A CUSTOMIZED ILP-BASED SOLVER FOR DESCRIPTION LOGIC REASONERS

Mina Kazemi Zanjani

A Thesis

in

The Department

of

Computer Science And Software Engineering

Presented In Partial Fulfillment Of The Requirements

For The Degree Of Master Of Computer Science

Concordia University

Montreal, Quebec, Canada

April 2014

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: **Mina Kazemi Zanjani**

Entitled**:** **A customized ILP-based solver for Description Logic reasoners**

and submitted in partial fulfillment of the requirement for the degree of

Master of Computer Science

complies with the regulations of the University and meets with the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Nematollah Shiri Chair

Dr. Brigitte Jaumard Examiner

Dr. Juergen Rilling Examiner

Dr. Volker Haarslev Supervisor

Approved by _____

Chair of Department or Graduate Program Director

_____ 20 _____

Dr. Christopher Trueman, Interim Dean

Faculty of Engineering and Computer Science

Abstract

A Customized ILP-based Solver for Description Logic Reasoners

Mina Kazemi Zanjani

Artificial intelligence based systems are known for conveying knowledge through machines. This knowledge is often represented using logic representation languages. One of the well-known families of such languages is called Description Logic (DL) which formally reasons and represents knowledge on the concepts, roles and individuals of an application domain. DL reasoners have been evolving through the years, however when it comes to handling more complicated ontologies with big values occurring in number restrictions, the current reasoners mostly fail to perform efficiently. One of the techniques used in DL reasoners is the so-called atomic decomposition technique which combines arithmetic and logical reasoning. This thesis presents a customized CPLEX-based solver for enhancing DL reasoners through optimizing the atomic decomposition technique. Furthermore, we provide evidence on how this method can improve the reasoning performance by optimizing atomic decomposition. For such purpose, an empirical evaluation of our system for a set of synthesized benchmarks is demonstrated.

Acknowledgments

I acknowledge, with gratitude, my supervisor: Dr. Volker Haarslev for his advice, guidance and encouragement. I would also like to thank my colleagues Jinan El-Hashem, Kejia Wu and Jocelyne Faddoul.

Lastly, I would like to thank my family for their unconditional love and support.

Contents

List of Figures

List of Tables

Chapter 1

1 Introduction

One of the main and old objectives of artificial intelligence based systems is conveying knowledge through machines. Such knowledge is derived from information which itself is based on and gathered from data. When the knowledge is obtained, its representation is often logic-based and passed to machines so that knowledge can be derived in a more efficient way. A well-known family of knowledge-based languages is called Description Logic (DL) which offers more expressive features compared to traditional propositional logic. The main application of DL systems in the artificial intelligence domain is formal reasoning on the concepts of an application domain and representing knowledge about the individuals and the relationship between them. These concepts are any kinds of object which are present in a an application domain. The reasoning process is performed through using a software called DL reasoner or reasoner engine which is capable of inferring logical consequences. To do so, a set of inference rules is required which is defined by

a description language or ontology language. An ontology mainly describes a particular application domain in terms of concepts, roles (the relationship between objects) and axioms (statements which formally describe concepts and roles). These axioms are in fact the logical statements which can be processed through reasoning. For example, an ontology could describe the structure of a university by using concepts such as Employee, Professor and Student. Part of a a university's hierarchy can be captured by an axiom stating that Professor is a subconcept of Employee. Such axioms can be used by the ontology reasoner to perform subsumption reasoning. Subsumption reasoning means reasoning about the hierarchy of ontology concepts. For example, concept B subsumes (is more general than) a concept C iff in every interpretation the set denoted by B is a superset of the set denoted by C. [20][3]. Semantic reasoners have been evolving and upgrading through the years, however when it comes to handling more complicated ontologies especially the ones with big values of number restrictions, the current reasoners mostly fail to perform the reasoning process and deliver inference outputs in an acceptable period of time. Number restrictions which are present in almost all existing systems, allow one determine the number of possible role-fillers of a particular role. For instance, such a restriction can express that a

student may be supervised by least 1 professor, by restricting the number of role-fillers of the is-supervised-by role to more or equal to 1 [1]. This is why reasoner optimization is an on-going research topic and is an open domain for further research. In DL languages, the numerical restrictions on relationships is expressed through number restrictions.

1.1 Number Restriction in Description Logic

In every DL model or scenario, there are relationships which are defined between individuals within that model. Through applying number restrictions it is possible to assign cardinality constraints on these relationships. For instance consider a model which describes a medical school admission requirements. In this model we may use the expressions: Student $\equiv$ ($\geq$ 15 $hasUndergradCredit$) and Student $\equiv$ ($\geq$ 3 $passScienceCourse$) which are used to indicate that every student should have at least 15 undergrad credits and at least pass 3 science courses in order to be admitted in a medical school. Number restrictions enrich DL languages by making the models more expressive, however the enrichment comes with the price of making the model more complex and therefore harder to handle in terms of time and speed during the reasoning process [10].

1.2 Research Objectives and Contributions

Previously, reasoners faced a big challenge in delivering an efficient performance when it came to handling large values of number restrictions as their approach was mainly based on a trial-and-error approach. This issue was addressed in [10] and [9] and to conquer this issue, an algorithm was suggested in [10] which converted the numerical semantics of DL into linear inequalities during a process which is called Atomic Decomposition. Although atomic decomposition has been proven to be effective in later publications and implementations such as [9], however there are still some complex models which could not be easily handled within a reasonable amount of time by the current reasoners. Moreover, tracing back the source of unsatisfiability is still not performed as efficiently as it is required. Therefore further improvement in this area is inevitable. We propose a customized linear inequality solver which adds more efficiency in terms of time to the current reasoners.

1.2.1 Objectives

In this research we pursue the following objectives:

- Develop a customized CPLEX-based solver which is embedded into a DL-based reasoner using the atomic decomposition technique. The cus-

tomized solver would solve the system of inequalities generated by the atomic decomposition process with superior efficiency in terms of time. The results are passed to the reasoner as input for later processing.

• If the system of inequalities is found to be not satisfiable, the source of unsatisfiability is detected by the customized solver and a minimal explanation is generated and sent back as feedback to the reasoner.

1.2.2 Contributions

We can summarize the contributions of this research as follows:

1. We present a customized CPLEX-based solver which can be integrated with a DL-based reasoner to optimize its performance.

2. We present a mechanism as part of our customized solver, which finds the source of detected unsatisfiabilities and provides the reasoner with a minimal explanation for simplifying the process of tracing back the cause of unsatifiability inside a model.

3. We present a system which creates a new virtual bridge between two different systems (CPLEX and DL) for more efficient performance.

4. We analyse the complexity of our proposed system in comparison with existing approaches.

5. We study the practical aspects of implementing such customized system with respect to the available CPLEX services which can be applied in our system.

6. We generate different prototypes of our proposed system and report on their performance and select the one showing the most efficient behaviour.

7. We study the behaviour of our system while working with the HARD reasoner [9] and report on its performance.

1.2.3 Thesis Organization

This thesis consists of seven chapters which are organized in the following order: In Chapter 2, description logic (DL) is defined and explained in detail and an introduction to the DL $\mathcal{ALC}$ and knowledge bases is provided. In Chapter 3, the atomic decomposition process in DL reasoners is studied and reviewed. In Chapter 4, we focus on introducing Integer Linear Programming and providing a detailed description of Simplex as one its most important solving algorithms. In this chapter some variations of the Simplex algorithm are discussed and their complexities studied. Moreover one of the best known simplex-based solvers called CPLEX is introduced and described in the final sections of Chapter 4. In Chapter 5, the architecture of our customized solver

is discussed in more detail and it is compared to other alternative solvers available, in terms of performance. Chapter 6 discuses different test cases and illustrates the results of the tests. Chapter 7 as the final chapter concludes this thesis, points out its achievements based on the theoretical analysis and empirical evaluation, mentions the system limitations and related future work which could pertain as an improvement to our research.

Chapter 2

2 Description Logics

In this chapter, first we will formally define and introduce Description Logics (DL) in Section 2.1. In Section 2.2 two basic DL languages called $\mathcal{AL}$ and $\mathcal{ALC}$ are discussed and in Section 2.3 a knowledge base is described in detail.

2.1 Introduction to DL systems

As defined in [2], Description Logics is a family of knowledge representation (KR) formalisms which is applied for representing the knowledge of an application domain (the "world"). For such purpose, at first, any relevant concept inside this domain is identified or in other words the terminology of the domain is defined. Once the terminology is identified, any concept introduced inside the domain could be employed for determining the properties of objects and individuals occurring in that specific domain. This could be considered as the process of describing the world. The basic fundamental features of DL includes its logic-based semantics along with the reasoning services it offers through these semantics. Any DL has three basic syntactic building blocks

or components. These components are atomic concepts (unary predicates), atomic roles (binary predicates) and individuals (constants).

Definition 1 (Concept). *Any subset of a particular domain is defined as a concept. A concept specifies a set of domain elements with similar characteristics and is represented by using a unary predicate symbol. For example Family is a domain and Child, Mother or Father are sample concepts in this domain.*

Definition 2 (Role). *Roles are defined to describe the binary relationship between individuals inside a domain. For example hasChild is a role which defines a binary relationship between the individuals of concepts Father and Child.*

Definition 3 (Individual). *Individuals are instances of concepts in a particular domain. For instance if John is an individual which belongs to concept Male then we could say John: Male. Furthermore individuals could have relationships with each other. For instance* $\langle John, Tommy \rangle$ *: hasChild states Johnny is the Father of Tommy.*

In Description Logic, classification of concepts provides us with subconcept and superconcept relationships which are called subsumption and classification of individuals distinguishes individuals from each other by determining

which concept they belong to. Such subsumption and classification services enable one to extract implicit knowledge about concepts and individuals from the knowledge that is declared in the knowledge base. Automatic knowledge inference is performed using DL languages. One simple example of such inference could be the following: If Father is a Male and has a relationship called *hasChild* with a Child and we have an individual John where John : Male and $\langle John, Tommy \rangle$: *hasChild*, through KR one could infer that *John : Father*. Such simple descriptions from basic concepts and roles are building components of more complex descriptions. Description Logic is the descendant of so-called "Structured Inheritance Networks" [15] and was originally developed and introduced in KR systems in order to minimize and overcome the ambiguities of the first DL-based KR system called KL-ONE [6] [12].

In the following sections, we will first introduce and describe the DL $\mathcal{AL}$ and then introduce $\mathcal{ALC}$ as the simplest propositionally complete subset of Description Logic. In the last section, an introduction to the basic formalism of Description Logic is provided through terminological (TBox) and assertional (ABox) formalisms.

2.2 DL $\mathcal{ALC}$

In this section, to represent abstract notations, we will use the letters A and B for atomic concepts, the letter R for atomic roles, and the letters C and D for concept descriptions. $\mathcal{ALC}$ is a simple variation of $\mathcal{AL}$-languages. $\mathcal{AL}$ (=attributive language) language has been introduced in [19] as a minimal language that is of practical interest. Any other language belonging to this family is just an extension of $\mathcal{AL}$. To better understand $\mathcal{AL}$ language, it is necessary to know and understand its grammar. Figure 1 shows the grammar of the $\mathcal{AL}$ language:

$C, D \longrightarrow$	$A \mid$	(atomic concept)
	$\top \mid$	(universal concept)
	$\bot \mid$	(bottom concept)
	$\neg A \mid$	(atomic negation concept)
	$C \sqcap D \mid$	(conjunction)
	$\forall R.C \mid$	(universal restriction)
	$\exists R.\top$	(qualified existential restriction)

Figure 1: Grammar of the $\mathcal{AL}$ language

$\mathcal{ALC}$ is one of the basic and propositionally complete DL languages which is also an extension of $\mathcal{AL}$. In $\mathcal{ALC}$, the following grammar is used to form concept descriptions, where $\top$ and $\bot$ are respectively represented by $(C \sqcup \neg C)$ and $(C \sqcap \neg C)$. Figure 2 shows the grammar for $\mathcal{ALC}$ language:

$C, D \longrightarrow$	$A \mid$	(atomic concept)
	$\neg C \mid$	(negation concept)
	$C \sqcap D \mid$	(conjunction)
	$C \sqcup D \mid$	(disjunction)
	$\forall R.C \mid$	(universal restriction)
	$\exists R.C$	(qualified existential restriction)

Figure 2: Grammar of the $\mathcal{ALC}$ language

2.3 Knowledge Base

Knowledge bases can be built based on the services provided by DL-based KR systems. Through DL-based KRs, one can reason and change the contents of any particular knowledge base. A typical knowledge base (KB) has two building components: TBox and Abox. Figure 3 shows the architecture of a knowledge representation system and its components based on Description Logic.

Definition 4 (TBox). *TBox defines the terminological part of a DL knowledge base. A TBox carries knowledge regarding concepts and roles. If C and*

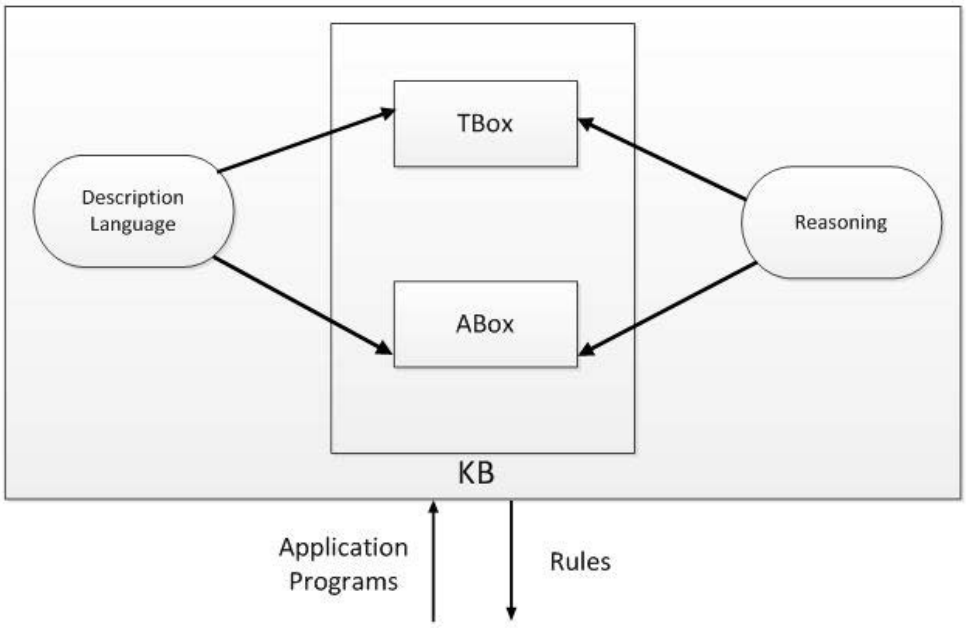

Figure 3: The architecture of a knowledge representation system based on Description Logic

D are concepts, a TBox $\mathcal{T}$ *is defined as a finite set of axioms in the form of* $C \sqsubseteq D$ *(General Concept Inclusion Axioms or GCIs), and/or* $C \equiv D$ *(placeholder for* $\{C \sqsubseteq D, D \sqsubseteq C\}$*).*

The first basic reasoning facility concerning the TBox is determining if a concept denotes nothing or in other words if a concept denotes the empty set in every interpretation. TBox second reasoning facility is computing the subsumption hierarchy [3].

Definition 5 (ABox). *ABox is the assertional part of a DL knowledge base. An ABox contains facts about individuals. An ABox* $\mathcal{A}$ *stands for a finite set*

of assertions of the forms $a : C$, $(a, b) : R$, *where* a *and* b *are the individuals occurring in an ABox* $\mathcal{A}$ *and* R *is a role.*

In other words a TBox introduces the vocabulary of an application domain whereas an ABox is more focused on providing assertions about named individuals in terms of the vocabulary [2]. The reasoning facilities which concerns both TBox and ABox are checking the following:

• Checking whether the represented knowledge is consistent,

• Given an individual of the ABox, the most specific concepts in the TBox which this individual is instance of, are computed,

• Given a concept, compute all the individuals of the ABox that are instances of this concept [3].

Figure 4 illustrates an example of a TBox and an ABox.

Now that we have introduced basic DL languages and knowledge bases,

TBox	*ABox*
$Female \sqsubseteq \neg Male$	$Anne : Human$
	$Anne : Female$
$Animal \equiv Female \sqcup Male$	$Sophie : Woman$
$Human \sqsubseteq Animal$	$Robert : Human$
$Woman \sqsubseteq Human \sqcap Female$	$David : Man$
$Man \sqsubseteq Human \sqcap \neg Female$	$\langle Sophie, Anne \rangle : hasChild$
$Mother \equiv Woman \sqcap \exists\ hasChild.\top$	$\langle Robert, David \rangle : hasChild$
$Father \equiv Man \sqcap \exists\ hasChild.\top$	
transitive(hasChild)	

Figure 4: Example of TBox and ABox

we will discuss one of the techniques used in DL reasoners called atomic decomposition in the next chapter.

Chapter 3

3 Atomic Decomposition

In this section we introduce Atomic Decomposition through an example of applying this method in DL reasoning and we explain why and how this method can be optimized.

3.1 Definition

Based on the definition of atomic decomposition described in [17] the consistency and subsumption problems of some concept formula can be mapped to equation solving problems. To perform such mapping the atomic decomposition technique plays an important role. Atomic decomposition was first introduced in [16] and was defined as a technique which translates cardinality information about finite sets into simple arithmetic terms. This provides a system with the ability to reason about such set cardinalities through solving arithmetic inequality problems. Through this technique, we separate a collection of sets into mutually disjoint components called atoms in a way that the cardinality of the sets are the sum of the cardinalities of their atoms.

Atomic decomposition not only makes it possible to have languages which combine arithmetic formula with set terms, but also it enables translating the formula of this combined logic into pure arithmetical formulas.

3.2 Example

The Venn diagram illustrated in Figure 5 shows the relationship between three sets s, t and v, where s is the set of soccer players, t is the set of tennis players and v is the set of volleyball players.

As observed in the Venn diagram in Figure 5, given 3 role filler, $2^3 = 8$

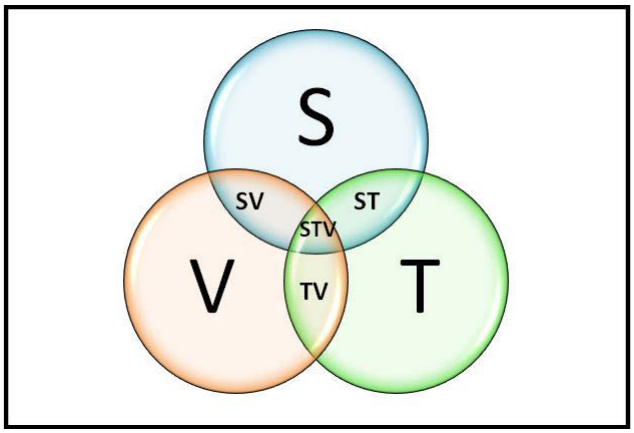

Figure 5: Venn diagram for Tennis,Volleyball and Soccer players

different areas are generated: s,t,v,st,sv,tv and stv. Informally and in simple terms each of these partitions have the following meanings attached to them:

s = plays only soccer, not volleyball, not tennis.

t = *plays only tennis, not volleyball, not soccer.*

v = *plays only volleyball, not soccer, not tennis.*

st = *plays only soccer and tennis, not volleyball.*

sv = *plays only soccer and volleyball, not tennis.*

tv = *plays only volleyball and tennis, not soccer.*

stv = *plays soccer and volleyball and tennis.*

The original sets could be rewritten based on their 'atomic' components in the following way:

$plays\text{-}soccer = s \cup sv \cup st \cup stv$

$plays\text{-}volleyball = v \cup sv \cup tv \cup stv$

$plays\text{-}tennis = t \cup st \cup tv \cup stv$

Since the above decomposition generates partitions (mutually disjoint sets), the sum of the cardinalities of the above roles are as follows:

$|plays\text{-}soccer| = |s| + |sv| + |st| + |stv|$

$|plays\text{-}volleyball| = |v| + |sv| + |tv| + |stv|$

$|plays\text{-}tennis| = |t| + |st| + |tv| + |stv|$

Having introduced the above cardinality terms, we can formulate inference problems. For instance if we know that no one plays tennis and at most 2 players play volleyball and at least 3 players play two games, then we can

conclude that at least 1 player is playing soccer:

$|t| \leq 0 \wedge |v| \leq 2 \wedge |sv| \geq 3 \Rightarrow |s| \geq 1$

Furthermore, as mentioned before, these sets are mutually disjoint which makes using cardinality terms rather irrelevant, therefore we could replace them with non-negative integer valued variables. Therefore we could conclude the formula below:

$x_t \leq 0 \wedge x_v \leq 2 \wedge x_{sv} \geq 3 \Rightarrow x_s \leq 1$

Considering the above example, in atomic decomposition, we start with the most general decomposition of the sets into their atomic components. During this process cardinality terms are converted to arithmetic terms which ultimately leads to a pure linear Diophantine equation problem. In the next step subset, disjointness and exhaustiveness relations between the different sets are exploited which makes some of the atoms empty, therefore simplifies the problem. In final step, we end up with a formula which can be submitted to an arithmetic equation solver [16].

3.3 Atomic Decomposition in Description Logic

Atomic Decomposition was originally introduced by [17] to handle arithmetic aspects of concept languages and to enrich formal systems which have

a language with a notion of (existentially quantified) variables such as mathematical programming systems for solving system of equalities or inequalities. In description logic, this method is useful in the role part and semantically, the sets which are decomposed and transformed are in fact the set of role fillers of a particular object. One of the significant merits of using atomic decomposition in DL systems is transforming complex subsumption and consistency problems into rather simple arithmetic equality problems. However, it is important to keep in mind that this method could become rather questionable in particular cases. For instance, the decomposition of a set with l elements yields 2^l, i.e. exponentially many atoms. This means even for small numbers of l, we can have cases which are unmanageably large. Hence it is better to exploit every possible way to optimize this technique. There are already some available optimization techniques for elevating the atomic decomposition technique, examples of such are *Relevancy Principle* (factoring out irrelevant boolean variables) and *Factoring Principle* (reducing the overall number of syntactic atoms by labelling them) [17]. One simpler way to improve the performance of atomic decomposition would be solving the inequality sets with minimum time. For such purpose, we have decided to perform the inequality solving process of the atomic decomposition proce-

dure through our customized system which is based on one of the powerful optimizing systems called CPLEX. More details of the system are presented in Chapter 5.

As we presented an introduction about DL systems and atomic decomposition, we now proceed to the next chapter to introduce linear programming along with one of its well-known algorithms called Simplex and ILOG CPLEX as a linear programming optimizer.

Chapter 4

4 Linear Programming (LP)

Linear programming (LP) is a special case of mathematical programming (optimization) which is defined as a method to gain the best outcome such as profit or lowest cost. In a typical LP all requirements are represented by linear relationships. Linear programming is considered as an important tool for combinatorial search problems. This is due to the fact that not only it solves efficiently a large class of important problems, but also because it is the basic block of some fundamental techniques in this area [7]. Most of linear programs can be practically solved in polynomial time and robust solvers are now available that solve large scale linear programs. One of the well-known optimizers which offers such solving service is IBM ILOG CPLEX optimizer. CPLEX is the first commercial linear optimizer written in the C language, providing great flexibility, reliability and performance efficiency for generating better and more efficient optimization algorithms, models and applications. In this chapter linear programming and its components are introduced in more detail mostly based on the materials presented in [11].

The Simplex algorithm has been the basis for the entire field of mathematical optimization and has served as the first practical method for solving linear programming problems, therefore we will discuss this algorithm in more detail in this chapter. Moreover ILOG CPLEX is presented and described as one of the optimization tools that targets these problems.

4.1 Introduction to LP

The concept behind every linear programming problem can be simply explained through four basic components. These components include decision variables, an objective function, constraints, and parameters. Decision variables are the quantities which are supposed to be determined. Objective functions determine the way the decision variables affect the optimization process (minimization or maximization). Constraints represent limits on decision variables. Parameters are responsible for quantifying the correlation between decision variables and the objective function as well as constraints. In linear programs, there is a linear relationship between decision variables in the objective function and the constraints. This feature of linear programs enables formulating real-world problems and facilitates an analytical decision-making process. In basic linear optimization problems, the variables

of the objective function are most often continuous in the mathematical sense, meaning there are no gaps between real values. However, there are some cases in which some or all the variables are restricted to be integers, this category of LPs is referred to as integer linear programming (ILP). LP can be formulated as follows:

$$Maximize \qquad Z = f(x), \tag{1}$$

$$Subject\ to: \qquad Ax \leq b, \tag{2}$$

$$and \qquad x \geq 0 \tag{3}$$

where equality (1) is called an objective function and Z represents the objective function to be maximized or minimized (in this case to be maximized). The inequalities (2) and (3) represent the constraints over which the objective function to be optimized. In these inequalities, x represents the vector of decision variables which are to be determined, A represents a known matrix of coefficients and b is a vector of known coefficients.

4.2 Simplex Algorithm

As stated in [11] in mathematical optimization, Simplex method is a general procedure which was originally developed by George Dantzig in 1947 for solving linear programming problems. Due to its efficiency, this method is the routine base used by so many applications to solve huge mathematical programming problems on computers. In the following sections we provide an introduction to the basics of Simplex and describe its main features.

Simplex method is originally an algebraic procedure based upon geometry. Before going through the algebraic aspects of Simplex, it is useful to understand its geometric basis. Therefore for better understanding of the algorithm, we will demonstrate Simplex algorithm through a simple example. First from a geometric and then from an algebraic point of view. The Linear Program (LP) model for our example is the following:

$$Maximize \qquad Z = 5x_1 + 8x_2, \tag{4}$$

$$Subject\ to: \qquad x_1 + x_2 \leq 6 \tag{5}$$

$$5x_1 + 9x_2 \leq 45 \tag{6}$$

$$and \qquad x_1, x_2 \geq 0 \tag{7}$$

In the above example, equality (4) represents the objective function and inequality (5), (6) and (7) represent the set of constraints imposed on the decision variables.

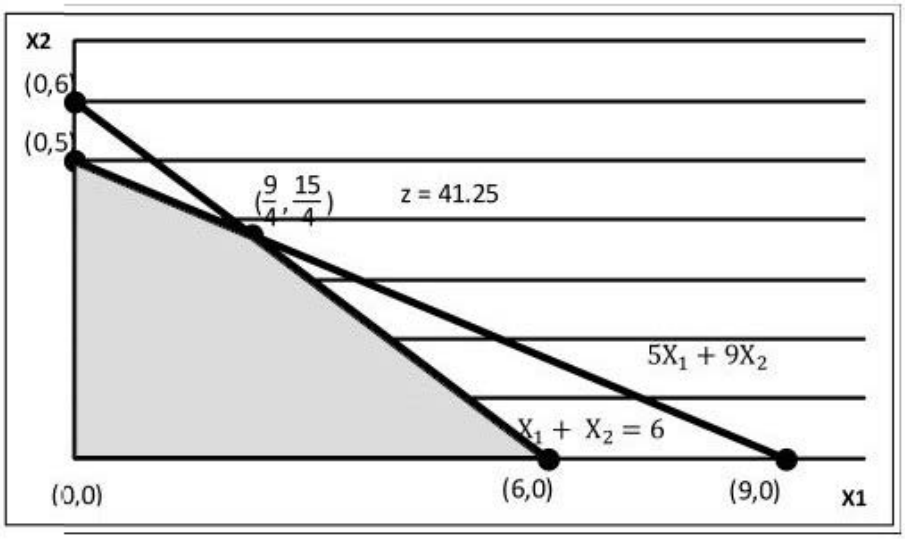

Figure 6: The graph of model (4)-(7)

4.2.1 Simplex Algorithm: geometric point of view

The graph of the model in the previous section is represented in Figure 6. The graph illustrates the two constraint boundaries and their intersecting points. Each of the constraint boundaries is represented by a line and the intersection of these two lines forms a boundary area that corresponds to the

values that are permitted by the imposed constraints. This area is called the feasible region and the intersection points are called the corner-point solutions of the problem. In our example, as displayed in Figure 6, there are four corner-point solutions: $(0,0)$,$(0,5)$,$(\frac{9}{4},\frac{15}{4})$ and $(6,0)$. Since these points lie on the feasible region, they are referred to as the corner-point feasible solutions (CPF solutions). The other corner point solutions are called the corner-point infeasible solutions: (i.e. $(0,6)$ and $(9,0)$). As illustrated in the graph, each CPF solution is connected to two other CPF solution, meaning each CPF solution is adjacent to another CPF solution through sharing an edge. The adjacent CPF solutions are important in the optimality test process. The optimal solution is defined below:

Definition 9 (Optimality Test). *For every linear programming problem with at least one optimal solution, if a CPF solution does not have any better adjacent CPF solutions (in terms of objective function value), then that CPF solution must be an optimal solution.*

Generally Simplex method consists of two major steps in order to solve linear programming problems, CPF solution selection and the optimality test. These two steps are repeated during iterations until an optimal solution is reached. We now explain briefly how Simplex performs from a geometric

point of view. Using the model presented in Section 4.2, we will have the following steps:

• *CPF solution selection:* corner-point (0,0) is chosen as the initial CPF solution.

• *Optimality Test:* (0,0) is not an optimal solution since (0,5) and (6,0) are better adjacent CPF solutions compared to (0,0).

• *CPF Solution Selection:* Moving from (0,0) toward the x_2 axis, we choose (0,5) as the next better CPF solution (since x_2 has greater coefficient compared to x_1 in the objective function ($8 > 5$), therefore moving toward the x_2 axis would maximize Z faster than moving toward the x_1 axis).

• *Optimality Test:* (0,5) is not an optimal solution since an adjacent CPF solutions is better.

• *CPF Solution Selection:* the next better adjacent CPF solution is located at the intersection of the two constraints at $(\frac{9}{4}, \frac{15}{4})$.

• *Optimality Test:* There is no better adjacent CPF compared to point $(\frac{9}{4}, \frac{15}{4})$ which leads us to the optimal solution of $Z = 41.25$.

Having introduced the geometric aspects of Simplex algorithm, we may look into its algebraic aspect in the next section.

4.2.2 Simplex Algorithm: Algebraic point of view

Since Simplex method is mostly run on a computer and computers can only follow algebraic instructions, it is crucial to translate the geometric version of Simplex into the algebraic form. To perform such a translation, the inequalities inside a particular model are converted to equalities in a general standard LP model. For the model introduced earlier in Section 4.2 or for any other maximization problem, slack variables should be introduced. In optimization problems where the objective function aims for maximization, a slack variable is defined as a variable which is being added to an inequality constraint in order to change it to an equality constraint [5]. We can create slack variable (s_i) for each constraint i in the above problem. For instance consider inequality (5) from the above example:

$$x_1 + x_2 \leq 6 \tag{5}$$

In the above inequality, the left hand side is less than or equal to 6. To convert the inequality to an equality we may add a non negative value to the left hand side. This value could be increased till the point where the left hand side is equal to the right hand side. This positive value is called

slack variable, meaning it compensates the slack on the left hand side of the inequality. Labelling this slack variable as x_3, the first slack variable could be defined as below:

$$x_3 = 6 - x_1 - x_2 \tag{8}$$

following the definition of the slack variable and knowing that they all have positive values, we can define a slack variable for inequality (6) as well:

$$x_4 = 45 - 5x_1 - 9x_2 \tag{9}$$

Having introduced slack variables we rewrite the problem into its augmented form, where the original form of the model has been augmented by the supplementary variables (slack variables):

$$Maximize \qquad Z - 5x_1 - 8x_2 = 0, \tag{10}$$

$$Subject\ to: \qquad x_1 + x_2 + x_3 - 6 = 0 \tag{11}$$

$$5x_1 + 9x_2 + x_4 - 45 = 0 \tag{12}$$

$$for \qquad x_i \geq 0 \qquad for\ i = 1, 2. \tag{13}$$

The above model is called the augmented form of the first model as it is augmented with supplementary variables called slack variables. The augmented form is more useful and convenient when it comes to finding solutions and that is the reason they are generated through introducing slack variables.

If a slack variable equals 0 in the current solution, then this solution lies on the constraint boundary for the corresponding functional constraint. A value greater than 0 means that the solution lies on the feasible side of this constraint boundary, whereas a value less than 0 means that the solution lies on the infeasible side of this constraint boundary. The above problem can have a *Basic* or a *Basic Feasible (BF) Solution*. The basic solution is an augmented corner-point (CP) solution and a basic feasible solution is defined as an augmented CPF solution. In the augmented form of model (4)-(7) there are 4 variables and 2 equations in overall. The difference between these two values would be:

$$\textit{Number of variables} - \textit{Number of equations} = 4 - 2 = 2$$

The above equation is interpreted as 2 degrees of freedom in finding a solution to the system, meaning any two variables could be set to arbitrary values in order to solve the two equations in terms of the remaining two variables. In Simplex, these values are set to zero and called non-basic vari-

ables. The other two variables are called basic variables. After introducing basic and non-basic variables, now we can proceed with Simplex algorithm. Like the geometric procedure, the algebraic procedure is based on two main steps: first deciding which variables should be set to zero (non-basic variables) and then performing the optimality test to decide whether an optimal solution has been achieved. Like the geometric procedure, these two steps are repeated in multiple iterations until the point where an optimal solution is obtained. Considering the augmented model, the following steps are taken during Simplex algorithm:

• *Choosing non-basic variables:* We first choose x_1 and x_2 to be the non-basic variables and set them to zero. This leads us to the BF solution: $(0, 0, 6, 45)$.

• *Optimality Test:* The solution is not optimal because increasing either one of the non-basic variables (x_1 or x_2) would increase Z.

• *Choosing non-basic variables:* Since x_2 has a higher coefficient in the objective function and it increases Z faster, it is chosen as the non-basic variable to be increased. While increasing x_2, the other variables values must be adjusted to satisfy the system of equalities. When one of the basic variables drops to zero, we stop increasing the value of x_2. In our case for $x_2 = 5$ (basic variable), x_4 would be zero (non-basic variable). The new BF solution

would be: $(0, 5, 1, 0)$. The new system of inequalities would be as follows:

$$Maximize \qquad Z = \frac{5}{9}x_1 - \frac{8}{9}x_4 + 40, \tag{14}$$

$$Subject\ to: \qquad \frac{4}{9}x_1 + x_3 - \frac{1}{9}x_4 = 1 \tag{15}$$

$$5x_1 + 9x_2 + x_4 = 45 \tag{16}$$

$$for \qquad x_i \geq 0 \qquad for \quad i = 1, 2. \tag{17}$$

• *Optimality Test:* The system is still not optimal since increasing the other non-basic variable x_1 increases Z.

• *Choosing non-basic variables:* x_1 and x_4 are set as non-basic variables and their values are zero. The value of x_1 is increased until one of the basic variables x_2 or x_3 is zero. Variable x_3 drops to zero (new non-basic variable) and x_1 is set as new basic variable. The next BF solution is $(\frac{9}{4}, \frac{15}{4}, 0, 0)$.

• *Optimality Test:* Since increasing either x_3 or x_4 would decrease Z, $(\frac{9}{4}, \frac{15}{4}, 0, 0)$ is the optimal solution with $Z = \frac{165}{4} = 41.25$.

4.3 Simplex Efficiency

Simplex algorithm operates through moving from one basic feasible solution to another one without returning to a previously visited solution, therefore the number of iterations in Simplex is at most the number of basic feasible solutions. This is true for all non-cycling variants of Simplex method. In the cycling variants of Simplex, the worst case is infinite. With respect to these facts, linear programming with simplex is proved to have expected $O(n^3)$ time complexity for average cases and $O(2^n)$ for the worst case [21] [23]. It is important to mention that although Simplex efficiency in most practical average cases is Non-deterministic Polynomial-time hard (NP-hard), however in some other cases, it has been recorded that worst-case complexity of simplex method can be exponential time [18].

4.4 Branch-and-Bound Algorithm

Branch-and-Bound algorithm is one of the ILP algorithms which improves the computational efficiency while complementing algorithms like Simplex. Branch-and-Bound algorithm is focused on implicitly enumerating feasible integer solutions [11]. Branch-and-Bound algorithm is done through three main steps: branching, fathoming and bounding. It is fundamentally based

on a divide and conquer approach, meaning it breaks very large problems that are difficult to solve into smaller and easier sub-problems. During the branching or dividing phase, the whole set of feasible solutions is partitioned into smaller subsets of feasible solutions. By performing LP relaxation, each of these smaller sub-problems is assigned a bound on how good its best feasible solution could be. Relaxation of a problem means deleting one set of constraints that makes the problem difficult to solve. More specifically in IP problems, relaxation most often refers to deleting integrality constraints. During fathoming or conquering, the best solution in each subset is bounded and in cases where the bounds do not allow the subset to contain an optimal solution for the main problem, that subset is discarded (fathomed). To better understand this algorithm, we will first define each of these steps in more details and then apply it to our earlier example model (4)-(7) from Section 4.2 by adding the integrality condition.

The branch-and-bound algorithm for a maximization problem like model (4)-(7) could be summarized in the following steps:

•*Initialization:* In the first step Z^* should be set to $-\infty$. Z^* is defined as the value of Z for the current incumbent solution. The incumbent refers to the best feasible integer solution found so far.

• *Branching:* Branching starts by selecting the most recently created and unfathomed sub-problem (breaking a tie based on the ones with larger bounds). In the next step, one of the variables with a fractional (non-integer) value, in the optimal solution for the LP relaxation of the sub-problem is chosen. Suppose the chosen variable is x_j and its value in the optimal solution is represented by ${x_j}^*$. We create two new sub-problems from the node for the sub-problem by using two constraints where $[{x_j}^*]$ denotes the largest integer less than or equal to x_j:

$$j \leq [{x_j}^*] \qquad and \tag{18}$$

$$x_j \geq [{x_j}^*] + 1 \tag{19}$$

• *Bounding:* In this stage, we apply Simplex or Dual Simplex (in case of re-optimizing) to LP relaxation of each of newly created sub-problems and use the value of Z in the optimal solution to obtain their upper bounds.

• *Fathoming:* In this phase, we apply three fathoming tests on each new sub-problem. If the sub-problem passes any of these three tests, it is fathomed and could be discarded.

Test 1: Its upper bound is $\leq Z^*$.

Test 2: LP relaxation is infeasible.

Test 3: The LP relaxation results are all integer solutions. In this case if the solution is better than the incumbent, then the incumbent is updated and the solution is assigned as the new incumbent. Then finally we would test all other unfathomed sub-problems which have a new and larger Z^* using test 1.

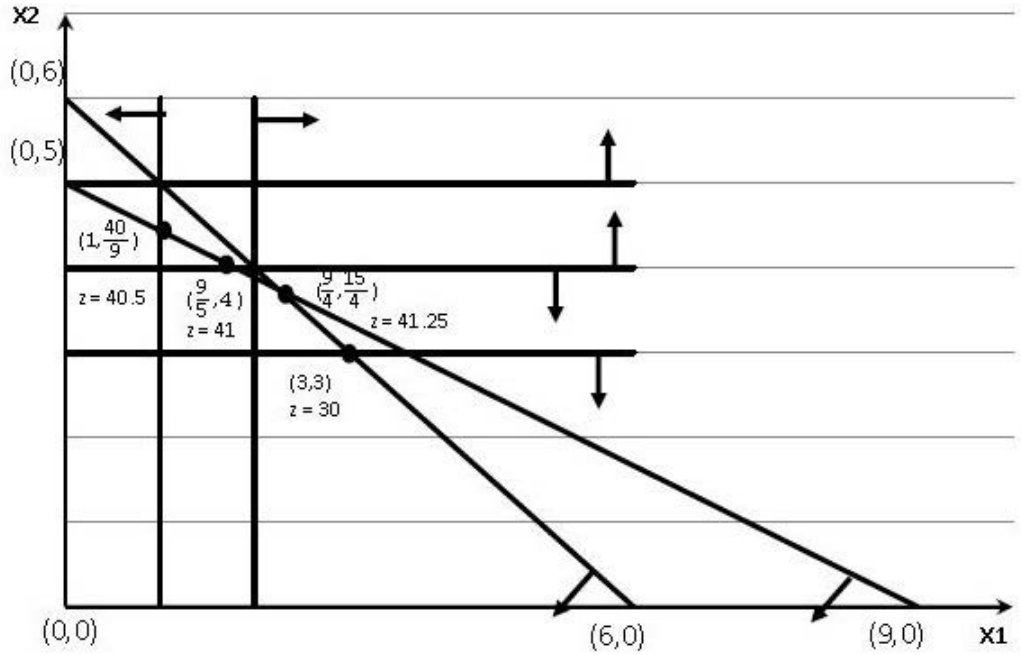

Figure 7: Branch-and-Bound Graph for IP model(4)-(7)

At the end of each iteration the optimality test is performed. In case there are no sub-problems left, we would stop the iteration and choose the current incumbent as the optimal one. In another scenario where there is no incumbent solution, then the problem is infeasible.

Now that we have discussed the algorithm methodology, we may apply it to our earlier example model (4)-(7) in section 4.2:

$$Maximize \qquad Z = 5x_1 + 8x_2, \tag{4}$$

$$Subject\ to: \qquad x_1 + x_2 \leq 6 \tag{5}$$

$$5x_1 + 9x_2 \leq 45 \tag{6}$$

$$and \qquad x_1, x_2 \geq 0 \tag{7}$$

Having $x^* = (\frac{9}{4}, \frac{15}{4})$ and $Z^* = 41.25$ and after performing LP relaxation, we could conclude that the upper bound is 41.

First Iteration :

Branching: Pick x_2 as the variable for branching. Knowing $x_2 = \frac{15}{4} = 3.75$

Let $x_2 \leq 3$, the original problem would be as below:

S_1 :

$$Maximize \qquad Z = 5x_1 + 8x_2, \tag{20}$$

$$Subject\ to: \qquad x_1 + x_2 \leq 6 \tag{21}$$

$$5x_1 + 9x_2 \leq 45 \tag{22}$$

$$x_2 \leq 3 \tag{23}$$

$$\textit{and} \qquad x_1, x_2 \geq 0 \tag{24}$$

Now let $x_2 \geq 4$ *, the original problem* becomes:

S_2 :

$$\textit{Maximize} \qquad Z = 5x_1 + 8x_2, \tag{25}$$

$$\textit{Subject to}: \qquad x_1 + x_2 \leq 6 \tag{26}$$

$$5x_1 + 9x_2 \leq 45 \tag{27}$$

$$x_2 \geq 4 \tag{28}$$

$$\textit{and} \qquad x_1, x_2 \geq 0 \tag{29}$$

1. Maximize $Z = 5x_1 + 8x_2$,

   subject to:

2. $x_1 + x_2 \leq 6$

3. $5x_1 + 9x_2 \leq 45$

4. $x_2 \geq 4$

   and

5. $x_1, x_2 \geq 0$ and integers

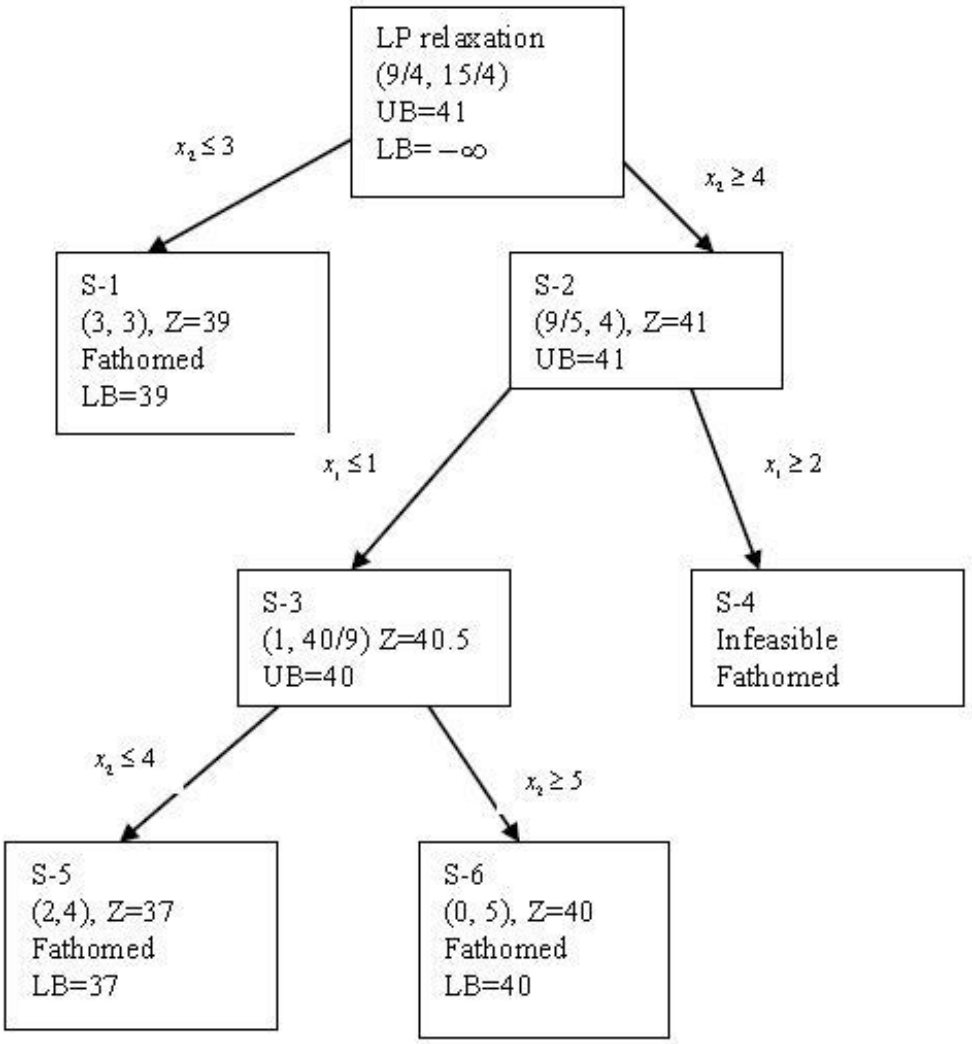

Figure 8: Branch-and-Bound Tree for Model(4)-(7)

Bounding: we would first solve S_1 and the optimal solution would be (3, 3) and Z=39. Having an integer solution, we can call this solution as the current incumbent solution meaning Z is the best lower bound for the IP

problem. In next step we would solve S_2 which results in optimal solution of $(\frac{9}{5}, 4)$ and $Z = 41$. $Z = 41$ would be selected as the best upper bound.

Fathom: S_1 is fathomed since its solution is is an integer. S_2 is not fathomed because further branching from S_2 might result in $Z > 39$ (current incumbent solution). We would continue the algorithm until the third iteration where the incumbent solution is (0, 5) and it is the optimal solution to the original IP problem. Figure 7 and 8 represent a branch-and-bound tree and a graph which summarise complete branch-and-bound algorithm iterations for the above example.

4.5 Dual Simplex Algorithm

Simplex method described in Section 4.2 is known as the Primal Simplex algorithm. In a tableau implementation of the Primal Simplex, all right hand side elements of each inequality are always non-negative, which results in feasible basic solutions in every iteration of the algorithm. In an alternative scenario where some of the right hand side elements have negative values, primal problem is infeasible. Primal Simplex starts with a feasible basis and looks for an optimal basis while maintaining feasibility. In an alternative algorithm known as Dual Simplex, the algorithm starts with an optimal

basis and looks for a feasible basis while maintaining optimality. In this kind of Simplex algorithm, in all iterations the basis is initially both primal infeasible and dual feasible. Dual feasible means neither the coefficients nor the right hand side of the objective function are negative. The solution at the final and optimal iteration should be both primal and dual feasible, meaning during Dual Simplex, the goal is to maintain dual feasibility and obtain primal feasibility. It is important to note that both Primal and Dual Simplex algorithms reach the same solution for the same problem, however they reach such solution through taking different directions. Dual Simplex is mostly applicable and suited in cases where the problem could easily reach an initial dual feasible solution. For instance, in cases where adding new constraints or changing parameters makes the previously optimal solution infeasible, Dual Simplex is an efficient algorithm for re-optimizing the problem [14]. Both Primal and Dual versions of the same problem share the same set of parameters except in different locations. The following shows the comparison between primal and dual versions of the problem more clearly:

•$PrimalProblem$:

$$Maximize \qquad Z = cx, \tag{30}$$

$$Subject\ to: \qquad Ax \leq b \tag{31}$$

$$and \qquad x \geq 0 \tag{32}$$

•$DualProblem$:

$$Maximize \qquad W = by, \tag{33}$$

$$Subject\ to: \qquad Ay \geq c \tag{34}$$

$$and \qquad x \geq 0 \tag{35}$$

4.6 ILOG CPLEX

The IBM ILOG CPLEX optimizer is an executable program which reads a problem interactively or from files in certain standard formats, solves these problems and then delivers a solution interactively or in form of text files. In general CPLEX offers C, C++, Java and .NET libraries which are designed to solve LP and LP related problems. More specifically, CPLEX solves those optimization problems which are linearly or quadratically constrained and their objective functions could be expressed in form of a linear function or a convex quadric function. These optimization models allow variables with continuous or integer only values [13]. Originally the IBM ILOG CPLEX studio combines high-performance ILOG CPLEX optimizer solvers with an integrated development environment (IDE) and a powerful Optimization Pro-

gramming Language (OPL). In the following sections, we will discuss CPLEX components and then briefly introduce different categories of mathematical problems which are targeted and solved by CPLEX.

4.6.1 CPLEX components

CPLEX is represented in various forms to meet a wide range of users' needs. It mainly consists of three components: Interactive Optimizer, Concert Technology and Callable Library. The CPLEX Interactive Optimizer is the component which is in charge of reading and solving the problem and delivering a relevant solution to that problem. Other important and powerful component of the CPLEX Optimizer is Concert Technology, a modelling layer which provides interfaces to programming languages. ILOG Concert technology consists of a set of modelling objects shared in common with OPL, IBM ILOG CPLEX Optimizer and IBM ILOG CPLEX CP Optimizer. There are three different Concert technology language implementations or class libraries: C++, Java and .NET (C# and Microsoft Visual Basic). These libraries provides us with an API. The API consists of modelling facilities which enables programmers to embed CPLEX optimizers in C++, Java, or .NET applications. For this purpose the Concert technology libraries have

to make use of CPLEX Callable Libraries. CPLEX Callable Library is itself a C library which allows the programmers to embed CPLEX optimizers in applications written in C, Visual Basic, FORTRAN, or any other language that needs to call C functions. CPLEX default settings allows programmers to call an optimizer that is appropriate for the class of problem being solved. However, it also provides alternative options to choose a different optimizer for special purposes. An LP problem can be solved by using Dual Dimplex, Primal simplex, Barrier, and perhaps also Network optimizer (in cases when the problem contains an extractable network substructure). It is important to keep in mind that the choice of optimizer or other parameter settings might have a strong impact on the solution speed of the particular class of problem being solved [13].

4.6.2 Types of problems solved by CPLEX

CPLEX is a solving tool designed for a large variety of optimization problems. Some of the best known categories of problems targeted by CPLEX are Linear Programming Problems (LP), Quadric Programs (QP), Problems with Quadratic Constraints (QCP), Mixed Integer Programs which itself contains Mixed Integer Linear Programs (MILP) and Mixed Integer Quadratic

Programs (MIQP) and Network problems.

LP problems which were introduced in detail in Section 4.1 are the most basic linear optimization problems which are solved by CPLEX. For LP problems, CPLEX optimizers are based on both Primal and Dual Simplex algorithms. QPs are those problems whose objective function is not linear but quadric. QCPs are similar to QPs, however in QCPs, one or more constraints might contain quadric terms and the objective functions do not necessarily contain quadric terms. In MIP problems, both continuous variables (e.g. reals) and discrete variables (e.g. integers) might be present in the objective functions and constraints. MIPs with linear objective terms are referred to as MILPs and MIPs with quadric objective terms are known as MIQPs. Network-flow problems which are partly or entirely structured as a network and they are handled by ILOG CPLEX Network optimizer. For those problems which are largely structured as a network, a Network optimizer for the populated LP object is applied. If however the entire problem contains network flows, a network object should be created, populated and then solved by the network optimizer.

4.6.3 ILOG Concert Technology for Java Applications

Since our customized CPLEX-based solver was implemented in Java, using the CPLEX API for Java was inevitable. Thus, we provide a brief abstract view of CPLEX design in Concert technology for Java applications. Figure 9 shows the design of Concert technology and its interaction with a user-application. Concert technology accomplishes this interaction through defining a set of interfaces for modelling objects. The interfaces defined by Concert technology do not actually consume memory (for this reason, the box in the figure has a dotted outline). To create a CPLEX model, any user-written application needs to create an *IloCplex* object which is stored in the ILOG CPLEX database. This object creates the variables, constraints and the objective function of the model by implementing a Concert technology modelling interface. The application can access all modelling objects, for instance the variables, through the Concert technology interface. *IloCplex* and the modelling interfaces of Concert technology then communicate with ILOG CPLEX internals (as shown in Figure 9). The computing environment, its communication channels and any objects of the problem are contained inside the ILOG CPLEX internal. One important advantage of using the design shown in Figure 9 is the fact that the code for creating the model through

Concert technology modelling interface could be used not only with *IloCplex* but also with any other classes which create objects. Therefore this feature allows us to be able to use other ILOG optimization technologies to solve our model [13].

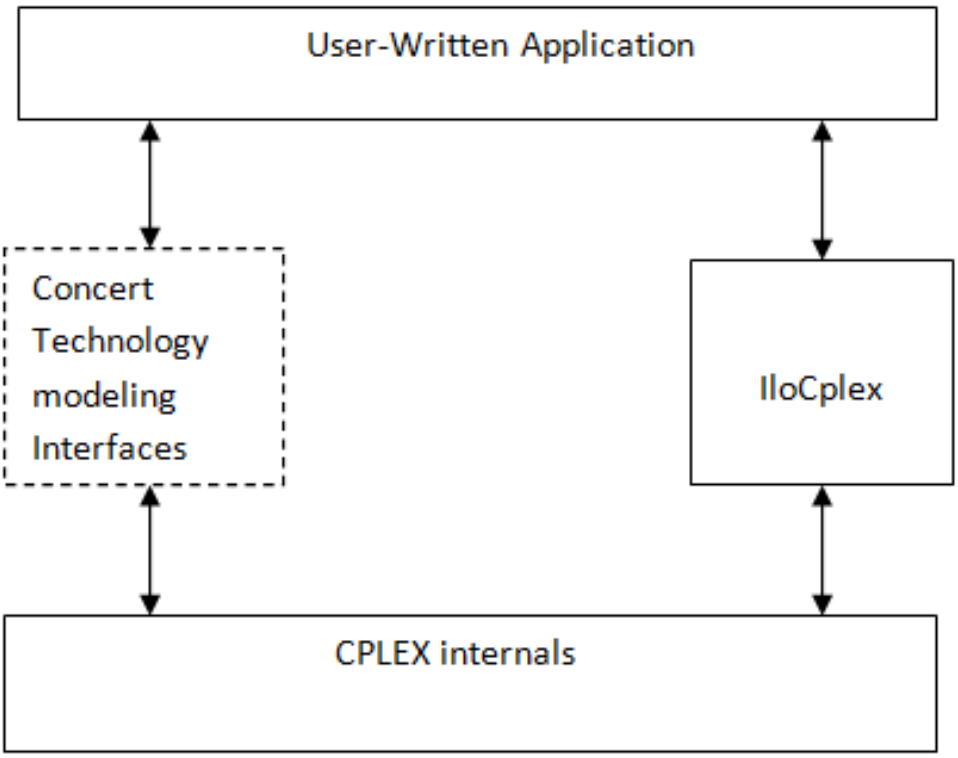

Figure 9: View of concert technology for Java applications

In the next chapter we will explain the methodology of our implemented system with respect to the information that was presented in this chapter and the previous ones.

Chapter 5

5 Customized API Description

In this chapter, we will explain the architecture of the implemented customized CPLEX based solver, which employs IBM CPLEX optimizer (introduced in Chapter 4) in a way that can improve the performance of DL semantic reasoners through enriching atomic decomposition procedure (presented in Chapter 3) during DL reasoning. The developed system is implemented in such way which later on can be integrated in forms of libraries in any DL reasoning engine. In the following sections, first, two main modules of the developed system and the underlying functions of these modules are discussed in more detail. In the final section, application of our system to solve a DL problem is illustrated through an example.

5.1 Architecture

As illustrated in Figure 10, the developed customized system consists of two main modules: the inequality set handler module and the solver module. The input of the system is initially generated from atomic decomposition

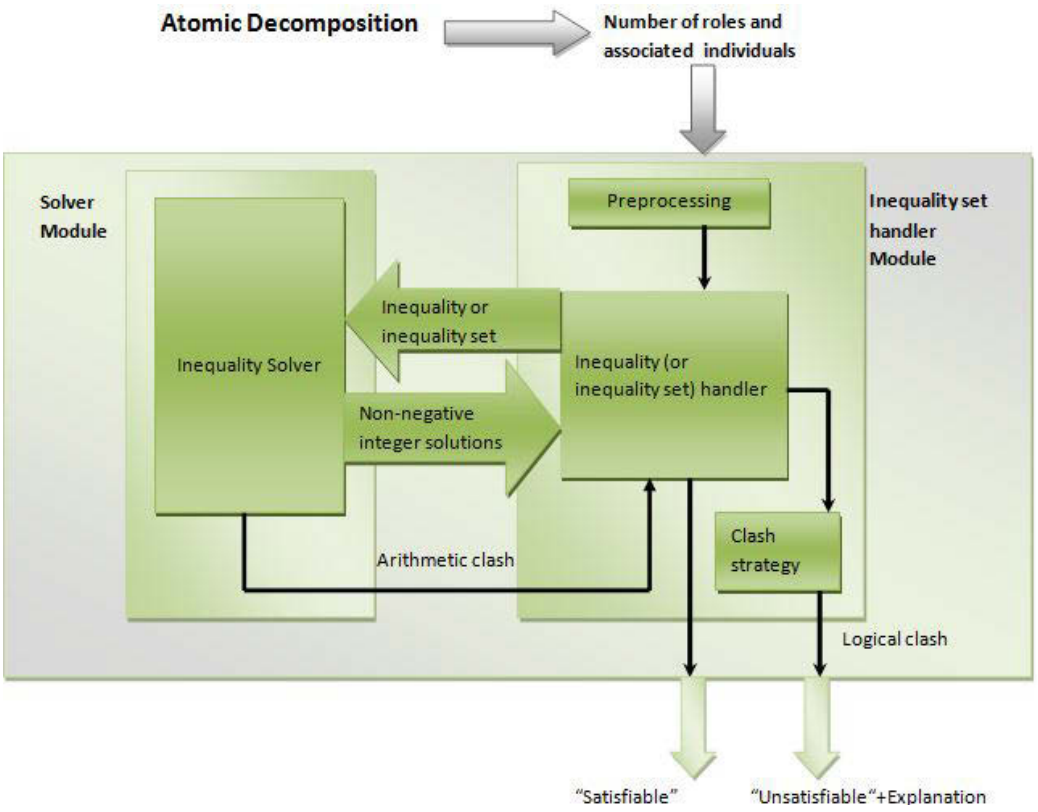

Figure 10: The architecture of customized CPLEX based solver

procedure during DL semantic reasoning process and it consists of a set of inequalities. These inequalities are then fed to the system in terms of number of roles and individuals associated with them. These values are then passed to and interpreted by the inequality set handler module. The inequality set is then passed to the solver module to decide whether the set is solvable or not. Through this communication between the two modules, the feasibility of the set is decided and if the set is mathematically feasible, the system provides *'satisfiable'* as the output plus the solutions of the inequality system. In

case of infeasibility, the output of the system would be 'unsatisfiable' along with an explanation determining the source of infeasibility. In the following sections the architecture of the system is explained in more detail.

5.1.1 Inequality Set Handler Module

This module serves as the initiating module which translates the users' input to define the problem to be solved by the solver. Based on the values given to the module (roles and associated individuals), the inequalities are generated in a sub-module named processor. The generated inequality set is then passed to the second sub-module. The inequality set handler communicates with the second module of the system (solver), through translating the inputs into inequalities which are to be passed to the solver and retrieving the solving feedback. The communication between the two modules is mainly based on the feasibility of the system of inequalities. If the system is feasible, the first module either continues to retrieve more input or it acknowledges the externals users (or system) that the inequality set is satisfiable. In cases where an arithmetic infeasibility is detected, arithmetic clashes are sent back to the handler and later on to the clash strategy sub-module for further investigation in order to detect the sources of infeasibility. After finding a

clash an explanation is provided explaining possible causes of the clash. The communication between the two modules can be done in an incremental or non-incremental fashion.

5.1.2 Solver Module and IBM CPLEX Optimizer

As mentioned in the previous section, after receiving the input from the semantic reasoner and creating a set of inequalities, this set is to be tested for feasibility. This module solves the inequality set through Dual Simplex algorithm by calling CPLEX API. The feedback is either a mathematical clash which is to be interpreted by the clash strategy module or it is a non-negative set of integer solutions.

5.1.3 Clash Strategy Module

As a clash is detected by the solver module and a clash feedback is sent back to the clash strategy module, CPLEX is invoked to make the inequality set feasible again. This is done through using one of CPLEX methods called *feasOpt()*. The model becomes feasible and more relaxed by removing some of the inequalities (constraints). These removed inequalities are stored in a an external database such as a text file. The feasible model along with the

removed inequalities is passed to the next iteration of the system so it can be combined with a newly added inequality. This process is repeated every time an infeasibility is detected. When all the inequalities are added to the system, based on the number of times each removed inequality is repeated in the database (a minimum of three times), the clash strategy module decides which inequalities caused the model to become infeasible. This module is useful for DL reasoners as it helps us detect the source of unsatisfiability during the reasoning process which leads to reducing redundancies in the backtracking process. Figure 11 shows how clashes are handled in the clash strategy module.

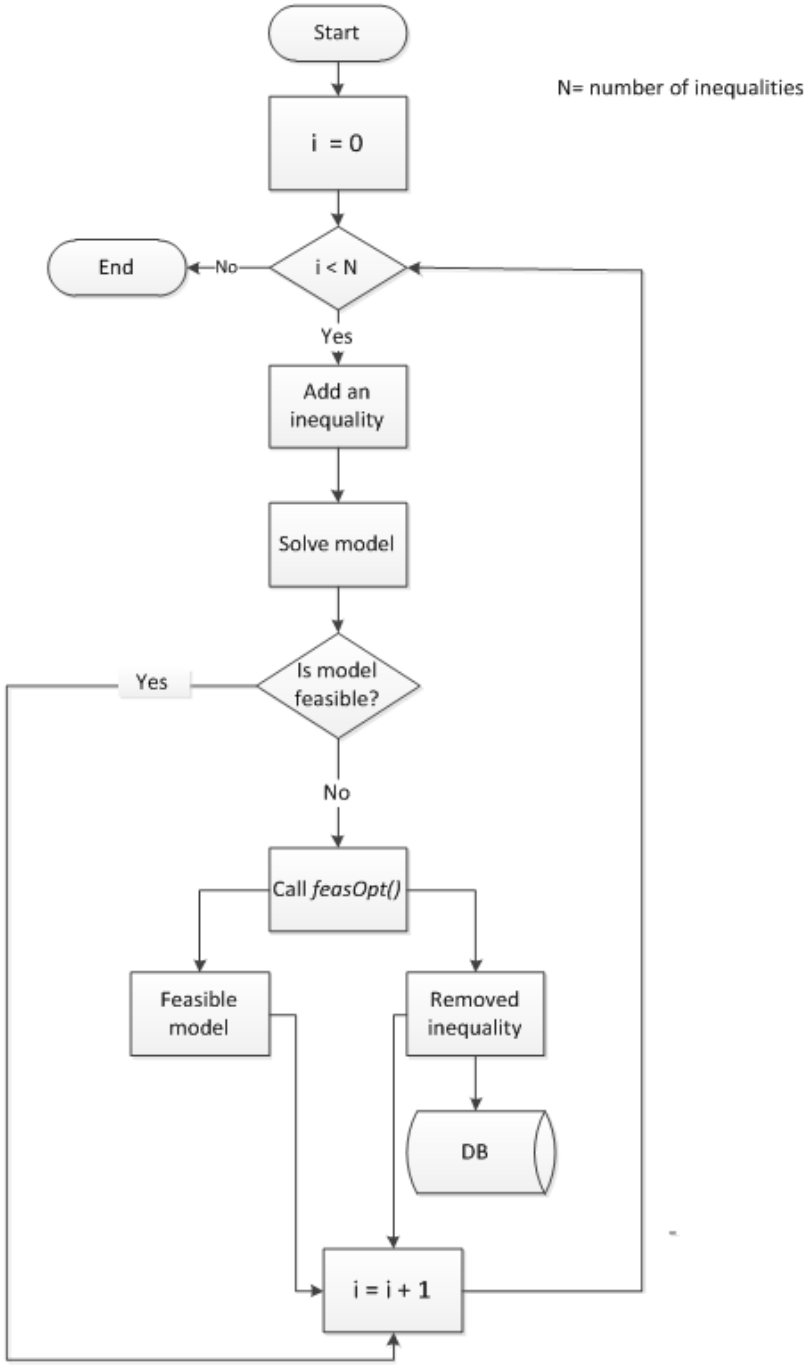

Figure 11: Clash handling flowchart

5.2 Free Alternative Solvers

Different optimization software modules can be easily tested and implemented on the same function *f*, meaning a given optimization software can be used for different functions *f*. In our case since the system is generated in Java, any optimization software which offers an internal library for Java-based applications can be a replacement to the CPLEX optimizer. In the following sections we look into two of these available solvers. The performance of these solvers may not be as efficient as CPLEX, however not only they are license-free solvers but also they are open source and hence they provide the users with the ability to change the implementation details of the solver operations through modifying its underlying code. Due to these reasons we suggest these solvers as possible alternatives to CPLEX.

5.2.1 Lpsolve

Lpsolve is a free Mixed Integer Linear Programming (MILP) solver which is fundamentally based on Simplex and Branch-and-Bound methods. Lpsolve is considered to have no limit on model size. However, as the size of the models gets bigger, it gets harder for the solver to handle the problems. It is possible to use Lpsolve as a library and call it through using different programming

languages like Java, C, .NET, VB, Delphi and etc. One can pass the data to the Lpsolve library through its API, input files and IDE. The latest version of this solver is Lpsolve 5.5.2.0. It is possible to use the Lpsolve API inside an application in two ways: dynamically or statically. In the dynamic version, the Lpsolve API is dynamically linked to application code enabling the code to use the API while the executable has started or the Lpsolve library is being called. Statically linking the API and the application does not require extra files because the code is already included in the executable. However, recompiling the program is needed if there is an update of Lpsolve. Since Lpsolve API is not object-oriented, to call the API in Java, a wrapper is provided though which Lpsolve API is available. Currently, the Lpsolve does not support infeasible constraint detection, therefore it is advised to anticipate and explicitly model the realistic sources of infeasibility [22].

5.2.2 The Cassowary

Cassowary is a linear arithmetic constraint solving algorithm which is developed to solve systems of constraints. This solver was originally created to target linear equality and inequality constraints which arise while specifying aspects of user interfaces, particularly layout and other geometric relations.

The cassowary algorithm is based on the Dual Simplex method and since its main focus is dealing with interactive graphical applications, it supports incremental and repetitive problem solving during which prior computations are exploited. The algorithm incrementally adds or deletes new constraints to the model and performs re-optimization in order to find a better solution. This algorithm has been implemented in Smalltalk, C++, and Java and its latest version (v0.60.) is freely available [4].

5.3 Implementation

The proposed system was implemented considering two main scenarios. These two include non-incremental and incremental implementations. In the following sections each of these scenarios is discussed in more detail.

5.3.1 Non-incremental Implementation

The first implementation scenario is more focused on solving the problem (system of inequalities) in a non-incremental fashion, meaning all the inequalities are added to the system once and then the solver is called to solve the whole system (Algorithm 1). The solver would try to perform all possible combinations to reach feasibility and return a satisfiable set of inequalities.

Once the permutation process is over, in case of infeasibility, a minimal explanation is presented to justify the source of infeasibility. Since in this scenario all the possible combinations are tested, the run time is expected to be high. Moreover, as our system is going to be in close collaboration with DL semantic reasoners and the nature of these reasoners is mostly incremental, (In semantic reasoners, new roles and concepts are frequently introduced to the system and this gives the system an incremental nature), for the purpose of consistency, it is more desirable to have a system whose solving process is performed in a non-incremental fashion. Due to the mentioned points, we expected a better and more efficient performance from an incremental implementation and decided to implement the system both incrementally and non-incrementally. Comparing the performance of these two scenarios could clarify the correctness of our assumptions. In the following section, the incremental implementation is discussed in more detail.

5.3.2 Incremental Implementation

In the incremental implementation, the system is implemented in such a manner that it can process the inequality set with the inequities being added to the system incrementally. Each time a new role is introduced during

the reasoning process, new inequalities and variables are introduced by the atomic decomposition process, therefore, the system needs to be able to re-check the feasibility of the system of inequalities after each change. If the changes do not alter the feasibility state of the current system, new inequalities can still be added. However, once an infeasibility is detected, the system needs to trace back to the previous feasible state to find the cause of infeasibility and provide an explanation regarding its source. The incremental implementation was implemented in two versions. In the first version, in each iteration once an inequality was added to or removed from the system, a CPLEX model was created (Algorithm 2). In the second version, instead of remodelling after each iteration, CPLEX methods were used to update the existing model (Algorithm 3).

Algorithm 1 Non-Incremental implementation

for all $i = 1$ to count, such that count = Number of inequalities **do**
  add each inequality to the model
end for
if model is solvable **then**
  Return the results
else {model is not solvable}
  for all inequality $i \in$ model **do**
    check the solvability of model without each inequality i through permutation
    if model is solvable **then**
      Remove inequality i
      Save the removed inequality in database
    end if
  end for
end if
Determine the cause of infeasibility based on the removed inequalities in database

Algorithm 2 Incremental Implementation Using Remodelling

Create an array of size count where count = number of inequalities
Initialize every inequality to -1
for all $i = 0$ to count, where count= number of inequalities **do**
  Add the inequality to the model
  Solve the model
  if model is not solvable **then**
    Remove the most recent inequality i
    Set the most recent inequality i to 0
    Remodel for all inequalities $i = -1$
    Save the removed inequality i in database
  end if
end for
Determine the cause of infeasibility based on the removed inequalities in database

Algorithm 3 Incremental Implementation Using Model Modification

```
for all i = 0 to count,where count= number of inequalities do
    add the inequality to the model
    solve the model
    if model is not solvable then
        Call CPLEX feasOpt method
        Save the removed inequality in database
        Add the removed inequality to the feasible model
        Pass the model to next iteration
    end if
end for
Determine the cause of infeasibility based on the removed inequalities in
database
```

5.4 Solving DL Problems Using the Customized API

Consider the scenario in which we want to design an ontology for a sport family domain. To formally define the children of such family, one can come up with the following definition:

$$Children \equiv (\geq 2\ Plays_{Soccer}) \sqcap (\geq 2\ Plays_{Tennis}) \sqcap (\leq 3\ Plays_{Game}) \quad (36)$$

Considering the Venn diagram for this scenario in Figure 12 and by using atomic decomposition, the definition (36) can be rewritten in terms of its atoms:

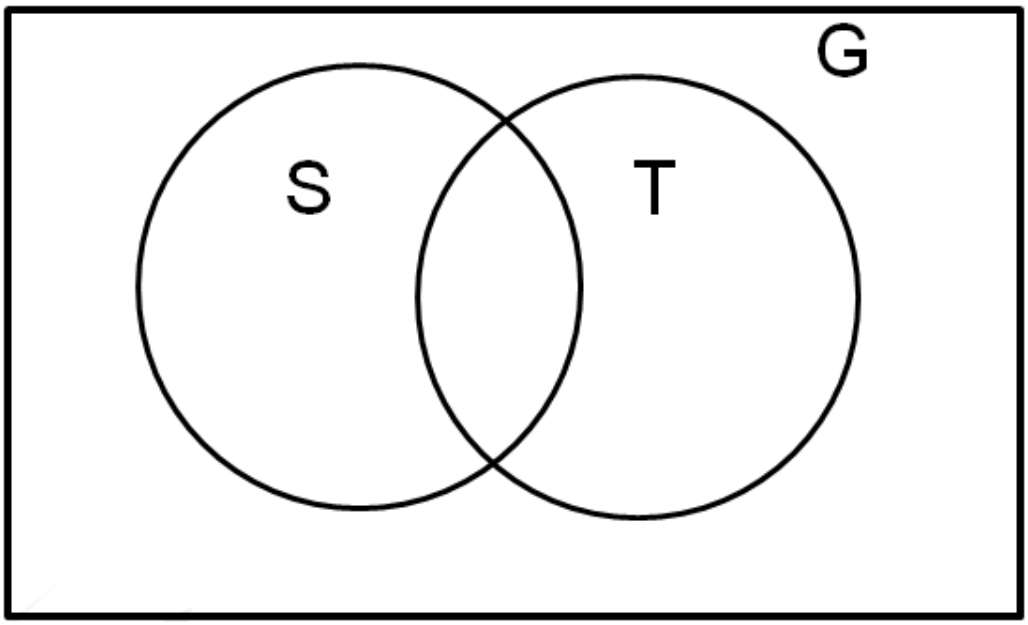

Figure 12: Venn diagram used for atomic decomposition technique

$$x_S + x_{ST} \geq 2 \tag{37}$$

$$x_T + x_{ST} \geq 2 \tag{38}$$

$$x_S + x_{ST} + x_T + x_G \leq 3 \tag{39}$$

As the above set of inequalities is generated through atomic decomposition, each one of these inequalities is added to the CPLEX model one by one (incremental implementation). After each addition, the feasibility of the model is tested. For instance, for the above inequality set, the solver detects the model to be feasible with the following solution results:

$$x_S = 1\ ,\ \ x_{ST} = 1\ ,\ \ x_T = 1\ ,\ \ x_G = 0 \tag{40}$$

If we suppose that no child is allowed to play both soccer and tennis, we will need to add the following inequality to the model:

$$x_{ST} \leq 0 \tag{41}$$

However adding the inequality (41) makes the model infeasible, which ultimately initiates clash handling module. To make the model feasible again, inequality (41) is removed from the model and is used for further inequality source detection purposes.

Now that we have explained the methodology of our system, we will present the evaluation results of our system in the next chapter.

Chapter 6

6 Evaluation

In this chapter, we are going to present the empirical results of implementing three different prototypes of our proposed CPLEX-based solver in Chapter 5. Each of these prototypes is briefly introduced in Section 6.1. In Section 6.2 the benchmarks and the contributing factors affecting the performance of the system are briefly introduced. The results of test cases implemented on these prototypes are illustrated through benchmarks in Section 6.3. Moreover the prototype with the superior performance is evaluated while dealing with real-world problems generated by the HARD reasoner. HARD is based on the algebraic tableau reasoning algorithm and was originally created as a test bed for ReAl DL (Reasoning Algebraically with DL). Given an ontology file, HARD can determine the consistency of the underlying ontology [8].

6.1 Prototype Description

As mentioned in Chapter 5, we consider incremental and non-incremental implementations while developing our system. These two scenarios were

explained in Chapter 5. To fully evaluate the efficiency of each of these scenarios, we have implemented three prototypes and tested each of them through different test cases. These prototypes include: Incremental Prototype Using Remodelling, Incremental Prototype Using Model Modification and Non-Incremental Prototype. The first prototype is based on the idea presented in Section 5.3.2, meaning it solves the system of inequalities in an incremental fashion. In each loop, as a new constraint is added to the CPLEX model, the model is solved and in case of any infeasibilities, the model is modified to become feasible by removing the constraints causing the infeasibilities. The second prototype works in the same incremental way, however it does not modify the model to make it feasible, but it creates a new feasible model which does not contain the constraints generating the infeasibilities. The last prototype works in a non-incremental way as described in section 5.3.1.

6.2 Benchmarking

In order to study the behaviour of our developed system, a set of synthetic benchmarks has been developed. Since the performance of the system is mainly dependent on the number of variables and inequalities that are in-

volved in each run, these two parameters are the permanent participants in each benchmark. In general, the following parameters can be identified as the ones contributing to the performance changes:

1. The number of variables
2. The number of inequalities
3. The feasibility or infeasibility of each state of the system

6.3 Evaluation Results

In this section through using benchmarks, we describe each test case and illustrate the result of each of them on all three prototypes and evaluate, explain and compare their performances in terms of time and efficiency. The following experiments were performed using Windows 64 on a standard PC with dual-core (2.10 GHz) processor and 8 GB of RAM. For achieving a more accurate and precise result, each experiment was executed in 5 runs.

6.3.1 Test Case 1: Linearly Increasing the Number of Inequalities

In this test case, with a fixed number of variables (=5), we increase the numbers of inequalities which are fed to the system with a linear growth of $f(x) = 10x$. The feasibility of the system changes randomly. The behaviour

of each of three prototypes for this test case is illustrated in Figure 13. Based on the results and as shown in Figure 13, both incremental prototypes perform better compared to the non-incremental one. Such result justifies our earlier assumption in Chapter 5. We predicted that solving the system of inequalities in an incremental fashion would be much faster compared to a non-incremental one as we do not have to consider all possible combinations. However the performance for the non-incremental prototype is mostly con-

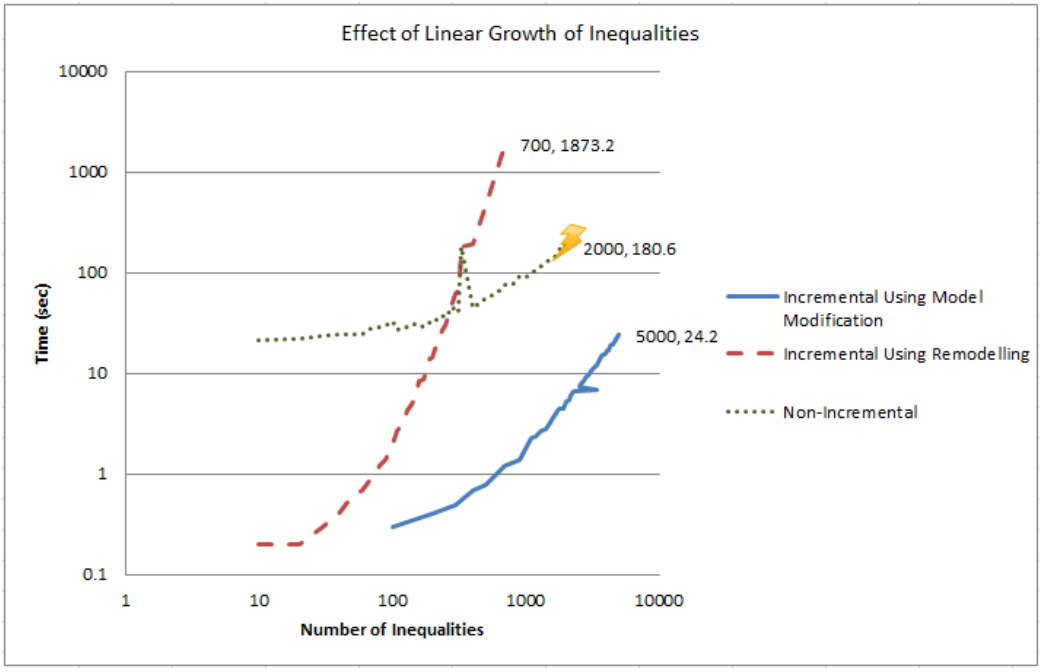

Figure 13: Behaviour of the prototypes: Linear growth of Inequalities

sistent. Between the two incremental prototypes, the one which updates the model rather than creating a new model, has a better performance for large number of inequalities. This means it is more efficient in this test case to

update the model after each iteration to reach a feasible state than creating a new one which is feasible.

6.3.2 Test Case 2: Exponentially Increasing the Number of Inequalities

In this test case, the number of variables is fixed (=5), however the number of inequalities is increased with an exponential growth of x^{10}. The feasibility of the system changes randomly in each iteration of incremental prototypes. As illustrated in Figure 14, for the first 100 inequalities, both incremental

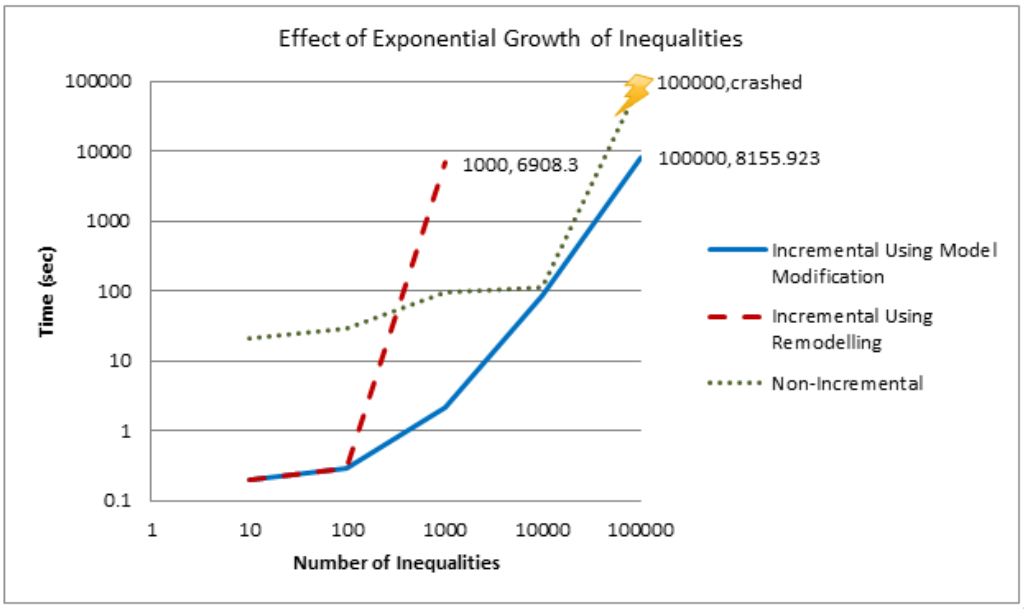

Figure 14: Behaviour of the prototypes: Exponential growth of Inequalities

prototypes show a similar behaviour, however the one updating the model, turns out to be faster as the number of inequalities grows higher. Both

incremental prototypes outperform the non-incremental one.

6.3.3 Test Case 3: Linearly Increasing the Number of Variables

Test case 3, tests the system while the number of inequalities is fixed (=10) and the number of variables fed as input to the system increases linearly with linear growth of $f(x) = 10x$. The infeasibility of the system changes randomly. Figure 15 shows the results for this test case on all three prototypes. The performances of two incremental prototypes are similar to each other, however the non-incremental prototype is almost hundred times slower.

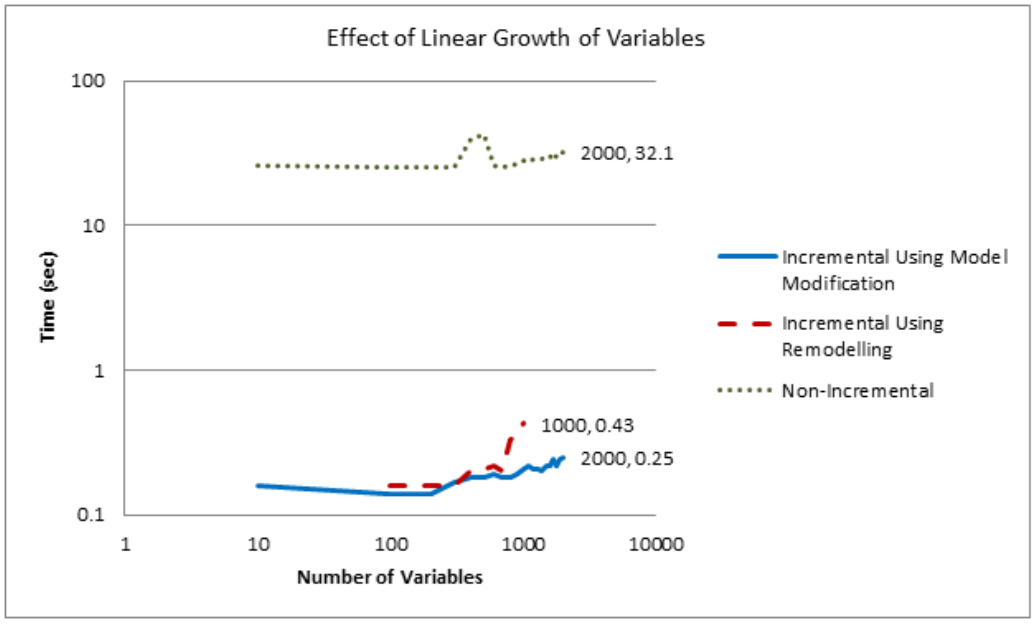

Figure 15: Behaviour of the prototypes: Linear growth of Variables

6.3.4 Test Case 4: Exponentially Increasing the Number of Variables

In this test case, the number of variables is increased with an exponential growth of $f(x) = 10^n$ and the number of inequalities is fixed (=10) with a random feasibility state.

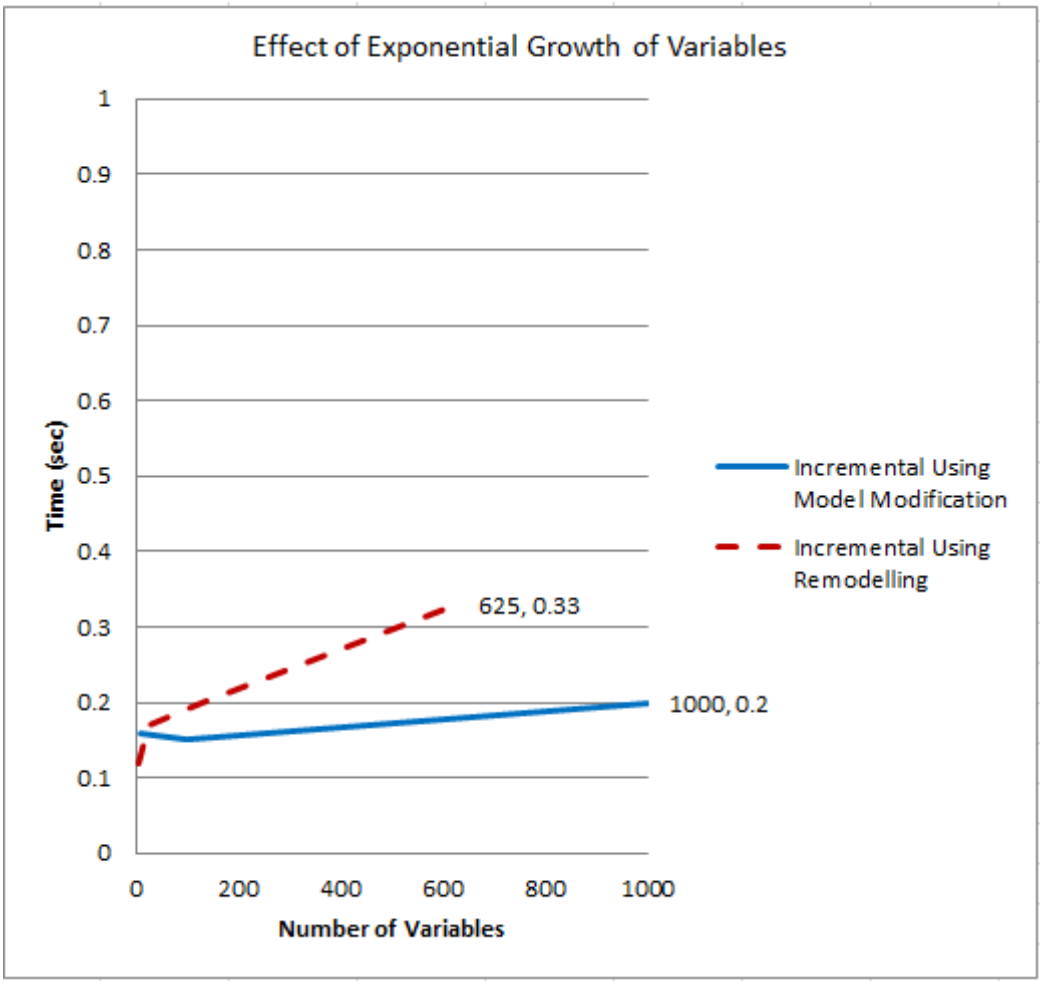

Figure 16: Behaviour of the prototypes: Exponential growth of Variables

We present the benchmark for the incremental prototypes and compare their performances. As shown in Figure 16, the execution time is longer for this test case compared to test case 3. However updating the CPLEX model still results in more efficient performance.

6.3.5 Test Case 5: Linearly Increasing the Number of Inequalities Using Feasible States

Test Case 5 is designed in order to monitor the effect of feasibility on the performance of the system in an incremental implementation. In this test case the number of variables is fixed, while the number of inequalities increases with a linear growth of $f(x) = 10x$. The system remains feasible after each iteration. As it is illustrated in Figure 17, the performance of both incremental prototypes for this test case is similar due to the fact that the state of the system does not change during the execution, therefore there is no need for remodelling or model modification.

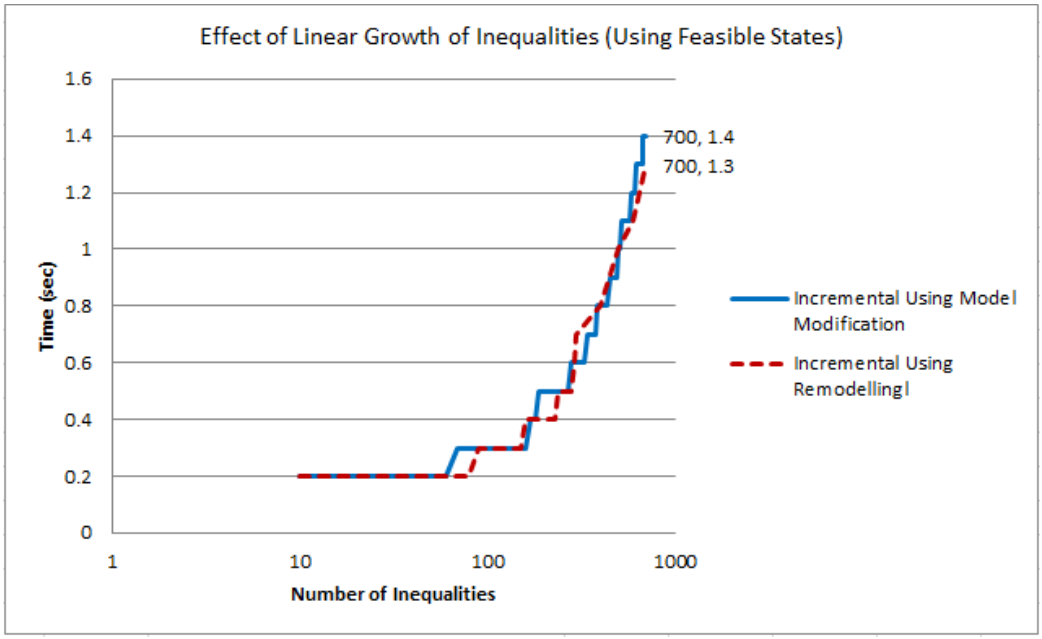

Figure 17: Behaviour of the prototypes: Linearly growth of Inequalities (Using Feasible model)

6.3.6 Test Case 6: Linearly Increasing the Number of Inequalities Using Infeasible States

In this test case, with a fixed number of variables (=5), we test the performance of our system while the number of inequalities increases with the linear growth of $f(x) = 10x$ and the system remains in an infeasible state after each iteration. Based on the benchmark of this test case in Figure 18, we conclude that in infeasible scenarios, updating the current model is more efficient compared to remodelling the current model.

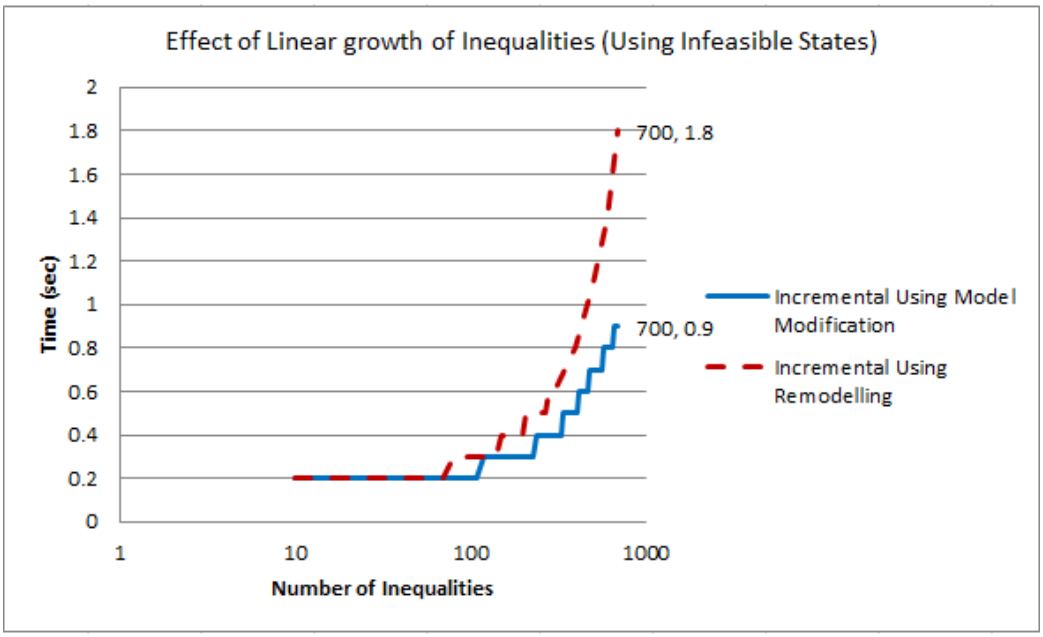

Figure 18: Behaviour of the prototypes: Linearly growth of Inequalities (Using Infeasible States)

6.3.7 Test Case 7: Linearly Increasing the Number of Inequalities Using Feasible or Infeasible States

The focus of this test case is on the behaviour of incremental prototypes when the system shifts from a feasible state to an infeasible one. For the same linear growth function of $f(x) = 10x$ and a fixed number of variables (=5), the incremental prototypes are tested in a feasible state for 700 iterations. In the first 350 iterations the state of the system is feasible and for the next 350 iteration it becomes infeasible. Figure 19 shows the changes made to the performance of our incremental systems. The benchmark shows the

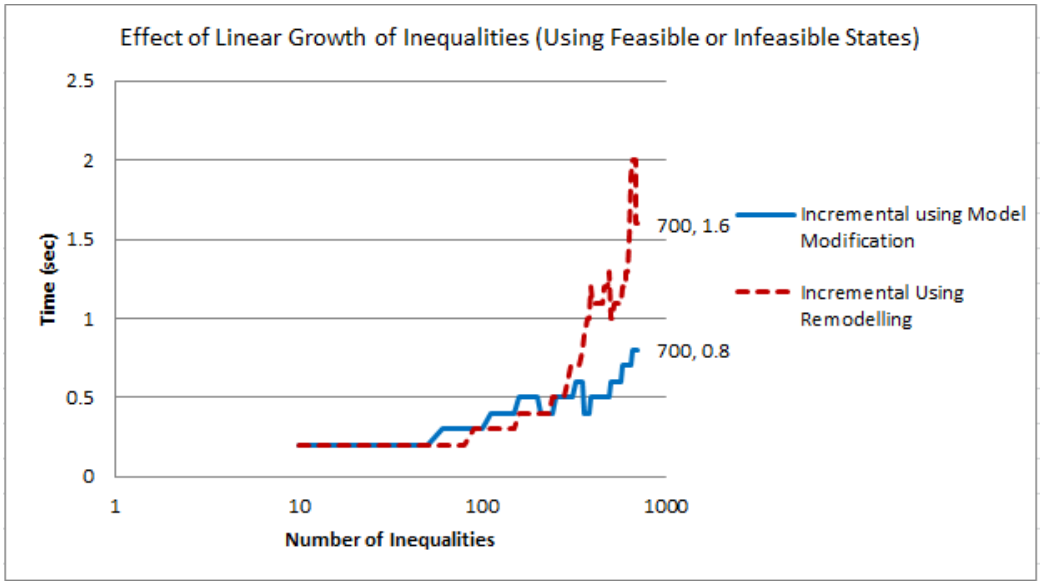

Figure 19: Behaviour of the prototypes: Linearly growth of Inequalities (Using Feasible or Infeasible States)

significant change as the system moves to infeasible state in both prototypes.

6.3.8 Test Case 8: Linearly Increasing the Number of Inequalities and Variables

This test case is implemented in our incremental prototypes and the number of variables and inequalities change in each iteration in a linear fashion with linear growth of $f(x) = 10x$ for the inequalities and linear growth of $f(y) = 5y$ for the variables. Feasibility of the system changes randomly. The results of this test case are presented in Tables 1 and 2. Based on these results, the increase in run time is much more tolerable compared to test cases in which the number of either variables or inequalities was constant.

Table 1: Effect of linear growth of inequalities and variables using Incremental Remodelling

Number of variables	Number of inequalities	Time(s)
5	10	0.2
10	20	0.2
20	30	0.3
30	40	0.3
40	50	0.3
50	60	0.2
60	70	0.3
70	80	0.3
80	90	0.4
90	100	0.4
100	110	0.5
110	120	0.5
120	130	0.6
130	140	0.7
140	150	0.7
150	160	0.8
160	170	0.8
170	180	0.9
180	190	1.0
190	200	1.1
200	210	1.2
210	220	1.3
220	230	1.4
230	240	1.5
240	250	1.6
250	260	1.7
260	270	1.8
270	280	1.9
280	290	2.0
290	300	2.3
300	310	2.3
400	410	3.7
500	510	6.3
600	610	9.5

Table 2: Effect of linear growth of inequalities and variables using Incremental Modification

Number of variables	Number of inequalities	Time(s)
5	10	0.2
10	20	0.2
20	30	0.2
30	40	0.2
40	50	0.2
50	60	0.2
60	70	0.2
70	80	0.2
80	90	0.3
90	100	0.3
100	110	0.3
110	120	0.3
120	130	0.3
130	140	0.3
140	150	0.3
150	160	0.4
160	170	0.4
170	180	0.4
180	190	0.4
190	200	0.4
200	210	0.4
210	220	0.4
220	230	0.5
230	240	0.5
240	250	0.5
250	260	0.5
260	270	0.5
270	280	0.5
280	290	0.6
290	300	0.6
300	310	0.6
400	410	0.8
500	510	1.1
600	610	1.3

6.4 Solving Time, Updating Time and Remodelling Time

One useful method to justify the changes observed in the results of our test cases in Section 6.3 is to monitor and analyse three important procedures which take place in each of the prototypes. These three procedures include solving the system of inequalities, updating this system (for instance removing the constraints imposed to the system) and also remodelling the system of inequalities to be solved. Observing the time spent to implement each of the mentioned procedures can provide us with a better vision in order to justify and analyse the changes to the time needed to process each system of inequality. In order to do so, a system of inequalities consisting of 100 inequalities and 5 variables was tested. Solving, updating, remodelling and total time for each run was recorded. Table 3 shows the test results for 5 sample runs. Based on the results, we conclude that remodelling is a slower process compared to model modification regardless of the feasibility state of the system.

Table 3: Performance of the system during Solving, Updating and Remodelling

	Solving Time(ms)	Updating Time(ms)	Remodelling Time(ms)	Total Time(ms)
1st Run	131	1	16	315
2nd Run	107	7	17	307
3rd Run	103	6	22	292
4th Run	94	3	18	290
5th Run	100	3	18	296

6.5 Testing The System With HARD-generated Inequalities

In this chapter we will present the results of testing our system on HARD-generated inequality sets. To perform these tests, HARD's performance was tested against some complex test cases that were presented in [8] and the inequality sets generated by HARD for these test cases were used as input sets for our system. For each test case we present the evaluation results for five runs in the following sections. In the tables, the terms *SAT* and *UNSAT* mean test cases with satisfiable and unsatisfiable ontologies.

6.5.1 Test Case 1: BackTracking (UNSAT)

This set of test cases points out the effect of backtracking in unsatisfiable cases. Table 4 shows the performance of our system for this test case.

Table 4: Performance of CPLEX-based solver for test case 1: BackTracking (UNSAT)

Test Case	Number of Variables	Number of Inequalities	Time(sec)
BT1-1	1	1	0.2
BT1-2	2	2	0.1
BT1-3	3	3	0.1
BT1-4	4	4	0.2
BT1-5	5	5	0.2
BT1-6	6	6	0.2
BT1-7	7	7	0.2
BT1-8	8	8	0.2
BT1-9	9	9	0.3
BT1-10	10	10	0.1
BT1-11	11	11	0.1
BT1-13	13	13	0.2
BT1-14	14	14	0.2
BT1-15	15	15	0.2

6.5.2 Test Case 2: C-lin-$\mathcal{ALCQ}$

This set of test cases shows the effect of increased numbers used in Qualified Cardinality Restrictions (QCRs) using the concept C and DL $\mathcal{ALCQ}$ (a DL language extending $\mathcal{ALC}$ with qualified number restrictions) when the numbers are increased linearly. Table 5 shows the performance of our system for this test case.

Table 5: Performance of CPLEX-based solver for test case 2: C-lin-$\mathcal{ALCQ}$

Test Case	Number of Variables	Number of Inequalities	Time(sec)
C-SAT-lin-ALCQ-1	3	4	0.1
C-SAT-lin-ALCQ-2	4	4	0.1
C-SAT-lin-ALCQ-3	4	4	0.1
C-SAT-lin-ALCQ-4	3	4	0.1
C-SAT-lin-ALCQ-5	3	4	0.1
C-SAT-lin-ALCQ-6	3	4	0.1
C-SAT-lin-ALCQ-7	3	4	0.1
C-SAT-lin-ALCQ-8	3	4	0.1
C-SAT-lin-ALCQ-9	3	4	0.1
C-SAT-lin-ALCQ-10	3	4	0.1
C-UnSAT-lin-ALCQ-1	3	4	0.1
C-UnSAT-lin-ALCQ-2	3	4	0.1
C-UnSAT-lin-ALCQ-3	3	4	0.1
C-UnSAT-lin-ALCQ-4	3	4	0.1
C-UnSAT-lin-ALCQ-5	3	4	0.1
C-UnSAT-lin-ALCQ-6	3	4	0.1
C-UnSAT-lin-ALCQ-7	3	4	0.1
C-UnSAT-lin-ALCQ-8	6	4	0.1
C-UnSAT-lin-ALCQ-9	6	4	0.1
C-UnSAT-lin-ALCQ-10	6	4	0.1

6.5.3 Test Case 3: C-exp-$\mathcal{ALCQ}$

This set of test cases shows the effect of increased numbers used in Qualified Cardinality Restrictions (QCRs) using the concept C and DL $\mathcal{ALCQ}$ when the numbers are increased exponentially. Table 6 shows the performance of our system for this test case.

Table 6: Performance of CPLEX-based solver for test case 3: C-exp-$\mathcal{ALCQ}$

Test Case	Number of Variables	Number of Inequalities	Time(sec)
C-SAT-exp-ALCQ-1	3	4	0.2
C-SAT-exp-ALCQ-2	3	4	0.1
C-SAT-exp-ALCQ-3	3	4	0.1
C-SAT-exp-ALCQ-4	3	4	0.1
C-SAT-exp-ALCQ-5	3	4	0.1
C-SAT-exp-ALCQ-6	3	4	0.1
C-UnSAT-exp-ALCQ-1	7	4	0.2
C-UnSAT-exp-ALCQ-2	7	4	0.2
C-UnSAT-exp-ALCQ-3	6	4	0.2
C-UnSAT-exp-ALCQ-4	7	4	0.2
C-UnSAT-exp-ALCQ-5	7	4	0.1
C-UnSAT-exp-ALCQ-6	7	4	0.2

6.5.4 Test Case 4: C-lin-$\mathcal{ALCHQ}$

This set of test cases shows the effect of increased numbers used in Qualified Cardinality Restrictions (QCRs) using the concept C and DL $\mathcal{ALCHQ}$ (a DL language extending $\mathcal{ALCQ}$ with role hierarchies) when the numbers are increased linearly. Table 7 shows the performance of our system for this test case.

Table 7: Performance of CPLEX-based solver for test case 4: C-lin-$\mathcal{ALCHQ}$

Test Case	Number of Variables	Number of Inequalities	Time(sec)
C-SAT-lin-ALCHQ-1	3	4	0.2
C-SAT-lin-ALCHQ-2	4	4	0.1
C-SAT-lin-ALCHQ-3	4	4	0.2
C-SAT-lin-ALCHQ-4	3	4	0.1
C-SAT-lin-ALCHQ-5	3	4	0.2
C-SAT-lin-ALCHQ-6	3	4	0.1
C-SAT-lin-ALCHQ-7	3	4	0.1
C-SAT-lin-ALCHQ-8	3	4	0.1
C-SAT-lin-ALCHQ-9	3	4	0.1
C-SAT-lin-ALCHQ-10	7	4	0.1
C-UnSAT-lin-ALCHQ-1	4	3	0.1
C-UnSAT-lin-ALCHQ-2	7	4	0.1
C-UnSAT-lin-ALCHQ-3	7	4	0.1
C-UnSAT-lin-ALCHQ-4	7	4	0.1
C-UnSAT-lin-ALCHQ-5	7	4	0.1
C-UnSAT-lin-ALCHQ-6	7	4	0.1
C-UnSAT-lin-ALCHQ-7	7	4	0.1
C-UnSAT-lin-ALCHQ-8	7	4	0.1
C-UnSAT-lin-ALCHQ-9	7	4	0.1
C-UnSAT-lin-ALCHQ-10	7	4	0.1

6.5.5 Test Case 5: C-exp-$\mathcal{ALCHQ}$

This set of test cases shows the effect of increased numbers used in Qualified Cardinality Restrictions (QCRs) using DL $\mathcal{ALCHQ}$ when the numbers are increased exponentially. Table 8 shows the performance of our system for this test case.

Table 8: Performance of CPLEX-based solver for test case 5: C-exp-$\mathcal{ALCHQ}$

Test Case	Number of Variables	Number of Inequalities	Time(sec)
C-SAT-exp-ALCHQ-1	3	4	0.2
C-SAT-exp-ALCHQ-2	3	4	0.2
C-SAT-exp-ALCHQ-3	3	4	0.2
C-SAT-exp-ALCHQ-4	3	4	0.2
C-SAT-exp-ALCHQ-5	3	4	0.2
C-SAT-exp-ALCHQ-6	3	4	0.2
C-UnSAT-exp-ALCHQ-1	7	4	0.2
C-UnSAT-exp-ALCHQ-2	7	4	0.2
C-UnSAT-exp-ALCHQ-3	7	4	0.3
C-UnSAT-exp-ALCHQ-4	7	4	0.2
C-UnSAT-exp-ALCHQ-5	7	4	0.2
C-UnSAT-exp-ALCHQ-6	7	4	0.3

6.5.6 Test Case 6: C-restr-num-$\mathcal{ALCHQ}$

This set of test cases shows the effect of increasing value of number restrictions using the concept C and DL $\mathcal{ALCHQ}$ when the numbers are increased linearly. Table 9 shows the performance of our system for this test case.

6.5.7 Conclusion

Based on the evaluation results illustrated in previous sections, we can conclude that our system's performance was trivial while using HARD-generated inequality sets.

Table 9: Performance of CPLEX-based solver for test case 6: C-restr-num-$\mathcal{ALCHQ}$

Test Case	Number of Variables	Number of Inequalities	Time(sec)	Comments
restr-num-1-1	4	4	0.1	
restr-num-1-2	2	3	0.1	
restr-num-1-3	2	3	0.1	
restr-num-1-4	3	6	0.1	
restr-num-1-5	7	10	0.1	
restr-num-1-6	2	7	0.1	Total value of number restrictions was initially 12 which led to HARD crashing. This was resolved by reducing the value of number restrictions to 7.
restr-num-1-7	2	9	0.1	Total value of number restrictions was initially 16 which led to HARD crashing. This was resolved by reducing the value of number restrictions to 9.
restr-num-1-8	2	10	0.1	Total value of number restrictions was initially 17 which led to HARD crashing. This was resolved by reducing the value of number restrictions to 10.
restr-num-1-9	2	11	0.1	Total value of number restrictions was initially 18 which led to HARD crashing. This was resolved by reducing the value of number restrictions to 11.

Chapter 7

7 Conclusion and Future Work

The presented CPLEX-based solver in this thesis can determine the feasibility of the system of inequalities generated by DL atomic decomposition and provide useful explanations regarding the sources of infeasibility. Utilizing ILOG CPLEX and applying its optimization features within a customized system, provides us with an efficient way to improve the performance of DL reasoners, especially when it comes to dealing with large values of number restrictions.

7.1 Conclusion

Our presented system enhances the performance of DL reasoners by speeding up one of the procedures which takes place during DL reasoning called atomic decomposition. Even for large values of number restrictions, our system is proven to be performing efficiently and handling complex scenarios with hundreds of inequalities or variables. The generated system could be easily integrated into any semantic DL-based reasoner which is implemented

in Java. The underlying functions and procedures of the system is consistent with the incremental nature of DL reasoners, which enables the system to immediately respond to any changes which are made to the number of variables or inequalities by the reasoning process. The system is able to respond to infeasible cases which are detected by CPLEX. As the system of inequalities becomes infeasible in any stage of solving process, the system provides a minimal explanation describing the cause of infeasibility. This feedback helps the DL reasoner to perform backtracking faster since it can ignore that paths which lead again to infeasible states. The system was proven to perform better when it was implemented incrementally. In an incremental implementation, modifying the problem model is much more efficient than re-creating the model. As there is no other system available right now which is developed and employed for the same purpose as our system, no comparison between our system and a similar existing ones was performed. However, the system was tested while using the input of one of current DL reasoners called HARD and the results of these tests were satisfactory as predicted.

7.2 Limitations

The limitations of our system could be summarized in the following points:

• Our system is designed in such way which can only target and optimize the performance of semantic DL reasoners that apply atomic decomposition technique in their reasoning process.

• Our system is implemented in Java, therefore it can only be applied in collaboration with semantic reasoners which are implemented in Java as well.

• Due to the absence of CPLEX implementation details, a customized utilization of CPLEX services is rather challenging.

• Testing the system is only possible in cases where the reasoner provides the input in form of a set of inequalities.

• The backtracking and clash handling procedure is mostly based on comparison method between different states of the system which can turn to be time consuming if the difference between the states is not minor in terms of inequality and variable quantity.

7.3 Future Work

Possible areas to extend and optimize our system can be focused on eliminating any of the system limitations mentioned in Section 7.2. The system

needs to be enhanced in such way that it can be embedded in other reasoners regardless of implementation language boundaries. Moreover, providing the ability for the system to use other state-of-the-art solvers adds to the flexibility of the system. Using solvers whose structural details are available can enhance the features of the system and provide more functionality. The backtracking and clash handling procedure provided in this thesis can be enhanced in terms of accuracy and efficiency. An optimized version of our clash handling module must not merely depend on the comparison method and it may use some smarter algorithm and also suggest corrective measures.

References

[1] Baader, F., Buchheit, M., Hollander, B.: Cardinality restrictions on concepts. Artificial Intelligence 88, 195–213 (December 1996)

[2] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, 2^{nd} edn. (2007)

[3] Baader, F., Hollander, B.: A terminological knowledge representation system with complete inference algorithms processing declarative knowledge. Lecture Notes in Computer Science 567, 67–86 (December 1991)

[4] Badros, G.J., Borning, A.: The cassowary linear arithmetic constraint solving algorithm pp. 67–92 (2002)

[5] Boyd, S.P., Vandenberghe, L.: Convex Optimization. Cambridge University Press, 1^{st} edn. (2004)

[6] Brachman, R., Schmolze, J.: An overview of the KL-one knowledge representation system pp. 171–216 (1985)

[7] Corporation, I.B.M.: IBM ILOG OPL language user's manual. Lecture Notes in Computer Science (V6.3), 29–51 (2009)

[8] Faddoul, J.: Reasoning algebraically with description logics. PhD thesis, Department of Computer Science and Software Engineering, Concordia University (September 2011)

[9] Faddoul, J., Haarslev, V.: Algebraic tableau reasoning for the description logic $\mathcal{SHOQ}$. Journal of Applied Logic (Special issue on Hybrid Logic) p. 8(4):334–355 (December 2010)

[10] Farsiniamarj, N., Haarslev, V.: Combining integer programming and tableau-based reasoning: A hybrid calculus for the description logic $\mathcal{SHQ}$. AI Communication 134, 5–22 (November 2008)

[11] Hillier, Frederick S.; Lieberman, G.J.: Hillier and Lieberman. Introduction to Operations Research. McGraw Hill, Boston, MA, 7^{th} edn. (2001)

[12] Hitzler, P., Krotzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Taylor and Francis, 1^{st} edn. (August 2009)

[13] IBM: ILOG CPLEX 10.0 user's manual pp. 67–92 (January 2006)

[14] Jensen, P.A., Bard, J.F.: Operations Research Models and Methods. Wiley (October 2002)

[15] Levesque, H.J., Brachman, R.J.: Expressiveness and tractability in knowledge representation and reasoning. Computational Intelligence pp. 78–93 (1987)

[16] Ohlbach, H.J., Koehler, J.: Reasoning about sets via atomic decomposition. Technical report TR-96-031 (August 1996)

[17] Ohlbach, H.J., Koehler, J.: Modal logics, description logics and arithmetic reasoning. Artificial Intelligence 109, 1–31 (June 1999)

[18] Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Dover Publications, unabridged edn. (1998)

[19] Schmidt-Schauls, M., Smolka, G.: Attributive concept descriptions with complements. Artificial Intelligence 109 (June 1999)

[20] Simančík, F., Motik, B., Horrocks, I.: Consequence-based and fixed-parameter tractable reasoning in description logics. Artificial Intelligence 209, 29–77 (April 2014)

[21] Spielman, D.A., Teng, S.H.: Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time (February 2008)

[22] Team, L.D.: Lpsolve reference guide (November 2008), http://lpsolve.sourceforge.net/5.5/

[23] Vanderbei, R.J.: Linear Programming, vol. 196. Springer, 1^{st} edn. (August 2014)