Modelling Arguments in the Dictator Game

Ricardo L. Parreira Duarte

BSc Dissertation
Department of Computer Science,
University of Liverpool, UK.

May 2009

Table of Contents

Abstract

In this project we use an approach to modelling reasoning in a simple scenario from experimental economics, the Dictator game, using preferences over social values to provide transparent justification of actions.

This approach does not require estimation of utilities and weights for different factors, instead it will consider how argumentation based on preferences relating to personal values of the subjects can affect the choices made by the dictator, these choices allow a different perspective and they can be used to analyse the game.

Using this model we can explain the behaviour of subjects in such experiments, and in particular, gain insight into the framing effect observed by some experimenters.

The Dictator game is a very simple game, or, more an economical experiment. There are two players involved: First is the dictator, who chooses how small sum of money is divided, and the other player just receives what the dictator has left. This player does not have choice but to accept whatever it was given by the dictator.

If only the economic well being of the Dictator was considered, the Dictator would have kept all the money, and there would be no reason for this experiment to exist in economics. But according to some experiments the Dictator often gives money away; [2], [1] this suggests that around 70% of dictators give a non-zero sum away.

This experiment can convince us that the majority of people do not act according to with their economic self-interest, in this way it is possible to believe that other reasons might affect the dictator choices, such as fairness, where the dictator does not appear selfish (i.e, they might care about what the person conducting the experiment thinks of them), etc.

Introduction

This project represents an approach to modelling reasoning in a simple scenario from experimental economics, the Dictator game. This approach pretends to justify choices using preferences over social values to provide transparent justification of actions, and simulates that different factors have to be taken into consideration, modelling its arguments and affecting its choices in what to do.

Aim and Objectives:

To create a simulator that using Action-based Alternating Transition System (AATS) [4] used as the underlying model for representing a valued as an approach to modelling arguments about action, it will generate all the arguments and the objections for each of one of the possible actions, these actions will reflect the dictator choice on the amount of money that it will give away.

Once all the arguments and objections are generated they will be used in a Valued-based Argumentation Framework (VAF), and the software should produce a preferred extension based on the value preference ordering of the Dictator, this allow to know which is the states that refer better to the dictator according with his preferences.

Challenges:

This project incurs into several challenges:

– The framework (AATS) in which the part of the software is based it hasn't never been produced, and there is no assurance that this framework is also mature enough to be implemented, consequently there is no other way to compare the results produced besides the research papers in which the project is based of.

– A part of this simulator also incurs on another framework (VAF) in which there is no knowledge of previous successful implementations for general cases.

– It was chosen to implement the simulator using ANSI C++ and QT 4.4.3, which I do not possess any previous knowledge about it.

– To implement this project it is necessary to acquire an excellent knowledge around Graph Theory to allow the creation of algorithms capable of generating the frameworks present in this project.

Solution Produced:

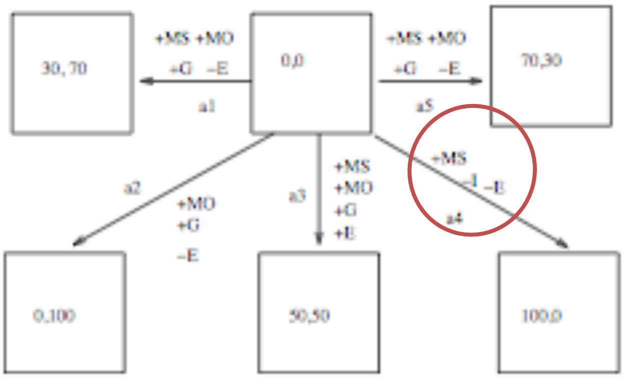
The software produced is capable of generate all arguments and objections present in a AATS diagram, and also capable of generate preferred extensions for VAF's graph with no cycles, and also VAF's with polychromatic and dichromatic cycles.

The software displays the arguments and objections in a list format ordered by joint action and in a table where it shows all the arguments and objections for each joint action.

It also produces statistics about the number of cycles, the total number of arguments and it displays how it the cycle detection works showing the DFS of the arguments graph.

Software Effectiveness:

The software produced can, generate all the arguments and objections that are present in the original research paper [1], and it managed to produce some other objections that were missed by the authors such a CQ9 in the joint action at a4 [1] (fig. 1) since there is two values demoted, different CQ11 where the I (image) is precluded at every action besides a4 (fig. 1).



It does not produce different objections with the same meaning as in [1] (per example obj1.6 and obj5.6 since these have the same meaning which would create unnecessary monochromatic loop that are currently unsolved by the software).

The software was built made taking in consideration correctness trading space for speed. Besides the fact that can solve more complex programs in seconds, it is necessary to consider that the program might run out of memory for extreme big problems.

Consequently it would allow the study of the different values present in the dictator choices and how influential they are in his actions through the experiment, allowing the prediction of different behaviour's.

The readability and the lack of the program at a first sight is also an issue since the program does not show the VAF graph neither the progress in a visual way, leaving to the user a text version of the progress and the solutions found. But it also informs the clearly the actions that the dictator would consider.

The program is also capable of generating arguments in the AATS with a time complexity of $O(n^4)$ for diagrams with just one active agent and for n active agents within other diagrams it would generate with $O(n^5)$.

The table of arguments in [1] is generated with a time complexity of O(n).

Prefered extensions for VAF graphs with no cycles can be generated with $O((n+m)^2)$. Since it takes O(n+m) to create a Depth first search tree which the algorithm runs through and it will take O(n+m) to go through the tree where m is the number of times the algorithm went back in the tree to correct the result.

The cycle deletion occurs with a time complexity of O(n) for each cycle detected, for both polychromatic and dichromatic cycles.

Description of anticipated components:

Background Research:

In this section it will be given a description of all the frameworks used in the project necessary for the creation of the simulator such as rules and definitions that will be applied and also some background information about the graph theory behind the project described all the background information.

Those frameworks are the Action-based Alternating Transition System (AATS), Value-based Argumentation Framework (VAF) and the graph theory algorithms. Not only these frameworks and algorithms will be described but it will be also explained why they were used (in the case of the graph algorithms), and how these definitions were applied.

Action-based Alternating Transition System (AATS):

According to [1] AATSs are used for modelling systems comprising multiple agents that can perform actions in order to modify and attempt to control the system in some way. It can be seen as a formal way to describe practical reasoning, about what is best for a particular agent in a given situation, based on pre-emptive justifications of actions through the, instantiation of an argument scheme, as defined in [4] represents a "stereotypical patterns of reasoning whereby the scheme contains premises that presumptively justify a conclusion. Each scheme has associated with it a set of characteristic critical questions that can be used to challenge the presumption justified by instantiated schemes and so identify counter arguments. The claim presumptively justified by the instantiated scheme is acceptable only so long as it can withstand the critical questioning." This scheme, called AS1, can be described as:

AS1 In the current circumstances R
We should perform action A
Which will result in new circumstances S
Which will realise goal G
Which will promote value V

Justifications of actions are subject to examination through a serial of critical questions (CQ) [4].

Critical questions will identify the ways in which the presumption may be challenged, and arguments grounding negative answers, used then as attacks to the original argument. For this particular experiment sixteen critical questions will be identified (CQ1 to CQ16), since there are no attacks from different agents (the Dictator chooses the amount to be shared and the other has not got any other option but to accept the money, so only the dictator behaviour will be analyzed).

The critical questions are divided in three sections:

Problem formulation: deciding what the propositions and values are relevant to the particular situation, and constructing the AATS. Critical questions from: CQ2, CQ3, CQ4, CQ12, CQ13,CQ14,CQ15 to CQ16.

Epistemic reasoning: determining the initial state in the structure formed in the previous stages, which is described at CQ1.

Choice of action: developing the appropriate arguments and counter arguments (objections), in terms of applications of the argument scheme and critical questions, and determining the status of the arguments with respect to other arguments and the value orderings, described using CQ5, CQ6, CQ7, CQ8, CQ9, CQ10 and CQ11.

AATS can be formally described in a (n+7) tuple S={$Q, q_0, Ag, Ac_1,\ldots, Ac_n, \rho, \tau, \phi, \pi$}, where:

- Q is a finite, non-empty set of *states*;
- q0 ∈ Q is the *initial state*;
- $Ag = \{1, \ldots, n\}$ is a finite, non-empty set of *agents*;
- Ac_i is a finite, non-empty set of actions, for each $i \in Ag$ where $Aci \cap Acj = \emptyset$ for all $i \neq j \in Ag$;
- $\rho : Ac_{Ag} \rightarrow 2^Q$ is an *action pre-condition function*, which for each action $\alpha \in Ac_{Ag}$ defines the set of states $\rho(\alpha)$ from which α may be executed;
- $\tau : Q \times J_{Ag} \rightarrow Q$ is a partial *system transition function*, which defines the state τ (q , j) that would result by the performance of j from state q —note that, as this function is partial, not all joint actions are possible in all states (cf. the pre-condition function above);
- Φ is a finite, non-empty set of *atomic propositions*; and
- $\pi : Q \rightarrow 2^\Phi$ is an interpretation function, which gives the set of primitive propositions satisfied in each state: if $p \in \pi(q)$, then this means that the propositional variable p is satisfied (equivalently, true) in state q .

According to [1] we will assume, perfect information from the dictator to the model and current state, this will allow the first two stages (problem formulation and epistemic reasoning) will be straightforward and uncontroversial. The aspect of interest here will the last stage the Choice of action, where the arguments and counter arguments will be put forward, based on the values that are promoted or demoted by the particular transitions detailed in each proposed action [1].

Values according to [1] are differentiated from goals, and defined as “some actual descriptive social attitude or interest, which an agent may or may not wish to uphold, promote or subscribe to.

Consequently the only set of critical questions that can provide interest to us are just the ones belonging to the choice of the action stage, which are studied with more detail, since the user will give apriori all the details necessary to build the first two stages, so the dictator will have a perfect view to the model and states.

We can describe the relevant set of CQ’s [4] through the following table;

	Description	Looks for
CQ5	Are there alternatives ways of realizing the same consequences?	Different Joint actions leading to same value.
CQ6	Are there alternatives ways of realizing the same goal?	Looks for different propositions that would reach the same goal
CQ7	Are there alternatives ways of promoting the same value?	Looks for different joint actions that promotes the same value
CQ8	Does doing the action have a side effect, which demotes the value?	Looks for a demoted value within an action.
CQ9	Does doing the action have a side effect, which demotes some other value?	Looks for other values demoted within an action.
CQ10	Does doing the action promote some other value?	Looks for different promoted values within an action
CQ11	Does doing the action preclude some other action, which would promote some other value?	Looks for promoted values within different actions that would cause the preclusion of others.

For a complete description of the CQs model and how they are used please refer to [4] or Appendix C.

(VAFs);

An argumentation framework is a pair:

$$AF = \langle AR, attacks \rangle$$

Where AR is a finite set of arguments and attacks is binary relation on AR.

AF can conveniently be modelled as directed graphs, with arguments as nodes and attacks as arcs (directed edges) showing which arguments attack one another. The notion of an argument is purely abstract, where no concern is given to the internal structure of arguments. Thus, the status of an argument can be determined by considering whether or not there is a set of arguments, which can defend itself from attacks by other arguments in the AF [3].

It is possible to observe that AFs do not offer any practical way to handle values, so its not possible to reach conclusions on argumentations where they come into it. Besides that, determining whether a dispute is resoluble is not in general a tractable problem, in fact when exists a plurality of preferred extensions it derives from the presence of cycles in the graph. When that happens in finite argument framework without self-attacks, there must be a simple cycle of even length.

This project requires the use of framework capable of deal with practical reasoning, capable of produce arguments in the AS1 form as described before:

To accomplish that VAF's will be used, why? In many cases of disagreement, particularly in situations involving practical reasoning, it is impossible to demonstrate conclusively that either party is wrong. In such cases the role of argument in such cases is to persuade rather than to prove demonstrate or refute. Argumentation Frameworks (AFs), have been a successful way of looking at systems of conflicting arguments, but fails to describe some practical reasoning, such debates where points of are defensible, and do not provide at all times a rational basis for preferring one argument over another [3].

VAF extend AFs by associating arguments with values that are promoted through acceptance of the argument, thus accommodating different audiences with different interests, VAFs define a notion of defeat of arguments different to that of an AF. In a VAF, an attack is distinguished from defeat for an audience whereby strengths of arguments for a particular audience are compared with reference to the values to which the arguments relate. Allowing a particular audience to choose to reject an attack, even if the attacking argument cannot itself be defeated, by preferring the value the argument promotes to that of its attacker.

VAF is capable of dealing with the type of argument described, to be able for us to record such a and offer a 5-tuple relation [3,11,12];

$$VAF = \langle AR, attacks, V, val, P \rangle$$

Where:

- *AR*, is a finite set of arguments;
- *Attacks,* is an irreflexive binary relation on AR;
- *V* is a nonempty set of values;
- *val* is a function which maps from elements of AR to elements of *V*
- *P* is the set of possible audiences.

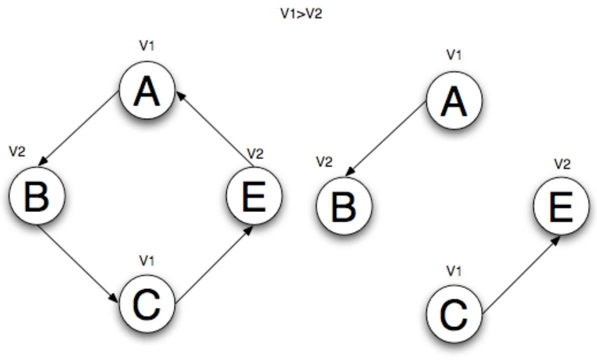
Within a VAF it is possible to distinguish different types: A VAF graph is said to be Dichromatic when it value set contains exactly two values. Polychromatic when it contains more than two values, and monochromatic when it only possesses one value.

VAF cycles:

Cycles in a VAF deserve a special attention since they have to be deleted before any preferred extension is created. Within cycles we can distinguish three types:

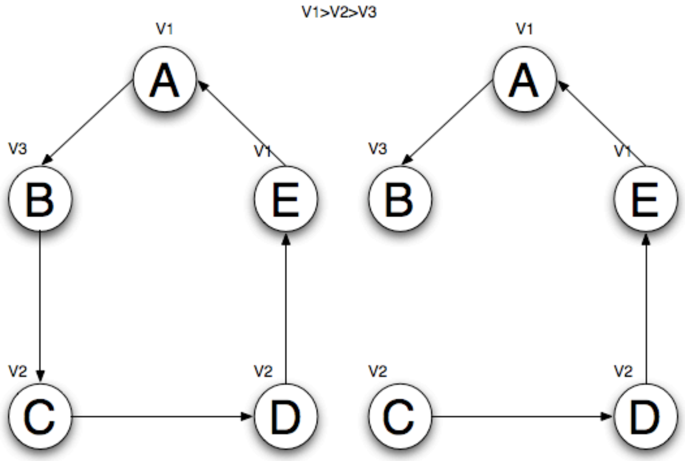
– Monochromatic cycles: of odd length (odd-cycles) will have the empty set as their preferred extension, cycles of even length will two preferred extensions as in AF. However we should feel somehow uncomfortable with the existence of odd monochromatic cycles in the graph, since they will have the nature of a paradox in which we cannot accept any argument. With the existence of even cycles they represent a dilemma in which we have a choice between two actions with no rational grounds[12], these even cycles can be broken and solved separately as if we had two different VAF graph, each one with its unique preferred extension.

– Dichromatic cycles (fig. 2a): will have a unique preferred extension since the cycle will be broken at each one of the attacking arguments with the most inferior value (fig. 2b).



(fig. 2a) (fig. 2b)

– Polychromatic cycles (fig.2a): can be easily solved since they will have a unique preferred extension, given some value ordering in a graph, it known that the cycle will be always broken in which the attacking argument will have the most inferior value(fig. 2b) [11].



(fig. 3a) (fig. 3b)

Further description can be found at [3,11,12].

Graph Algorithms:

To create this simulator it is necessary to fully understand the following Graph algorithms representations:

Adjacency list Graph representation:

The directed graph representation to be used to recreate the AATS framework and the VAF will be represented through an adjacency list, this will contain a list of nodes, where each node will contain a list with all the edges in which it directs to (fig.4).

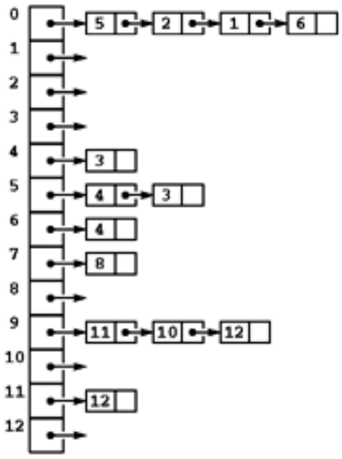


Fig.4 – graphical representation of adjacency list.

This type of representation ensures a O(V+E) for the used space for the graph with insertion of edge as O(1) [10].

Transpose Graph:

If graph G = (V, E) is a directed graph, its transpose, $G^T = (V, E^T)$ is the same as graph G with all arrows reversed. We define the transpose of a adjacency matrix $A = (a_{ij})$ to be the adjacency matrix $A^T = ({}^Ta_{ij})$ given by ${}^Ta_{ij} = a_{ji}$. In other words, rows of matrix A become columns of matrix A^T and columns of matrix A becomes rows of matrix A^T [13].

Formally, the transpose of a directed graph G = (V, E) is the graph G^T (V, E^T), where E^T = {(u, *v*) Î V×V: (u, *v*) ÎE. Thus, G^T is G with all its edges reversed [14].

We can compute G^T from G in the adjacency matrix representations and adjacency list representations of graph G.

For this project it was discovered that it is possible to find start of a chain in a VAF using the transpose graph, when reversing the arrows of each edge on a directed graph the edge with no outgoing edges in the reverse will be the edge in the original graph without a edge directing to itself, what in the VAF will be the edge with that does not have any attacks:

Lets consider the graph chain in (fig. 5) and it’s adjacency list (fig. 6).

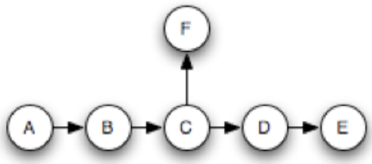


Fig. 5

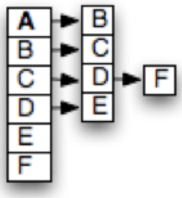


Fig. 6

Now consider its reverse graph (fig. 7) and its adjacency list representation (fig. 8), now it is possible to observe that A does not have any node in it's adjacency list representation (fig. 8), being the only node with no edges directed to itself in the original graph (fig. 5).

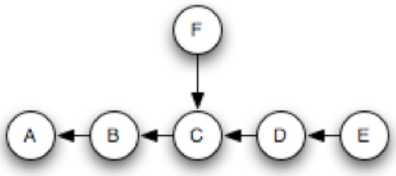


Fig. 7

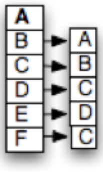


Fig. 8

Depth First Search (DFS) in Directed Graphs [10]:

DFS in directed graphs allows them to be represented in a tree, making possible to go through chains faster and easier, DFS also allows detection of strong connected components within a graph.
To visit a node in a DFS we mark that component as visited, then we go from that component to the one connected to itself, and so on. If we visit a component already marked as visited we found a back edge, what will represent a cycles in the graph. A DFS will always visit each element using a preorder selection and then go as deep as possible in that path, from left to right in the tree representing the graph.

DFS will have a very important role in the software since will allow to solve more efficiently VAF chains since all the arguments will be reallocated in a tree using preorder, also will allow cycle detection in the graph marking all the back edges in the DFS tree.

Dictator game Background Information:

This experiment use the frameworks described above (AATS's and VAF's respectively), in order to explain the different subjects behaviours.
We will focus mainly in checking the value ordering for maximiser (which acts accordingly with his self economical interest), or satisfier agents (tries to satisfy the other economical interest).

The experiment will consider five values:
– Money: The value of money is distinguished as the dictator's money (MS), or the others money (MO)
– Giving (G): According to [1] "It can be held that giving a gift is a source of pleasure, and this is what motivates the dictator to share"
– Image (I): According to [1] "Some have argued that there is a desire not to appear mean before the experimenter that motivates sharing. It could even be that one does not want to appear mean to oneself".
– Equality (E): As defined by an equal distribution, characterizes a sense of fairness.

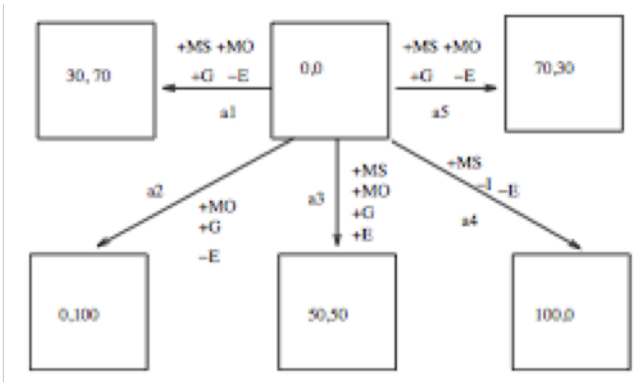


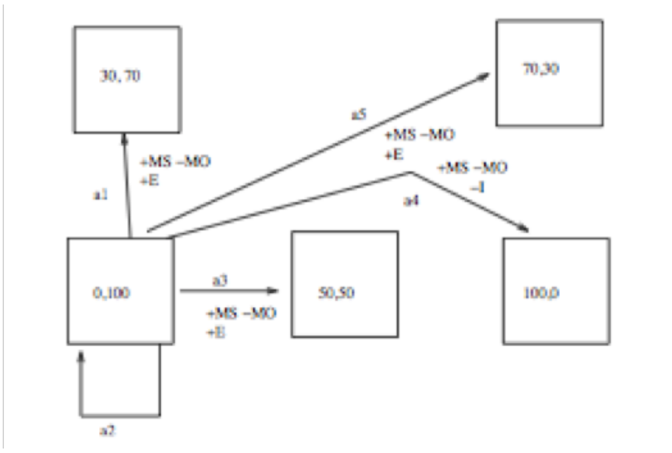
Fig.9 Dictator Game

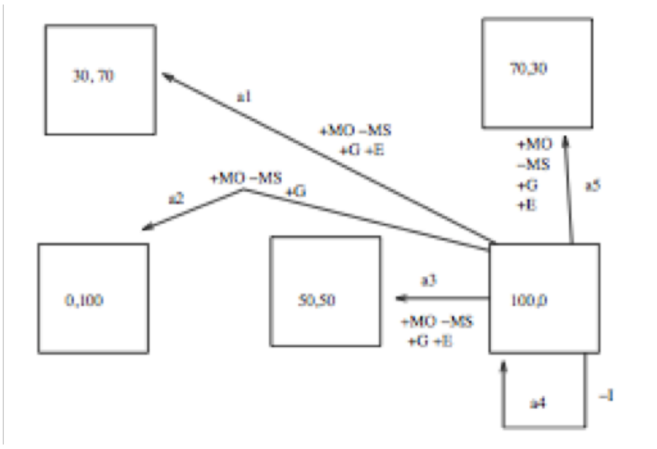
As it is possible to observe from the fig. 9 and [1] the amount of values being used will never be more than 5 values in the dictator game.
The graph created through the VAF process, will be of a considerable size such as most of the economical experiments and it will be have around 40 to 50 arguments (edge number in the graph) and objections included, depending on approach used in the model.

According to [1] it is necessary to explore the different ways in which the problem is framed to the subjects, Bradley writes:

“ Experimental dictator games have been used to explore unselfish behaviour. Evidence is presented here, however, that allowing them to take a partner’s money can reverse subjects’ generosity. Dictator game giving therefore does not reveal concern for consequences to others existing independently of the environment, as posited in rational choice theory. It may instead be an artefact of experimentation.”

In [1] to framing effect to have an impact, the choice of action cannot be determined by the expected utility of the target state, since the utilities of the states are unchanged and the effects of action certain. This lead to the change of the initial state from (0,0) where the dictator and the other player start with nothing, to states where one starts with money (100,0) fig. 10 or (0,100) fig. 11.





ıg
e

[1],
ı

Fig.10 Taking Game

Fig. 11 Dictator Game with start state at 100,0

Project Requirements:

The requirements established are:

- The software will have to able to read the AATS diagrams at [1];
- It will have to recreate all the arguments and objections in [1] according with the critical questions in [4], validating this way the content in [1] or not;
- It will have to display the arguments and objections found in a clear layout, similar to the one found in [1], providing a complete of the arguments and objections found, and table layout organized by action per argument/objection.
- It will have to evaluate all the arguments and objections found, creating this way a VAF, providing a preferred extension and the preferred actions found.
- It should provide a way to solve VAF either with cycles (dichromatic or polychromatic) or without any (arguments chains).
- It is necessary to provide a graphical view of the VAF graph and a way to change the value ordering for each agent.

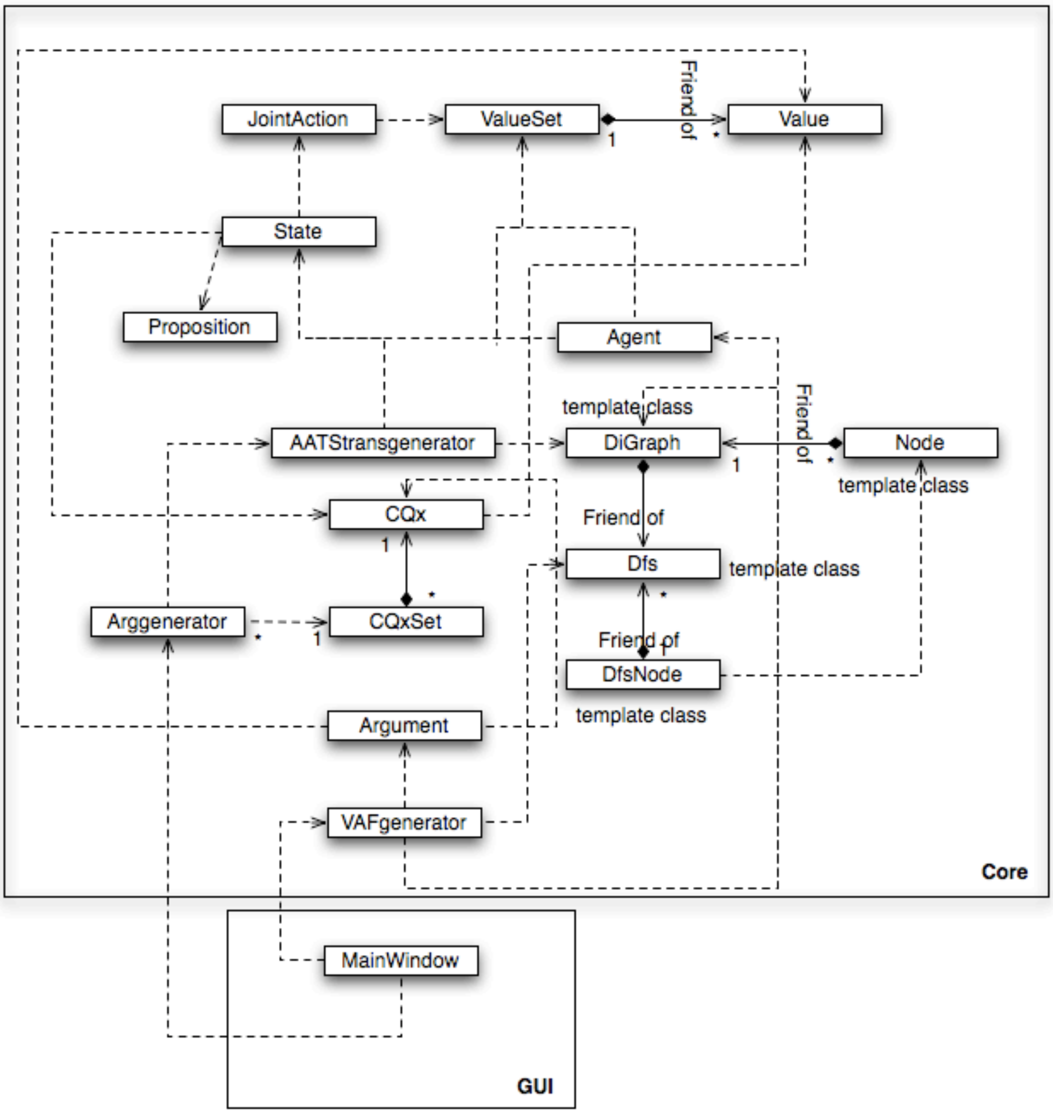
Existing solutions/approaches:

There are no known existing approaches to this framework, and consequently there is no possible way to compare the solutions created. Besides this it will be essential that software will provide as much information as possible at this stage offering as much as possible a way to

Design

Data Structures to be used:

The program will be implemented using an OOP approach, dividing its functionalities by class and methods, creating a modular approach at the same time, reducing the error probability and allowing early checking. The simulator class list is presented in the following diagram;



NOTE: This diagram has suffered a few changes, some because of limitations on the C++ object oriented model, while most of the design would suit a OOP language such as JAVA, others changes were done in order to correct design issues at the design stage of the project.

The changes were:

- Node class: this class is no longer described as vertex class but as node, all the functionality and implementation are the same as the ones meant originally for the class vertex, with some minor changes described below.

- DiGraph and Node classes: are no longer extended by other classes, instead these classes are templates that can adopt any type, and can used to describe the AATS and the VAF graphs.

- ValueSet class: were created allowing the implementation of set of values to be used with agents or in the joint-actions.

- CQxSet class: as a similar functionality as ValueSet allowing to define sets of CQ's when looking for arguments and objections.

- GUI classes: most of the gui classes were chosen to not be implemented since the software will no longer have a way to display visually the graphs and the AATS as defined in the design stage, since it would be unrealistic to have everything completed by the deadline. Leaving the MainWindow class to display all the functionality.

Further description will be displayed below:

Classes' description:

– Node: Template subclass of DiGraph, this class will have all the nodes in which it directs at, this class is capable of use any type for node.

– Digraph: Template class that uses a Adjacency List to represent all the nodes and edges of a graph, this graph contains a list of nodes, and each node will contain a list of edges that a directed from each one of those nodes. This class will be used to represent the graphs for the AATS and for the VAF

– Dfs: Another template class, this class is a subclass (friend class) of DiGraph that will create a depth first search of a directed graph, keeping it in a list, it also keeps the index of each cycle found in the list containing the Dfs, and also the pre-order in which all the nodes were found and the post-order of the edges found in the graph.

– DfsNode: Subclass of the class Dfs, this class contains, an instance of node corresponding to the edge in the graph, the depth in the tree and the edge type.

– Agent: this object will have a value container with a preference order, which will act as the dictator or the other (in games such the ultimatum game).
It will have to analyze its preferences as well, for example if it is a maximiser it will have to check which state will be better to itself, or if it is a satisfier, etc.

– Value: Will contain the description of what is a value, as well if it is promoted (will correspond to value=1 in valuation variable) demoted (value=-1) or stays equal (value=0).
– ValueSet: This class will contain a map that will sort each value per its priority in the set, from the smallest one to the highest one rated value. The set will accept values between 0 and 100, for value sets that do not have a rank it will be given a priority -1 (per example the value set in each joint action)

– JointAction: Will contain all the values promoted, demoted or that will stay equal through each action within the transiction diagram, besides a description.

– State: This class will be used as a type in DiGraph class, where each state will correspond to a node in the graph.
Each state will a joint action or not, it can be defined as a normal state, or as a initial state or goal state.

– Proposition: Class created for ilustrative purpose only, this class can receive any container of ints type that will describe a proposition within a state.

– AATStransgenerator: will generate the transition diagram that describes the AATS problem, such values, joint actions and states as in [4] and [1].

– CQx: this class will offer a way to describe any CQ (from CQ5 to CQ11 as in [4]) it will be used when looking for the arguments and objections in AATS diagram. Storing between one and two states, depending on the CQ in question [4] and storing a description of the CQ found, and its name.

– CQxSet: Offers similar functionality as ValueSet, where it allows to store a set of different CQ's, this will insert the arguments and objections separating them by joint action.

– Arggenerator: object that will generate arguments and objections from CQ5 until CQ11, and will be able to output them in a CQxSet, or in string list with the description, and also will generate the arguments directed graph to be used in the VAF to generate a preferred extension.

– Argument: used as type in the DiGraph and will contain with similar functionality as State class. This class will define each argument in the VAF directed graph.

– VAFgenerator: Responsible for solving VAF's for a specified agent, based on its value ordering, it will solve cycled VAF with polychromatic and dichromatic cycles, and will create the preferred extension.

– MainWindow: will contain the program window where all information will be displayed.

For a specific description of each class and each method can be found at appendix B.

Classes’ and Methods Diagram:

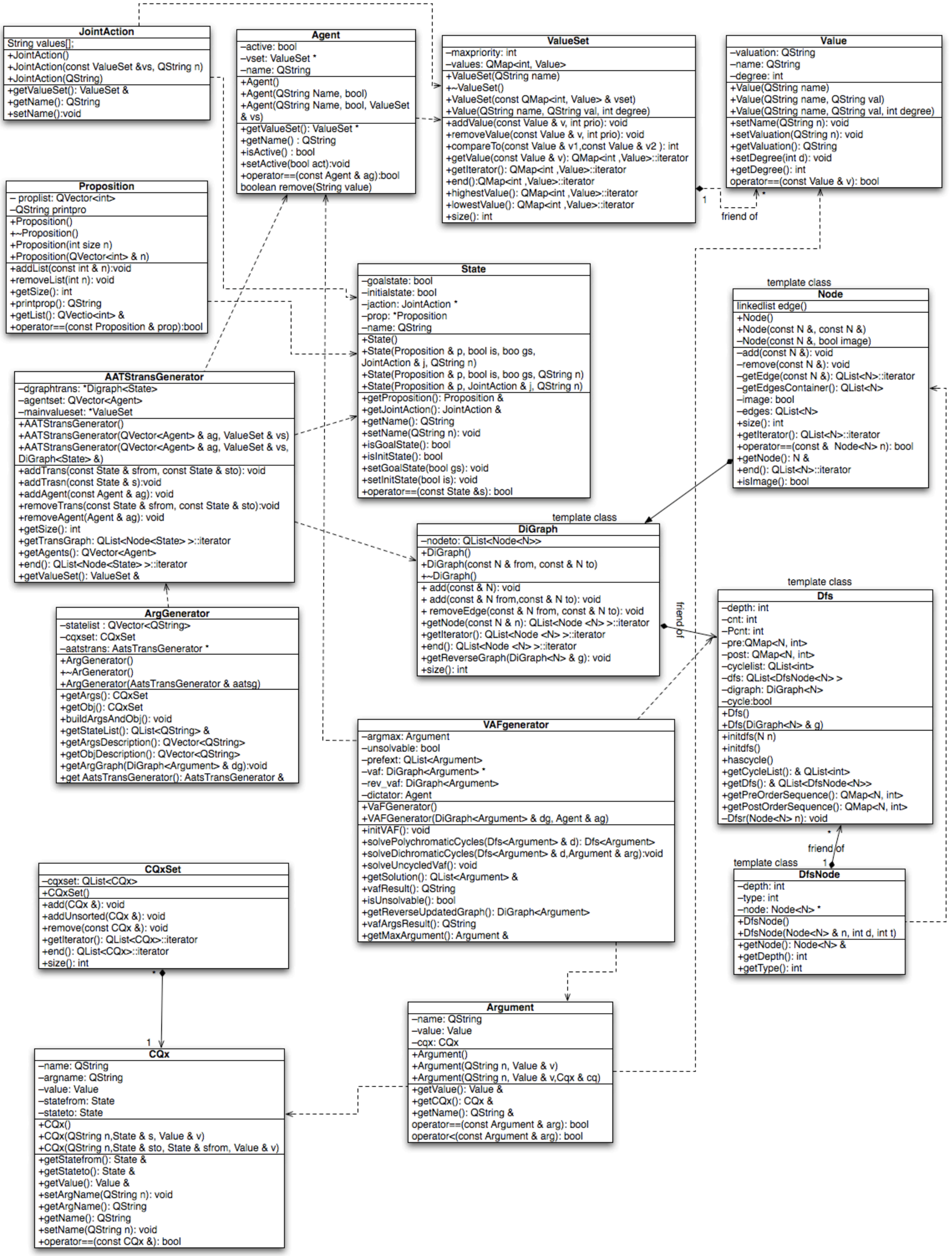

JointAction
String values[];
+JointAction()
+JointAction(const ValueSet &vs, QString n)
+JointAction(QString)
+getValueSet(): ValueSet &
+getName(): QString
+setName():void
Agent
–active: bool
–vset: ValueSet *
–name: QString
+Agent()
+Agent(QString Name, bool)
+Agent(QString Name, bool, ValueSet & vs)
+getValueSet(): ValueSet *
+getName() : QString
+isActive() : bool
+setActive(bool act):void
+operator==(const Agent & ag):bool
boolean remove(String value)
ValueSet
–maxpriority: int
–values: QMap<int, Value>
+ValueSet(QString name)
+~ValueSet()
+ValueSet(const QMap<int, Value> & vset)
+Value(QString name, QString val, int degree)
+addValue(const Value & v, int prio): void
+removeValue(const Value & v, int prio): void
+compareTo(const Value & v1,const Value & v2): int
+getValue(const Value & v): QMap<int ,Value>::iterator
+getIterator(): QMap<int ,Value>::iterator
+end():QMap<int ,Value>::iterator
+highestValue(): QMap<int ,Value>::iterator
+lowestValue(): QMap<int ,Value>::iterator
+size(): int
Value
–valuation: QString
–name: QString
–degree: int
+Value(QString name)
+Value(QString name, QString val)
+Value(QString name, QString val, int degree)
+setName(QString n): void
+setValuation(QString n): void
+getValuation(): QString
+setDegree(int d): void
+getDegree(): int
operator==(const Value & v): bool
1
*
friend of
Proposition
– proplist: QVector<int>
–QString printpro
+Proposition()
+~Proposition()
+Proposition(int size n)
+Proposition(QVector<int> & n)
+addList(const int & n):void
+removeList(int n): void
+getSize(): int
+printprop(): QString
+getList(): QVectio<int> &
+operator==(const Proposition & prop):bool
State
–goalstate: bool
–initialstate: bool
–jaction: JointAction *
–prop: *Proposition
–name: QString
+State()
+State(Proposition & p, bool is, boo gs, JointAction & j, QString n)
+State(Proposition & p, bool is, boo gs, QString n)
+State(Proposition & p, JointAction & j, QString n)
+getProposition(): Proposition &
+getJointAction(): JointAction &
+getName(): QString
+setName(QString n): void
+isGoalState(): bool
+isInitState(): bool
+setGoalState(bool gs): void
+setInitState(bool is): void
+operator==(const State &s): bool
template class
Node
linkedlist edge()
+Node()
+Node(const N &, const N &)
–Node(const N &, bool image)
–add(const N &): void
–remove(const N &): void
–getEdge(const N &): QList<N>::iterator
–getEdgesContainer(): QList<N>
–image: bool
–edges: QList<N>
+size(): int
+getIterator(): QList<N>::iterator
+operator==(const & Node<N> n): bool
+getNode(): N &
+end(): QList<N>::iterator
+isImage(): bool
AATStransGenerator
–dgraphtrans: *Digraph<State>
–agentset: QVector<Agent>
–mainvalueset: *ValueSet
+AATStransGenerator()
+AATStransGenerator(QVector<Agent> & ag, ValueSet & vs)
+AATStransGenerator(QVector<Agent> & ag, ValueSet & vs, DiGraph<State> &)
+addTrans(const State & sfrom, const State & sto): void
+addTrasn(const State & s):void
+addAgent(const Agent & ag): void
+removeTrans(const State & sfrom, const State & sto):void
+removeAgent(Agent & ag): void
+getSize(): int
+getTransGraph: QList<Node<State> >::iterator
+getAgents(): QVector<Agent>
+end(): QList<Node<State> >::iterator
+getValueSet(): ValueSet &
template class
DiGraph
–nodeto: QList<Node<N>>
+DiGraph()
+DiGraph(const N & from, const & N to)
+~DiGraph()
+ add(const & N): void
+ add(const & N from,const & N to): void
+ removeEdge(const & N from, const & N to): void
+getNode(const N & n): QList<Node <N> >::iterator
+getIterator(): QList<Node <N> >::iterator
+end(): QList<Node <N> >::iterator
+getReverseGraph(DiGraph<N> & g): void
+size(): int
friend of
template class
Dfs
–depth: int
–cnt: int
–Pcnt: int
–pre:QMap<N, int>
–post: QMap<N, int>
–cyclelist: QList<int>
–dfs: QList<DfsNode<N> >
–digraph: DiGraph<N>
–cycle:bool
+Dfs()
+Dfs(DiGraph<N> & g)
+initdfs(N n)
+initdfs()
+hascycle()
+getCycleList(): & QList<int>
+getDfs(): & QList<DfsNode<N>>
+getPreOrderSequence(): QMap<N, int>
+getPostOrderSequence(): QMap<N, int>
–Dfsr(Node<N> n): void
ArgGenerator
–statelist : QVector<QString>
–cqxset: CQxSet
–aatstrans: AatsTransGenerator *
+ArgGenerator()
+~ArGenerator()
+ArgGenerator(AatsTransGenerator & aatsg)
+getArgs(): CQxSet
+getObj(): CQxSet
+buildArgsAndObj(): void
+getStateList(): QList<QString> &
+getArgsDescription(): QVector<QString>
+getObjDescription(): QVector<QString>
+getArgGraph(DiGraph<Argument> & dg):void
+get AatsTransGenerator(): AatsTransGenerator &
VAFgenerator
–argmax: Argument
–unsolvable: bool
–prefext: QList<Argument>
–vaf: DiGraph<Argument> *
–rev_vaf: DiGraph<Argument>
–dictator: Agent
+VaFGenerator()
+VAFGenerator(DiGraph<Argument> & dg, Agent & ag)
+initVAF(): void
+solvePolychromaticCycles(Dfs<Argument> & d): Dfs<Argument>
+solveDichromaticCycles(Dfs<Argument> & d,Argument & arg):void
+solveUncycledVaf(): void
+getSolution(): QList<Argument> &
+vafResult(): QString
+isUnsolvable(): bool
+getReverseUpdatedGraph(): DiGraph<Argument>
+vafArgsResult(): QString
+getMaxArgument(): Argument &
friend of
template class
1
DfsNode
–depth: int
–type: int
–node: Node<N> *
+DfsNode()
+DfsNode(Node<N> & n, int d, int t)
+getNode(): Node<N> &
+getDepth(): int
+getType(): int
CQxSet
–cqxset: QList<CQx>
+CQxSet()
+add(CQx &): void
+addUnsorted(CQx &): void
+remove(const CQx &): void
+getIterator(): QList<CQx>::iterator
+end(): QList<CQx>::iterator
+size(): int
*
1
CQx
–name: QString
–argname: QString
–value: Value
–statefrom: State
–stateto: State
+CQx()
+CQx(QString n,State & s, Value & v)
+CQx(QString n,State & sto, State & sfrom, Value & v)
+getStatefrom(): State &
+getStateto(): State &
+getValue(): Value &
+setArgName(QString n): void
+getArgName(): QString
+getName(): QString
+setName(QString n): void
+operator==(const CQx &): bool
Argument
–name: QString
–value: Value
–cqx: CQx
+Argument()
+Argument(QString n, Value & v)
+Argument(QString n, Value & v,Cqx & cq)
+getValue(): Value &
+getCQx(): CQx &
+getName(): QString &
operator==(const Argument & arg): bool
operator<(const Argument & arg): bool

Fig. 12 represents a complete diagram with all methods and classes within the program. Please note that there are significant changes at this level, most of the classes have their functionality expanded offering a complete framework offering the possibility to expand the software easily with further versions, that can offer more functionalities and a better user interaction.

Pseudo-code and event diagrams:

Pseudo-code and transition diagrams

In this section it will be introduced the pseudocode for the most important methods and structures in the simulator.

Directed Graph:

As it was explained before the graphs representation will be implemented using an adjacency list representation [10].

To implement this approach it will be considered two classes, node and digraph as it is possible to describe in the previous class diagrams.

Node:

This class will contain the details of a node, and will contain all the node references that its edges will point at.

This class will contain a sequential container that will contain all the node references.

NOTE: As referred before the name from this class has changed from Vertex, and it is chosen to describe this time the methods add and remove.

DiGraph:

This class will contain a list with that will reference to the class node.

Each node in the graph will have two functions: if the node as edges directed from itself the node will be a normal node, but if the node does not have any edges directed from itself, and only has edges directed to itself, the node in the graph container will defined as an image.

The pseudocode for the methods add and remove edge is:

Method add(Node from, node to)	Method removeEdge(nfrom,nto)

tmpfrom=Search for from in graph;
tmpto=Search for to in graph;
if(tmpfrom!=null)
    tmpfrom.add(to);
else
    from.add(to)
    graph.add(from)
if(tmpto==null)
    graph.add(to)
	tmpfrom=Search for from in graph;
	tmpfrom.remove(nto);
	It(tmpfrom.size==0)
	    set tmpfrom to image

Dfs:

This class will implement depth-first search for the directed graphs, while this is one of the main changes from the original design since it allows cycle detection and in which all the VAF chain will be solved.

While this algorithm is based in DFS for directed graphs in [10] it, this version will have, in which will allow the start of the DFS from any node in the graph, and will keep track of all edges where the cycle starts, besides storing the DFS in a list instead of printing the algorithm.

The original algorithm keeps track of pre-order visit and the post-order of the nodes, using the pre-order to detect cycles.

Method initDfs(Node n)	Method DfsR(Node node)

if(pre.size()==0)
dfsR(n(n.v,n.w));
for(i=graph start; to graph end)
if(pre.find(i)==null)
dfsR(n(i.v,i.v));
	w=node.w;
	dfs.add(node,depth,”tree”);
	pre[w]=cnt++;
	depth++;
	tmp nodea=search for node w;
	for(i=nodes in a; to end)
	v=i;
	n=node(w,v);
	if(pre.find(v)==null) dfsR(n);
	else if(post.find(v)==null)
	dfs.add(n,depth,”back”);
	cyclelist.add(dfs.size-1)
	cycle=true;
	else
	dfs.add(node,depth,”cross”);
	post[w]=cntP++;
	depth--;

AATStransgenerator:

NOTE: This class originally should extent the class graph and use state class as a class that would extend node. But since C++ object model implements classes and inheritance in a different way to java the language in which we were used to work on, it was decided to take another approach, this approach will use the class Node and DiGraph as templates where they each node can be of any type and the class digraph will allow a to describe directed graphs with any type as well.

Using this approach the class AATStransgenerator will have a digraph class instance that will accept they type of the State class, allowing the representation of AATS diagrams.

ArgGenerator:

This class was originally created to only generate all the arguments and objections found within the AATS diagram from the AATStransgenerator class, but it was decided that this class would also generate the arguments graph for the VAF.

The arguments generation method in the previous design suffered some slight modifications in order to correct some errors found in the generation of some CQ's (CQ11, CQ10 and CQ7). Consequently a new pseudocode was created.

```
function getArgsandObj()
for(each agent a in A) do
   if(a is active) then
       for(each node n in DiGraph D) do
           prevn=n
           if(n≠from first in D && n.isgoalstate() && prevn.getProposition() ≠
n.getProposition() ) then
              hval=find highest common value in n prevn
              cqxset.add(CQx("CQ6",n,hval))
           for(each node j in n) do
              if(n=j and n.valueset.valueset.size()>0) do
                 highv=find highest value in n
                 cqxset.add(CQx("CQ5",n,highv))
              dvalue=0
              vcommon;
              for( each value v in j) do
                 if(j.isinitstate()) then
                     if(v.getvaluation="-" && dvalue>0) then
                        cqxset.add(CQx("CQ9",j,v))
                     if(v.getvaluation="-" && dvalue=0) then
                        cqxset.add(CQx("CQ8",j,v))
                        dvalue++
                     if(v.getvaluation="+" && dvalue>0) then
                        tmpcqxset.add(CQx("",n,j,v))
              for( each value v in agent value set) do
                 vcommon=0
                 for(each value vj in j) do
                     if(vj==v)
                        vcommon++
                 if(vcommon==0)
                     if(n.isinitstate() && n≠j)
                        cqxset.add(CQx("CQ11",j,v))
prevs= stateto in first of tmpcqxset
prevvaal
for(each cq i in tmpcqxset-1) do
   if(i.statefrom.isinitstate() && prevs==i.getstateto() && i.getValue()≠prevval)
      cqxset.add(CQx("CQ10",i.getStateto(),i.getValue()))
   for(each cq j in tmpcqxset) do

      if(i.getstateto()≠j.getstateto() && i.getValue()=j.getValue())
         if(j.getValue().getDegree==-1 ||
i.getstateto.getvalue.degree>i.getstateto.getvalue.degree) do
             cqxset.add(CQx("CQ7",j.getStateto(),i.getstateto(),i.getValue()))
         if(i.getstateto.getvalue.degree<i.getstateto.getvalue.degree) do
             cqxset.add(CQx("CQ7",i.getStateto(),j.getstateto(),i.getValue()))
   prevval=i.getValue()
if(!i==tmpcqxset.end() && i.getstatefrom.isinitstate() && prevs=i.getStateto() &&
!(i.getValue()≠prevval)
   cqxset.add(CQx("CQ10",i.getStateto(),i.getValue()))
```

To understand how each cq are obtained we will explain them separately, since the whole process is computationally expensive it will be chosen to obtain all the arguments in one single method, in order to achieve better efficiency.

– First the algorithm looks for agents that are active (in which there action has to be considered) [4].

– CQ6: simply checks if a state is a goalstate, since DiGraph will contain all nodes in graph, it will always check all nodes.

– CQ5: checks for actions that direct to the same state [4].

– CQ8: checks for the first demoted value in a action [4].

– CQ9: checks for all the other demoted values in a action [4].

– CQ11: compares all values with each value of the agents on each action looking for values that are precluded(or that do not appear in the action) [4].

To look for CQ10 and CQ7 all the promoted values are gathered and treated individually, since it would be more costly to look at CQ7 inside the others loops.

– CQ10: looks for values promoted in action without repeating them.
Please note that for performance reasons, when this exceptional cases occur it is expected to the user to insert this values with same priorities and are inserted sequentially.

– CQ7: looks for values that are promoted within other actions and compares them, to check if they are more promoted less or equally [1].

VAFGenerator:

Note: This class as completely changed from the previous design, the only functionality that this class will offer, will be to solve VAF and eliminate cycles.

To consider this new functionality lets define which cycle type the algorithm can solve, dichromatic and polychromatic cycles[12] and [11].

Polychromatic cycles:

As described before these cycles, can simply be solved by removing the element in the cycle with the least important value.
In order to remove these cycles from a VAF it was chosen a greedy method approach algorithm, which uses a recursive call on a DFS to remove the cycles.
The pseudocode to remove this type of cycles is:

```
function solvePolychromaticCycles(dfs){
  if(cyclelist.size()>0){
    for(each cycle j in the cyclelist starting at the end){
     Value vcur=dfslist[j].w.getValue();
     if(j=cyclelist.begin()){
       argmin=argprev=dfslist[j];
       cycle=dfslist[j].v);
       if(vcycle.find(vcur.getName())=vcycle.end()){
         Value tmpv=dfslist[j].w.getValue();
         vcycle.insert(vcur.getName(),dictator value importance in
vcur);
       }
```

```
      }
      else if(argprev.w=dfslist[j].v){
        cyclenode++;
        if(vcycle.find(vcur.getName())==vcycle.end()){
           Value tmpv=dfslist[j].getNode().getNode().getValue();
           vcycle.insert(vcur.getName(),dictator value importance in
vcur);
        }
        compvalue=dictator.getValueSet()-
>compareTo(dfslist[j].w.getValue(),argmin.v.getValue());
        if(compvalue<0){
           argmin=dfslist[j];
        }
        argprev=dfslist[j];
      }
      if(cycle==dfslist[j].w){
        endcycle=true;
        break;
      }
     }
     if(vcycle.size()>2 && endcycle){
      vaf->removeEdge(argmin.w ,argmin.v);
      result=(Dfs(vaf));
      result.initdfs();
      solvePolychromaticCycles(result);
     }
     else if(vcycle.size()=2 && endcycle){
      solveDichromaticCycles(dfs,argmin.w);
     }
     else{
      unsolvable=true;
     }
  }
  else{
     dfs=Dfs(vaf);
  }
  return dfs;
}
```

Dichromatic cycles:

This cycle type, will deleted using the method describe in the previous section. To be able to remove this cycle type, it will know before hand that the current cycle is a dichromatic, from the previous function. Then it will proceed removing the cycle using the approach described in background.

```
function solveDichromaticCycles(dfs, argmin){
    if(cyclelist.size()>0){
      for( each cycle j in the cyclelist starting at the end){
    if(j=cyclelist.begin()){
        argprev=dfslist[j];
        cycle=dfslist[j].v;
       }
       else if(argprev.w==dfslist[j]v){
           if(dfslist[j].w.getValue()=vmin &&
dfslist[j].v.getValue()≠vmin){
             vaf->removeEdge(dfslist[j].w ,*(dfslist[j].v));
           }
           if(argprev.w.getValue()=vmin &&
!((argprev.v.getValue())=vmin)){
             vaf->removeEdge(argprev.w ,argprev.v);
           }
        argprev=dfslist[j];
       }
       if(cycle=dfslist[j].w){
        endcycle=true;
        break;
       }
      }
      if(endcycle){
       result=(Dfs(vaf));
       result.initdfs();
       solvePolychromaticCycles(result);
      }
    }
    else{
      solvePolychromaticCycles(dfs);
    }
}
```

VAF chains:

To solve a DFS chain it’s necessary to know, which node will be the one in which the chain will start solving, to do this it’s necessary to find the non-attacked node in the graph with the highest value. To accomplish we use the transverse graph, and we check in the graph which node will be the one with highest value. Since this is a very expensive computational procedure, it was chosen to keep the VAF in the DFS list since it will offer a better performance under sparse graphs.

Since the DFS can start from any point in the graph (the node with the highest value ordering) not just from the ideal node or from its actual root. This algorithm will solve chains using another greedy approach with a recursive call. The algorithm will take only in consideration the previous node and the current one, it save and previous nodes that had been visited in the DFS.

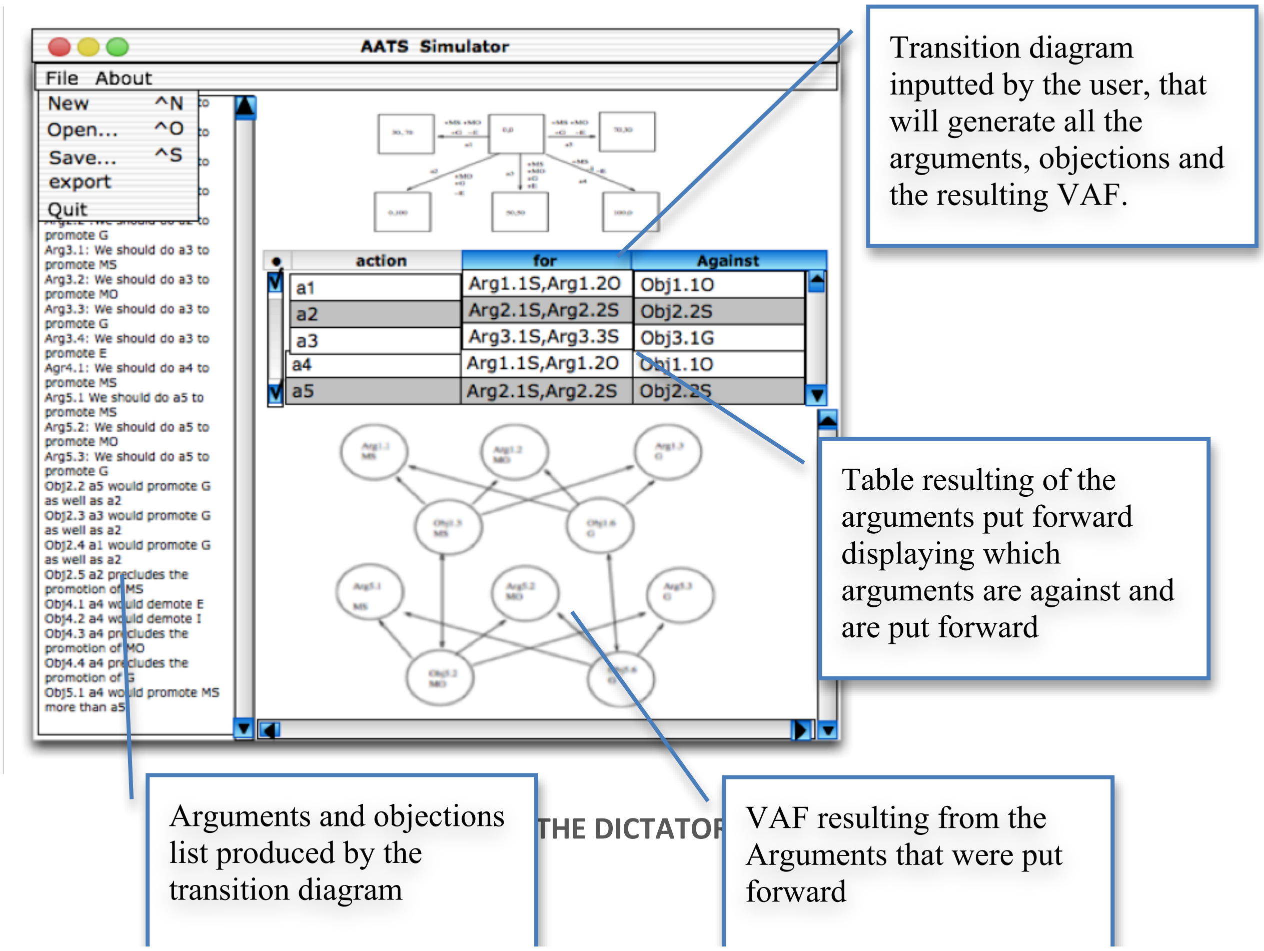
When an element that it’s proven to not belong to the preferred extension anymore it will make another check through the piece of the chain involved removing or adding new arguments to the preferred extension as its possible to see from the pseudocode.

```
functio solve_ext(start,end,dfslist){
   if(dfslist[start].w≠dfslist[start].v)
    visitedto.insert(dfslist[start].w,start);
    for(start i to end in dfslist){
       DfsNode<Argument> cur=dfslist[i];
       compvalue=dictator.getValueSet().compareTo(cur.w.
getValue(),cur.v->getValue());
       if(i==start){
         prev=cur;
       }
       if((cur.w=cur.v && cur.getDepth()=0)){
         if(prefext.indexOf(cur.w,0)==-1){
           prefext.append(cur.w);
         }
       }
       else{
         if(visitedfrom.find(cur.w)=visitedfrom.end()){
           visitedfrom.insert(cur.w,i);
       }
       if(compvalue>=0){
         int argpos=prefext.indexOf(*cur.v,0);
         if(prefext.indexOf(cur.w,0)=-1 && visitedto.find(
cur.w)!=visitedto.end()){
            if(argpos=-1 && visitedto.find(cur.v )=visitedto.end()){
               prefext.append(cur.v);
            }
         }
         if(argpos>=0 && visitedto.find(cur.w)!=visitedto.end()){
            prefext.removeAt(argpos);
            if(visitedfrom.find(cur.v)!=visitedfrom.end()){
               solve_ext(visitedfrom.value(cur.v,i,dfslist);
            }
           }
         }
         else{
           if(prefext.indexOf(cur.v,0)==-1 && visitedto.find(cur.v)=
visitedto.end()){
            prefext.append(cur.v);
           }
         }
       }
       if(visitedto.find(*cur.v)==visitedto.end()){
         visitedto.insert(*cur.v,i);
       }
       prev=cur;
    }
}
```

Interface Design:

The way that the user will interact with the program is described on the diagram below, where we can see, that the only role that the user has is to start the simulation and insert the AATS transition diagram.

The previous expected interface will be:



promote G
Arg3.1: We should do a3 to promote MS
Arg3.2: We should do a3 to promote MO
Arg3.3: We should do a3 to promote G
Arg3.4: We should do a3 to promote E
Agr4.1: We should do a4 to promote MS
Arg5.1 We should do a5 to promote MS
Arg5.2: We should do a5 to promote MO
Arg5.3: We should do a5 to promote G
Obj2.2 a5 would promote G as well as a2
Obj2.3 a3 would promote G as well as a2
Obj2.4 a1 would promote G as well as a2
Obj2.5 a2 precludes the promotion of MS
Obj4.1 a4 would demote E
Obj4.2 a4 would demote I
Obj4.3 a4 precludes the promotion of MO
Obj4.4 a4 precludes the promotion of G
Obj5.1 a4 would promote MS more than a5

action	for	Against
a1	Arg1.1S,Arg1.2O	Obj1.1O
a2	Arg2.1S,Arg2.2S	Obj2.2S
a3	Arg3.1S,Arg3.3S	Obj3.1G
a4	Arg1.1S,Arg1.2O	Obj1.1O
a5	Arg2.1S,Arg2.2S	Obj2.2S

Transition diagram inputted by the user, that will generate all the arguments, objections and the resulting VAF.

Table resulting of the arguments put forward displaying which arguments are against and are put forward

Arguments and objections list produced by the transition diagram

VAF resulting from the Arguments that were put forward

Realisation

Project Implementation:

At this stage we will go through each stage of the project implementation, and we will go through all the problems that occurred and solutions.

DiGraph class:

It was chosen to represent direct graphs using an adjacency list representation, it was decided at the beginning of the implementation a similar implementation as in [10], but that representation offered some issues, to our simulator:

- **Issue:** The implementation in the book uses a linkedlist approach,
- **Solution:** According with [5] the linkedlist implementation used in Qt only offers better performance that QList if the containers will have more than 200 nodes, since in this project it never happens it was chosen to use QList instead of the original LinkedList.

- **Issue:** According to OOP model in C++, when a class is extended from another one, it can use the other class functionality and access to its methods, but if it contains a container that stores elements from the first class, they will only store it's components not the components that are native from the extended class. Per example if state extended node as in the original design, all the information about state stored in the node container in the class node would just be, the methods that are common to node, not the ones created in state.
- **Solution:** Because of this it was chosen to create the class Node and DiGraph as templates that can accept any type. This way a Node can contain elements of the Class State or of Arguments Class.

- **Issue:** To use the implementation in [10] we would have to recreate our own linkedlist implementation, what was decided to be avoided, since it would be time consuming to create an implementation that would offer the same stability and performance and efficiency.
- **Solution:** It was chosen to implement a different directed graph class.

While implementing a different approach to this class, it noted some problems such:

- **Issue:** It was observed that a node with no edges directing to or from itself, would have the same role as a node that is just an image (a node that does not contain edges directed from itself) because of this issue
- **Solution:** It was decided that each node should contain a Boolean property to distinguish this, and the property would change automatically if the role of the node changes (if a new edge is inserted into an image node, or an edge is deleted).

The only issues that currently exist in this class are related with the pointers relation within the class Node and the class DiGraph, what affects an optimal memory management but this issue does not affect its correctness. Because of the project had a limited time it was chosen to leave this issue since it wouldn't affect particularly the performance.

Besides being capable of representing directed graphs with any time, this class offers the possibility to generate reverse graphs. To generate a reverse graph the method will take an empty graph and copy its content to this empty graph, where an edge (w,v) will be inserted like in (v,w).

The source code for this method is:

```
template<typename N>
void DiGraph<N>::getReverseGraph(DiGraph<N> &reverse)
  for(typename QList<Node<N> >::iterator i=nodeto.begin();i!=end();i++){
      if(i->size()==0 && !i->isImage())
        reverse.add(i->getNode());
      for(typename QList<N>::iterator j=i->getIterator();j!=i->end();j++){
            reverse.add((*j),i->getNode());
      }
  }
}
```

The full source code for this class will be in appendix D.

DFS class:

This class was the biggest change from the design stage to the implementation, since briefly after the design stage was concluded it was found severe flaws within the design that would allow to solve the preferred extension for VAF, the original design had immense flaws, the cycle detection was wrongly implemented, and the original algorithm did not take into consideration the different types of cycles and how they had to be deleted, to solve the VAF.

After some deep research through Graph theory books [10] and [15], after a few failures with algorithms that could use cycle detection such Tarjan's algorithm or Gabow's or Kosaraju's algorithms it was decided that all this algorithms could not offer enough information about cycles in directed graphs, to implement the VAF class, since they could not inform in which nodes the cycles was occurring, to be possible to remove them according with is necessary in the VAF theory.

The only way found to detect cycles and keep track was using the DFS algorithm found in [10], this algorithm besides flattening the graph into a sequential container (QList was the chosen one), it finds all the cycles allowing to keep track of back edges, what allowed to go from node to node in that cycle.

This algorithm was changed to check faster for cycles in the DFS, storing all the index of the back edges in the dfslist container. And was modified not just be able to accept nodes of any type, but also to start the DFS from any node in the graph, and going still through all the edges in the graph.

This algorithm considers 3 different edges types [10]:

- Those representing a recursive call (tree edges back edges represent with a number 2 in the dfsnode class).
- Those from a vertex to an ancestor in its DFS tree (back edges represent with a number -1).

– Those from a vertex to a descendant in its DFS tree (down edges represented as 1).

– Those from a vertex to another vertex that is neither an ancestor nor a descendant in its DFS tree (cross edges represented with 0).

To distinguish each node in the dfslist container was created a class that would represent each one of these nodes, storing the edge type, the depth and the edge itself.

The source code used to initialize the DFS is:

```
void initdfs(){
  for(typename QList<Node<N> >::iterator
i=digraph.getIterator();i!=digraph.end();i++){
      if(pre.find(i->getNode())==pre.end()){
        dfsR(Node<N>(i->getNode(),i->getNode()));
      }
  }
}
```

This method basically initializes the DFS from the first node in the DiGraph class, it is used when the node to start from isn't important or unknown.

The method basically goes through the graph nodes and checks if they are visited using a preorder way to go through the graph, every time that it didn't found a node in the preorder container it will run the method dfsR to start creating the DFS list.

Source code used to initialize the DFS from a specific node.

```
void initdfs(N n){
typename QList<Node<N> >::iterator nodestart=digraph.getNode(n);
  if(pre.size()==0){
      if(!(nodestart==digraph.end())){
        dfsR(Node<N>((*nodestart).getNode(),(*nodestart).getNode()));
      }
  for(typename QList<Node<N> >::iterator
i=digraph.getIterator();i!=digraph.end();i++){
      if(pre.find(i->getNode())==pre.end()){
         dfsR(Node<N>(i->getNode(),i->getNode()));
      }
  }
}
```

This method it's very similar to the one before but before it starts search through the graph for nodes that aren't in the preorder container, it starts solving the DFS straight from the node N.

The following source shows how the dfs gets solved.

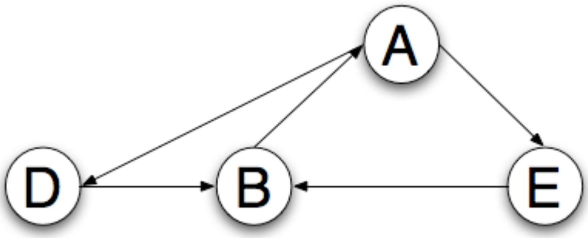
```
void dfsR(Node<N> node){
  N w=(*node.getIterator());
  dfs.append(DfsNode<N>(*(new Node<N>((node.getNode()),w)),depth,2));
  pre[w]=cnt++;depth++;
  typename QList<Node<N> >::iterator a=digraph.getNode(w);
  for(typename QList<N>::iterator i=(*a).getIterator();!(i==(*a).end());i++){
     N v=(*i);
     Node<N> n=Node<N>(w,v);
     if(pre.find((*i))==pre.end()) dfsR(n);
```

```
      else if(post.find((*i))==post.end()){
        dfs.append(DfsNode<N>(*(new Node<N>(*node.getIterator(),(*i))),depth,-
1));
        cyclelist.append(dfs.size()-1);
        cycle=true;
      }
      else dfs.append(DfsNode<N>(*(new
Node<N>(*node.getIterator(),(*i))),depth,0));
   }
   post[*node.getIterator()]=cntP++;depth--;
}
```

This method solves the DFS through recursive calls on the edges that aren't visited using the preorder of visiting. If a node is found in its preorder way of visiting but using the postorder way, it will be marked as a back edge and it will be stored where this back edge was found in the dfslist, otherwise will be a cross edge.

While implementing this the VAFgenerator class it was found a bug that affects this algorithm from [10] but it does not affect the correctness of the cycle solving algorithms in the Class.

The algorithm in 10 present in dfsR does not detect all cycles when a digraph contains two cycles starting and finishing at the same node per example:



(fig. 13)

In fig. 13 it is possible to observe that there is two cycles from {A,E,B,A} and {A,D,B,A} but the algorithm do not detect both cycles, both end at A. When this bug was found in the algorithm it was no time left to find another solution for this problem keeping the correctness of the algorithm, so it was chosen to create another DFS every time, that a cycle is deleted to update the DFS looking for the missing loop.

The output of this issue will be displayed in the Testing section. It can also be found the correct output for a VAF with type of cycles.

Value class:

This class suffered some slight changes from the design to the implementation mainly at the methods instantiation level, where it was decided to create more methods that will more adequate to represent a value in different circumstances or evaluate its properties.

To understand how Values are represented it is necessary to describe it's main properties.

Each value will have a Name, which will identify each value, a valuation that will describe if the value is being promoted (represented with +), demoted (represented with -) and the degree in which a value gets promoted or demoted. The degree is particular important in this experiment since for proposition the Dictator gives some money away, using the degree attached to each others value it is possible to know which value as being more promoted than others based on the degree. Per example in fig.9 a1 proposition says that Dictator will have 30% of the money and the other 70%, so the degree to value of MS (dictators money) will be 70 and MO 30.

Each constructor in the Value class allows the representation of Values in the VAF, and in each joint action at the AATS.

Constructors' source code:

```
Value::Value(){
      this->name="";
      this->valuation="";
      this->degree=-1;
}
```

The constructor above will initialize a Value with no name, no valuation and no degree.

```
Value::Value(QString n){
      (*this).name=n;
      this->valuation="";
      this->degree=-1;
}
```

However this constructor will initialize a Value with a name to identify.

```
Value::Value(QString n,QString s){
      (*this).name=n;
      setValuation(s);
      this->degree=-1;
}
```

This constructor goes a step further and allows a Value to instantiated with a name and a valuation.

```
Value::Value(QString n,QString s, int d){
      (*this).name=n;
      setValuation(s);
      this->degree=d;
}
```

This last constructor allows a Value to be instantiated with all the possible properties, name, valuation and degree.

ValueSet class:

This class offer the possibility to handle sets of Values ordering them by importance, from the least important to the most important value. All the methods in this class offer a way to compare values the values importance. This class specifies that the minimum importance value is -1 and its maximum 100. This set accepts values with same importance degree, or with difference importance degree.

The source code for its main methods is:

```
bool ValueSet::addValue(const Value & value,int p){
    if(p>=-1){
       values.insertMulti(p,value);
    }
    else return false;
    return true;
}

void ValueSet::removeValue(const Value &value,int p){
   for(QMap<int, Value>::iterator i=values.find(p);i!=values.end();i++){
      if((*i)==value)
            values.erase(i);
      }
   }

int ValueSet::compareTo(const Value &v1,const Value &v2){
   int v1pos=getValue(v1).key();
   int v2pos=getValue(v2).key();
   if (v1pos<v2pos)
      return -1;
   else if(v1pos>v2pos)
      return 1;
   else
      return 0;
}

QMap<int,Value>::iterator ValueSet::highestValue(){
      return values.end()-1;
}

QMap<int,Value >::iterator ValueSet::lowestValue(){
      return values.lowerBound(0);
}
```

This class was first implement using a Vector where the index of the values in the vector were its importance within the set, but since a set of values can include as well different values with the same importance it was necessary to find another way that would represent the importance of values within a set while keeping a order of importance. Since QMap can insert a key and value, it was decided that the importance will be the map key, and instances of values will be value in the map. This way QMap will organize the values by importance from the lowest one to highest one.

NOTE: this class wasn’t included in the original design, and was created because the original design could not handle sets of values.

Agent class:

In this experiment only one agent has the control over the actions, so for this experiment itself this class does not offer much functionality from implementing an agent as a subclass of AATSTransGenerator, but since one of the main goals of this project wasn’t only to implement the Dictator experiment but also others experiments such the one described in [4] where the environment has to take into consideration different agents with an active role in the experiment and different value ordering.

Because this it was chosen to create a specific class that could represent active or passive agents within different experiments.

The source code for this class is:

```
  Agent::Agent(){ }

  Agent::Agent(QString n,bool actv){
      name=n;
      active=actv;
  }

  Agent::Agent(QString n,bool actv, ValueSet & v){
      name=n;
      vset=&v;
      active=actv;
  }

  ValueSet *Agent::getValueSet(){
  return vset;
  }
  QString Agent::getName(){
      return name;
  }
  bool Agent::isActive(){
  return active;
  }
  void Agent::setActive(bool actv){
      active=actv;
  }
  const bool Agent::operator==(const Agent &ag) const {
      Agent prt=*&ag;
      if(name==prt.getName())
              return true;
  return false;
  }
```

No issues were found while implementing this class, its functionality has bee kept from the previous design.

JointAction class:

This class will represent a joint-action within two states, it will have a set of Values that will be promoted or demoted, as well as a name that will identify the joint action. A state will have a jointaction if there is an edge that directs to itself.

The source code for this class:

```
JointAction::JointAction(QString n){
        name=n;
}

JointAction::JointAction(const ValueSet &v,QString n){
        name=n;
        vset=v;
}

ValueSet & JointAction::getValueSet(){
        return vset;
}
QString JointAction::getName(){
        return name;
}
```

```
void JointAction::setName(QString n){
      name=n;
}
```

State class:

This class will represent states within an AATS diagram, this class will be able to contain an instance joint-action that directs to other states, will contain an instance of proposition to describe the proposition in the state, and will contain a name.

The source code for the constructors is:

```
State::State(){
      prop=(new Proposition());
      jaction=(new JointAction());
      }
```

The constructor above initializes a state with a proposition with no description and a jointaction with no values or name.

```
State::State(Proposition& p,bool is,bool gs,JointAction &j,QString n){
      prop=&p;
      jaction=&j;
      name=n;
      initialstate=is;
      goalstate=gs;
}
```

The constructor above the instantiation of a State, with a proposition, a Jointaction, a name (the name for the state should be always the same as the Jointaction), allows setting the state as initial and goal state.

```
State::State(Proposition& p,bool is,bool gs,QString n) {
      prop=&p;
      jaction=(new JointAction());
      name=n;
      initialstate=is;
      goalstate=gs;
}
```

The constructor above will allow the instantiation of a state with a proposition, a name and will allow the setting it as a goal state or initial state. This constructor will be used mainly when the state hasn’t got any edge directing to itself.

```
State::State(Proposition& p,JointAction &j,QString n) {
      prop=&p;
      jaction=&j;
      name=n;
      goalstate=false;
      initialstate=false;
}
```

The constructor above allows the instantiation of a state with a proposition a jointaction and a name, setting is goalstate and initialstate to false.

The methods source code is:

```
Proposition & State::getProposition(){
      return *prop;
}

bool State::operator==(const State& exp) const{
      State prt=*&exp;
      JointAction jnametmp=*jaction;
      JointAction j1nametmp=prt.getJointAction();
      if(name==prt.getName() &&
jnametmp.getName()==j1nametmp.getName())
            return true;
      else
            return false;
}
```

```
QString State::getName(){
return name;
}

void State::setName(QString n){
      name=n;
}
bool State::isGoalState(){
      return goalstate;
}
bool State::isInitState(){
      return initialstate;
}

void State::setGoalState(bool gs){
      goalstate=gs;
}

void State::setInitState(bool is){
      initialstate=is;
}

JointAction & State::getJointAction(){
      return *jaction;
}
```

No issues were found while implementing this class, no issues are known after completion.

AATSTransGenerator class:

This class will represent the AATS diagram. It will contain an instance of all agents in the experiment, a valueset instance with all the values present in the AATS diagram, and a DiGraph instance that will accept the states for nodes.

The only changes in the implementation for this class are that it doesn't extend the class DiGraph for reasons described above (see DiGraph class description). Instead of extending DiGraph class this class will use it with State as type and this way will allow the representation of the AATS diagram.

The source code for the constructors in this class is:

```
AatsTransGenerator::AatsTransGenerator(QVector<Agent>& ag,ValueSet
&vs){
    agentset=QVector<Agent>(ag);
    mainvalueset=&vs;
    dgraphtrans=new DiGraph<State> ();
}
```

The above constructor will create an instance of an AATS diagram with an empty graph, a set of agents in a QVector container and a ValueSet that will contain all the main set of values.

```
AatsTransGenerator::AatsTransGenerator(QVector<Agent> & ag ,ValueSet
&vs, DiGraph<State> &dg){
    agentset=QVector<Agent>(ag);
    mainvalueset=&vs;
    dgraphtrans=&dg;
}
```

The constructor above will do the same as the first constructor but will create an instance with a Graph that might exist.

The source code for all the methods in the class is:

```
void AatsTransGenerator::addTrans(const State &transfrom, const State
&transto){
    (*dgraphtrans).add(transfrom,transto);
}

void AatsTransGenerator::addTrans(const State &transfrom){
      (*dgraphtrans).add(transfrom);
}

void AatsTransGenerator::addAgent(const Agent &ag){
    agentset.append(ag);
}

void AatsTransGenerator::removeTrans(const State& transfrom,const
State&transto){
    (*dgraphtrans).removeEdge(transfrom,transto);
}

void AatsTransGenerator::removeAgent(const Agent& ag

   for(QVector<Agent>::iterator i=agentset.begin();
i!=agentset.end();i++){
       if((*i)==*&ag)
          agentset.erase(i);
   }
}

QList<Node<State> >::iterator AatsTransGenerator::getTransGraph(){
   return (*dgraphtrans).getIterator();
}

QVector<Agent> & AatsTransGenerator::getAgents(){
   return agentset;
}

QList<Node <State> >::iterator AatsTransGenerator::end(){
   return (*dgraphtrans).end();
}

ValueSet & AatsTransGenerator::getValueSet(){
   return *mainvalueset;
}
```

No issues were found while implementing this class, since the design revisions solved all the problems around it.

CQx class:

The cqx class was constructed to be able create an instance from any CQ from CQ5 to CQ11, allowing to keep the all the necessary information about the CQ found and the state where it was found as the value that is in question as well, This class will be mainly used in the ArgGenerator class and in the argument Class.

The class will provide several constructors and methods to allow changes and the visualization of the information contained in itself.

NOTE: The current implementation of this class follows some changes on the previous design, CQx class will be able to represent a CQ and CQxSet class will be responsible to handle sets of CQx within the ArgGenerator class.

To understand how this class tries to represent a CQ we will describe this class properties and their functionality in this class;

- name: This property will be used to describe the name of the CQ in use.
- value: This property will contain the value in which the CQ uses to itself.
- statefrom: This property has two different uses, when used with CQ10 or CQ7, it does not get used with any other CQ. When used with CQ10 this property will contain the state where the joint-action comes from, according with [4] a CQ10 to exist it needs to come from the initial state.

  In CQ7 this property will be used to distinguish the different states where the value is promoted as well, this property in conjuction with stateto will define the two states necessary to identify a CQ7.
- stateto: this property will contain the state with the jointaction where CQ was found, this property is used in all CQ's in this project.
- argname: This property contains the index of the argument in the set of arguments generated, this index is similar to the one found in [1] in the list of arguments generated.

The source code for the constructors in this class is:

```
CQx::CQx(QString n ,State & s,Value & v){
      stateto=s;
      statefrom=*(new State());
      value=v;
      name=n;
      argname="";
}
```

The constructor above will create an instance of CQx with a state, a name and a value. This constructor is one used in all the CQ's besides CQ10 and CQ9.

```
CQx::CQx(QString n ,State & sfrom,State & sto,Value & v){
      statefrom=sfrom;
      stateto=sto;
      value=v;
      name=n;
      argname="";
}
```

The constructor above will create an instance of CQx with two states and a value. Used to describe CQ10 and CQ7.

The source code for the methods in this class is:

```
QString CQx::getName(){
return name;
}
State & CQx::getStatefrom(){
      return statefrom;
}
State & CQx::getStateto(){
      return stateto;
}

void CQx::setName(QString n){
      name=n;
}

Value & CQx::getValue(){
      return value;
}

void CQx::setArgName(QString n){
      argname=n;
}

QString CQx::getArgName(){
      return argname;
}

const bool CQx::operator==(const CQx & cqx) const{
      CQx prt=*&cqx;
      if((value)==prt.getValue() && (statefrom)==prt.getStatefrom()
&& (stateto)==prt.getStateto())
            return true;
      return false;
}
```

CQxSet class:

This class handles sets of CQ’s, and its mainly used within the Class ArgGenerator, IT offers the possibility of insert CQ’s sorted in a similar way to the one presented in [1] or to insert them in an unsorted way.

This class wasn’t present in the old design document, it was decided that it would be necessary that it was necessary that representation of a CQ’s set and CQ’s themselves should be in different classes. This allows a straightforward way to describe CQ’s in a set and giving a more correct and better OOP design.

The source code for the method add which will insert the CQs sorted is:

```
void CQxSet::add(CQx &cqx){
   if(cqxset.size()==0 || cqxset.size()==1){
      cqxset.append(cqx);
      return;
   }
   for(QList<CQx>::iterator i=cqxset.begin();i!=cqxset.end();i++){
      if(cqx.getName()=="CQ10" && i->getName()=="CQ10" &&
cqx.getStateto()==i->getStateto()) {
            cqxset.insert(i,cqx);
            return;
      }
      else if(cqx.getName()!="CQ7" && cqx.getStateto()==i->getStateto()){
            cqxset.insert(i+1,cqx);
            return;
      }
      else if(cqx.getName()=="CQ7"){
            if(cqx.getValue().getDegree()==-1 && cqx.getStateto()==i-
>getStateto()){
                  cqxset.insert(i+1,cqx);
                  return;
            }
            Value vfrom= (*cqx.getStatefrom().getJointAction()
.getValueSet().getValue(cqx.getValue().getName()));
                  Value vto= (*cqx.getStateto().getJointAction()
.getValueSet().getValue(cqx.getValue().getName()));
            if(vfrom.getDegree()>vto.getDegree() && cqx.getStateto()==i-
>getStateto()){
                  cqxset.insert(i+1,cqx);
                  return;
            }
            if(vfrom.getDegree()<vto.getDegree() && cqx.getStatefrom()==i-
>getStateto()){
                  cqxset.insert(i+1,cqx);
                  return;
            }
      }

   }
   cqxset.append(cqx);
}
```

ArgGenerator class:

This is one of the main Classes in the simulator, it will be responsible for the generation of all arguments and objections in [1], and it offers the possibility to display all arguments and objections in separated containers, as well as a description similar to the one existing in [1], once all the arguments and objections are generated this class can also generate the arguments graph, the VAF graph.

This class suffered some slight changes to the original design, since now it can generate the VAF graph, and generate a complete description of all the arguments in the AATS. The changes in the method buildArgandObjs (previously called buildArgs) are mainly of correctness, the changes are already enumerated in the design document, but while testing the AATS and after some reflection it was realized that the AATS arguments and objections at this stage, do NOT depend on the agents, why? According with [4] at this stage the agents should already established an agreement with the set of values that will be part of the AATS, so once following the rules for the CQ5 to CQ11 it's possible to observe that they will not interfere with the generation of these CQ's, since they agree with the same set of values, promotion, demotion and preclusion of values it's dependent of the AATS that was created and on the actions in it.

AS2 in [4] says:

> The initial state $q_0 = q_x \in Q$,
> Agent $i \in Ag$ should participate in joint action $j_n \in J_{Ag}$ where $j_n^i = \alpha_i$,
> Such that $\tau(q_x, j_n)$ is q_y,
> Such that $p_a \in \pi(q_y)$ and $p_a \notin \pi(q_x)$, or $p_a \notin \pi(q_y)$ and $p_a \in \pi(q_x)$,
> Such that for some $v_u \in Av_i, \delta(q_x, q_y, v_u)$ is +.

But in fact if AS2 is used this stage (choice of action) on an experiment with more active agents (agents with an active role in the experiment) the fact that arguments and objections should be generated based on some Agent *i* [4], it would generate repeated arguments that would generate "clutter" in the VAF graph.

If we observe the pseudocode that would generate all the arguments and objections it's possible to observe that only with CQ11 it is used the agent set of values to find precluded values, but since the agents already agreed apriori the set of Values that will belong to the experiment their valueset is equal to the one existing in the AATS.

Consequences: This fact arises a few consequences:

1st: If the argument list will be generated per agent and each agent generate the same arguments and objections then it will generate duplicate arguments.

2nd: This will waste space and time, since the algorithm has to go through the AATS diagram for each agent.

3rd: Readability, it will be harder to read trough all the arguments and objections and find specific information if the same information is generated *n* times (where n is the number of active agents)

4th: This is one the most important reasons, if the list contains repeated arguments they will not affect the result of the VAF, since we argue that if two repeated arguments belong to the preferred extension because of some value ordering, they will bring any new information if they constitute the same argument in fact. But with another value ordering that would defeat one of the arguments we argue that the other will be defeated as well. Repeated arguments do not bring any new information to the VAF, since or they will be both defeated or will belong to the preferred extension, what in the last case for one to belong to the VAF, having the second one in the preferred extension will not constitute a better defence since they mean the same.

Besides the current conclusions our algorithm still will follow the steps evolving the agents. However this issue will not be detected experiment for the project, since the Dictator is the only agent with that can contribute to the experiment.
But on other experiments such the one in [4] this issue would in fact appear.
It was decided to the authors to look into this issue and look for new conclusions.

It was chosen to highlight the following methods in this class:

```
void ArgGenerator::buildArgsAndObj(){
 AatsTransGenerator aats=(*aatstrans);
 CQxSet tmpcqxset;
 State tmpstate;
 for(QVector<Agent>::iterator a=aats.getAgents().begin();a!=aats.getAgents().end();a++){
  if((*a).isActive()){
     for(QList<Node<State> >::iterator i=aats.getTransGraph(); !(i==aats.end());i++){
      State tmpstatei=(*i).getNode();
      JointAction tmpjacti=tmpstatei.getJointAction();
      if(i!=aats.getTransGraph() && tmpstatei.isGoalState() && !(tmpstatei.getProposition()
==tmpstate.getProposition())){
            Value tmpval1=tmpjacti.getValueSet().highestCommonValue(
tmpstate.getJointAction().getValueSet());
            cqxset.add(*(new CQx("CQ6",tmpstatei,tmpval1)));//done
     }
     for(QList<State>::iterator j=(*i).getIterator();!(j==(*i).end());j++){
     State tmpstatej=(*j);
     statelist.append(j->getJointAction().getName());
     JointAction tmpjactj=tmpstatej.getJointAction();
     if(tmpstatei==tmpstatej && tmpjacti.getValueSet().size()>0){
      Value tmpval=*tmpjacti.getValueSet().highestValue();
      cqxset.add(*(new CQx("CQ5",tmpstatej,tmpval)));//done
     }
     int dvalue=0;
     int vcommon;
     int exp;
     for(QMap<int,Value >::iterator z=tmpjactj.getValueSet().getIterator();
z!=tmpjactj.getValueSet().end();z++){
      if(tmpstatei.isInitState()){//check if the state is a initial state
        if((*z).getValuation()=="-" && dvalue>0){
            cqxset.addUnsorted(*(new CQx("CQ9",(*j),(*z))));//done
        }
        if((*z).getValuation()=="-" && dvalue==0){
            cqxset.add(*(new CQx("CQ8",(*j),(*z))));//done
            ++dvalue;
        }
      }
      if((*z).getValuation()=="+"){
            tmpcqxset.addUnsorted(*(new CQx("",(*i).getNode(),(*j),(*z))));
      }
     }
     for(QMap<int,Value >::iterator v=(*(*a).getValueSet()).getIterator();
v!=(*(*a).getValueSet()).end();v++){
        exp=vcommon=0;
        for(QMap<int,Value >::iterator y=tmpjactj.getValueSet().getIterator();
y!=tmpjactj.getValueSet().end();y++){
            if((*y)==(*v)){
              vcommon++;
            }
        }
        if(vcommon==0)
            if(tmpstatei.isInitState() && !(tmpstatej==tmpstatei)){
              cqxset.add(*(new CQx("CQ11",(*j),(*v))));
```

```
           }
        }
     }
    }
   }
  }
  State prevSto=(*tmpcqxset.getIterator()).getStateto();
  Value prevval;
  QList<CQx>::iterator i=tmpcqxset.getIterator();
  for(;!(i==tmpcqxset.end()-1);i++){//need to put cq10 bullet proof
   if((*i).getStatefrom().isInitState() && ((prevSto==(*i).getStateto() &&
!((*i).getValue()==prevval)) || !(prevSto==(*i).getStateto()))){
       (cqxset).add(*(new CQx("CQ10",(*i).getStateto(),(*i).getValue())));
   }
   QList<CQx>::iterator j=i;//tmpcqxset.getIterator();
   for(;!(j==tmpcqxset.end());j++){
       if(!((*i).getStateto()==(*j).getStateto())  && (*i).getValue()==(*j).getValue()){
         QMap<int,Value>::iterator vfromj=j->getStateto().getJointAction().getValueSet()
.getValue(i->getValue().getName());
         QMap<int,Value>::iterator vtoj=i->getStateto().getJointAction().getValueSet()
.getValue(i->getValue().getName());
         if((j->getValue().getDegree()==-1 || vfromj->getDegree()>vtoj->getDegree())){
           cqxset.add(*(new CQx("CQ7",(*j).getStateto(),(*i).getStateto(),
(*i).getValue())));
         }
         else if(vfromj->getDegree()<vtoj->getDegree())
           cqxset.add(*(new CQx("CQ7",(*i).getStateto(),(*j).getStateto(),
(*i).getValue())));
         }
       }
       prevval=i->getValue();
   }
   if(!(i==tmpcqxset.end()) && (*i).getStatefrom().isInitState() &&
((prevSto==(*i).getStateto() && !((*i).getValue()==prevval)) ||
!(prevSto==(*i).getStateto())))
       (cqxset).add(*(new CQx("CQ10",(*i).getStateto(),(*i).getValue())));
  }
```

The method above generates all the arguments and objections in [1], with some exceptions and will find a CQ9 in a4 (fig. 9) what wasn't present in the original document [1] the output of the tests proceed for this method will be presented in testing.

The exceptions mentioned before are the ones mentioned in the design of this algorithm, in [1] its possible to observe the existence of arguments with the same meaning, these arguments besides offering different perspectives over the actions they mean the same and they both create the dilemma present in the VAF in [1].

In [1] in the argument list include the following arguments:

- Obj1.6 a5 is as good as a1 with respect to G

But:

- Obj 5.6 a1 would promote G as well as a5

As its possible to observe these two arguments have the same meaning they both say that a1 and a5 promote G equally.

This is possible to observe with Obj3.6 and Obj2.3, Obj1.7 and Obj 3.7, Obj3.5 and Obj5.8 and with a few more Objections involving G.

The existence of Obj1.6 and Obj5.6 are the cause of the monochromatic cycle in [1], but these cycles would not need to exist if it did not exist pairs of arguments posed differently with the same meaning. Because of this it was chosen to implement the generation of CQ7 in such a way that would not pose repeated arguments never.

The following method will implement the VAF graph:

```
void ArgGenerator::getArgGraph(DiGraph<Argument> &argGraph){
   CQxSet argsl=this->getArgs();
   CQxSet objl=this->getObj();
   for(QList<CQx>::iterator i=objl.getIterator();i!=objl.end();i++){
      if((i->getName()=="CQ8" || i->getName()=="CQ9"|| i-
>getName()=="CQ11"|| i->getName()=="CQ7")){
         Argument argfrom=(Argument(i->getArgName(),i->getValue(),(*i)));
         argGraph.add(argfrom);
         for(QList<CQx>::iterator j=cqxset.getIterator();j!=cqxset.end();
j++){
          if(j->getName()=="CQ7" ){
             if(i->getStateto()==j->getStatefrom())
              argGraph.add(argfrom,Argument(j->getArgName(),j-
>getValue(),(*j)));
          }
          else if((j->getName()=="CQ5" || j->getName()=="CQ6"|| j-
>getName()=="CQ10") && i->getStateto()==j->getStateto())
             argGraph.add(argfrom,Argument(j->getArgName(),j-
>getValue(),(*j)));
         }
       }
     }
   }
```

This method will start from creating nodes from an objections container creating this way the attacking nodes, then it looks to the main set of arguments and objections and inserts the arguments that will be attacked from the objections inserted before.

Currently there is only one issue in this class that was chosen from the creators to be left to the creators for further research, the agent set issue when generating the AATS arguments and objections.

Currently there are no other issues and the performance of this class is considered optimal. There is no known way of implementing the methods in this class in a more efficient way.

VAFGenerator class:

This class follows the revised design specification created to be able to solve VAFs, using the greedy algorithm approach described before, this class will provide a way to solve cycles (polychromatic and dichromatic) and chains, being able to give a preferred extension after solving the VAF.

To solve a VAF this class will go through at the most two steps, cycle deletion if cycles exist in the VAF and solving chains in the VAF to obtain the preferred extension.

- Cycle deletion:

To delete a cycle the according with [11], [3] and [12] first it's necessary to run a DFS then if cycles exist then the algorithm will use the DFS list and will run this list backwards from the back edge to where it found it for the first time, while it goes through the arguments backwards the algorithm counts how many values are present, and which node is has the least important value.

If only one value is encountered then the algorithm halts and declares the VAF as unsolvable by the simulator, if it counts two values it calls a function that will explicit solve this type of cycle, and finally if it counts three values will remove the edge from the weakest one to the other in the cycle.

If the cycle contained two values it would go through the cycle again and would remove all edges that would go from the least important to the most important as explain before in the Background section.

- Solve Arguments chains:

To solve chains a different procedure to the one in [11] and [12] are used, instead of looking through even or odd chains the algorithm simply, chooses the most important un-attacked value in the VAF and starts solving the VAF from there creating an updated DFS, then it recursively solves all the chains in the VAF taking in consideration the elements visited, the elements already in the preferred extension (in case it needs to delete them), and the previous and current node.

When an argument in the preferred extension is deleted: In this case the algorithm will look to the index where it found the element deleted and where it is now, solving the resulting chain again. This way it is possible to solve chains in VAF.

This changes were fundamental to the original design without them it would not be possible to solve VAF's since the previous design could only find the unnattacked node in the VAF.

This is the source code for the main methods in this class;

```
void VAFGenerator::solveUncycledVaf(){
  DiGraph<Argument> rev_garg;
  vaf->getReverseGraph(rev_garg);
  Argument argmax;
  bool init=false;
  for(QList<Node<Argument> >::iterator i=rev_garg.getIterator();
i!=rev_garg.end();i++){
    Argument curarg=i->getNode();
    if(i->size()==0 && !init){
      argmax=i->getNode();
      init=true;
    }
    else if(i->size()==0){
      int compvalue=dictator.getValueSet()->compareTo(curarg.getValue(),
argmax.getValue());
        if(compvalue>0){
          argmax=curarg;
        }
      }
    }
    Dfs<Argument> dfsexp(*vaf);
    dfsexp.initdfs(argmax);
    QList<DfsNode<Argument> > dfslist=dfsexp.getDfs();
    this->solve_ext(0,dfslist.size(),dfslist);
  }
```

The method above creates a reverse graph from the VAF graph, and looks for the un-attacked argument with, the most important value, once the argument is found the dfs is created starting from that argument. Leaving the DFS list ready to be used and solve the VAF.

```
void VAFGenerator::solve_ext(int start,int end,  QList<DfsNode< Argument> > &
dfslist){
   DfsNode<Argument> prev;
   QMap<Argument,int> visitedto;
   QMap<Argument,int> visitedfrom;
   if(!(dfslist[start].getNode().getNode()==*dfslist[start].getNode().getIterator()
))
    visitedto.insert(dfslist[start].getNode().getNode(),start);
    for(int i=start;i<end;i++){
       DfsNode<Argument> cur=dfslist[i];
       int compvalue=dictator.getValueSet()->compareTo(cur.getNode().getNode().
getValue(),cur.getNode().getIterator()->getValue());
       if(i==start){
         prev=cur;
       }
       if((cur.getNode().getNode()==*cur.getNode().getIterator() &&
cur.getDepth()==0)){
         if(prefext.indexOf(cur.getNode().getNode(),0)==-1){
           prefext.append(cur.getNode().getNode());
         }
       }
       else{
         if(visitedfrom.find(cur.getNode().getNode())==visitedfrom.end()){
           visitedfrom.insert(cur.getNode().getNode(),i);
       }
       if(compvalue>=0){
         int argpos=prefext.indexOf(*cur.getNode().getIterator(),0);
         if(prefext.indexOf(cur.getNode().getNode(),0)==-1 && visitedto.find(
cur.getNode().getNode())!=visitedto.end()){
            if(argpos==-1 && visitedto.find(*cur.getNode().getIterator()
)==visitedto.end()){
               prefext.append(*cur.getNode().getIterator());
            }
         }
         if(argpos>=0 && visitedto.find(*cur.getNode().getIterator()
)==visitedto.end()){
            prefext.removeAt(argpos);
            if(visitedfrom.find(*cur.getNode().getIterator())!=visitedfrom.end()){
               this->solve_ext(visitedfrom.value(*cur.getNode().getIterator())
,i,dfslist);
            }
           }
         }
         else{
           if(prefext.indexOf(*cur.getNode().getIterator(),0)==-1 &&
visitedto.find(*cur.getNode().getIterator())==visitedto.end()){
            prefext.append(*cur.getNode().getIterator());
           }
         }
       }
       if(visitedto.find(*cur.getNode().getIterator())==visitedto.end()){
         visitedto.insert(*cur.getNode().getIterator(),i);
       }
       prev=cur;
    }
}
```

This is the method that will allow solving VAF chains, this algorithm follows the revised project design, and offers a recursive way to solve chains in VAF.

```
Dfs<Argument> & VAFGenerator::solvePolychromaticCycles(Dfs<Argument>
& dfs){
  Dfs<Argument> result;
    QList<int> cyclelist=dfs.getCycleList();
    QList<DfsNode<Argument> > dfslist=dfs.getDfs();
    DfsNode<Argument> argmin;
    DfsNode<Argument> argprev;
    Argument cycle;
    QMap<QString,int> vcycle;
    bool endcycle=false;
    int cyclenode=0;
    if(cyclelist.size()>0){
     for(int j=*cyclelist.begin();0<=j;j--){
       Value vcur=dfslist[j].getNode().getNode().getValue();
       if(j==*cyclelist.begin()){
         argmin=argprev=dfslist[j];
         cycle=*(dfslist[j].getNode().getIterator());
         if(vcycle.find(vcur.getName())==vcycle.end()){
           Value tmpv=dfslist[j].getNode().getNode().getValue();
           vcycle.insert(vcur.getName(),(dictator.getValueSet()-
>getValue(vcur)).key());
         }
       }
       else if(argprev.getNode().getNode()==*(dfslist[j].getNode()
.getIterator())){
         cyclenode++;
         if(vcycle.find(vcur.getName())==vcycle.end()){
           Value tmpv=dfslist[j].getNode().getNode().getValue();
           vcycle.insert(vcur.getName(),(dictator.getValueSet()-
>getValue(vcur)).key());
         }
         int compvalue=dictator.getValueSet()-
>compareTo(dfslist[j].getNode().getNode().getValue(),argmin.getNode()
.getNode().getValue());
         if(compvalue<0){
           argmin=dfslist[j];
         }
         argprev=dfslist[j];
       }
       if(cycle==dfslist[j].getNode().getNode()){
         endcycle=true;
         break;
       }
     }
     if(vcycle.size()>2 && endcycle){
       vaf->removeEdge(argmin.getNode().getNode()
,*(argmin.getNode().getIterator()));
       result=(Dfs<Argument>(*vaf));
       result.initdfs();
       solvePolychromaticCycles(result);
     }
     else if(vcycle.size()==2 && endcycle){
       solveDichromaticCycles(dfs,argmin.getNode().getNode());
     }
     else{
       unsolvable=true;
     }
    }
```

```
     else{
      dfs=Dfs<Argument>(*vaf);
     }
     return dfs;
  }
```

This method will solve polychromatic cycles within VAF's this method is very important since it finds how many values each contains, and creates a call to solve dichromatic cycle solver method, or declares the VAF unsolvable if an monochromatic cycles [11].

```
void VAFGenerator::solveDichromaticCycles(Dfs<Argument> &
dfs,Argument &argmin){
    Dfs<Argument> result;
    QList<int> cyclelist=dfs.getCycleList();
    QList<DfsNode<Argument> > dfslist=dfs.getDfs();
    DfsNode<Argument> argprev;
    Argument cycle;
    Value vmin=argmin.getValue();
    bool endcycle=false;
    if(cyclelist.size()>0){
      for(int j=*cyclelist.begin();0<=j;j--){
       if(j==*cyclelist.begin()){
        argprev=dfslist[j];
        cycle=*(dfslist[j].getNode().getIterator());
       }
       else if(argprev.getNode().getNode()==*(dfslist[j].getNode().
getIterator())){
           if(dfslist[j].getNode().getNode().getValue()==vmin &&
!((dfslist[j].getNode().getIterator()->getValue())==vmin)){
             vaf->removeEdge(dfslist[j].getNode().getNode()
,*(dfslist[j].getNode().getIterator()));
           }
           if(argprev.getNode().getNode().getValue()==vmin &&
!((argprev.getNode().getIterator()->getValue())==vmin)){
             vaf->removeEdge(argprev.getNode().getNode()
,*(argprev.getNode().getIterator()));
           }
        argprev=dfslist[j];
       }
       if(cycle==dfslist[j].getNode().getNode()){
        endcycle=true;
        break;
       }
      }
      if(endcycle){
       result=(Dfs<Argument>(*vaf));
       result.initdfs();
       solvePolychromaticCycles(result);
      }
    }
    else{
      solvePolychromaticCycles(dfs);
    }
}
```

The method above removes dichromatic cycles in VAF, this method does not detect this cycles since the polychromatic cycles method detects them. This method only removes them.

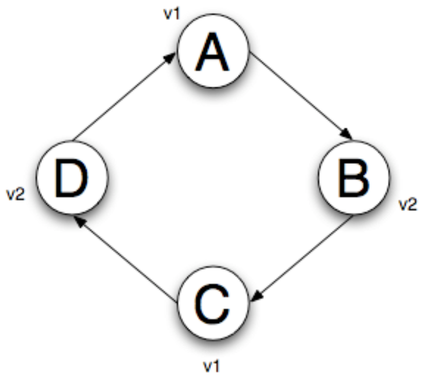
Testing:

In this chapter we will introduce all the testing and results done for VAF and AATS as well as the different outputs produced by the software on dictator game and the taking game. It is chosen to show the outputs of the two mechanisms in order to prove that everything works and produces the correct results, since it is almost impossible to verify manually the result of the VAF in the dictator game and the taking game.

VAF testing:

To test the VAF it was chosen to test it against different VAF with cycles and without, and it will be tested against the graph where the DFS outputs the wrong result in algorithm, to prove that it was found a way around the issue. It was chosen to show four tests for the VAF.

VAF cycle	Source code that inputs the Graph.

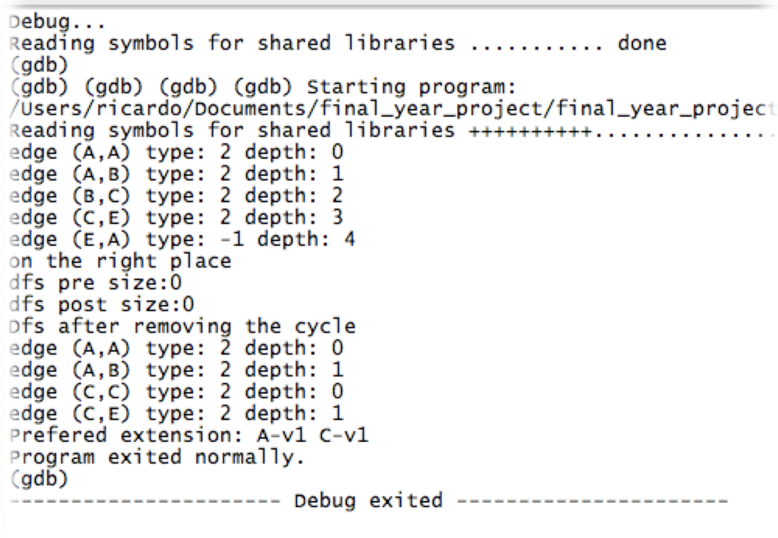

(fig. 14)
	`DiGraph<Argument> d1;`
	`Value v1("v1");`
	`Value v2("v2");`
	`ValueSet v_vafset;`
	`v_vafset.addValue(v1,3);`
	`v_vafset.addValue(v2,2);`
	`Agent vaf1("vaf1",true,v_vafset);`
	`Argument a("A",v1);`
	`Argument b("B",v2);`
	`Argument c("C",v1);`
	`Argument d("E",v2);`
	`d1.add(a,b);`
	`d1.add(b,c);`
	`d1.add(c,d);`
	`d1.add(d,a);`
	`VAFGenerator vafgen(d1, vaf1);`
	`vafgen.initVAF();`

DFS output before removing the cycle	DFS output after removing the cycle

`edge (A,A) type: 2 depth: 0`
`edge (A,B) type: 2 depth: 1`
`edge (B,C) type: 2 depth: 2`
`edge (C,D) type: 2 depth: 3`
`edge (D,A) type: -1 depth: 4`
	`edge (A,A) type: 2 depth: 0`
	`edge (A,B) type: 2 depth: 1`
	`edge (C,C) type: 2 depth: 0`
	`edge (C,D) type: 2 depth: 1`

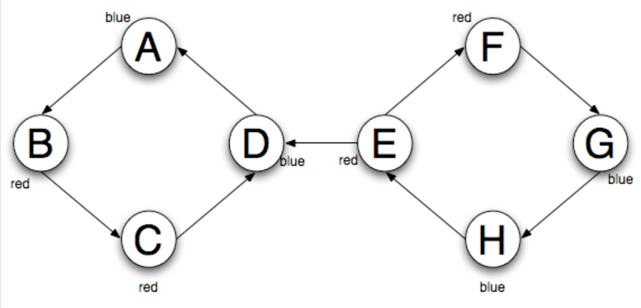
Preferred extension: {A,C}

As it's possible to see on the DFS before the cycle the removal it was found a cycle at edge (D,A) with a type -1 (back edge). The cycle is removed by taking the edges in which they direct to an argument with a strong value, such (B,C) and (D,A), giving the preferred extension {A,C}



```
Debug...
Reading symbols for shared libraries ........... done
(gdb)
(gdb) (gdb) (gdb) (gdb) Starting program:
/Users/ricardo/Documents/final_year_project/final_year_projec
Reading symbols for shared libraries ++++++++++...............
edge (A,A) type: 2 depth: 0
edge (A,B) type: 2 depth: 1
edge (B,C) type: 2 depth: 2
edge (C,E) type: 2 depth: 3
edge (E,A) type: -1 depth: 4
on the right place
dfs pre size:0
dfs post size:0
Dfs after removing the cycle
edge (A,A) type: 2 depth: 0
edge (A,B) type: 2 depth: 1
edge (C,C) type: 2 depth: 0
edge (C,E) type: 2 depth: 1
Prefered extension: A-v1 C-v1
Program exited normally.
(gdb)
---------------------- Debug exited ----------------------
```

VAF cycle	Source code that inputs the Graph.

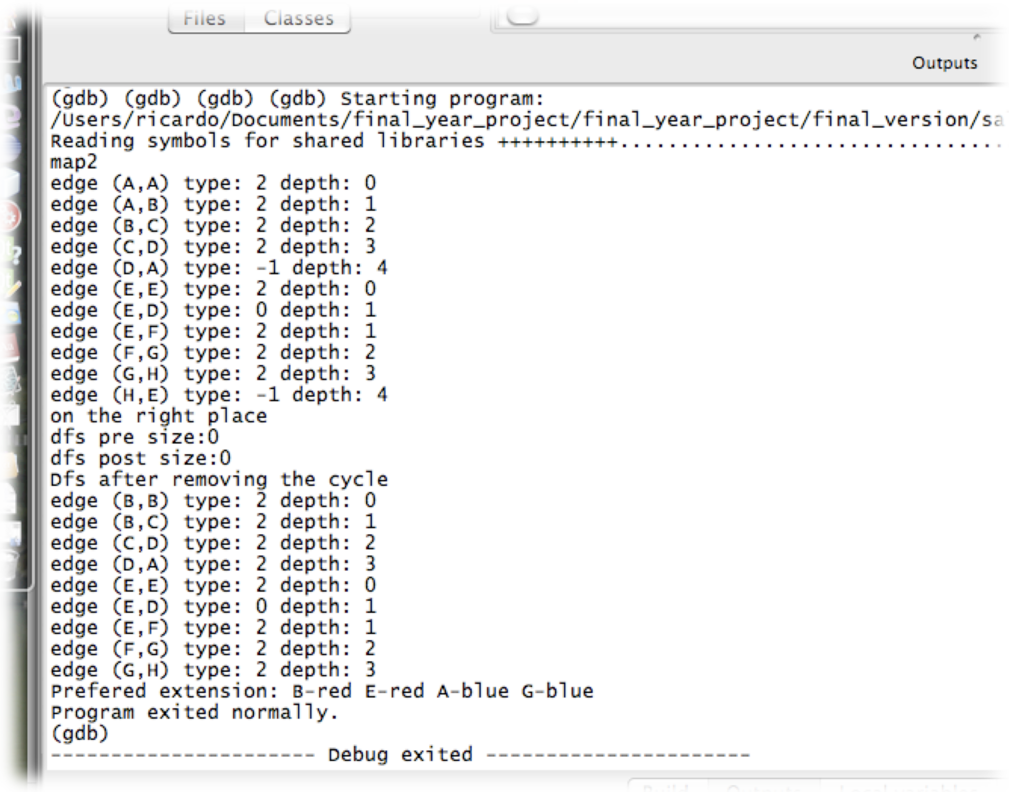
 blue
A
red
F
B
red
D
blue
red
E
G
blue
C
red
H
blue
(fig. 15)
	`DiGraph<Argument> d1;`
	`Value v1("blue");`
	`Value v2("red");`
	`ValueSet v_vafset;`
	`v_vafset.addValue(v1,2);`
	`v_vafset.addValue(v2,3);`
	`Agent`
	`vaf1("vaf1",true,v_vafset);`
	`Argument a("A",v1);`
	`Argument b("B",v2);`
	`Argument c("C",v2);`
	`Argument d("D",v1);`
	`Argument e("E",v2);`
	`Argument f("F",v2);`
	`Argument g("G",v1);`
	`Argument h("H",v1);`
	`d1.add(a,b);`
	`d1.add(b,c);`
	`d1.add(c,d);`
	`d1.add(d,a);`
	`d1.add(e,d);`
	`d1.add(e,f);`
	`d1.add(f,g);`
	`d1.add(g,h);`
	`d1.add(h,e);`
	`VAFGenerator`
	`vafgen(d1,vaf1);`
	`vafgen.initVAF();`

DFS output before removing the cycle	DFS output after removing the cycle
`edge (A,A) type: 2 depth: 0`	`edge (B,B) type: 2 depth: 0`
`edge (A,B) type: 2 depth: 1`	`edge (B,C) type: 2 depth: 1`
`edge (B,C) type: 2 depth: 2`	`edge (C,D) type: 2 depth: 2`
`edge (C,D) type: 2 depth: 3`	`edge (D,A) type: 2 depth: 3`
`edge (D,A) type: -1 depth: 4`	`edge (E,E) type: 2 depth: 0`
`edge (E,E) type: 2 depth: 0`	`edge (E,D) type: 0 depth: 1`
`edge (E,D) type: 0 depth: 1`	`edge (E,F) type: 2 depth: 1`
`edge (E,F) type: 2 depth: 1`	`edge (F,G) type: 2 depth: 2`

edge (F,G) type: 2 depth: 2 edge (G,H) type: 2 depth: 3 edge (H,E) type: -1 depth: 4	edge (G,H) type: 2 depth: 3

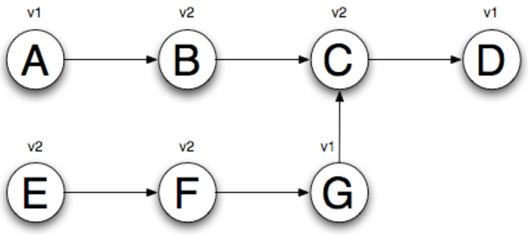
Preferred extension: {B,E,A,G}

It is possible to observe on the DFS before it was found a cycle at edges (D,A) and (H,E) with a type -1 (back edge). After the VAF checks for the cycle it removes the edges that direct from the nodes with least important values such (A,B) and (H,E), giving the preferred extension {B,E,A,G}.



```
(gdb) (gdb) (gdb) (gdb) Starting program:
/Users/ricardo/Documents/final_year_project/final_year_project/final_version/sa
Reading symbols for shared libraries ++++++++++...............................
map2
edge (A,A) type: 2 depth: 0
edge (A,B) type: 2 depth: 1
edge (B,C) type: 2 depth: 2
edge (C,D) type: 2 depth: 3
edge (D,A) type: -1 depth: 4
edge (E,E) type: 2 depth: 0
edge (E,D) type: 0 depth: 1
edge (E,F) type: 2 depth: 1
edge (F,G) type: 2 depth: 2
edge (G,H) type: 2 depth: 3
edge (H,E) type: -1 depth: 4
on the right place
dfs pre size:0
dfs post size:0
Dfs after removing the cycle
edge (B,B) type: 2 depth: 0
edge (B,C) type: 2 depth: 1
edge (C,D) type: 2 depth: 2
edge (D,A) type: 2 depth: 3
edge (E,E) type: 2 depth: 0
edge (E,D) type: 0 depth: 1
edge (E,F) type: 2 depth: 1
edge (F,G) type: 2 depth: 2
edge (G,H) type: 2 depth: 3
Prefered extension: B-red E-red A-blue G-blue
Program exited normally.
(gdb)
---------------------- Debug exited ----------------------
```

VAF cycle	Source code that inputs the Graph.

 v1
v2
v2
v1
A
B
C
D
v2
v2
v1
E
F
G
(fig. 16)
	`DiGraph<Argument> d1;`
	`Value v1("v1");`
	`Value v2("v2");`
	`ValueSet v_vafset;`
	`v_vafset.addValue(v1,3);`
	`v_vafset.addValue(v2,2);`
	`Agent`
	`vaf1("vaf1",true,v_vafset);`
	`Argument a("A",v1);`
	`Argument b("B",v2);`
	`Argument c("C",v2);`
	`Argument d("D",v1);`
	`Argument e("E",v2);`
	`Argument f("F",v2);`
	`Argument g("G",v1);`
	`d1.add(a,b);`
	`d1.add(b,c);`

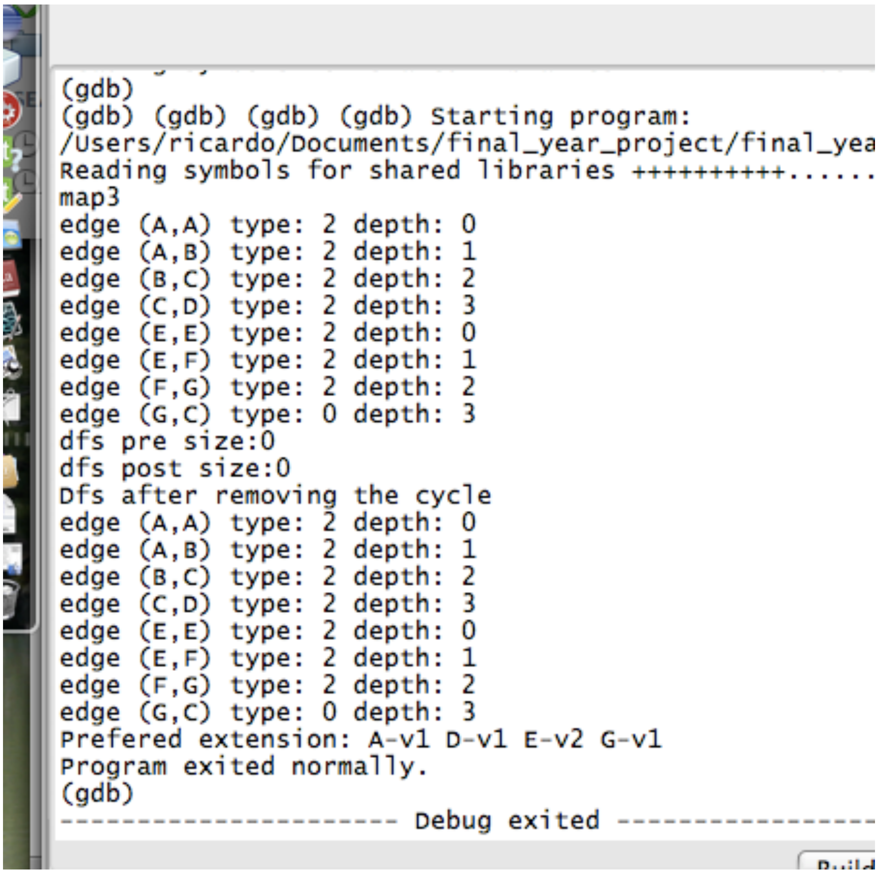
	`d1.add(c,d);`
	`d1.add(e,f);`
	`d1.add(f,g);`
	`d1.add(g,c);`
	`VAFGenerator vafgen(d1, vaf1);`
	`vafgen.initVAF();`

DFS output after starting solving the VAF

```
edge (A,A) type: 2 depth: 0
edge (A,B) type: 2 depth: 1
edge (B,C) type: 2 depth: 2
edge (C,D) type: 2 depth: 3
edge (E,E) type: 2 depth: 0
edge (E,F) type: 2 depth: 1
edge (F,G) type: 2 depth: 2
edge (G,C) type: 0 depth: 3
```

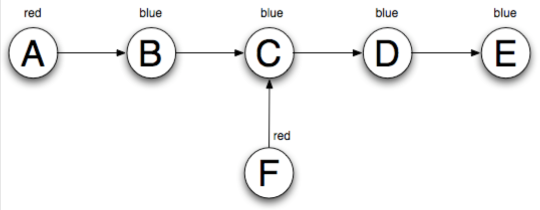
Preferred extension: {A,D,E,G}

In this example it’s possible to observe that the VAF starts solving from A the argument with the highest value order in the set. Giving the preferred extension {A,D,E,G}



```
(gdb)
(gdb) (gdb) (gdb) (gdb) Starting program:
/Users/ricardo/Documents/final_year_project/final_yea
Reading symbols for shared libraries ++++++++++......
map3
edge (A,A) type: 2 depth: 0
edge (A,B) type: 2 depth: 1
edge (B,C) type: 2 depth: 2
edge (C,D) type: 2 depth: 3
edge (E,E) type: 2 depth: 0
edge (E,F) type: 2 depth: 1
edge (F,G) type: 2 depth: 2
edge (G,C) type: 0 depth: 3
dfs pre size:0
dfs post size:0
Dfs after removing the cycle
edge (A,A) type: 2 depth: 0
edge (A,B) type: 2 depth: 1
edge (B,C) type: 2 depth: 2
edge (C,D) type: 2 depth: 3
edge (E,E) type: 2 depth: 0
edge (E,F) type: 2 depth: 1
edge (F,G) type: 2 depth: 2
edge (G,C) type: 0 depth: 3
Prefered extension: A-v1 D-v1 E-v2 G-v1
Program exited normally.
(gdb)
---------------------- Debug exited ----------------
```

VAF cycle



(fig. 17)

Source code that inputs the Graph.

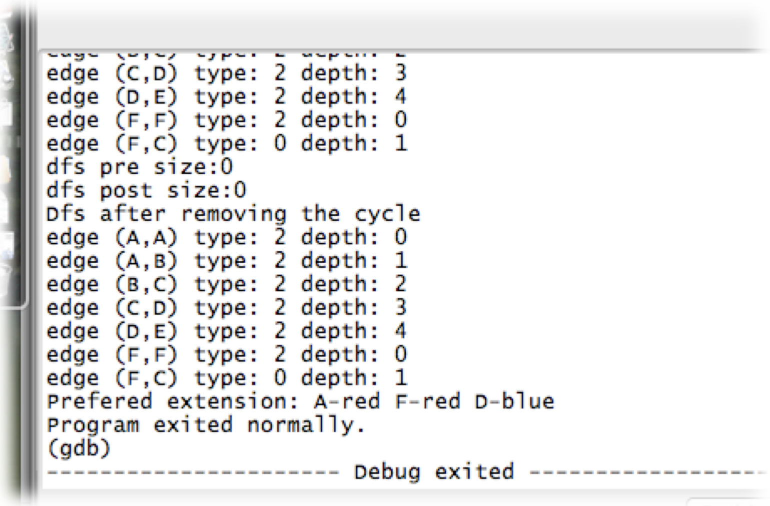
```
DiGraph<Argument> d1;
Value v1("blue");
Value v2("red");
ValueSet v_vafset;
v_vafset.addValue(v1,3);
v_vafset.addValue(v2,2);
Agent
vaf1("vaf1",true,v_vafset);
Argument a("A",v2);
Argument b("B",v1);
Argument c("C",v1);
Argument d("D",v1);
Argument e("E",v1);
Argument f("F",v2);
Argument g("g",v1);
Argument h("H",v2);
d1.add(a,b);
d1.add(b,c);
d1.add(c,d);
d1.add(d,e);
d1.add(f,c);
VAFGenerator vafgen(d1, vaf1);
vafgen.initVAF();
```

DFS output after starting solving the VAF

```
edge (A,A) type: 2 depth: 0
edge (A,B) type: 2 depth: 1
edge (B,C) type: 2 depth: 2
edge (C,D) type: 2 depth: 3
edge (D,E) type: 2 depth: 4
edge (F,F) type: 2 depth: 0
edge (F,C) type: 0 depth: 1
```

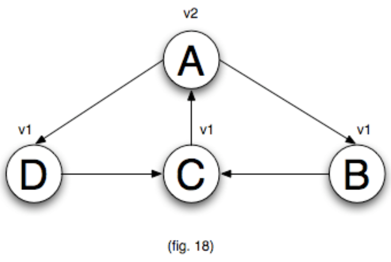
Preferred extension: {A,F,D}

In this example it starts solving the VAF from A the element with the highest value ordering, solving all the elements in the chain, but it is possible to observe that the chain is broken with the last node found (F,C), because F is stronger than C, the algorithm removes C from the chain and creates a recursive call at C to the preferred extension.



```
edge (C,D) type: 2 depth: 3
edge (D,E) type: 2 depth: 4
edge (F,F) type: 2 depth: 0
edge (F,C) type: 0 depth: 1
dfs pre size:0
dfs post size:0
Dfs after removing the cycle
edge (A,A) type: 2 depth: 0
edge (A,B) type: 2 depth: 1
edge (B,C) type: 2 depth: 2
edge (C,D) type: 2 depth: 3
edge (D,E) type: 2 depth: 4
edge (F,F) type: 2 depth: 0
edge (F,C) type: 0 depth: 1
Prefered extension: A-red F-red D-blue
Program exited normally.
(gdb)
---------------------- Debug exited ------------------
```

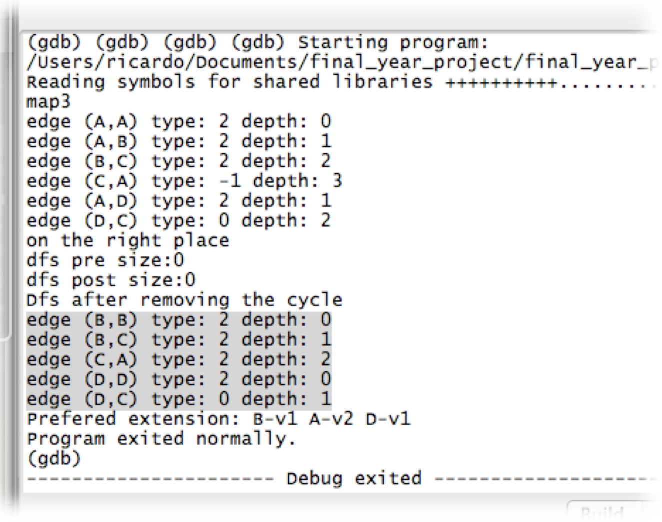
VAF cycle	Source code that inputs the Graph.


(fig. 18)
	DiGraph<Argument> d1;
	Value v1("blue");
	Value v2("red");
	ValueSet v_vafset;
	v_vafset.addValue(v1,3);
	v_vafset.addValue(v2,2);
	Agent vaf1("vaf1",true,v_vafset);
	Argument a("A",v2);
	Argument b("B",v1);
	Argument c("C",v1);
	Argument d("D",v1);
	d1.add(a,b);
	d1.add(b,c);
	d1.add(c,a);
	d1.add(a,d);
	d1.add(d,c);
	VAFGenerator vafgen(d1, vaf1);
	vafgen.initVAF();

DFS output before removing the cycle	DFS output after removing the cycle
edge (A,A) type: 2 depth: 0 edge (A,B) type: 2 depth: 1 edge (B,C) type: 2 depth: 2 edge (C,A) type: -1 depth: 3 edge (A,D) type: 2 depth: 1 edge (D,C) type: 0 depth: 2	edge (B,B) type: 2 depth: 0 edge (B,C) type: 2 depth: 1 edge (C,A) type: 2 depth: 2 edge (D,D) type: 2 depth: 0 edge (D,C) type: 0 depth: 1

Preferred extension: {B,A,D}

This example tests the existing bug in DFS algorithm and shows how besides this the issue is fixed for the purposes of this project. It is possible to observe that the DFS only detects one cycle instead of the two existing cycles on the VAF. On the DFS after all the cycle removal it is possible to observe that the edges (A,D) and (A,C), removing all the necessary edges, that create cycles. This way it is possible to prove the correctness of the algorithm.



```
(gdb) (gdb) (gdb) (gdb) Starting program:
/Users/ricardo/Documents/final_year_project/final_year_
Reading symbols for shared libraries ++++++++++........
map3
edge (A,A) type: 2 depth: 0
edge (A,B) type: 2 depth: 1
edge (B,C) type: 2 depth: 2
edge (C,A) type: -1 depth: 3
edge (A,D) type: 2 depth: 1
edge (D,C) type: 0 depth: 2
on the right place
dfs pre size:0
dfs post size:0
Dfs after removing the cycle
edge (B,B) type: 2 depth: 0
edge (B,C) type: 2 depth: 1
edge (C,A) type: 2 depth: 2
edge (D,D) type: 2 depth: 0
edge (D,C) type: 0 depth: 1
Prefered extension: B-v1 A-v2 D-v1
Program exited normally.
(gdb)
---------------------- Debug exited -------------------
```

Arguments Testing:

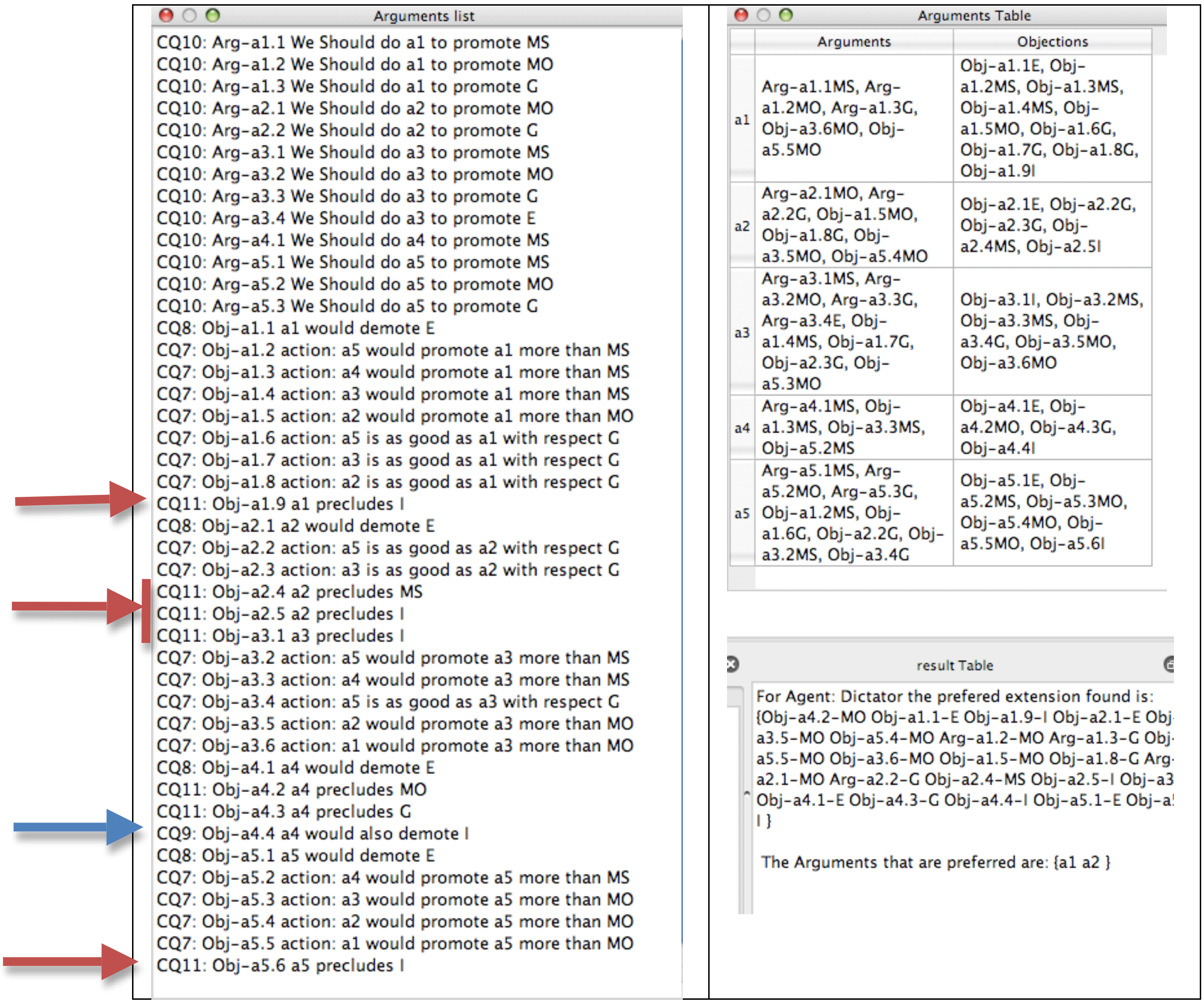
To test the different arguments generation it will be used two different values ordering each for the AATS in the dictator game and the taking game. According to [1] it is possible to take into consideration two aspects of the taking game, where it starts at a2 [0,100] and where it starts at a4 [100,0].

– Dictator Game:

The dictator agent in this test had a value ordering of: {{MO,G}>E>{MS,I}}.

Dictator source code:

```
vset_agdictator.addValue(v_ms,2);
vset_agdictator.addValue(v_i,2);
vset_agdictator.addValue(v_mo,5);
vset_agdictator.addValue(v_g,5);
vset_agdictator.addValue(v_e,4);
Agent agdictator("Dictator",true,vset_agdictator);
```



```
CQ10: Arg-a1.1 We Should do a1 to promote MS
CQ10: Arg-a1.2 We Should do a1 to promote MO
CQ10: Arg-a1.3 We Should do a1 to promote G
CQ10: Arg-a2.1 We Should do a2 to promote MO
CQ10: Arg-a2.2 We Should do a2 to promote G
CQ10: Arg-a3.1 We Should do a3 to promote MS
CQ10: Arg-a3.2 We Should do a3 to promote MO
CQ10: Arg-a3.3 We Should do a3 to promote G
CQ10: Arg-a3.4 We Should do a3 to promote E
CQ10: Arg-a4.1 We Should do a4 to promote MS
CQ10: Arg-a5.1 We Should do a5 to promote MS
CQ10: Arg-a5.2 We Should do a5 to promote MO
CQ10: Arg-a5.3 We Should do a5 to promote G
CQ8: Obj-a1.1 a1 would demote E
CQ7: Obj-a1.2 action: a5 would promote a1 more than MS
CQ7: Obj-a1.3 action: a4 would promote a1 more than MS
CQ7: Obj-a1.4 action: a3 would promote a1 more than MS
CQ7: Obj-a1.5 action: a2 would promote a1 more than MO
CQ7: Obj-a1.6 action: a5 is as good as a1 with respect G
CQ7: Obj-a1.7 action: a3 is as good as a1 with respect G
CQ7: Obj-a1.8 action: a2 is as good as a1 with respect G
CQ11: Obj-a1.9 a1 precludes I
CQ8: Obj-a2.1 a2 would demote E
CQ7: Obj-a2.2 action: a5 is as good as a2 with respect G
CQ7: Obj-a2.3 action: a3 is as good as a2 with respect G
CQ11: Obj-a2.4 a2 precludes MS
CQ11: Obj-a2.5 a2 precludes I
CQ11: Obj-a3.1 a3 precludes I
CQ7: Obj-a3.2 action: a5 would promote a3 more than MS
CQ7: Obj-a3.3 action: a4 would promote a3 more than MS
CQ7: Obj-a3.4 action: a5 is as good as a3 with respect G
CQ7: Obj-a3.5 action: a2 would promote a3 more than MO
CQ7: Obj-a3.6 action: a1 would promote a3 more than MO
CQ8: Obj-a4.1 a4 would demote E
CQ11: Obj-a4.2 a4 precludes MO
CQ11: Obj-a4.3 a4 precludes G
CQ9: Obj-a4.4 a4 would also demote I
CQ8: Obj-a5.1 a5 would demote E
CQ7: Obj-a5.2 action: a4 would promote a5 more than MS
CQ7: Obj-a5.3 action: a3 would promote a5 more than MO
CQ7: Obj-a5.4 action: a2 would promote a5 more than MO
CQ7: Obj-a5.5 action: a1 would promote a5 more than MO
CQ11: Obj-a5.6 a5 precludes I
```

	Arguments	Objections
a1	Arg-a1.1MS, Arg-a1.2MO, Arg-a1.3G, Obj-a3.6MO, Obj-a5.5MO	Obj-a1.1E, Obj-a1.2MS, Obj-a1.3MS, Obj-a1.4MS, Obj-a1.5MO, Obj-a1.6G, Obj-a1.7G, Obj-a1.8G, Obj-a1.9I
a2	Arg-a2.1MO, Arg-a2.2G, Obj-a1.5MO, Obj-a1.8G, Obj-a3.5MO, Obj-a5.4MO	Obj-a2.1E, Obj-a2.2G, Obj-a2.3G, Obj-a2.4MS, Obj-a2.5I
a3	Arg-a3.1MS, Arg-a3.2MO, Arg-a3.3G, Arg-a3.4E, Obj-a1.4MS, Obj-a1.7G, Obj-a2.3G, Obj-a5.3MO	Obj-a3.1I, Obj-a3.2MS, Obj-a3.3MS, Obj-a3.4G, Obj-a3.5MO, Obj-a3.6MO
a4	Arg-a4.1MS, Obj-a1.3MS, Obj-a3.3MS, Obj-a5.2MS	Obj-a4.1E, Obj-a4.2MO, Obj-a4.3G, Obj-a4.4I
a5	Arg-a5.1MS, Arg-a5.2MO, Arg-a5.3G, Obj-a1.2MS, Obj-a1.6G, Obj-a2.2G, Obj-a3.2MS, Obj-a3.4G	Obj-a5.1E, Obj-a5.2MS, Obj-a5.3MO, Obj-a5.4MO, Obj-a5.5MO, Obj-a5.6I

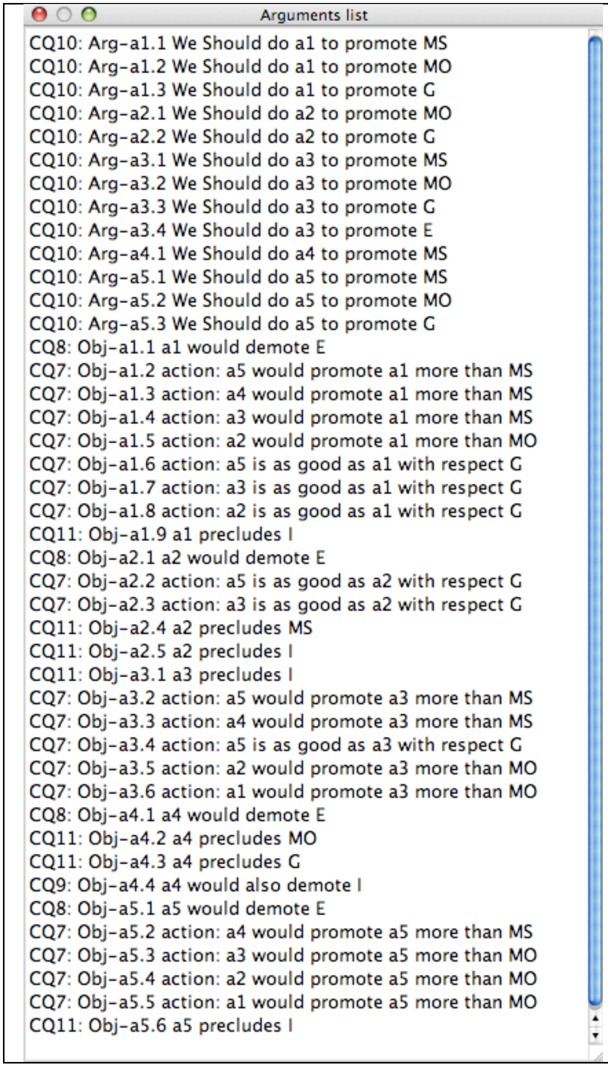
```
For Agent: Dictator the prefered extension found is:
{Obj-a4.2-MO Obj-a1.1-E Obj-a1.9-I Obj-a2.1-E Obj-
a3.5-MO Obj-a5.4-MO Arg-a1.2-MO Arg-a1.3-G Obj-
a5.5-MO Obj-a3.6-MO Obj-a1.5-MO Obj-a1.8-G Arg-
a2.1-MO Arg-a2.2-G Obj-a2.4-MS Obj-a2.5-I Obj-a3
Obj-a4.1-E Obj-a4.3-G Obj-a4.4-I Obj-a5.1-E Obj-a
I }

 The Arguments that are preferred are: {a1 a2 }
```

In this test it is possible to observe all the arguments and objections generated in the argument list, as well as all the CQ11 that weren't in the original paper (red arrow) and the CQ9 that isn't also in the original paper (blue arrow). The arguments preferred in this example are {a1,a2} that are satisfiers actions for the dictator. This way it is proven that the dictator is in fact a satisfier.

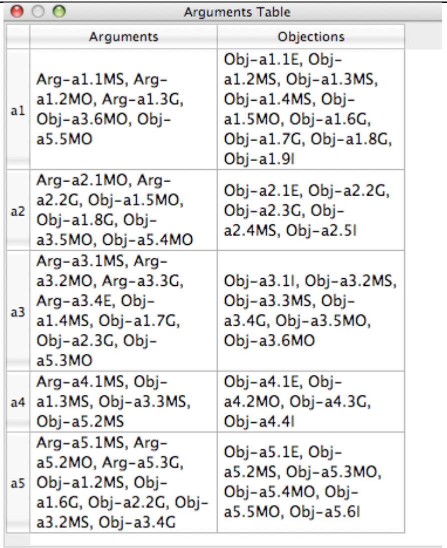
The dictator agent in this test had a value ordering of: {MS>I>{MO,G,E}}

Dictator source code:

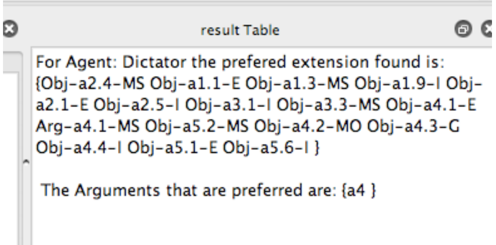
```
vset_agdictator.addValue(v_ms,5);
vset_agdictator.addValue(v_i,4);
vset_agdictator.addValue(v_mo,3);
vset_agdictator.addValue(v_g,3);
vset_agdictator.addValue(v_e,3);
Agent agdictator("Dictator",true,vset_agdictator);
```



```
CQ10: Arg-a1.1 We Should do a1 to promote MS
CQ10: Arg-a1.2 We Should do a1 to promote MO
CQ10: Arg-a1.3 We Should do a1 to promote G
CQ10: Arg-a2.1 We Should do a2 to promote MO
CQ10: Arg-a2.2 We Should do a2 to promote G
CQ10: Arg-a3.1 We Should do a3 to promote MS
CQ10: Arg-a3.2 We Should do a3 to promote MO
CQ10: Arg-a3.3 We Should do a3 to promote G
CQ10: Arg-a3.4 We Should do a3 to promote E
CQ10: Arg-a4.1 We Should do a4 to promote MS
CQ10: Arg-a5.1 We Should do a5 to promote MS
CQ10: Arg-a5.2 We Should do a5 to promote MO
CQ10: Arg-a5.3 We Should do a5 to promote G
CQ8: Obj-a1.1 a1 would demote E
CQ7: Obj-a1.2 action: a5 would promote a1 more than MS
CQ7: Obj-a1.3 action: a4 would promote a1 more than MS
CQ7: Obj-a1.4 action: a3 would promote a1 more than MS
CQ7: Obj-a1.5 action: a2 would promote a1 more than MO
CQ7: Obj-a1.6 action: a5 is as good as a1 with respect G
CQ7: Obj-a1.7 action: a3 is as good as a1 with respect G
CQ7: Obj-a1.8 action: a2 is as good as a1 with respect G
CQ11: Obj-a1.9 a1 precludes I
CQ8: Obj-a2.1 a2 would demote E
CQ7: Obj-a2.2 action: a5 is as good as a2 with respect G
CQ7: Obj-a2.3 action: a3 is as good as a2 with respect G
CQ11: Obj-a2.4 a2 precludes MS
CQ11: Obj-a2.5 a2 precludes I
CQ11: Obj-a3.1 a3 precludes I
CQ7: Obj-a3.2 action: a5 would promote a3 more than MS
CQ7: Obj-a3.3 action: a4 would promote a3 more than MS
CQ7: Obj-a3.4 action: a5 is as good as a3 with respect G
CQ7: Obj-a3.5 action: a2 would promote a3 more than MO
CQ7: Obj-a3.6 action: a1 would promote a3 more than MO
CQ8: Obj-a4.1 a4 would demote E
CQ11: Obj-a4.2 a4 precludes MO
CQ11: Obj-a4.3 a4 precludes G
CQ9: Obj-a4.4 a4 would also demote I
CQ8: Obj-a5.1 a5 would demote E
CQ7: Obj-a5.2 action: a4 would promote a5 more than MS
CQ7: Obj-a5.3 action: a3 would promote a5 more than MO
CQ7: Obj-a5.4 action: a2 would promote a5 more than MO
CQ7: Obj-a5.5 action: a1 would promote a5 more than MO
CQ11: Obj-a5.6 a5 precludes I
```

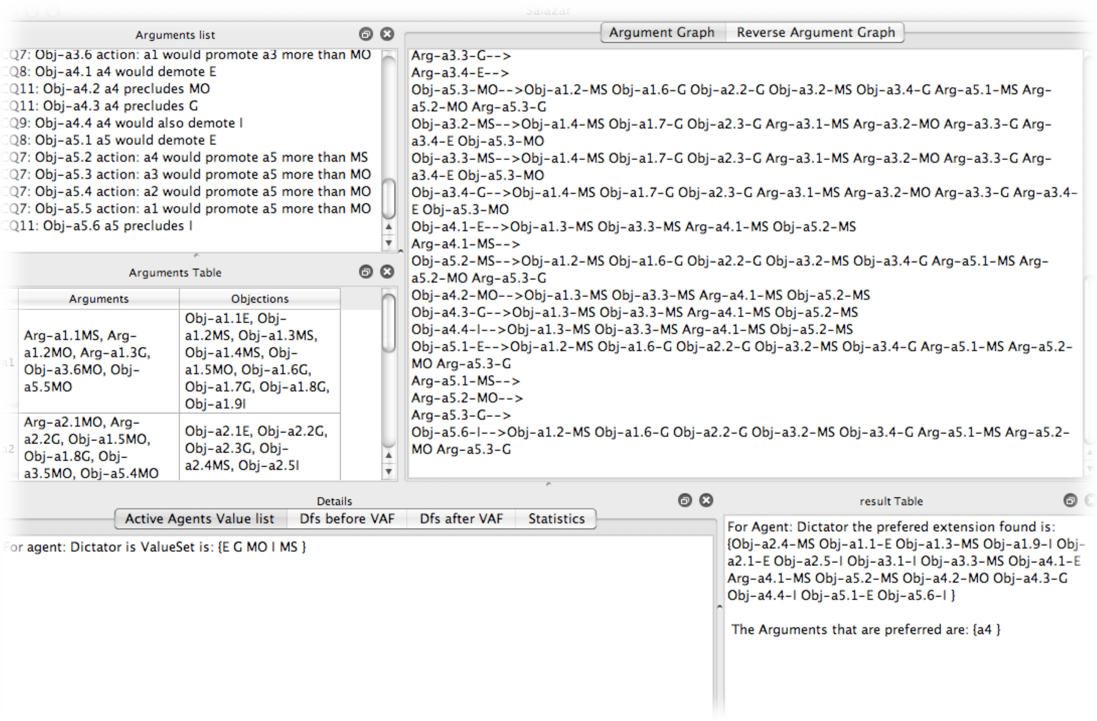


	Arguments	Objections
a1	Arg-a1.1MS, Arg-a1.2MO, Arg-a1.3G, Obj-a3.6MO, Obj-a5.5MO	Obj-a1.1E, Obj-a1.2MS, Obj-a1.3MS, Obj-a1.4MS, Obj-a1.5MO, Obj-a1.6G, Obj-a1.7G, Obj-a1.8G, Obj-a1.9I
a2	Arg-a2.1MO, Arg-a2.2G, Obj-a1.5MO, Obj-a1.8G, Obj-a3.5MO, Obj-a5.4MO	Obj-a2.1E, Obj-a2.2G, Obj-a2.3G, Obj-a2.4MS, Obj-a2.5I
a3	Arg-a3.1MS, Arg-a3.2MO, Arg-a3.3G, Arg-a3.4E, Obj-a1.4MS, Obj-a1.7G, Obj-a2.3G, Obj-a5.3MO	Obj-a3.1I, Obj-a3.2MS, Obj-a3.3MS, Obj-a3.4G, Obj-a3.5MO, Obj-a3.6MO
a4	Arg-a4.1MS, Obj-a1.3MS, Obj-a3.3MS, Obj-a5.2MS	Obj-a4.1E, Obj-a4.2MO, Obj-a4.3G, Obj-a4.4I
a5	Arg-a5.1MS, Arg-a5.2MO, Arg-a5.3G, Obj-a1.2MS, Obj-a1.6G, Obj-a2.2G, Obj-a3.2MS, Obj-a3.4G	Obj-a5.1E, Obj-a5.2MS, Obj-a5.3MO, Obj-a5.4MO, Obj-a5.5MO, Obj-a5.6I



```
For Agent: Dictator the prefered extension found is:
{Obj-a2.4-MS Obj-a1.1-E Obj-a1.3-MS Obj-a1.9-I Obj-
a2.1-E Obj-a2.5-I Obj-a3.1-I Obj-a3.3-MS Obj-a4.1-E
Arg-a4.1-MS Obj-a5.2-MS Obj-a4.2-MO Obj-a4.3-G
Obj-a4.4-I Obj-a5.1-E Obj-a5.6-I }

 The Arguments that are preferred are: {a4 }
```

The argument preferred in this example is {a4} this argument is a maximiser action. Looking at the agent value set it is possible to observe that the agent is in fact a maximiser.



```
CQ7: Obj-a3.6 action: a1 would promote a3 more than MO
CQ8: Obj-a4.1 a4 would demote E
CQ11: Obj-a4.2 a4 precludes MO
CQ11: Obj-a4.3 a4 precludes G
CQ9: Obj-a4.4 a4 would also demote I
CQ8: Obj-a5.1 a5 would demote E
CQ7: Obj-a5.2 action: a4 would promote a5 more than MS
CQ7: Obj-a5.3 action: a3 would promote a5 more than MO
CQ7: Obj-a5.4 action: a2 would promote a5 more than MO
CQ7: Obj-a5.5 action: a1 would promote a5 more than MO
CQ11: Obj-a5.6 a5 precludes I
```

	Arguments	Objections
a1	Arg-a1.1MS, Arg-a1.2MO, Arg-a1.3G, Obj-a3.6MO, Obj-a5.5MO	Obj-a1.1E, Obj-a1.2MS, Obj-a1.3MS, Obj-a1.4MS, Obj-a1.5MO, Obj-a1.6G, Obj-a1.7G, Obj-a1.8G, Obj-a1.9I
a2	Arg-a2.1MO, Arg-a2.2G, Obj-a1.5MO, Obj-a1.8G, Obj-a3.5MO, Obj-a5.4MO	Obj-a2.1E, Obj-a2.2G, Obj-a2.3G, Obj-a2.4MS, Obj-a2.5I

```
Arg-a3.3-G-->
Arg-a3.4-E-->
Obj-a5.3-MO-->Obj-a1.2-MS Obj-a1.6-G Obj-a2.2-G Obj-a3.2-MS Obj-a3.4-G Arg-a5.1-MS Arg-
a5.2-MO Arg-a5.3-G
Obj-a3.2-MS-->Obj-a1.4-MS Obj-a1.7-G Obj-a2.3-G Arg-a3.1-MS Arg-a3.2-MO Arg-a3.3-G Arg-
a3.4-E Obj-a5.3-MO
Obj-a3.3-MS-->Obj-a1.4-MS Obj-a1.7-G Obj-a2.3-G Arg-a3.1-MS Arg-a3.2-MO Arg-a3.3-G Arg-
a3.4-E Obj-a5.3-MO
Obj-a3.4-G-->Obj-a1.4-MS Obj-a1.7-G Obj-a2.3-G Arg-a3.1-MS Arg-a3.2-MO Arg-a3.3-G Arg-a3.4-
E Obj-a5.3-MO
Obj-a4.1-E-->Obj-a1.3-MS Obj-a3.3-MS Arg-a4.1-MS Obj-a5.2-MS
Arg-a4.1-MS-->
Obj-a5.2-MS-->Obj-a1.2-MS Obj-a1.6-G Obj-a2.2-G Obj-a3.2-MS Obj-a3.4-G Arg-a5.1-MS Arg-
a5.2-MO Arg-a5.3-G
Obj-a4.2-MO-->Obj-a1.3-MS Obj-a3.3-MS Arg-a4.1-MS Obj-a5.2-MS
Obj-a4.3-G-->Obj-a1.3-MS Obj-a3.3-MS Arg-a4.1-MS Obj-a5.2-MS
Obj-a4.4-I-->Obj-a1.3-MS Obj-a3.3-MS Arg-a4.1-MS Obj-a5.2-MS
Obj-a5.1-E-->Obj-a1.2-MS Obj-a1.6-G Obj-a2.2-G Obj-a3.2-MS Obj-a3.4-G Arg-a5.1-MS Arg-a5.2-
MO Arg-a5.3-G
Arg-a5.1-MS-->
Arg-a5.2-MO-->
Arg-a5.3-G-->
Obj-a5.6-I-->Obj-a1.2-MS Obj-a1.6-G Obj-a2.2-G Obj-a3.2-MS Obj-a3.4-G Arg-a5.1-MS Arg-a5.2-
MO Arg-a5.3-G
```

```
For agent: Dictator is ValueSet is: {E G MO I MS }
```

```
For Agent: Dictator the prefered extension found is:
{Obj-a2.4-MS Obj-a1.1-E Obj-a1.3-MS Obj-a1.9-I Obj-
a2.1-E Obj-a2.5-I Obj-a3.1-I Obj-a3.3-MS Obj-a4.1-E
Arg-a4.1-MS Obj-a5.2-MS Obj-a4.2-MO Obj-a4.3-G
Obj-a4.4-I Obj-a5.1-E Obj-a5.6-I }

 The Arguments that are preferred are: {a4 }
```

– Taking Game:

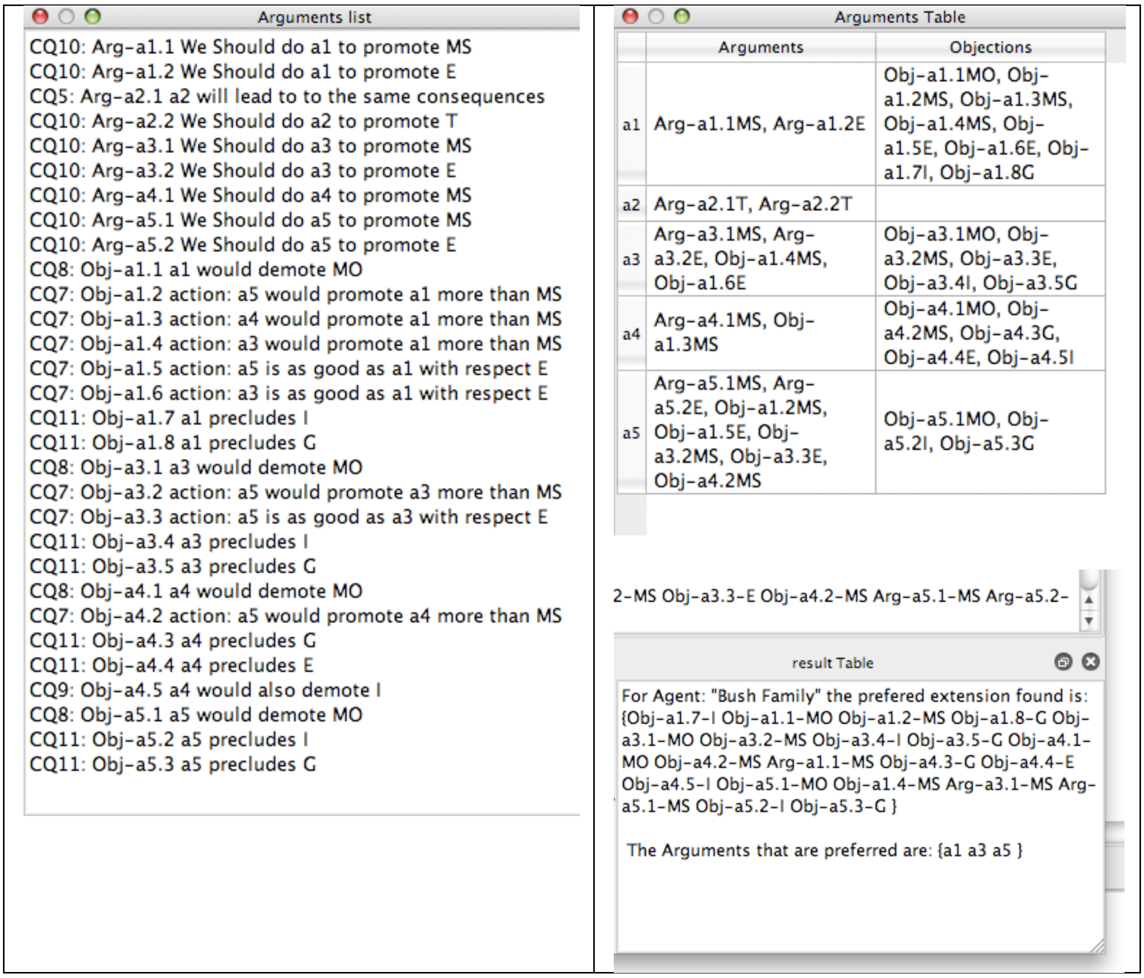
In this test it will be explored the framing effects of [1].

– With start at [0,100] a2:

The dictator agent in this test had a value ordering of: {MS>I>{MO,G,E}}.

Dictator source code:

```
ValueSet vset_agdictator;
vset_agdictator.addValue(v_ms,5);
vset_agdictator.addValue(v_i,4);
vset_agdictator.addValue(v_mo,3);
vset_agdictator.addValue(v_g,3);
vset_agdictator.addValue(v_e,3);
Agent agdictator("\"Bush Family\"",true,vset_agdictator);
```



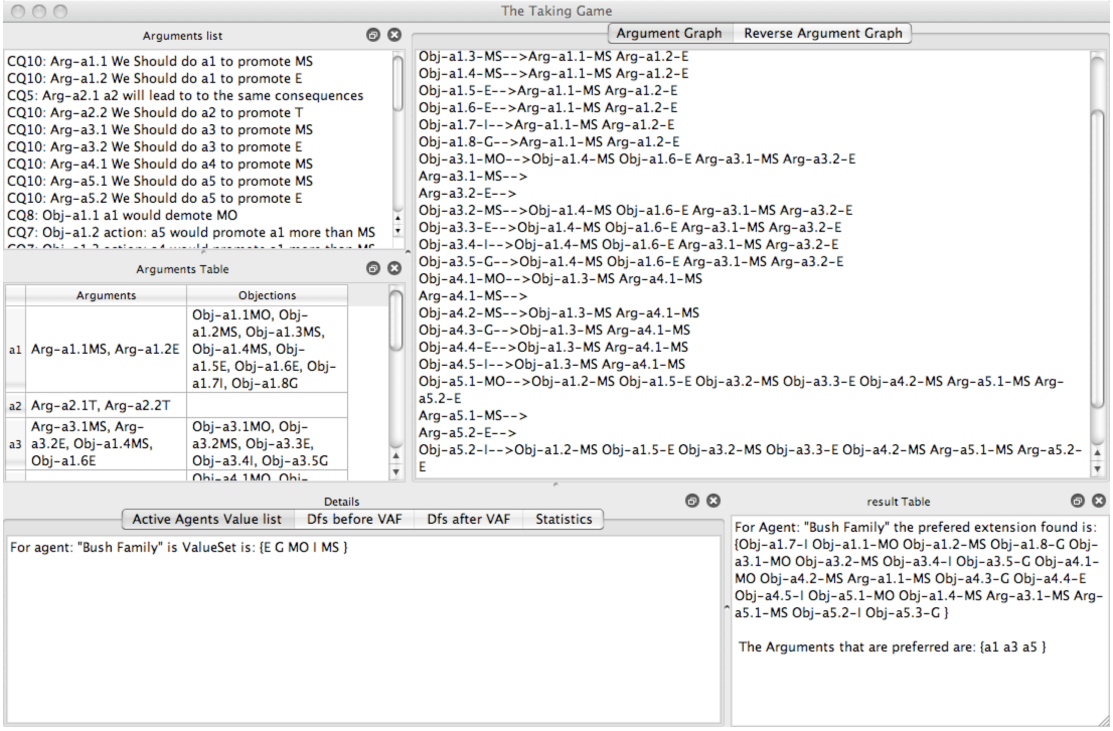
```
CQ10: Arg-a1.1 We Should do a1 to promote MS
CQ10: Arg-a1.2 We Should do a1 to promote E
CQ5: Arg-a2.1 a2 will lead to to the same consequences
CQ10: Arg-a2.2 We Should do a2 to promote T
CQ10: Arg-a3.1 We Should do a3 to promote MS
CQ10: Arg-a3.2 We Should do a3 to promote E
CQ10: Arg-a4.1 We Should do a4 to promote MS
CQ10: Arg-a5.1 We Should do a5 to promote MS
CQ10: Arg-a5.2 We Should do a5 to promote E
CQ8: Obj-a1.1 a1 would demote MO
CQ7: Obj-a1.2 action: a5 would promote a1 more than MS
CQ7: Obj-a1.3 action: a4 would promote a1 more than MS
CQ7: Obj-a1.4 action: a3 would promote a1 more than MS
CQ7: Obj-a1.5 action: a5 is as good as a1 with respect E
CQ7: Obj-a1.6 action: a3 is as good as a1 with respect E
CQ11: Obj-a1.7 a1 precludes I
CQ11: Obj-a1.8 a1 precludes G
CQ8: Obj-a3.1 a3 would demote MO
CQ7: Obj-a3.2 action: a5 would promote a3 more than MS
CQ7: Obj-a3.3 action: a5 is as good as a3 with respect E
CQ11: Obj-a3.4 a3 precludes I
CQ11: Obj-a3.5 a3 precludes G
CQ8: Obj-a4.1 a4 would demote MO
CQ7: Obj-a4.2 action: a5 would promote a4 more than MS
CQ11: Obj-a4.3 a4 precludes G
CQ11: Obj-a4.4 a4 precludes E
CQ9: Obj-a4.5 a4 would also demote I
CQ8: Obj-a5.1 a5 would demote MO
CQ11: Obj-a5.2 a5 precludes I
CQ11: Obj-a5.3 a5 precludes G
```

	Arguments	Objections
a1	Arg-a1.1MS, Arg-a1.2E	Obj-a1.1MO, Obj-a1.2MS, Obj-a1.3MS, Obj-a1.4MS, Obj-a1.5E, Obj-a1.6E, Obj-a1.7I, Obj-a1.8G
a2	Arg-a2.1T, Arg-a2.2T	
a3	Arg-a3.1MS, Arg-a3.2E, Obj-a1.4MS, Obj-a1.6E	Obj-a3.1MO, Obj-a3.2MS, Obj-a3.3E, Obj-a3.4I, Obj-a3.5G
a4	Arg-a4.1MS, Obj-a1.3MS	Obj-a4.1MO, Obj-a4.2MS, Obj-a4.3G, Obj-a4.4E, Obj-a4.5I
a5	Arg-a5.1MS, Arg-a5.2E, Obj-a1.2MS, Obj-a1.5E, Obj-a3.2MS, Obj-a3.3E, Obj-a4.2MS	Obj-a5.1MO, Obj-a5.2I, Obj-a5.3G

```
2-MS Obj-a3.3-E Obj-a4.2-MS Arg-a5.1-MS Arg-a5.2-
```

```
For Agent: "Bush Family" the prefered extension found is:
{Obj-a1.7-I Obj-a1.1-MO Obj-a1.2-MS Obj-a1.8-G Obj-
a3.1-MO Obj-a3.2-MS Obj-a3.4-I Obj-a3.5-G Obj-a4.1-
MO Obj-a4.2-MS Arg-a1.1-MS Obj-a4.3-G Obj-a4.4-E
Obj-a4.5-I Obj-a5.1-MO Obj-a1.4-MS Arg-a3.1-MS Arg-
a5.1-MS Obj-a5.2-I Obj-a5.3-G }

The Arguments that are preferred are: {a1 a3 a5 }
```

In this example it possible to observe that with the value set chosen for the dictator, the values preferred would be {a1,a3,a5}.



```
CQ10: Arg-a1.1 We Should do a1 to promote MS
CQ10: Arg-a1.2 We Should do a1 to promote E
CQ5: Arg-a2.1 a2 will lead to to the same consequences
CQ10: Arg-a2.2 We Should do a2 to promote T
CQ10: Arg-a3.1 We Should do a3 to promote MS
CQ10: Arg-a3.2 We Should do a3 to promote E
CQ10: Arg-a4.1 We Should do a4 to promote MS
CQ10: Arg-a5.1 We Should do a5 to promote MS
CQ10: Arg-a5.2 We Should do a5 to promote E
CQ8: Obj-a1.1 a1 would demote MO
CQ7: Obj-a1.2 action: a5 would promote a1 more than MS
```

	Arguments	Objections
a1	Arg-a1.1MS, Arg-a1.2E	Obj-a1.1MO, Obj-a1.2MS, Obj-a1.3MS, Obj-a1.4MS, Obj-a1.5E, Obj-a1.6E, Obj-a1.7I, Obj-a1.8G
a2	Arg-a2.1T, Arg-a2.2T	
a3	Arg-a3.1MS, Arg-a3.2E, Obj-a1.4MS, Obj-a1.6E	Obj-a3.1MO, Obj-a3.2MS, Obj-a3.3E, Obj-a3.4I, Obj-a3.5G

```
Obj-a1.3-MS-->Arg-a1.1-MS Arg-a1.2-E
Obj-a1.4-MS-->Arg-a1.1-MS Arg-a1.2-E
Obj-a1.5-E-->Arg-a1.1-MS Arg-a1.2-E
Obj-a1.6-E-->Arg-a1.1-MS Arg-a1.2-E
Obj-a1.7-I-->Arg-a1.1-MS Arg-a1.2-E
Obj-a1.8-G-->Arg-a1.1-MS Arg-a1.2-E
Obj-a3.1-MO-->Obj-a1.4-MS Obj-a1.6-E Arg-a3.1-MS Arg-a3.2-E
Arg-a3.1-MS-->
Arg-a3.2-E-->
Obj-a3.2-MS-->Obj-a1.4-MS Obj-a1.6-E Arg-a3.1-MS Arg-a3.2-E
Obj-a3.3-E-->Obj-a1.4-MS Obj-a1.6-E Arg-a3.1-MS Arg-a3.2-E
Obj-a3.4-I-->Obj-a1.4-MS Obj-a1.6-E Arg-a3.1-MS Arg-a3.2-E
Obj-a3.5-G-->Obj-a1.4-MS Obj-a1.6-E Arg-a3.1-MS Arg-a3.2-E
Obj-a4.1-MO-->Obj-a1.3-MS Arg-a4.1-MS
Arg-a4.1-MS-->
Obj-a4.2-MS-->Obj-a1.3-MS Arg-a4.1-MS
Obj-a4.3-G-->Obj-a1.3-MS Arg-a4.1-MS
Obj-a4.4-E-->Obj-a1.3-MS Arg-a4.1-MS
Obj-a4.5-I-->Obj-a1.3-MS Arg-a4.1-MS
Obj-a5.1-MO-->Obj-a1.2-MS Obj-a1.5-E Obj-a3.2-MS Obj-a3.3-E Obj-a4.2-MS Arg-a5.1-MS Arg-
a5.2-E
Arg-a5.1-MS-->
Arg-a5.2-E-->
Obj-a5.2-I-->Obj-a1.2-MS Obj-a1.5-E Obj-a3.2-MS Obj-a3.3-E Obj-a4.2-MS Arg-a5.1-MS Arg-a5.2-
E
```

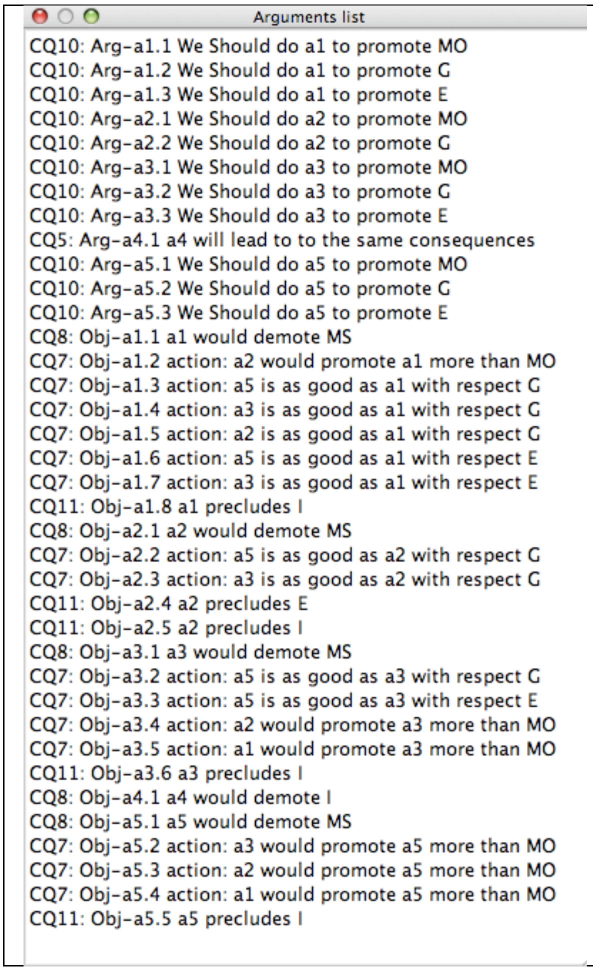
```
For agent: "Bush Family" is ValueSet is: {E G MO I MS }
```

```
For Agent: "Bush Family" the prefered extension found is:
{Obj-a1.7-I Obj-a1.1-MO Obj-a1.2-MS Obj-a1.8-G Obj-
a3.1-MO Obj-a3.2-MS Obj-a3.4-I Obj-a3.5-G Obj-a4.1-
MO Obj-a4.2-MS Arg-a1.1-MS Obj-a4.3-G Obj-a4.4-E
Obj-a4.5-I Obj-a5.1-MO Obj-a1.4-MS Arg-a3.1-MS Arg-
a5.1-MS Obj-a5.2-I Obj-a5.3-G }

The Arguments that are preferred are: {a1 a3 a5 }
```

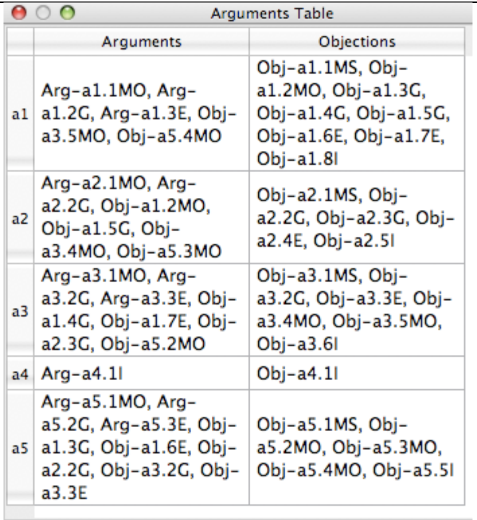
– Taking game starting at [100,0] a4;
The dictator agent in this test had a value ordering of: {MS>I>{MO,G,E}}

Dictator source code:

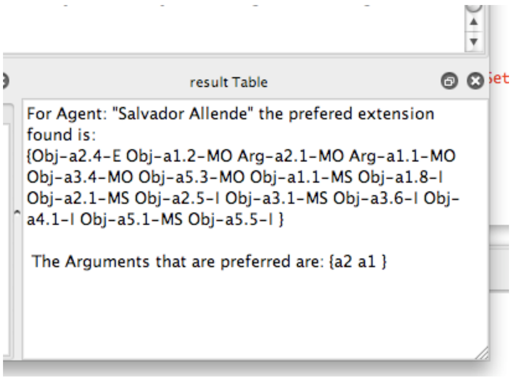
```
ValueSet vset_agdictator;
vset_agdictator.addValue(v_mo,5);
vset_agdictator.addValue(v_i,3);
vset_agdictator.addValue(v_ms,2);
vset_agdictator.addValue(v_g,4);
vset_agdictator.addValue(v_e,4);
Agent agdictator("\"Salvador Allende\"",true,vset_agdictator);
```



```
CQ10: Arg-a1.1 We Should do a1 to promote MO
CQ10: Arg-a1.2 We Should do a1 to promote G
CQ10: Arg-a1.3 We Should do a1 to promote E
CQ10: Arg-a2.1 We Should do a2 to promote MO
CQ10: Arg-a2.2 We Should do a2 to promote G
CQ10: Arg-a3.1 We Should do a3 to promote MO
CQ10: Arg-a3.2 We Should do a3 to promote G
CQ10: Arg-a3.3 We Should do a3 to promote E
CQ5: Arg-a4.1 a4 will lead to to the same consequences
CQ10: Arg-a5.1 We Should do a5 to promote MO
CQ10: Arg-a5.2 We Should do a5 to promote G
CQ10: Arg-a5.3 We Should do a5 to promote E
CQ8: Obj-a1.1 a1 would demote MS
CQ7: Obj-a1.2 action: a2 would promote a1 more than MO
CQ7: Obj-a1.3 action: a5 is as good as a1 with respect G
CQ7: Obj-a1.4 action: a3 is as good as a1 with respect G
CQ7: Obj-a1.5 action: a2 is as good as a1 with respect G
CQ7: Obj-a1.6 action: a5 is as good as a1 with respect E
CQ7: Obj-a1.7 action: a3 is as good as a1 with respect E
CQ11: Obj-a1.8 a1 precludes I
CQ8: Obj-a2.1 a2 would demote MS
CQ7: Obj-a2.2 action: a5 is as good as a2 with respect G
CQ7: Obj-a2.3 action: a3 is as good as a2 with respect G
CQ11: Obj-a2.4 a2 precludes E
CQ11: Obj-a2.5 a2 precludes I
CQ8: Obj-a3.1 a3 would demote MS
CQ7: Obj-a3.2 action: a5 is as good as a3 with respect G
CQ7: Obj-a3.3 action: a5 is as good as a3 with respect E
CQ7: Obj-a3.4 action: a2 would promote a3 more than MO
CQ7: Obj-a3.5 action: a1 would promote a3 more than MO
CQ11: Obj-a3.6 a3 precludes I
CQ8: Obj-a4.1 a4 would demote I
CQ8: Obj-a5.1 a5 would demote MS
CQ7: Obj-a5.2 action: a3 would promote a5 more than MO
CQ7: Obj-a5.3 action: a2 would promote a5 more than MO
CQ7: Obj-a5.4 action: a1 would promote a5 more than MO
CQ11: Obj-a5.5 a5 precludes I
```

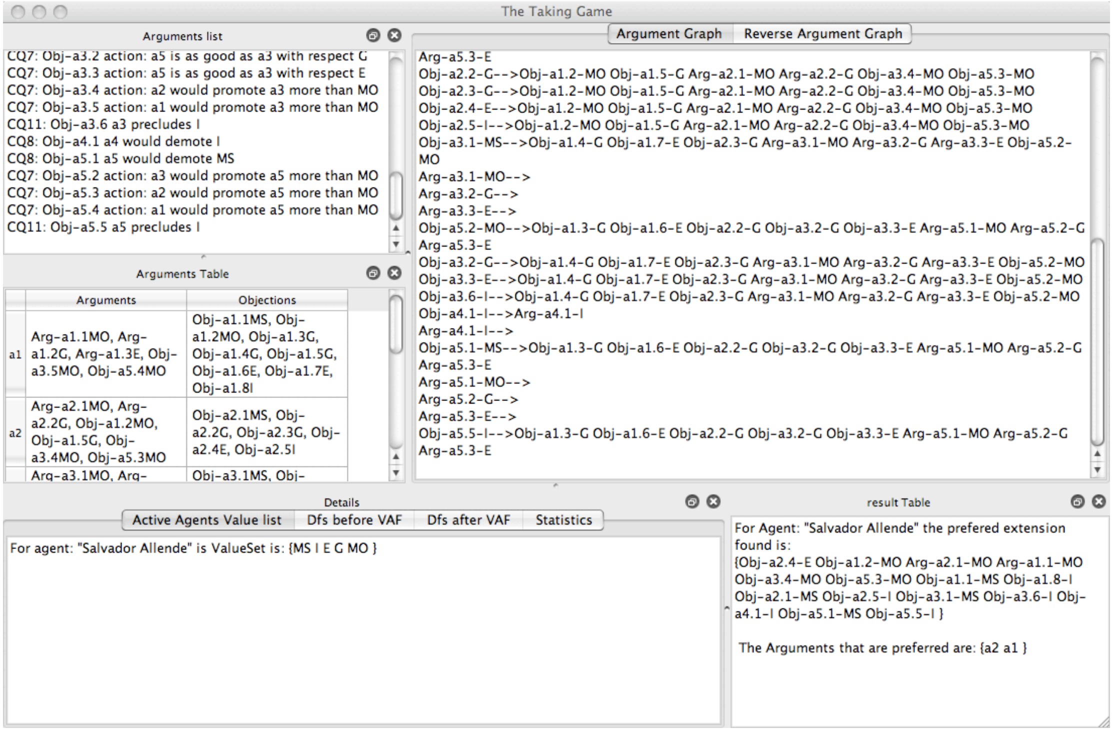


	Arguments	Objections
a1	Arg-a1.1MO, Arg-a1.2G, Arg-a1.3E, Obj-a3.5MO, Obj-a5.4MO	Obj-a1.1MS, Obj-a1.2MO, Obj-a1.3G, Obj-a1.4G, Obj-a1.5G, Obj-a1.6E, Obj-a1.7E, Obj-a1.8I
a2	Arg-a2.1MO, Arg-a2.2G, Obj-a1.2MO, Obj-a1.5G, Obj-a3.4MO, Obj-a5.3MO	Obj-a2.1MS, Obj-a2.2G, Obj-a2.3G, Obj-a2.4E, Obj-a2.5I
a3	Arg-a3.1MO, Arg-a3.2G, Arg-a3.3E, Obj-a1.4G, Obj-a1.7E, Obj-a2.3G, Obj-a5.2MO	Obj-a3.1MS, Obj-a3.2G, Obj-a3.3E, Obj-a3.4MO, Obj-a3.5MO, Obj-a3.6I
a4	Arg-a4.1I	Obj-a4.1I
a5	Arg-a5.1MO, Arg-a5.2G, Arg-a5.3E, Obj-a1.3G, Obj-a1.6E, Obj-a2.2G, Obj-a3.2G, Obj-a3.3E	Obj-a5.1MS, Obj-a5.2MO, Obj-a5.3MO, Obj-a5.4MO, Obj-a5.5I



```
For Agent: "Salvador Allende" the prefered extension
found is:
{Obj-a2.4-E Obj-a1.2-MO Arg-a2.1-MO Arg-a1.1-MO
Obj-a3.4-MO Obj-a5.3-MO Obj-a1.1-MS Obj-a1.8-I
Obj-a2.1-MS Obj-a2.5-I Obj-a3.1-MS Obj-a3.6-I Obj-
a4.1-I Obj-a5.1-MS Obj-a5.5-I }

 The Arguments that are preferred are: {a2 a1 }
```

In this example it possible to observe that with the value set chosen for the dictator, the values preferred would be {a2,a1}.



```
CQ7: Obj-a3.2 action: a5 is as good as a3 with respect G
CQ7: Obj-a3.3 action: a5 is as good as a3 with respect E
CQ7: Obj-a3.4 action: a2 would promote a3 more than MO
CQ7: Obj-a3.5 action: a1 would promote a3 more than MO
CQ11: Obj-a3.6 a3 precludes I
CQ8: Obj-a4.1 a4 would demote I
CQ8: Obj-a5.1 a5 would demote MS
CQ7: Obj-a5.2 action: a3 would promote a5 more than MO
CQ7: Obj-a5.3 action: a2 would promote a5 more than MO
CQ7: Obj-a5.4 action: a1 would promote a5 more than MO
CQ11: Obj-a5.5 a5 precludes I
```

	Arguments	Objections
a1	Arg-a1.1MO, Arg-a1.2G, Arg-a1.3E, Obj-a3.5MO, Obj-a5.4MO	Obj-a1.1MS, Obj-a1.2MO, Obj-a1.3G, Obj-a1.4G, Obj-a1.5G, Obj-a1.6E, Obj-a1.7E, Obj-a1.8I
a2	Arg-a2.1MO, Arg-a2.2G, Obj-a1.2MO, Obj-a1.5G, Obj-a3.4MO, Obj-a5.3MO	Obj-a2.1MS, Obj-a2.2G, Obj-a2.3G, Obj-a2.4E, Obj-a2.5I
	Arg-a3.1MO, Arg-	Obj-a3.1MS, Obj-

```
Arg-a5.3-E
Obj-a2.2-G-->Obj-a1.2-MO Obj-a1.5-G Arg-a2.1-MO Arg-a2.2-G Obj-a3.4-MO Obj-a5.3-MO
Obj-a2.3-G-->Obj-a1.2-MO Obj-a1.5-G Arg-a2.1-MO Arg-a2.2-G Obj-a3.4-MO Obj-a5.3-MO
Obj-a2.4-E-->Obj-a1.2-MO Obj-a1.5-G Arg-a2.1-MO Arg-a2.2-G Obj-a3.4-MO Obj-a5.3-MO
Obj-a2.5-I-->Obj-a1.2-MO Obj-a1.5-G Arg-a2.1-MO Arg-a2.2-G Obj-a3.4-MO Obj-a5.3-MO
Obj-a3.1-MS-->Obj-a1.4-G Obj-a1.7-E Obj-a2.3-G Arg-a3.1-MO Arg-a3.2-G Arg-a3.3-E Obj-a5.2-
MO
Arg-a3.1-MO-->
Arg-a3.2-G-->
Arg-a3.3-E-->
Obj-a5.2-MO-->Obj-a1.3-G Obj-a1.6-E Obj-a2.2-G Obj-a3.2-G Obj-a3.3-E Arg-a5.1-MO Arg-a5.2-G
Arg-a5.3-E
Obj-a3.2-G-->Obj-a1.4-G Obj-a1.7-E Obj-a2.3-G Arg-a3.1-MO Arg-a3.2-G Arg-a3.3-E Obj-a5.2-MO
Obj-a3.3-E-->Obj-a1.4-G Obj-a1.7-E Obj-a2.3-G Arg-a3.1-MO Arg-a3.2-G Arg-a3.3-E Obj-a5.2-MO
Obj-a3.6-I-->Obj-a1.4-G Obj-a1.7-E Obj-a2.3-G Arg-a3.1-MO Arg-a3.2-G Arg-a3.3-E Obj-a5.2-MO
Obj-a4.1-I-->Arg-a4.1-I
Arg-a4.1-I-->
Obj-a5.1-MS-->Obj-a1.3-G Obj-a1.6-E Obj-a2.2-G Obj-a3.2-G Obj-a3.3-E Arg-a5.1-MO Arg-a5.2-G
Arg-a5.3-E
Arg-a5.1-MO-->
Arg-a5.2-G-->
Arg-a5.3-E-->
Obj-a5.5-I-->Obj-a1.3-G Obj-a1.6-E Obj-a2.2-G Obj-a3.2-G Obj-a3.3-E Arg-a5.1-MO Arg-a5.2-G
Arg-a5.3-E
```

```
For agent: "Salvador Allende" is ValueSet is: {MS I E G MO }
```

```
For Agent: "Salvador Allende" the prefered extension
found is:
{Obj-a2.4-E Obj-a1.2-MO Arg-a2.1-MO Arg-a1.1-MO
Obj-a3.4-MO Obj-a5.3-MO Obj-a1.1-MS Obj-a1.8-I
Obj-a2.1-MS Obj-a2.5-I Obj-a3.1-MS Obj-a3.6-I Obj-
a4.1-I Obj-a5.1-MS Obj-a5.5-I }

 The Arguments that are preferred are: {a2 a1 }
```

Evaluation

To evaluate the project it was decided to create a table to show how well defined where the project is assessed through its different characteristics and how efficiently it displays the results.

The table will pretend to assess the project displaying the results in a scale of 1 to 5 where 5 is excellent and 1 very poor or inexistent.

It will be measured also how well the information outputted is displayed and readable to the user.

Finally a list will created that will show areas in the project of possible improvement.

	1	2	3	4	5
Main Methods Correctness					
AATsTransgenerator					x
VAFGenerator				x	
ArgGenerator					x
DiGraph			x		
CQxSet					x
ValueSet					x
DFS				x	
Other Methods					x
Readability and Use of the simulator					
Arguments and Objections readability					x
AATS diagram from input readability	x				
VAF graph readability		x			
Options present in simulator	x				
Information displayed					x
Quality of the information displayed				x	

Possible improvements on the project:

DFS: besides the efficiency in which a dfs is created this algorithm from [10] modified to this project cannot detect certain cycles as explained before in design and realization, this class wasn't modified since it was no time left to find or create an algorithm as efficient as this one and more correct at the same time. Because of this issue it is necessary to rebuild the DFS every time a cycle is removed in the VAF graph. This represents efficiency issues not visible with this experiment, but if the program is used with bigger experiments this will represent an unnecessary delay.

VAFGenerator: If the DFS class is fixed, this class can create preferred extensions in a more efficient way, since it would not have to create a DFS list every time a cycle is removed.

If the project was longer it is possible to merge the polychromatic cycle and the dichromatic cycle methods together, trading memory for run-time. Making this algorithm more efficient.

DiGraph: this class has memory management issues between it self and its friend class Node, this class besides not creating references in the nodes to create the edges in the graph, it also does not allow the deletion of Nodes with edges at the same time, resulting in a kernel exception error. But the lack of time and the difficulties encountered to implement the class VAFGenerator didn't allow to fix these two issues.

MultiThreading: The program does not have any threads in it self, since it wasn't found time to implement them. The simulator could also see some performance improvements if it provided with thread handling.

Error Handling: The program does not have any error handling in it self, this is one of the reasons why there is no possibility to let the user to change some options or the AATS itself.

The inexistence of error handling in the project was also caused by lack of time to implement it.

Gui Layout: The layout of the GUI was seriously affected by the by all the previous factors, and to create a GUI, it was necessary to chose between finish to implement a general algorithm that would solve VAF or to create a more readable GUI that a user could use. Since this simulator is mainly to be used for research it was decided that it was more important to create a simple gui that would help the research as much as possible, and finish the implementation of the VAF. If more time were given to create this program, it would be possible to create a drag-n-drop interface that would allow the creation of AATS diagram and could display the Graph for the VAF.

Learning Points

Carrying out a research project is challenging task in itself, since there is no knowledge if the project will be successful or if instead of a piece of software it will be delivered a document and a Powerpoint presentation.
Not having an idea of the dimension and responsibility that exists in research projects I realize today that I took an immense risk, but a risk that gave me an immense pleasure and gave me a an excellent background in all the theory involved around this project.

To deliver a complete specification and design and implement it was a complete challenge specially because the existing documentation around the subject isn't big.

To have a supervisor and a set deadline at all times gave a good feedback of how it would be to work for a company. It would not be possible to accomplish all the deadlines without the supervisor helps and organization (Dr Katie Atkinson), the existing relation with the supervisor allowed all success and the implementation of the project.

Before this project I didn't possess any knowledge about AATS or VAF, I would even think that Argumentation was an area of research, after this 8 months I became quite familiar with this two frameworks and the theory behind them, I realize now their use, and important this are and the frameworks are to our society.

My knowledge about graph theory is allot more extended, then before, not just because the code implemented here but for all the research done, while searching for a way to implement VAF and AATS, it read about topological sort, DFS, strong connectivity in graphs, different ways of representing a graph, differences in representing directed graphs and normal graphs, and the problems that directed graphs constitute.

Besides of all this challenges it was also decided that ANSI C++ would be the language of choice, while Qt framework would allow having the program running on different OS's, This was one of the major challenges in the project since it is very different to program in C++ or in Java, the language that I was used to work.

Professional Issues

During the development of this project, several principles were taken into consideration for implementing the project. According to the British Computer Society (BCS), each software programmer-developer should try to apply those principles in the work, which is produced. Some of that involves, data manipulation, data misuse, data protection etc.

Furthermore, as the Data Protection and misuse act state, all data should be kept and manipulated for the purposes of their nature.

This simulator were created taking in consideration the programming principle of information hiding, were the all the source code is hidden from the implementation, not allowing data misuse, or anything that might modify the results created.

No unauthorized persons should be allowed to view the source of the program and therefore be able to change the program input. Since this is the only to change the program input.

Moreover, the law states that all data kept and provided should be valid and up-to-date. Since there is no way changing the input in the program, all the testing done in the software proves the correctness of the output. Leaving the information valid and correct at anytime.

Bibliography

References:

[1] K. Atkinson and T. Bench-Capon (2008): Value-based arguments in the dictator game. In: Proceedings of the Fourth Multidisciplinary Workshop on Advances in Preference Handling (M-PREF 2008), Chicago, USA, pp. 1-6. AAAI Press, Technical Report WS-08-09

[2] Wikipedia link: http://en.wikipedia.org/wiki/Dictator_game#cite_note-Henrich-0, Last time this link was visited: 19/11/2008

[3] T.J.M. Bench-Capon, (2003). Persuasion in Practical Argument Using Value Based Argumentation Frameworks. Journal of Logic and Computation. Volume 13 No 3 pp429-448.

[4] K. Atkinson and T. Bench-Capon (2007): Practical reasoning as presumptive argumentation using action based alternating transition systems. Artificial Intelligence. Special Issue on Argumentation, edited by P. E. Dunne and T. Bench-Capon. Vol. 171 (10-15), pp. 855-874.

[5] Trolltech creator of qt4 cross-platform application framework, that can be ported to Apple Mac OS, Windows, Windows CE, Linux (KDE is built using this framework), Embedded Linux, www.trolltech.com

[6]

[7] Wikipedia: Test plan, Link: http://en.wikipedia.org/wiki/Test_plan, Last time this link was visited: 19/11/2008

[8] Wikipedia IEEE 829-1998, Link: http://en.wikipedia.org/wiki/IEEE_829, Last time this link was visited: 19/11/2008

[9] Evaluation Process: link: http://issco-www.unige.ch/ewg95/node73.html, Last time this link was visited:19/11/2008

[10] Sedgewick, Robert, 2006 Algorithms in C++ part 5 graph algorithms. – 3rd edition

[11]ULCS-02-001: Trevor J. M. Bench-Capon and Paul E. Dunne: Value Based Argumentation Frameworks.

[12] Trevor J.M. Bench-Capon: Valued Based Argumentations Frameworks, 2002 link: http://arxiv.org/pdf/cs/0207059v1 last visited 16/04/2009 or http://people.cs.uu.nl/henry/add02/papers/tbc.ps last visited 16/04/2009

[13] Wikipedia Transpose Graph: http://en.wikipedia.org/wiki/Transpose_graph last visited 17/04/2009

[14] Transpose graph link: http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/graphIntro.htm, last visited 17/04/2009

[15] Introduction to algorithms / Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, 1990 Cambridge, MA. : MIT Press, 2001. 2nd edition.

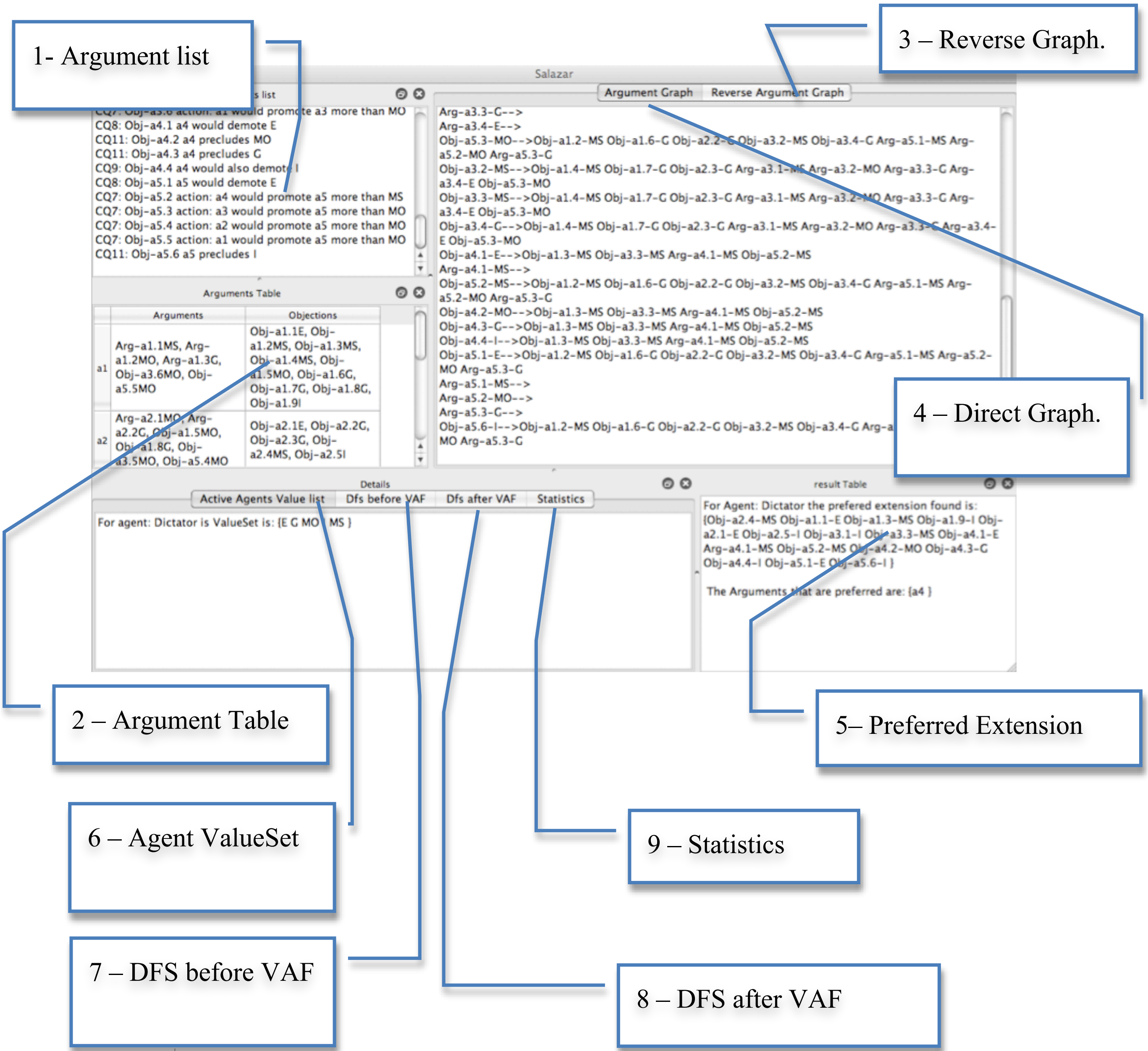
Appendices:

Appendix A – User Manual:

In this appendix it will be described the different sections of the software, how to install Qt and how to modify it's source code to insert new AATS.

User Manual:

To understand the software it is necessary to explain the different sections that exist in the GUI, and what information is displayed in it.



```
CQ7: Obj-a3.6 action: a1 would promote a3 more than MO
CQ8: Obj-a4.1 a4 would demote E
CQ11: Obj-a4.2 a4 precludes MO
CQ11: Obj-a4.3 a4 precludes G
CQ9: Obj-a4.4 a4 would also demote I
CQ8: Obj-a5.1 a5 would demote E
CQ7: Obj-a5.2 action: a4 would promote a5 more than MS
CQ7: Obj-a5.3 action: a3 would promote a5 more than MO
CQ7: Obj-a5.4 action: a2 would promote a5 more than MO
CQ7: Obj-a5.5 action: a1 would promote a5 more than MO
CQ11: Obj-a5.6 a5 precludes I
```

	Arguments	Objections
a1	Arg-a1.1MS, Arg-a1.2MO, Arg-a1.3G, Obj-a3.6MO, Obj-a5.5MO	Obj-a1.1E, Obj-a1.2MS, Obj-a1.3MS, Obj-a1.4MS, Obj-a1.5MO, Obj-a1.6G, Obj-a1.7G, Obj-a1.8G, Obj-a1.9I
a2	Arg-a2.1MO, Arg-a2.2G, Obj-a1.5MO, Obj-a1.8G, Obj-a3.5MO, Obj-a5.4MO	Obj-a2.1E, Obj-a2.2G, Obj-a2.3G, Obj-a2.4MS, Obj-a2.5I

```
Arg-a3.3-G-->
Arg-a3.4-E-->
Obj-a5.3-MO-->Obj-a1.2-MS Obj-a1.6-G Obj-a2.2-G Obj-a3.2-MS Obj-a3.4-G Arg-a5.1-MS Arg-
a5.2-MO Arg-a5.3-G
Obj-a3.2-MS-->Obj-a1.4-MS Obj-a1.7-G Obj-a2.3-G Arg-a3.1-MS Arg-a3.2-MO Arg-a3.3-G Arg-
a3.4-E Obj-a5.3-MO
Obj-a3.3-MS-->Obj-a1.4-MS Obj-a1.7-G Obj-a2.3-G Arg-a3.1-MS Arg-a3.2-MO Arg-a3.3-G Arg-
a3.4-E Obj-a5.3-MO
Obj-a3.4-G-->Obj-a1.4-MS Obj-a1.7-G Obj-a2.3-G Arg-a3.1-MS Arg-a3.2-MO Arg-a3.3-G Arg-a3.4-
E Obj-a5.3-MO
Obj-a4.1-E-->Obj-a1.3-MS Obj-a3.3-MS Arg-a4.1-MS Obj-a5.2-MS
Arg-a4.1-MS-->
Obj-a5.2-MS-->Obj-a1.2-MS Obj-a1.6-G Obj-a2.2-G Obj-a3.2-MS Obj-a3.4-G Arg-a5.1-MS Arg-
a5.2-MO Arg-a5.3-G
Obj-a4.2-MO-->Obj-a1.3-MS Obj-a3.3-MS Arg-a4.1-MS Obj-a5.2-MS
Obj-a4.3-G-->Obj-a1.3-MS Obj-a3.3-MS Arg-a4.1-MS Obj-a5.2-MS
Obj-a4.4-I-->Obj-a1.3-MS Obj-a3.3-MS Arg-a4.1-MS Obj-a5.2-MS
Obj-a5.1-E-->Obj-a1.2-MS Obj-a1.6-G Obj-a2.2-G Obj-a3.2-MS Obj-a3.4-G Arg-a5.1-MS Arg-a5.2-
MO Arg-a5.3-G
Arg-a5.1-MS-->
Arg-a5.2-MO-->
Arg-a5.3-G-->
Obj-a5.6-I-->Obj-a1.2-MS Obj-a1.6-G Obj-a2.2-G Obj-a3.2-MS Obj-a3.4-G Arg-a
MO Arg-a5.3-G
```

```
For agent: Dictator is ValueSet is: {E G MO I MS }
```

```
For Agent: Dictator the prefered extension found is:
{Obj-a2.4-MS Obj-a1.1-E Obj-a1.3-MS Obj-a1.9-I Obj-
a2.1-E Obj-a2.5-I Obj-a3.1-I Obj-a3.3-MS Obj-a4.1-E
Arg-a4.1-MS Obj-a5.2-MS Obj-a4.2-MO Obj-a4.3-G
Obj-a4.4-I Obj-a5.1-E Obj-a5.6-I }

 The Arguments that are preferred are: {a4 }
```

1 – Argument list: This section of the software will display all the arguments and objections enumerating them and separating them by arguments and objections. It will show which CQ is found in the argument as well as the action and a brief description

2 – Argument Table: This section shows all the arguments and objections organizing them in this order and by action as well.

3 – Reverse Graph: This section of the program will display all the arguments found in the graph after the cycle removal, allowing to show which argument could be chosen to start solving the VAF graph.

4 – Direct Graph: This section will display the graph generated from all the arguments and objections from (1) and (2), creating the graph with the notion that objections attack arguments. These way objections will direct to arguments in the AATS argument list (1).

5 – Preferred Extension: This section shows the preferred extension created from solving the VAF. It also shows which actions are preferred for a set of values, within a preferred set of values.

6 – Agent ValueSet: This section shows the agent value in the order from the least important to the most important ones.

7 – DFS before VAF: This section shows the DFS of the argument graph (4) before it is solved by the VAF, this sections intents to display how the program sees a DFS and show also all the different edge types, found in the DFS. It is also possible to see the index in the VAF of all the back edges in the DFS.

8 – DFS after VAF: This section displays the DFS after the VAF is solved, and displays the DFS without the cycles, this DFS is useful to show where the cycle removal occurred.

Since the program does not offer any options, or any other way to interact with the user, it makes the program usability is straightforward. Where the user only run the program to obtain the results.

Qt Installation:

To change the value ordering of the agent, or to insert another AATS it is necessary to change the source of /src/core/Main.cpp at the project folder source. To be able to change the program source its necessary to install the Qt framework on your operating system.
To install Qt on:

Mac – follow the instructions at: http://doc.trolltech.com/4.4/install-mac.html.
Windows – follow the instructions at: http://doc.trolltech.com/4.4/install-win.html.
X11 – follow the instructions at: http://doc.trolltech.com/4.4/install-x11.html.

For X11 and Windows it is possible to have eclipse integration for Qt, or its possible to install QDevelop for a dedicated Qt editor at: http://qdevelop.org/ this application is available for Windows Mac and X11, or it is possible to install Qt Creator at: http://www.qtsoftware.com/downloads this application is also available for Windows Mac and X11.

How to change the Value Ordering for the Dictator:

To change the value ordering for the Dictator it is necessary to change the values of the Values for the dictator in this part of the code:

```
93 ValueSet vset_agdictator;
94 vset_agdictator.addValue(v_ms,5);
95 vset_agdictator.addValue(v_i,4);
96 vset_agdictator.addValue(v_mo,3);
97 vset_agdictator.addValue(v_g,3);
98 vset_agdictator.addValue(v_e,3);
99 Agent agdictator("Dictator",true,vset_agdictator);
```

If it is necessary to change the valueset of the dictator the user only has to change the values in lines 94 to 98 and input a value from 0 to 99, this value will decide how important it is, where as high the value is as more important it is.

To insert new states and or new actions within the AATS it is necessary to insert each action on the following order:

```
36 ValueSet vset_a1;
37 vset_a1.addValue(*(new Value("MS","+",30)),0);
38 vset_a1.addValue(*(new Value("MO","+",70)),0);
39 vset_a1.addValue(*(new Value("G","+")),-1);
40 vset_a1.addValue(*(new Value("E","-")),-1);
…
64 JointAction a1(vset_a1,"a1");
…
69 QVector<int> v_s1;
70 v_s1 << 30<<70;
…
82 Proposition p_s1(v_s1);
…
88 State s1(p_s1,a1,"s1");
```

From lines 37 to 40 values are inserted to an action, where the values can be promoted, demoted or stay equal. On line 64 the valueset create at 36 is used in the jointaction at 64 to define a new action this action is inserted after into the state.

Please note that to insert states it is necessary to follow this order where jointaction will be the edge in the AATS diagram that directs to state created.

For a complete description on how to insert new different state types please refer to Realization where is described the different constructors of the classes used before.

To insert the actions into the AATS it is necessary to do the following:

```
122 AatsTransGenerator aats(agset,vset_main);
123 aats.addTrans(s0,s1);
```

The line 122 creates an AATS diagram representation without any edges. Line 123 creates a transiction between state s0 and s1.

```
111 QVector<Agent> agset;
112 agset.append(agother);
113 agset.append(agdictator);
```

To insert new agents please follow the steps from above in how to change the value ordering of an Agent and simply insert the agent into the agent set lines 112 to 113.

Then it is necessary to compile and run the project.

Appendix B – Full Class and Methods List:

In this appendix it will be described all the methods and classes for this project.

Class Node:

```
QList<N>::iterator getIterator();
```
returns the beginning of an STL iterator allowing to go through all the edges that go from this node.

```
int size();
```
Return the node size

```
bool operator==(const Node<N>&) const;
```
Returns true if the node to compare as parameter is equal to itself.

```
N& getNode();
```
Returns the node that the class is representing.

```
QList<N>::iterator end();
```
Returns the end of STL iterator.

```
bool isImage();
```
returns if true if the node is an image of an node.

```
Node(const N &,const N &);
```
this constructor will create an instance of node with an edge between itself and another node.

```
Node(const N &,bool);
```
this constructor will create an instance of node that will define if it is an image or not.

```
void setImage(bool i);
```
allows to set the image property to true or false.

```
void add(const N &);
```
this node appends a node to this node creating an edge.

```
bool remove(const N &);
```
this method removes a node in the nodes container, removing this way an edge.

```
typename QList<N>::iterator getEdge(const N &);
```
searchs and returns the iterator that points at the node

```
QList<N> & getEdgesContainer();
```
returns the node container.

Class DiGraph:

```
DiGraph();
```
Creates an empty instance of DiGraph.

```
DiGraph(const N &,const N &);
```
Creates an instance of Digraph with an edge.

```
void add(const N &);
```
inserts a node without edge into the graph.

```
void add(const N &,const N &);
```
This method inserts an edge into the graph, using two nodes.

```
removeEdge(const N &,const N &);
```
remove an edge from a node.

```
QList<Node<N> >::iterator  getIterator();
```
Returns the beginning of the Graph STL iterator.

```
QList<Node<N> >::iterator  getNode(const N &);
```
Returns the node in the array to look at the outgoing edges.

```
QList<Node<N> >::iterator end();
```
Returns the end of the iterator

```
getReverseGraph(DiGraph<N> &);
```
Creates a reverse graph from current class into the graph inserted as parameter

```
int size();
```
returns the size of the graph.

Class DfsNode:

```
DfsNode(Node<N> & n, int d, int t){//
```
Creates an instance of DfsNode (v,w) with a depth in the tree and the node type.

```
Node<N> & getNode(){
```
Returns the node in the DfsNode.

```
int getDepth()
```
Returns the depth of the DfsNode

```
int getType(){
```
Returns the type of the DfsNode, where type=2 tree type=1 down edge, type=0 cross edge, type=-1 back edge.

Class Dfs:

```
Dfs(DiGraph<N> &g)
```

Creates an instance of the Dfs with a graph.

```
void initdfs(N n)
```

This method will start a DFS from a node n.

```
void initdfs()
```

This will start a Dfs from the first node in the graph.

```
bool hascycle()
```

Returns true if the DFS is strong connected.

```
QList<int> & getCycleList()
```

Returns the cyclelist container, this container will contain all the positions of the back edges in the DFS.

```
QList<DfsNode<N> > & getDfs()
```

Returns the Dfs container, will all the edges of the graph.

```
QMap<N,int> &getPreOrderSequence()
```

Returns the preorder in which the nodes in the Graph were visited.

```
QMap<N,int> &getPostOrderSequence()
```

Returns the postorder in which the nodes in the Graph were visited.

```
void dfsR(Node<N> node){
```

solves the DFS and creates the tree looking for cycles and different types of edges

Class Agent:

```
Agent(QString, bool);
```

Create an instance of agent with a name and defines it as active or passive.

```
Agent(QString, bool,ValueSet &);
```

Creates an instance of agent with a name and defines it as active or passive, with also a Valueset.

```
ValueSet * getValueSet();
```

Returns the set of values for this agent.

```
QString getName();
```

Returns the agent Name.

```
bool isActive();
```

Returns true if the agent is active or false if is passive.

```
void setActive(bool);
```
Sets the agent to passive with false or active with true.

```
const bool operator==(const Agent &) const
```
Returns equal if the agent in the parameter is equal to itself.

Class JointAction:

```
void setName(QString);//sets a name for the jointaction.
```

```
JointAction();
```
Creates an instance with an empty jointaction with no valueset and no name

```
JointAction(const ValueSet& ,QString);
```
Creates an instance of jointaction with a ValueSet and a name

```
JointAction(QString);
```
Creates an instance of a jointaction with just a name and no ValueSet.

```
ValueSet & getValueSet();
```
Returns the valueset of the jointaction

```
QString getName();
```
Returns the name of the jointaction

```
void setName(QString);
```
Sets the name for the jointaction.

Class AatsTransGenerator:

```
AatsTransGenerator(QVector<Agent>&,ValueSet &);
```
Creates an instance of AatsTransGenerator with an Agent set and a value set.

```
AatsTransGenerator(QVector<Agent> &,ValueSet
&,DiGraph<State>&);
```
Creates an instance of AatsTransGenerator with an Agent set, a value set and DiGraph.

```
void addTrans(const State &, const State &);
Adds a transition to the AATS container.
```

```
void addTrans(const State &);
Add a state to the AATS container.
```

`void addAgent(const Agent &);`
Adds an agent to the AATS representation.

`void removeTrans(const State&,const State&);`
Removes transition from a given state.

`void removeAgent(const Agent&);`
`Removes an agent from the agent container.`

`int getSize()`
Returns the size of the aats diagram.

`QList<Node<State> >::iterator getTransGraph();`
`Returns the beginning of the STL iterator in the AATS graph.`

`QVector<Agent>& getAgents();`
`Return the beginning of the STL iterator with all the agents.`

`QList<Node<State> >::iterator end();`
`Returns the last item in the AATS graph STL iterator.`

`ValueSet &getValueSet();`
Returns the main value set of the AATS, each agent has to compile with these values.

Class ArgGenerator:

`ArgGenerator(AatsTransGenerator &);`
Creates an instance of ArgGenerator with AATS instance as parameter.

`CQxSet getArgs();`
Returns the arguments found for CQ5,CQ6,CQ10 in a CQxSet.

`CQxSet getObj();`
Returns the objections found for CQ7,CQ8,CQ9,CQ11 in a CQxSet.

`void buildArgsAndObj();`
Generates all the arguments and objections in the AATS diagram.

`QList<QString> & getStateList();`
Returns all the states found in a QList

`QVector<QString> getArgsDescription();`
Returns a container with all the arguments description

`QVector<QString> getObjDescription();`
Returns a container with all the objections description

`void getArgGraph(DiGraph<Argument> &);`
Copies the arguments and objections to a digraph inputted as parameter

`AatsTransGenerator & getAatsTransGenerator();`
Returns the AATs instance.

Class CQx:

`CQx(QString,State&,State&,Value &);`
Creates an instance of CQx with a name and two states and a value that is related with.

`CQx(QString,State&);`
Creates an instance of CQx with a name and a state

`CQx(QString,State&,Value &);`
Creates an instance of CQx with a name and a state and value.

`State & getStatefrom();`
Returns the statefrom in the CQx this state is used in the CQ10 and CQ7

`State & getStateto();`
Returns the stateto in the CQx used in all the other CQ’s.

`Value & getValue();`
Returns the value in the CQx

`void setArgName(QString);`
Sets the CQ description used to identify the argument

`QString getArgName();`
Returns the description used to identify the CQ.

`void setName(QString);`
Sets the Name of the CQx.

`const bool operator==(const CQx &) const`
Compares two cqx for equality on the name and the states attached are taken into consideration

Class CQxSet:

`CQxSet();`
Creates and empty set of CQ’s.

```
void add(CQx &);
```

Inserts a new CQ in a sorted way into the set.

```
void addUnsorted(CQx &);
```

Inserts a CQ into the set not taken into consideration any order.

```
void remove(const CQx &);
```

Removes a CQ of the set.

```
QList<CQx>::iterator getIterator();
```

Returns the beginning of the STL iterator for convenience,

```
QList<CQx>::iterator getIterator();
```

Returns the beginning of STL iterator for convenience.

```
QList<CQx>::iterator end();
```

Returns the end of the STL iterator in the set.

```
int size();
```

Returns the size of the CQ set.

Class Proposition:

```
Proposition(int size);
```

Creates an instance of proposition with a size.

```
Proposition(QVector<int> &);
```

Creates an instance of proposition with a vector that will be the proposition

```
void addList(const int&);
```

Adds an element to the proposition.

```
void removeList(int);
```

Removes an element from the proposition.

```
int getSize();
```

Returns the size of the list in the proposition.

```
string printprop();
```

Returns the content of the list representing the proposition

```
QVector<int>& getlist();
```

Returna a container that will represent the list in the proposition

```
bool operator==(const Proposition &) const;
```

Returns true if a proposition is equal to the one inserted as parameter

Class State:

```
State(Proposition &,bool, bool,JointAction &, QString );
```

Creates an instance of State with a proposition, a jointaction and name. It defines if the state is initial and if it is a goal state.

```
State(Proposition &,bool, bool, QString );
```

Creates an instance of State with a proposition and name. It defines if the state is initial, if it is a goal state.

```
State(Proposition &,JointAction &, QString );
```

Creates an instance of State with a proposition, a jointaction and name.

```
Proposition & getProposition();
```

Returns the proposition in the state.

```
JointAction & getJointAction();
```

Returns the jointaction in the state.

```
QString getName();
```

Returns the name of the State.

```
void setName(QString);
```

Sets the name of the state.

```
bool isGoalState();
```

Returns true if the state is a goal state or false otherwise.

```
bool isInitState();
```

Returns false if the state is an initial state or false otherwise.

```
void setGoalState(bool);
```

Sets the state to goal or not.

```
void setInitState(bool);
```

Sets the state to initial or not.

```
bool operator==(const State&) const;
```

Returns true if the state inserted as parameter is true or not.

Class Argument:

```
Argument(QString,Value &);
```

Creates an instance of argument with a name and value.

```
Argument(QString,Value &,CQx &);
```

Creates an instance of argument with a name, a value, and CQx

```
Value & getValue();
```

Returns the value in the argument.

```
CQx & getCQx();
```

Returns the cqx in the argument.

```
QString & getName();
```

Returns the name in the argument.

```
bool operator==(const Argument&) const;
```

Returns true if the inserted as parameter argument is equal to itself

```
bool operator<(const Argument&) const;
```

Returns true if the argument less that the one inserted as parameter.

Class Value:

```
Value(QString);
```

Creates an instance of Value with a name.

```
Value(QString,QString);
```

Creates an instance of value with a name and a valuation.

```
Value(QString,QString, int);
```

Creates an instance of value with a name, a valuation and a degree.

```
void setName(QString);
```

Sets the name to value.

```
QString getName();
```

Returns the value name.

```
void setValuation(QString);
```

Sets the valuation for the value (+,-,=).

```
QString getValuation();
```

Returns the valuation of the value.

```
void setDegree(int);
```

Sets the degree of the value.

```
int getDegree();
```

Returns the degree of the value.

```
bool operator==(const Value&) const;
```
Returns true if the value inserted as parameter is equal.

Class ValueSet:

```
ValueSet(const QMap< int,Value > &);
```
Creates an instance of a valueset from a Map.

```
ValueSet();
```
Creates an instance of an empty valueset.

```
bool addValue(const Value &,int);
```
Adds a Value with an order into the set.

```
void removeValue(const Value &, int);
```
Removes a value with an order.

```
int compareTo(const Value&,const Value&);
```
Compares two values to see which one has the biggest priority. The function returns -1 if the first is smaller, 0 if equal or 1 otherwise.

```
Value highestCommonValue(const ValueSet &);
```
Compares the highest common value between the set and the inserted as parameter.

```
QMap<int,Value >::iterator getValue(const Value &);
```
Search for a Value and returns the iterator position.

```
QMap<int,Value >::iterator getIterator();
```
Returns the beginning of the STL iterator.

```
QMap<int,Value >::iterator end();
```
Returns the end of the STL iterator.

```
QMap<int,Value >::iterator highestValue();
```
Returns the highest value in the set.

```
QMap<int,Value >::iterator lowestValue();
```
Returns the lowest value in the set.

```
int size();
```
Returns the size of the ValueSet.

Class VAFGenerator:

VAFGenerator(DiGraph<Argument> &,Agent &);
Creates a vaf solution for an agent and a graph

void initVAF();
starts the solving the VAF

Dfs<Argument> & solvePolychromaticCycles(Dfs<Argument> &);
Returns an updated DFS. This function solves polychromatic cycles recursively.

void solveDichromaticCycles(Dfs<Argument> &,Argument &);
This function solves dichromatic cycles recursively as part of the solvePolychromaticCycles method.

void solveUncycledVaf();
This method prepares the VAF to be solved and call a method to solve the DFS.

QList<Argument> & getSolution();
Returns a list of arguments in the solution.

QString vafResult();
Returns the result in a string.

bool isUnsolvable();
Returns true if it's unsolvable or false otherwise.

DiGraph<Argument> & getReverseUpdatedGraph();
Returns the reverse updated graph.

QString vafArgsResult();
Returns the arguments that are part of the vaf preferred extension.

Argument & getMaxArgument();
Return the maximum argument when looking at the reverse graph.

void solve_ext(**int**,**int**,QList<DfsNode<Argument> > &);
Solves recursive problems with the vaf.

Appendix C – Critical questions description:

Full description of the CQs to be taken into consideration in this experiment;
CQ5, CQ6 and CQ7, all consider possible alternatives to the original action proposed with each of those critical questions considering the effects of any, such alternative actions upon the consequences, goal and value.
Formal way (F) that describes CQ5- Agent i Ag can participate in joint action j_m J_{Ag} , where
$j_n \neq j_m$, such that τ (q_x , j_m) is q_y .
Description (D): Are there alternatives ways of realizing the same consequences?

CQ6;
F: Agent i Ag can participate in joint action j_m J_{Ag} , where $j_n \neq j_m$, such that τ (q_x , j_m) is q_y, such that p_a π (q_y) and $p_a \notin \pi (q_x)$ or $p_a \notin \pi (q_y)$ and p_a π (q_x).

D: Are there alternatives ways of realizing the same goal?

CQ7:
F: Agent i Ag can participate in joint action jm JAg , where jn jm , such that τ $(qx ,$ $jm)$ is qz , such that
δ (q_x, q_z, v_u) is +.
D: Are there alternatives ways of promoting the same value?

CQ8, CQ9 and CQ10 are all concerned with the side effects of the proposed action where CQ8 and CQ9 draw attention to possible negative side effects, CQ10 can be seen as more of a supporting argument that identifies positive side effects of the action that endorse rather than the performance of the action itself.

CQ8;
F: In the initial state qx Q, if agent i Ag participates in joint action jn JAg , then τ (qx , jn) is qy , such that pb π (qy), where $pa \neq pb$, such that δ (qx , qy , vu) is –.
D: Does doing the action have a side effect, which demotes the value?

CQ9;
F: In the initial state qx Q, if agent i Ag participates in joint action jn JAg , then τ (qx , jn) is qy , such that δ (qx , qy , vw) is –, where vu≠ vw .
D: Does doing the action have a side effect, which demotes some other value?

CQ10;
F: In the initial state qx Q, if agent i Ag participates in joint action jn JAg , then τ (qx , jn) is qy , such that δ (qx , qy , vw) is +, where vu ≠ vw .
D: Does doing the action promote some other value?

CQ11 identifies a clash between the action proposed and other desirable action, Arises when the goal stated achieved by proposed action is incompatible with the goal state of some other action, that promotes a desirable value, so that only one of the actions can be executed.

F: In the initial state qx Q, if agent i Ag participates in joint action jn JAg , then τ (qx , jn) is qy and

δ (qx , qy , vu) is +. But, there is some other joint action jm J*Ag* , where jn ≠ jm , such that τ (qx , jm) is qz , such that δ (qx , qz , vw) is +, where vu ≠ vw .
D: Does doing the action preclude some other action, which would promote some other value?