / Heidelberg, 2009, pp. 81-85.

[22] S. R. Abidi, “A Conceptual Framework for Ontology Based Automating and Merging of Clinical Pathways of Comorbidities,” in *Knowledge Management for Health Care Procedures*, vol. 5626, D. Riaño, Ed. Springer Berlin / Heidelberg, 2009, pp. 55–66.

[23] K. R. Apt and M. Wallace, *Constraint Logic Programming using Eclipse*. New York, NY, USA: Cambridge University Press, 2007.

[24] M. M. Hing, M. Michalowski, S. Wilk, W. Michalowski and K. Farion, "Identifying inconsistencies in multiple clinical practice guidelines for a patient with comorbidity," in *IEEE International Conference on Bioinformatics and Biomedicine Workshops (BIBMW),* 2010, pp. 447-452.

[25] S. Wilk, M. Michalowski, M. M. Hing, W. Michalowski and K. Farion, "Reconciliation of concurrently applied clinical practice guidelines using constraint logic programming," in *Proceedings of the 6th International Symposium on Health Informatics and Bioinformatics, (HIBIT 2011), Izmir, Turkey,* 2011, pp. 138-144.

[26] F. Real and D. Riaño, "Automatic combination of formal intervention plans using SDA* representation model," in *Proceedings of the 2007 conference on Knowledge management for health care procedures*, 2008, vol. Amsterdam, The Netherlands, pp. 75–86.

[27] D. Riano, "The SDA model: A set theory approach," in Twentieth IEEE International Symposium on Computer-Based Medical Systems, 2007, pp. 563-568.

[28] F. Real, D. Riano and J. Bohada, "Automatic generation of formal intervention plans based on the SDA* representation model," in *Computer-Based Medical Systems, 2007. CBMS '07. Twentieth IEEE International Symposium on,* 2007, pp. 575-580.

[29] F. Real and D. Riano, "An autonomous algorithm for generating and merging clinical algorithms," in *Knowledge Management for Health Care Procedures*, D. Riaño, Ed. Springer Berlin / Heidelberg, 2009, pp. 13-24.

[30] S. S. R. Abidi and S. Shayegani, "Modeling the form and function of clinical practice guidelines: An ontological model to computerize clinical practice guidelines," in *Knowledge Management for Health Care Procedures*, Springer, 2009, pp. 81–91.

[31] J. Bock, P. Haase, Q. Ji, and R. Volz, "Benchmarking OWL reasoners," in *Proc. of the ARea2008 Workshop, Tenerife, Spain*, 2008.

[32] P. F. Patel-Schneider, P. Hayes and I. Horrocks, "Semantics and Abstract Syntax," http://www.w3.org/TR/owl-semantics/

[33] L. Ohno-Machado, J. H. Gennari, S. N. Murphy, N. L. Jain, S. W. Tu, D. E. Oliver, E. Pattison-Gordon, R. A. Greenes, E. H. Shortliffe and G. O. Barnett, "The GuideLine Interchange Format: A Model for Representing Guidelines," *American Medical Informatics Association,* vol. 5, pp. 357-372, 1998.

[34] S. W. Tu, M. A. Musen, R. Shankar, J. Campbell, K. Hrabak, J. McClay, S. M. Huff, R. McClure, C. Parker, R. Rocha, R. Abarbanel, N. Beard, J. Glasgow, G. Mansfield, P. Ram, Q. Ye, E. Mays, T. Weida, C. G. Chute, K. McDonald, D. Molu, M. A. Nyman, S. Scheitel, H. Solbrig, D. A. Zill and M. K. Goldstein, "Modeling guidelines for integration into clinical workflow," *Stud.Health Technol.Inform.,* vol. 107, pp. 174-178, 2004.

[35] M. Kifer and G. Lausen, "F-logic: A higher-order language for reasoning about objects, inheritance, and scheme," in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data,* Portland, Oregon, United States, 1989, pp. 134-146.

[36] M. Benerecetti, P. Bouquet and C. Ghidini, "On the dimensions of context dependence: Partiality, approximation, and perspective," in *Proceedings of the Third International and Interdisciplinary Conference on Modeling and using Context,* 2001, pp. 59-72.

[37] J. Euzenat and P. Shvaiko, *Ontology Matching.* Springer-Verlag New York Inc, 2007.

[38] W. W. Cohen, P. Ravikumar and S. E. Fienberg, "A comparison of string distance metrics for name-matching tasks," in *Proceedings of the IJCAI-2003 Workshop on Information Integration on the Web (IIWeb-03),* 2003, pp. 73–78.

[39] G. A. Miller, "WordNet: a lexical database for English," *Commun ACM,* vol. 38, pp. 39-41, 1995.

[40] A. Budanitsky and G. Hirst, "Semantic distance in WordNet: An experimental, application-oriented evaluation of five measures," in *Workshop on Wordnet and Other Lexical Resources, Second Meeting of the North American Chapter of the Association for Computational Linguistics,* 2001, .

[41] J. Euzenat and P. Valtchev, "Similarity-based ontology alignment in OWL-lite," in *Proceeding of ECAI 2004: 16th European Conference on Artificial Intelligence,* Valencia, Spain, 2004, pp. 333-337.

[42] J. English and S. Nirenburg, "Calculating concept similarity heuristics for ontology learning from text," University of Maryland, 2007.

[43] L. Lee, L. H. Yang, W. Hsu and X. Yang, "XClust: Clustering XML schemas for effective integration," in *Proceedings of the Eleventh International Conference on Information and Knowledge Management,* McLean, Virginia, USA, 2002, pp. 292-299.

[44] R. R. Sokal, "Distance as a measure of taxonomic similarity," *Syst. Biol.,* vol. 10, pp. 70, 1961.

[45] S. Melnik, H. Garcia-Molina and E. Rahm, "Similarity flooding: A versatile graph matching algorithm and its application to schema matching," in *Proceeding of 18th International Conference on Data Engineering,* 2002, pp. 117-128.

[46] A. Isaac, L. Van Der Meij, S. Schlobach and S. Wang, "An empirical study of instance-based ontology matching," in *Proceedings of the 6th International the Semantic Web and 2nd Asian Conference on Asian Semantic Web Conference,* Busan, Korea, 2007, pp. 253-266.

[47] P. Bouquet, L. Serafini and S. Zanobini, "Peer-to-peer semantic coordination," *J. Web Sem.,* vol. 2, pp. 81-97, 12/1, 2004.

[48] P. Bouquet, L. Serafini and S. Zanobini, "Semantic coordination: A new approach and an application," in *The Semantic Web - ISWC 2003*, D. Fensel, K. Sycara and J. Mylopoulos, Eds. Springer Berlin / Heidelberg, 2003, pp. 130-145.

[49] F. Giunchiglia, P. Shvaiko and M. Yatskevich, "S-match: An algorithm and an implementation of semantic matching," in *The Semantic Web: Research and Applications*, C. Bussler, J. Davies, D. Fensel and R. Studer, Eds. Springer Berlin / Heidelberg, 2004, pp. 61-75.

[50] D. Calvanese, F. Baader, P. Patel-Schneider, D. L. McGuinness and D. Nardi, *The Description Logic Handbook: Theory, Implementation, and Applications.* New York, USA: Cambridge University Press, 2003.

[51] J. H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row Publishers, Inc., 1985.

[52] E. Goldberg and Y. Novikov, "BerkMin: A fast and robust SAT-solver," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition.* 2002, pp. 142-149.

[53] J. P. Marques-Silva and K. A. Sakallah, "GRASP: a search algorithm for propositional satisfiability," *IEEE Trans. Comput.,* vol. 48, pp. 506-521, 1999.

[54] Sotnykova, C. Vangenot, N. Cullot, N. Bennacer and M. Aufaure, "Semantic mappings in description logics for spatio-temporal database schema integration," in *Journal on Data Semantics III*, S. Spaccapietra and E. Zimányi, Eds. Springer Berlin / Heidelberg, 2005, pp. 586-586.

[55] Meilicke, H. Stuckenschmidt and A. Tamilin, "Improving automatically created mappings using logical reasoning." in *Ontology Mapping Workshop at ISWC,* Athens, GA, USA, 2006.

[56] Calvanese, G. D. Giacomo and M. Lenzerini, "Ontology of integration and integration of ontologies," *Description Logics,* vol. 49, pp. 10-19, 2001.

[57] H. Stuckenschmidt, L. Serafini and H. Wache, "Reasoning about ontology mappings," ITC-IRST, Trento, 2005.

[58] P. Bouquet, F. Giunchiglia, F. van Harmelen, L. Serafini and H. Stuckenschmidt, "C-OWL: Contextualizing ontologies." in *International Semantic Web Conference,* 2003, pp. 164-179.

[59] P. Bouquet, L. Serafini, S. Zanobini and S. Sceffer, "Bootstrapping semantics on the web: Meaning elicitation from schemas," in *Proceedings of the 15th International Conference on World Wide Web,* Edinburgh, Scotland, 2006, pp. 505-512.

[60] D. Dou, D. McDermott and P. Qi, "Ontology translation by ontology merging and automated reasoning," in *Proceeding of Workshop on Ontologies for MultiAgent Systems*,2005, pp. 73–94.

[61] D. Dou, D. McDermott and P. Qi, "Ontology translation on the semantic web," in *Journal on Data Semantics II*, S. Spaccapietra, E. Bertino, S. Jajodia, R. King, D. McLeod, M. Orlowska and L. Strous, Eds. Springer Berlin / Heidelberg, 2005, pp. 35-57.

[62] A. Maedche, B. Motik, N. Silva and R. Volz, "MAFRA - A Mapping FRAmework for distributed ontologies," in Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web, 2002, pp. 235-250.

[63] K. Yang and R. Steele, "A framework for ontology mapping for the semantic web," in *Proceeding of the International Conference on Information Technology in Asia (CITA07),* 2007.

[64] N. F. Noy, "Semantic integration: a survey of ontology-based approaches," *SIGMOD Rec.,* vol. 33, pp. 65-70, December, 2004.

[65] M. Sabou, M. D'Aquin and E. Motta, "Exploring the Semantic Web as background knowledge for ontology matching," *Knowledge,* vol. 11, pp. 156-190, 2008.

[66] N. F. Noy and M. A. Musen, "Anchor-prompt: Using non-local context for semantic matching," in Proceedings of the Workshop on Ontologies and Information Sharing at the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2001), 2001, pp. 63-70.

[67] J. Madhavan, P. A. Bernstein, P. Domingos and A. Y. Halevy, "Representing and reasoning about mappings between domain models," in *Eighteenth National Conference on Artificial Intelligence,* Edmonton, Alberta, Canada, 2002, pp. 80-86.

[68] E. Jérôme, S. François and S. Luciano, "Specification of the delivery alignment format," Knowledge web, 2005.

[69] H. Thomas, D. Sullivan and R. Brennan, "Ontology Mapping Representations: a Pragmatic Evaluation," *Management,* pp. 228-232, 2009.

[70] J. d. Bruijn and A. Polleres, "Towards an ontology mapping specification language for the semantic web," DERI - DIGITAL ENTERPRISE RESEARCH INSTITUTE, Tech. Rep. DERI-2004-06-30, 2004.

[71] B. Omelayenko, "RDFT: A mapping meta-ontology for business integration," in *Proc. of the Workshop on Knowledge Transformation for the Semantic Web at the 15th European Conference on Artificial Intelligence (KTSW2002)*, 2002, pp. 77–84.

[72] F. Scharffe, "Instance transformation for semantic data mediation," in *Proceeding of the International Semantic Web and Web Services Conference SWWS,* 2006.

[73] F. Scharffe and J. d. Bruijn, "A language to specify mappings between ontologies," in *Proceeding of the Internet Based Systems IEEE Conference (SITIS05),* 2005, pp. 267-271.

[74] Y. Lei, "An instance mapping ontology for the semantic web," in *Proceedings of the 3rd International Conference on Knowledge Capture,* Banff, Alberta, Canada, 2005, pp. 67-74.

[75] Haase and B. Motik, "A mapping system for the integration of OWL-DL ontologies," in *Proceedings of the First International Workshop on Interoperability of Heterogeneous Information Systems,* Bremen, Germany, 2005, pp. 9-16.

[76] Burton-Jones, V. C. Storey, V. Sugumaran and P. Ahluwalia, "A semiotic metrics suite for assessing the quality of ontologies," *Data Knowl. Eng.,* vol. 55, pp. 84-102, October, 2005.

[77] Gangemi, C. Catenacci, M. Ciaramita and J. Lehmann, "A theoreticalframework for ontologyevaluation and validation," in *Proceedings of the 2nd ItalianSemantic Web Workshop, SWAP 2005 - Semantic Web Applications and Perspectives,* Trento, Italy, 2005.

[78] J. Völker, D. Vrandečić and Y. Sure, "Automatic evaluation of ontologies (AEON)," in *The Semantic Web – ISWC 2005*, Y. Gil, E. Motta, V. Benjamins and M. Musen, Eds. Springer Berlin / Heidelberg, 2005, pp. 716-731.

[79] D. Isern and A. Moreno, "Computer-based execution of clinical guidelines: a review," *Int. J. Med. Inf.*, vol. 77, no. 12, pp. 787–808, 2008.

[80] L. Parthiban and R. Subramanian, "Intelligent heart disease prediction system using CANFIS and genetic algorithm," *Intl. J. Bio. Biomed. Med. Sci.*, vol. 3, no. 3, 2008.

[81] R. Dybowski, V. Gant, P. Weller, and R. Chang, "Prediction of outcome in critically ill patients using artificial neural network synthesised by genetic algorithm," *Lancet*, vol. 347, no. 9009, pp. 1146–1150, 1996.

[82] H. Yan, Y. Jiang, J. Zheng, C. Peng, and Q. Li, "A multilayer perceptron-based medical decision support system for heart disease diagnosis," *Expert Syst.Appl.*, vol. 30, no. 2, pp. 272–281, 2006.

[83] V. L. Yu, L. M. Fagan, S. M. Wraith, W. J. Clancey, A. C. Scott, J. Hannigan, R. L. Blum, B. G. Buchanan, and S. N. Cohen, "Antimicrobial selection by a computer. A blinded evaluation by infectious diseases experts.," *J. Am. Med. Inform. Assoc.*, vol. 242, no. 12, p. 1279, 1979.

[84] M. Suganthi and M. Madheswaran, "An improved medical decision support system to identify the breast cancer using mammogram," *J. Med. Syst.*, vol. 36, no. 1, pp. 79–91, 2012.

[85] N. Mehdi, S. Mehdi, N. T. Adel, and M. Azra, "Hepatitis disease diagnosis using hybrid case based reasoning and particle swarm optimization," *ISRN Artificial Intelligence*, vol. 2012, 2012.

[86] K. A. Kumar, Y. Singh, and S. Sanyal, "Hybrid approach using case-based reasoning and rule-based reasoning for domain independent clinical decision support in ICU," *Expert Syst.Appl.*, vol. 36, no. 1, pp. 65–71, 2009.

[87] W. J. Clancey, *Knowledge-based tutoring: the GUIDON program*. Cambridge, MA, USA: MIT Press, 1987.

[88] M. Peleg, S. Tu, J. Bury, P. Ciccarese, J. Fox, R. A. Greenes, R. Hall, P. D. Johnson, N. Jones, A. Kumar, S. Miksch, S. Quaglini, A. Seyfang, E. H. Shortliffe, and M. Stefanelli, "Comparing computer-interpretable guideline models: a case-study approach," *J. Am. Med. Inform. Assoc.*, vol. 10, no. 1, pp. 52–68, Feb. 2003

[89] P. A. de Clercq, J. A. Blom, H. H. Korsten, and A. Hasman, "Approaches for creating computer-interpretable guidelines that facilitate decision support," *Artif. Intell. Med.*, vol. 31, no. 1, pp. 1–27, 2004.

[90] S. W. Tu and M. A. Musen, "Representation formalisms and computational methods for modeling guideline-based patient care," *Stud.Health Technol.Inform.*, pp. 115–132, 2001.

[91] P. De Clercq, K. Kaiser, and A. Hasman, "Computer-interpretable guideline formalisms," *Stud.Health Technol.Inform.*, vol. 139, p. 22, 2008.

[92] M. Sordo, O. Ogunyemi, A. A. Boxwala, and R. A. Greenes, "GELLO: an object-oriented query and expression language for clinical decision support," in *the AMIA Annual Symposium Proceedings*, 2003, vol. 2003, p. 1012.

[93] P. J. Lucas, N. C. de Bruijn, K. Schurink, and A. Hoepelman, "A probabilistic and decision-theoretic approach to the management of infectious disease at the ICU," *Artif.Intell.Med.*, vol. 19, no. 3, pp. 251–279, 2000

[94] S. Heiler, "Semantic interoperability," *ACM Comput. Surv.*, vol. 27, no. 2, pp. 271–273, 1995.

[95] M. Klein and L. B. Methlie, *Knowledge-based decision support systems with applications in business: a decision support approach*. Chichester: John Wiley & Sons, 2009.

[96] S. Haag, M. Cummings, and J. Dawkins, "Management information systems," *Multimedia Syst.*, vol. 279, pp. 280,297–298, 1998.

[97] S. S. R. Abidi, "Medical knowledge morphing: towards case-specific integration of heterogeneous medical knowledge resources," in *Computer-Based Medical Systems, 2005. Proceedings. 18th IEEE Symposium on*, 2005, pp. 208–213.

[98] M. Michalowski, S. Wilk, W. Michalowski, D. Lin, K. Farion, and S. Mohapatra, "Using Constraint Logic Programming to Implement Iterative Actions and Numerical Measures during Mitigation of Concurrently Applied Clinical Practice Guidelines," in *Artificial Intelligence in Medicine*, Springer, 2013, pp. 17–22.

[99] I. Sánchez-Garzón, J. Fdez-Olivares, E. Onaindía, G. Milla, J. Jordán, and P. Castejón, "A Multi-agent Planning Approach for the Generation of Personalized Treatment Plans of Comorbid Patients," in *Artificial Intelligence in Medicine*, Springer, 2013, pp. 23–27.

[100] E. Lozano, M. Marcos, B. MartÃnez-Salvador, A. Alonso, and J. Alonso, “Experiences in the Development of Electronic Care Plans for the Management of Comorbidities,” vol. 5943, D. RiaÃo, A. Teije, S. Miksch, and M. Peleg, Eds. Springer Berlin Heidelberg, 2010, pp. 113–123.

[101] N. Mulyar, W. M. P. van der Aalst, and M. Peleg, “A Pattern-based Analysis of Clinical Computer-interpretable Guideline Modeling Languages,” *J. Am. Med. Inform. Assoc.*, vol. 14, no. 6, pp. 781–787, 2007.

[102] N. Russell, W. M. van der Aalst, and N. Mulyar, “Workflow Control-Flow Patterns: A Revised View,” *BPM Center Report BPM-06-22*, 2006.

[103] M. Peleg, S. Tu, and J. Bury, “Comparing models of decision and action for guideline-based decision support: a case-study approach,” *J. Am. Med. Inform. Assoc.*, vol 1, no. 10, pp. 52-68, 2002.

[104] S. Hussain and S. Abidi, “An ontology-based framework for authoring and executing clinical practice guidelines for clinical decision support systems,” *J. Inf. Technol. Healthc*, vol. 6, no. 1, pp. 8–22, 2008.

[105] S. Hussain and S. S. R. Abidi, “Ontology driven CPG authoring and execution via a Semantic Web framework,” in *the System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, 2007, pp. 135–135

[106] A. Daniyal, S. R. Abidi, and S. S. R. Abidi, “Computerizing clinical pathways: ontology-based modeling and execution.,” *Stud.Health Technol.Inform.*, vol. 150, pp. 643–647, 2009.

[107] S. R. Abidi, “A Knowledge Management Framework to Develop, Model, Align and Operationalize Clinical Pathways to Provide Decision Support for Comorbid Diseases,” Dalhousie University, Halifax, NS, 2010.

[108] Y. Shahar, S. Miksch, and P. Johnson, “The Asgaard project: a task-specific framework for the application and critiquing of time-oriented clinical guidelines,” *Artif. Intell. Med.*, vol. 14, no. 1–2, pp. 29–51, 1998.

[109] Z. Aleksovski, M. Klein, W. ten Kate, and F. van Harmelen, “Matching Unstructured Vocabularies using a Background Ontology,” in *Proceedings of the 15th International Conference on Knowledge Engineering and Knowledge Management*, 2006, pp. 182–197.

[110] Z. Aleksovski, W. ten Kate, and F. van Harmelen, “Using multiple ontologies as background knowledge in ontology matching,” in Procedding of Collective Intelligence & the Semantic Web (*CISWeb 2008*), p. 35, 2008.

[111] F. Giunchiglia, P. Shvaiko, and M. Yatskevich, “Discovering Missing Background Knowledge in Ontology Matching,” in *Proceedings of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence*, Riva del Garda, Italy, 2006, pp. 382–386.

[112] Z. Aleksovski, W. ten Kate, and F. van Harmelen, “Exploiting the Structure of Background Knowledge Used in Ontology Matching.,” in *Proceeding of Internation Workshop of Ontology Matching*, 2006, p. 13-21.

[113] D. Wang, M. Peleg, S. W. Tu, A. A. Boxwala, R. A. Greenes, V. L. Patel, and E. H. Shortliffe, “Representation primitives, process models and patient data in computer-interpretable clinical practice guidelines: A literature review of guideline representation models,” *Int. J. Med. Inf.*, vol. 68, no. 1, pp. 59–70, 2002.

[114] D. Wang, M. Peleg, D. Bu, M. Cantor, G. Landesberg, E. Lunenfeld, S. W. Tu, G. E. Kaiser, G. Hripcsak, and V. L. Patel, “GESDOR–a generic execution model for sharing of computer-interpretable clinical practice guidelines,” in *AMIA Annual Symposium Proceedings*, 2003, vol. 2003, p. 694.

[115] H. Wang, N. Noy, A. Rector, M. Musen, T. Redmond, D. Rubin, S. Tu, T. Tudorache, N. Drummond, M. Horridge, and J. Seidenberg, “Frames and OWL Side by Side,” in *9th International Protégé Conference*, 2006.

[116] Y. Kitamura, M. Ikeda, and R. Mizoguchi, “A causal time ontology for qualitative reasoning,” in *International Joint Conferences on Artificial Intelligence*, 1997, vol. 1, pp. 501–507.

[117] J. R. Hobbs and F. Pan, “An ontology of time for the semantic web,” *ACM Trans. Asian Lang. Inf. Process.*, vol. 3, no. 1, pp. 66–85, 2004.

[118] I. Paulsen, D. Mainz, K. Weller, I. Mainz, J. Kohl, and A. von Haeseler, “Ontoverse: Collaborative Knowledge Management in the Life Sciences Network,” in *Proceedings of the German e-Science Conference 2007 GES 2007*, 2007.

[119] C. Tempich, E. Simperl, M. Luczak, R. Studer, and H. S. Pinto, “Argumentation-based ontology engineering,” *IEEE Intell. Syst.*, vol. 22, no. 6, pp. 52–59, 2007.

[120] Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, and D. Wenke, “OntoEdit: Collaborative Ontology Development for the Semantic Web,” in *Proceedings of the First Semantic Web Conference*, 2002.

[121] M. Fernández-López, A. Gómez-Pérez, and N. Juristo, “Methontology: from ontological art towards ontological engineering,” in *Proc. Symposium on Ontological Engineering of AAAI*, 1997.

[122] H. S. Pinto, S. Staab, and C. Tempich, "DILIGENT: Towards a fine-grained methodology for DIstributed, Loosely-controlled and evolvInG Engineering of oNTologies," in *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, 2004.

[123] E. P. B. Simperl, M. Mochol, T. Bürger, and I. O. Popov, "Achieving Maturity: The State of Practice in Ontology Engineering in 2009.," in *OTM Conferences 2*, 2009, vol. 5871, pp. 983–991.

[124] M. M. F. López, "Overview of Methodologies for Building Ontologies," in *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem Solving Methods (KRR5) Stockholm, Sweden, August 2, 1999*, 1999.

[125] N. NOY, "Ontology Development 101: A Guide to Creating Your First Ontology: Knowldege Systems Laboratory, Stanford University," *Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880*, 2001.

[126] C. Knight, D. Gasevic, and G. Richards, "An ontology-based framework for bridging learning design and learning content," *Educ. Technol. Soc.*, vol. 9, no. 1, pp. 23–37, 2006.

[127] G. V. Gkoutos, E. C. Green, A.-M. Mallon, J. M. Hancock, and D. Davidson, "Using ontologies to describe mouse phenotypes," *Genome Biol.*, vol. 6, no. 1, p. R8, 2004.

[128] A. Kim, J. Luo, and M. Kang, "Security ontology for annotating resources," in *Proceedings of the 2005 OTM Confederated international conference on On the Move to Meaningful Internet Systems: CoopIS, COA, and ODBASE*, 2005, pp. 1483–1499.

[129] G. Dobson and A. Sanchez-Macian, "Towards Unified QoS/SLA Ontologies," in *Proceedings of the IEEE Services Computing Workshops*, 2006, pp. 169–174.

[130] A. Seaborne and E. Prud'hommeaux, "SPARQL Query Language for RDF," W3C, W3C Recommendation, Jan. 2008.

[131] M. L. Sbodio, C. Moulin, "SPARQL as an expression language for OWL-S," In OWL-S: Experiences and Directions, a workshop at the 4th European Semantic Web Conference (ESWC 2007), 2007.

[132] J. M. Grimshaw and I. T. Russell, "Effect of clinical guidelines on medical practice: a systematic review of rigorous evaluations," *Lancet*, vol. 342, no. 8883, pp. 1317–1322, 1993.

[133] A. A. Boxwala, M. Peleg, S. Tu, O. Ogunyemi, Q. T. Zeng, D. Wang, V. L. Patel, R. A. Greenes, and E. H. Shortliffe, “GLIF3: a representation format for sharable computer-interpretable clinical practice guidelines,” *J. Biomed. Inform.,* vol. 37, no. 3, pp. 147–161, 2004.

[134] P. A. de Clercq, A. Hasman, J. A. Blom, and H. H. M. Korsten, “Design and implementation of a framework to support the development of clinical guidelines,” *Int. J. Med. Inform.*, vol. 64, no. 2–3, pp. 285–318, 2001.

[135] P. A. de Clercq, A. Hasman, J. A. Blom, and H. H. M. Korsten, “The application of ontologies and problem-solving methods for the development of shareable guidelines,” *Artif. Intell. Med.*, vol. 22, no. 1, pp. 1–22, 2001.

[136] J. Fox, N. Johns, and A. Rahmanzadeh, “Disseminating medical knowledge: the PROforma approach,” *Artif. Intell. Med.*, vol. 14, no. 1–2, pp. 157–182, 1998.

[137] M. A. Musen, S. W. Tu, A. K. Das, and Y. Shahar, “EON: A Component-Based Approach to Automation of Protocol-Directed Therapy,” *J. Am. Med. Inform. Assoc.*, vol. 3, no. 6, pp. 367–387, 1996.

[138] D. R. Sutton and J. Fox, “The Syntax and Semantics of the PROforma Guideline Modeling Language,” *J. Am. Med. Inform. Assoc.*, vol. 10, no. 5, pp. 433–443, 2003.

[139] S. W. Tu and M. A. Musen, “A flexible approach to guideline modeling,” in *Proceeding of AMIA 1999 Annual Symposium*, 1999, vol. 2, Washington DC, pp. 420–424.

[140] D. Wang, M. Peleg, S. W. Tu, A. A. Boxwala, O. Ogunyemi, Q. Zeng, R. A. Greenes, V. L. Patel, and E. H. Shortliffe, “Design and implementation of the GLIF3 guideline execution engine,” *J. Biomed. Inform.*, vol. 37, no. 5, pp. 305–318, 2004.

[141] J. Lee, J. Kim, I. Cho, and Y. Kim, “Integration of workflow and rule engines for clinical decision support services,” *Stud.Health Technol.Inform.*, vol. 160, no. Pt 2, pp. 811–815, 2010.

[142] “OWL web ontology language guide”, http://www.w3.org/TR/owl-guide/

[143] M. A. Casteleiro and J. J. Des Diz, “Clinical practice guidelines: A case study of combining OWL-S, OWL, and SWRL,” *Knowl-Based Syst.*, vol. 21, no. 3, pp. 247–255, 2008.

[144] S. Hussain and S. S. R. Abidi, “A Semantic Web Based Framework for Modelling Clinical Practice Guidelines to Develop Clinical Decision Support Systems,” in *12th World Congress on Health (Medical) Informatics; Building Sustainable Health Systems (Medinfo 2007)*, 2007, Brisbane, Australia.

[145] M. A. Din, S. S. Abidi, and B. Jafarpour, “Ontology based modeling and execution of Nursing Care Plans and Practice Guidelines,” *Stud.Health Technol.Inform.*, vol. 160, no. Pt 2, pp. 1104–1108, 2010.

[146] Z. Aleksovski, M. Klein, W. t. Kate and F. v. Harmelen, "Matching unstructured vocabularies using a background ontology," in Proceedings of the 15th International Conference on Knowledge Engineering and Knowledge Management, 2006, pp. 182-197.

[147] M. Sabou, M. D’Aquin and E. Motta, "Exploring the Semantic Web as background knowledge for ontology matching," *Knowledge*, vol. 11, pp. 156-190, 2008.

[148] A. Daniyal and S. S. R. Abidi, “Semantic Web-Based Modeling of Clinical Pathways Using the UML Activity Diagrams and OWL-S,” in *Knowledge Representation for Health-Care. Data, Processes and Guidelines*, vol. 5943, D. Riaño, A. ten Teije, S. Miksch, and M. Peleg, Eds. Springer Berlin / Heidelberg, 2010, pp. 88–99.

[149] M. Arguello Casteleiro, J. Des, M. J. Prieto, R. Perez, and H. Paniagua, “Executing medical guidelines on the web: Towards next generation healthcare,” *Knowl-Based Syst.*, vol. 22, no. 7, pp. 545–551, 2009.

[150] M. Ceccarelli, A. Donatiello, and D. Vitale, “KON3: A clinical decision support system, in oncology environment, based on knowledge management,” in the 20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2008), 2008, vol. 2, pp. 206-210.

[151] “Qualified cardinality restrictions (QCR),” http://www.w3.org/2001/sw/BestPractices/OEP/QCR/

[152] “OWL 2 Web Ontology Language Profiles”, http://www.w3.org/TR/owl2-profiles/

[153] B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, and U. Sattler, “OWL 2: The next step for OWL,” *J. Web Sem.*, vol. 6, no. 4, pp. 309–322, 2008.

[154] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof and M. Dean, “SWRL: A semantic web rule language combining OWL and RuleML,” http://www.w3.org/Submission/SWRL/.

[155] S. W. Tu and M. A. Musen, “The EON model of intervention protocols and guidelines,” in the Proceeding of AMIA Annual Fall Symposium, 1996, vol. Philadelphia, pp. 587–591.

[156] S. Quaglini, M. Stefanelli, A. Cavallini, G. Micieli, C. Fassino, and C. Mossa, “Guideline-based careflow systems,” *Artif. Intell. Med.*, vol. 20, no. 1, pp. 5–22, 2000.

[157] “InferMed Arezzo technical white paper,” http://www.infermed.com/

[158] J. Fox and S. K. Das, *Safe and sound - artificial intelligence in hazardous applications.* MIT Press, 2000.

[159] D. Beckett and T. Berners-Lee, “Turtle - terse RDF triple language,” http://www.w3.org/TeamSubmission/turtle/.

[160] B. McBride, “Jena: A Semantic Web Toolkit,” *IEEE Internet Comput.*, vol. 6, no. 6, pp. 55–59, 2002.

[161] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, “Pellet: A practical OWL-DL reasoner,” *J. Web Sem.*, vol. 5, no. 2, pp. 51–53, 2007.

[162] B. Motik, U. Sattler, and R. Studer, “Query Answering for OWL-DL with rules,” *J. Web Sem.*, vol. 3, no. 1, pp. 41–60, 2005.

[163] M. A. Din, “Ontology modeling for nursing care plans and clinical practice guidelines,” Master’s dissertation, Dalhousie University, Halifax, NS, 2009.

[164] J. Euzenat, “An API for Ontology Alignment,” in *The Semantic Web “ ISWC 2004*, vol. 3298, S. McIlraith, D. Plexousakis, and F. Harmelen, Eds. Springer Berlin Heidelberg, 2004, pp. 698–712.

[165] J. de Bruijn, D. Foxvog, and K. Zimmermann, “Ontology mediation patterns library v1,” *Deliverable D4*, vol. 3, 2004.

[166] J. Euzenat, F. Scharffe, and A. Zimmermann, “Expressive alignment language and implementation,” Knowledge Web Project Report D 2.2.10, 2007.

[167] B. Jafarpour, S. Abidi, and S. Abidi, “Exploiting OWL Reasoning Services to Execute Ontologically-Modeled Clinical Practice Guidelines,” vol. 6747, M. Peleg, N. LavraÄ, and C. Combi, Eds. Springer Berlin Heidelberg, 2011, pp. 307–311.

[168] D. Riaño, F. Real, J. A. López-Vallverdú, F. Campana, S. Ercolani, P. Mecocci, R. Annicchiarico, and C. Caltagirone, “An ontology-based personalization of health-care knowledge to support clinical decisions for chronically ill patients,” *J.Biomed.Inform.*, vol. 45, no. 3, pp. 429–446, 2012.

[169] "Osteoartherathis Treatment Algorithm," National Health Services Tayside of Scotland, http://www.nhstaysideadtc.scot.nhs.uk/TAPG%20html/pdf%20docs/Section%20%2010%20osteoalgor2.pdf

[170] "Hypertension Pathway," National Institute for Health and Care Excelence, http://pathways.nice.org.uk/pathways/hypertension

[171] "Diabetes Pathway," National Institute for Health and Care Excelence, http://pathways.nice.org.uk/pathways/diabetes

[172] S. A. Hunt, W. T. Abraham, M. H. Chin, A. M. Feldman, G. S. Francis, T. G. Ganiats, M. Jessup, M. A. Konstam, D. M. Mancini, and K. Michl, "ACC/AHA 2005 guideline update for the diagnosis and management of chronic heart failure in the adult a report of the American College of Cardiology/American Heart Association Task Force on Practice Guidelines (Writing Committee to Update the 2001 Guidelines for the Evaluation and Management of Heart Failure)," *Circulation*, vol. 112, no. 12, pp. e154–e235, 2005.

[173] A. Gómez-Pérez, "Ontology evaluation," in *Handbook on ontologies*, Springer, 2004, pp. 251–273.

[174] J. Brank, M. Grobelnik, and D. Mladenić, "A survey of ontology evaluation techniques," in *Proceedings of the Conference on Data Mining and Data Warehouses (SiKDD 2005), 2005.*

[175] J. M. Arnold, P. Liu, C. Demers, P. Dorian, N. Giannetti, H. Haddad, G. A. Heckman, J. G. Howlett, A. Ignaszewski, D. E. Johnstone, P. Jong, R. S. McKelvie, G. W. Moe, J. D. Parker, V. Rao, H. J. Ross, E. J. Sequeira, A. M. Svendsen, K. Teo, R. T. Tsuyuki, M. White, and Canadian Cardiovascular Society, "Canadian Cardiovascular Society consensus conference recommendations on heart failure 2006: diagnosis and management," *Can. J. Cardiol.*, vol. 22, no. 1, pp. 23–45, 2006.

[176] M. Di Bari, C. Pozzi, M. C. Cavallini, F. Innocenti, G. Baldereschi, W. De Alfieri, E. Antonini, R. Pini, G. Masotti, and N. Marchionni, "The diagnosis of heart failure in the community. Comparative validation of four sets of criteria in unselected older adults: the ICARe Dicomano Study,"*J. Am. Coll. Cardiol.*, vol. 44, no. 8, pp. 1601–1608, 2004.

[177] J. García, F. Jose'García-Peñalvo, and R. Therón, "A survey on ontology metrics," in *Knowledge management, information systems, E-learning, and sustainability research*, Springer, 2010, pp. 22–27.

[178] S. Tartir, I. B. Arpinar, M. Moore, A. P. Sheth, and B. Aleman-Meza, "OntoQA: Metric-based ontology quality analysis," in *IEEE Workshop on Knowledge Acquisition from Distributed, Autonomous, Semantically Heterogeneous Data and Knowledge Sources*, 2005, vol. 9.

[179] N. R. Jennings, K. Sycara, and M. Wooldridge, "A roadmap of agent research and development," *Auton.Agents Multi-Agent Syst.*, vol. 1, no. 1, pp. 7–38, 1998.

[180] "Computational Properties of OWL," http://www.w3.org/TR/owl2-profiles/#Computational_Properties

[181] Y.-B. Kang, Y.-F. Li, and S. Krishnaswamy, "Predicting Reasoning Performance Using Ontology Metrics," in *The Semantic Web – ISWC 2012*, vol. 7649, P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. Parreira, J. Hendler, G. Schreiber, A. Bernstein, and E. Blomqvist, Eds. Springer Berlin Heidelberg, 2012, pp. 198–214.

[182] H. Lin and E. Sirin, "Pellint - A Performance Lint Tool for Pellet," in *the OWL: Experiences and Directions (OWLED), Fourth International Workshop*, 2008, Washington, DC, USA.

[183] D. E. Foulser, M. Li and Q. Yang, "Theory and algorithms for plan merging," *Artif. Intell.*, vol. 57, pp. 143-181, October, 1992.

[184] "Health Level Seven," http://www.hl7.org/

[185] P. L. Elkin, S. H. Brown, C. S. Husser, B. A. Bauer, D. Wahner-Roedler, S. T. Rosenbloom, and T. Speroff, "Evaluation of the content coverage of SNOMED CT: ability of SNOMED clinical terms to represent clinical problem lists," in *Mayo Clinic Proceedings*, 2006, vol. 81, pp. 741–748.

[186] W. Ren, R. W. Beard, and E. M. Atkins, "A survey of consensus problems in multi-agent coordination," in *American Control Conference, 2005. Proceedings of the 2005*, 2005, pp. 1859–1864.

[187] S. Yoshikawa, S. Kenji, and A. Konagaya, "Drug interaction ontology (DIO) for inferences of possible drug-drug interactions," *Stud.Health Technol.Inform.*, vol. 107, pp. 454–458, 2004.

[188] "GLINDA: GuideLine INteraction Detection Architecture," http://glinda-project.stanford.edu/guidelineinteractionontology.html