Computing the Stratified Minimal Models Semantic

Mauricio Osorio[1], Angel Marin-George[2], and Juan Carlos Nieves[3]

[1] Universidad de las Américas - Puebla
CENTIA, Sta. Catarina Mártir, Cholula, Puebla, 72820 México
osoriomauri@googlemail.com
[2] Benemérita Universidad Atónoma de Puebla
Facultad de Ciencias de la Computación,
Puebla, Puebla, México
misterilei@hotmail.com
[3] Universitat Politècnica de Catalunya
Software Department (LSI)
c/Jordi Girona 1-3, E08034, Barcelona, Spain
jcnieves@lsi.upc.edu

Abstract. It is well-known, in the area of argumentation theory, that there is a direct relationship between extension-based argumentation semantics and logic programming semantics with negation as failure. One of the main implication of this relationship is that one can explore the implementation of argumentation engines by considering logic programming solvers. Recently, it was proved that the argumentation semantics CF2 can be characterized by the stratified minimal model semantics (MM^r). The stratified minimal model semantics is also a recently introduced logic programming semantics which is based on a recursive construction and minimal models.
In this paper, we introduce a solver based on MINISAT algorithm for inferring the logic programming semantics MM^*. As one of the applications of the MM^r solver, we will argue that this solver is a suitable tool for computing the argumentation semantics CF2.
Keywords: Non-monotonic reasoning, extension-based argumentation semantics and logic programming.

1 Introduction

Argumentation theory has become an increasingly important and exciting research topic in Artificial Intelligence (AI), with research activities ranging from developing theoretical models, prototype implementations, and application studies [3]. The main purpose of argumentation theory is to study the fundamental mechanism, humans use in argumentation, and to explore ways to implement this mechanism on computers.

Argumentation is also a formal discipline within Artificial Intelligence (AI) where the aim is to make a computer assist in or perform the act of argumentation. In fact, during the last years, argumentation has been gaining increasing importance in Multi-Agent Systems (MAS), mainly as a vehicle for facilitating *rational interaction* (*i.e.* interaction which involves the giving and receiving of reasons). A single agent may also use argumentation techniques to perform its individual reasoning because it needs to make decisions under complex preferences policies, in a highly dynamic environment.

Dung's approach, presented in [7], is a unifying framework which has played an influential role on argumentation research and AI. This approach is mainly orientated to manage the interaction of arguments. The interaction of the arguments is supported by four extension-based argumentation semantics: *stable semantics*, *preferred semantics*, *grounded semantics*, and *complete semantics*. The central notion of these semantics is the *acceptability of the arguments*. It is worth mentioning that although these argumentation semantics represents different pattern of selection of arguments, all these argumentation semantics are based on the concept of admissible set.

An important point to remark *w.r.t.* the argumentation semantics based on admissible sets is that these semantics exhibit a variety of problems which have been illustrated in the literature [17, 2, 3]. For instance, let AF be the argumentation framework which appears in Figure 1. We can see that there are five arguments: a, b, c, c and e. The arrows in the figure represent conflict between arguments. For example, we can see that the argument e is attacked by the argument d, the argument d is attacked by the arguments a, b and c. Some authors, as Prakken and Vreeswijk [17], Baroni *et al*[2], suggest that the argument e can be considered as an acceptable argument since it is attacked by the argument d which is attacked by three arguments: a, b, c. Observe that the arguments a, b and c form a cyclic of attacks. However, none of the argumentation semantics suggested by Dung is able to infer the argument e as acceptable.

We can recognize two major branches for improving Dung's approach. On the one hand, we can take advantage of graph theory; on the other hand, we can take advantage of logic programming with negation as failure.

With respect to graph theory, the approach suggested by Baroni *et al*, in [2] is maybe the most general solution defined until now for improving Dung's approach. This approach is based on a solid concept in graph theory which is a *strongly connected component* (SCC). Based on this concept, Baroni *et al*, describe a recursive approach for generating new argumentation semantics. For instance, the argumentation semantics CF2 suggested in [2] is able to infer the argument e as an acceptable argument from the argumentation framework of Figure 1.

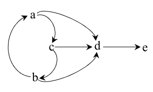

Fig. 1. Graph representation of the following argumentation framework: $\langle\{a, b, c, d, e\}, \{(a, c), (c, b), (b, a), (a, d), (c, d), (b, d), (d, e)\}\rangle$.

Since Dung's approach was introduced in [7], it was viewed as a special form of logic programming with *negation as failure*. For instance, in [7] it was proved that the grounded semantics can be characterized by the well-founded semantics [9], and the stable argumentation semantics can be characterized by the stable model semantics [10]. Also in [4], it was proved that the preferred semantics can be characterized by the p-stable semantics [16]. In fact, the preferred semantics can be also characterized by the minimal models and the stable models of a logic program [13]. By regarding an argumentation framework in terms of logic programs, it has been shown that one can construct intermediate argumentation semantics between the grounded and preferred semantics [11]. Also it is possible to define extensions of the preferred semantics [14].

Recently, it was proved that the argumentation semantics CF2 can be characterized by the stratified minimal model semantics (MM^r) [15]. MM^r in is an interesting logic programming semantic which satisfies some relevant properties as it is always defined and satisfies that property of *relevance*. The construction of MM^r is based on a recursive function and minimal models. These features allow the construction of a MM^r's solver based on algorithms of general purpose as UNSAT algorithms.

In this paper, we introduce a solver of MM^r. This solver is based on the MINISAT solver [8] and standard graph's algorithms. We will see that this solver presents quite efficient running time executions that suggest that the actual version of our MM^r's solver is an efficient implementation.

As we have pointed out, MM^r is a logic programming semantics which is able to characterize the argumentation semantics CF2. Hence, we argue that our MM^r's solver is a quite efficient implementation of CF2. Therefore, one can consider the MM^r's solver for building rational agents whose rational process could be based on CF2 and MM^r. It is worth mentioning, that to the best of our knowledge there is not an open implementation of CF2.

The rest of the paper is divided as follows: In §2, we present introduce some basic concepts *w.r.t.* logic programming and argumentation theory. In §3, the stratified argumentation semantics is introduced. In §4, we present how by considering the stratified minimal model semantics one can perform argumentation reasoning. In particular, we show that MM^r is able to characterize CF2. In §5, we describe a little in detail the implementation of the MM^r's solver. In §6, we presents our conclusions. In Appendix A, we present the general algorithms that where implemented in the MM^r's solver.

2 Background

In this section, we define the syntax of the logic programs that we will use in this paper and some basic concepts of logic programming semantics and argumentation semantics.

2.1 Syntax and some operations

A signature $\mathcal{L}$ is a finite set of elements that we call atoms. A *literal* is either an atom a, called *positive literal*; or the negation of an atom $\neg a$, called *negative literal*. Given a set of atoms $\{a_1, ..., a_n\}$, we write $\neg\{a_1, ..., a_n\}$ to denote the set of atoms $\{\neg a_1, ..., \neg a_n\}$. A *normal* clause, C, is a clause of the form

$$a \leftarrow b_1 \wedge \ldots \wedge b_n \wedge \neg b_{n+1} \wedge \ldots \wedge \neg b_{n+m}$$

where a and each of the b_i are atoms for $1 \leq i \leq n + m$. In a slight abuse of notation we will denote such a clause by the formula $a \leftarrow \mathcal{B}^+ \cup \neg\mathcal{B}^-$ where the set $\{b_1, \ldots, b_n\}$ will be denoted by $\mathcal{B}^+$, and the set $\{b_{n+1}, \ldots, b_{n+m}\}$ will be denoted by $\mathcal{B}^-$. We define a *normal program* P, as a finite set of normal clauses. If the body of a normal clause is empty, then the clause is known as a *fact* and can be denoted just by: $a \leftarrow$.

We write $\mathcal{L}_P$, to denote the set of atoms that appear in the clauses of P. We denote by $HEAD(P)$ the set $\{a|a \leftarrow \mathcal{B}^+, \neg\mathcal{B}^- \in P\}$.

A program P induces a notion of $dependency$ between atoms from $\mathcal{L}_P$. We say that a *depends immediately on* b, if and only if, b appears in the body of a clause in P, such that a appears in its head. The two place relation *depends on* is the transitive closure of *depends immediately on*. The set of dependencies of an atom x, denoted by *dependencies-of*(x), corresponds to the set $\{a \mid x\ depends\ on\ a\}$. We define an equivalence relation $\equiv$ between atoms of $\mathcal{L}_P$ as follows: $a \equiv b$ if and only if $a = b$ or ($a\ depends\ on\ b$ and $b\ depends\ on\ a$). We write $[a]$ to denote the equivalent class induced by the atom a.

Example 1. Let us consider the following normal program,

$$S = \{e \leftarrow e,\ \ c \leftarrow c,\ \ a \leftarrow \neg b \wedge c,\ \ b \leftarrow \neg a \wedge \neg e,\ \ d \leftarrow b\}.$$

The dependency relations between the atoms of $\mathcal{L}_S$ are as follows:
dependencies-of$(a) = \{a, b, c, e\}$; *dependencies-of*$(b) = \{a, b, c, e\}$; *dependencies-of*$(c) = \{c\}$; *dependencies-of*$(d) = \{a, b, c, e\}$; and *dependencies-of*$(e) = \{e\}$.
We can also see that, $[a] = [b] = \{a, b\}$, $[d] = \{d\}$, $[c] = \{c\}$, and $[e] = \{e\}$.

We take $<_P$ to denote the strict partial order induced by $\equiv$ on its equivalent classes. Hence, $[a] <_P [b]$, if and only if, b *depends-on* a and $[a]$ is not equal to $[b]$. By considering the relation $<_P$, each atom of $\mathcal{L}_P$ is assigned an order as follows:

- An atom a *is of order* 0, if $[a]$ is minimal in $<_P$.
- An atom a *is of order* $n + 1$, if n is the maximal order of the atoms on which a depends.

We say that a program P is of order n, if n is the maximum order of its atoms. We can also break a program P of order n into the disjoint union of programs P_i with $0 \leq i \leq n$, such that P_i is the set of rules for which the head of each clause is of order i (*w.r.t.* P). We say that $P_0, \ldots, P_n$ are the *relevant modules* of P.

Example 2. By considering the equivalent classes of the program S in Example 1, the following relations hold: $\{c, e\} <_S \{a, b\} <_S \{d\}$. We also can see that: a is of order 1, d is of order 2, b is of order 1, e is of order 0, and c is of order 0. This means that S is a program of order 2.

The following table illustrates how the program S can be broken into the disjoint union of the following relevant modules S_0, S_1, S_2:

S	S_0	S_1	S_2
$e \leftarrow e.$	$e \leftarrow e.$		
$c \leftarrow c.$	$c \leftarrow c.$		
$a \leftarrow \neg b \wedge c.$		$a \leftarrow \neg b \wedge c.$	
$b \leftarrow \neg a \wedge \neg e.$		$b \leftarrow \neg a \wedge \neg e.$	
$d \leftarrow b.$			$d \leftarrow b.$

Now we introduce a single reduction for any normal program. The idea of this reduction is to remove from a normal program any atom which has already fixed to some true value. In fact, this reduction is based on a pair of sets of atoms $\langle T; F\rangle$ such that the set T contains the atoms which can be considered as true and the set F contains the atoms which can be considered as false. Formally, this reduction is defined as follows:

Let $A = \langle T; F \rangle$ be a pair of sets of atoms. The reduction $R_{WFS}(P, A)$ is obtained by 2 steps:

1. Let $R(P, A)$ the program obtained in the following steps:
   (a) We replace every atom x that occurs in the bodies of P by 1 if $x \in T$, and we replace every atom x that occurs in the bodies of P by 0 if $x \in F$;
   (b) we replace every occurrence of $\neg 1$ by 0 and $\neg$ 0 by 1;
   (c) every clause with a 0 in its body is removed;
   (d) finally we remove every occurrence of 1 in the body of the clauses.
2. $R_{WFS}(P, A) = norm_{CS}(R(P, A))$ such that CS is a rewriting system formed by the transformation rules: RED^+, RED^-, $Success$, $Failure$ and $Loop$ (the definition of these transformation rules can be founded in [6]) and $norm_{CS}(P)$ denotes the uniquely determined normal form of a program P with respect to the system $\mathcal{CS}$.

We want to point out that this reduction does not coincide with the Gelfond-Lifschitz reduction [10].

Example 3. Let us consider the normal program S of Example 1. Let P be the normal program $S \setminus S_0$, and let A be the pair of sets of atoms $\langle \{c\}; \{e\} \rangle$. This means that we obtain the following programs:

P:	$R(P, A)$:
$a \leftarrow \neg b \wedge c.$	$a \leftarrow \neg b.$
$b \leftarrow \neg a \wedge \neg e.$	$b \leftarrow \neg a.$
$d \leftarrow b.$	$d \leftarrow b.$

2.2 Semantics

From now on, we assume that the reader is familiar with the single notion of *minimal model*. In order to illustrate this basic notion, let P be the normal program $\{a \leftarrow \neg b,\ \ b \leftarrow \neg a,\ \ a \leftarrow \neg c,\ \ c \leftarrow \neg a\}$. As we can see, P has five models: $\{a\}$, $\{b, c\}$, $\{a, c\}$, $\{a, b\}$, $\{a, b, c\}$; however, P has just two minimal models: $\{b, c\}$, $\{a\}$. We will denote by $MM(P)$ the set of all the minimal models of a given logic program P. Usually MM is called *minimal model semantics*.

A semantics SEM is a mapping from the class of all programs into the powerset of the set of (2-valued) models. SEM assigns to every program P a (possible empty) set of (2-valued) models of P. If $SEM(P) = \emptyset$, then we informally say that SEM is undefined for P.

Given a set of interpretations Q and a signature $\mathcal{L}$, we define Q *restricted to* $\mathcal{L}$ as $\{M \cap \mathcal{L} \mid M \in Q\}$. For instance, let Q be $\{\{a, c\}, \{c, d\}\}$ and $\mathcal{L}$ be $\{c, d, e\}$, hence Q *restricted to* $\mathcal{L}$ is $\{\{c\}, \{c, d\}\}$.

Let P be a program and $P_0, \ldots, P_n$ its relevant modules. We say that a semantics S satisfies the property of relevance if for every i, $0 \leq i \leq n$, $S(P_0 \cup \cdots \cup P_i) = S(P)$ restricted to $\mathcal{L}_{P_0 \cup \cdots \cup P_i}$.

2.3 Argumentation basics

Now, we present some basic concepts with respect to extended-based argumentation semantics. The first concept that we consider is the one of *argumentation framework*. An argumentation framework captures the relationships between the arguments.

Definition 1. *[7] An argumentation framework is a pair $AF = \langle AR, \text{attacks}\rangle$, where* AR *is a finite set of arguments, and* attacks *is a binary relation on* AR, i.e. attacks $\subseteq AR \times AR$. *We write $\mathcal{AF}_{AR}$ to denote the set of all the argumentation frameworks defined over AR.*

We say that *a attacks b* (or b is attacked by a) if $(a, b) \in attacks$ holds. Usually an extension-based argumentation semantics S_{Arg} is applied to an argumentation framework AF in order to infer sets of acceptable arguments from AF. An extension-based argumentation semantics S_{Arg} is a function from $\mathcal{AF}_{AR}$ to 2^{AR}. S_{Arg} can be regarded as a pattern of selection of sets of arguments from a given argumentation framework AF.

Given an argumentation framework $AF = \langle AR, attacks\rangle$, we will say that an argument $a \in AR$ is acceptable, if $a \in E$ such that $E \in S_{Arg}(AF)$.

3 Stratified Minimal Model Semantics

In this section, we introduce the *stratified minimal model semantics*. This semantics has some interesting properties as: it satisfies the property of relevance, and it agrees with the stable model semantics for the well-known class of stratified logic programs (the proof of this property can be found in [11, 12]).

In order to define the stratified minimal model semantics MM^r, we define the operator $*$ and the function $freeTaut$ as follows:

- Given Q and L both sets of interpretations, we define $Q * L := \{M_1 \cup M_2 \mid M_1 \in Q, M_2 \in L\}$.
- Given a logic program P, $freeTaut$ denotes a function which removes from P any tautology.

The idea of the function $freeTaut$ is to remove any clause which is equivalent to a tautology in classical logic.

Definition 2. *Given a normal logic program P, we define the s*stratified minimal model semantics MM^r *as follows:* $MM^r(P) = MM^r_c(freeTaut(P) \cup \{x \leftarrow x \mid x \in \mathcal{L}_P \setminus HEAD(P)\}$ *such that $MM^r_c(P)$ is defined as follows:*

1. *if P is of order* 0, $MM^r_c(P) = MM(P)$.
2. *if P is of order $n > 0$,* $MM^r_c(P) = \bigcup_{M \in MM(P_0)} \{M\} * MM^r_c(R_{WFS}(Q, A))$ *where $Q = P \setminus P_0$ and $A = \langle M; \mathcal{L}_{P_0} \setminus M\rangle$.*

We call a model in $MM^r(P)$ a stratified minimal model *of P.*

Observe that the definition of the stratified minimal model semantics is based on a recursive construction where the base case is the application of MM. It is not difficult to see that if one changes MM by any other logic programming semantics S, as the stable model semantics, one is able to construct a relevant version of the given logic programming semantics (see [11, 12] for details).

In order to introduce an important theorem of this paper, let us introduce some concepts. We say that a normal program P is basic if every atom x that belongs to $\mathcal{L}_P$, then x occurs as a fact in P. We say that a logic programming semantics SEM is *defined for basic programs*, if for every basic normal program P then $SEM(P)$ is defined.

4 Stratified Argumentation Semantics

In this section, we show that by considering the stratified minimal model semantics, one can perform argumentation reasoning based on extension-based argumentation semantics style.

As the stratified minimal model semantics is a semantics for logic programs, we require a function mapping able to construct a logic program from an argumentation framework. Hence, let us introduce a simple mapping to regard an argumentation framework as a normal logic program. In this mapping, we use the predicates $d(x)$, $a(x)$. The intended meaning of $d(x)$ is: "the argument x is defeated" (this means that the argument x is attacked by an acceptable argument), and the intended meaning of $a(X)$ is that the argument X is accepted.

Definition 3. *Let $AF = \langle AR, attacks \rangle$ be an argumentation framework, $P^1_{AF} = \{d(a) \leftarrow \neg d(b_1), \ldots, d(a) \leftarrow \neg d(b_n) \mid a \in AR$ and $\{b_1, \ldots, b_n\} = \{b_i \in AR \mid (b_i, a) \in attacks\}\}$; and $P^2_{AF} = \bigcup_{a \in AR}\{a(a) \leftarrow \neg d(a)\}$. We define: $P_{AF} = P^1_{AF} \cup P^2_{AF}$.*

The intended meaning of the clauses of the form $d(a) \leftarrow \neg d(b_i)$, $1 \leq i \leq n$, is that an argument a will be defeated when anyone of its adversaries b_i is not defeated. Observe that, essentially, P^1_{AF} is capturing the basic principle of *conflict-freeness* (this means that any set of acceptable argument will not contain two arguments which attack each other). The idea P^2_{AF} is just to infer that any argument a that is not defeated is accepted.

Example 4. Let AF be the argumentation framework of Figure 1. We can see that $P_{AF} = P^1_{AF} \cup P^2_{AF}$ is:

P^1_{AF} :	P^2_{AF} :
$d(a) \leftarrow \neg d(b).$	$a(a) \leftarrow \neg d(a).$
$d(b) \leftarrow \neg d(c).$	$a(b) \leftarrow \neg d(b).$
$d(c) \leftarrow \neg d(a).$	$a(c) \leftarrow \neg d(c).$
$d(d) \leftarrow \neg d(a).$	$a(d) \leftarrow \neg d(d).$
$d(d) \leftarrow \neg d(b).$	$a(e) \leftarrow \neg d(e).$
$d(d) \leftarrow \neg d(c).$	
$d(e) \leftarrow \neg d(d).$	

Two relevant properties of the mapping P_{AF} are that the stable models of P_{AF} characterize the stable argumentation semantics and the well founded model of P_{AF} characterizes the grounded semantics [11].

Once we have defined a mapping from an argumentation framework into logic programs, we are going to define a candidate argumentation semantics which is induced by the stratified minimal model semantics.

Definition 4. *Given an argumentation framework A, we define a stratified extension of AF as follows: A_m is a stratified extension of AF if exists a stratified minimal model M of P_{AF} such that $A_m = \{x | a(x) \in M\}$. We write $MM^r_{Arg}(AF)$ to denote the set of stratified extensions of AF. This set of stratified extensions is called stratified argumentation semantics.*

In order to illustrate the stratified argumentation semantics, we are going to presents an example.

Example 5. Let AF be the argumentation framework of Figure 1 and P_{AF} be the normal program defined in Example 4. In order to infer the stratified argumentation semantics, we infer the stratified minimal models of P_{AF}. As we can see P_{AF} has three stratified minimal models : $\{d(a), d(b), d(d), a(c), a(e)\}\{d(b), d(c), d(d), a(a), a(e)\}\{d(a), d(c), d(d), a(b), a(e)\}$, this means that AF has three stratified extensions which are: $\{c, e\}$, $\{a, e\}$ and $\{b, e\}$. Observe that the stratified argumentation semantics coincides with the argumentation semantics CF2.

In [15], it was proved that the stratified argumentation semantics and the argumentation semantics CF2 coincide.

Theorem 1. *[15] Given an argumentation framework $AF = \langle AR, Attacks \rangle$, and $E \in AR$, $E \in MM^r_{Arg}(AF)$ if and only if $E \in CF2(AF)$.*

5 Implementation of the Stratified Minimal Model Semantics

In this section we describe our implementation of a MM^r solver[4]. The implementation was made in C++ and despite the use of an external SAT-solver(MINISAT) to find the minimal models, which implies the duplication of the data, we got a good performance.

We started implementing a specific-CF2 prototype solver which was a little faster than the current version of MM^r solver (when inputting CF2 programs of course).

The difference between a CF2 solver and a MM^r solver is that with a CF2 solver we only have rules r with $|B(r)| = 1$, for a MM^r solver we also may have rules r with $|B(r)| > 1$.

To explain our MM^r solver we first give the theoretical justification and then the implemented algorithms.

[4] This implementation was made as part of a project that also includes the implementation of a *p-stable* solver. The $p\text{-}stablesemantics$ is a logic programming semantics based on Paraconsistent Logic [16]

5.1 Theoretical Justification

From the definition 2 we can design an algorithm that computes the MM^r semantics. To compute a stratified minimal model of P using the definition 2, first the input program P is split into its relevant modules $P_0, ..., P_n$, then compute a minimal model M of P_0 and then compute a stratified minimal model of $R_{WFS}(Q, A)$ where $Q = P \setminus P_0$ and $A = \langle M; \mathcal{L}_{P_0} \setminus M \rangle$, which involves the computation of the relevant modules of $R_{WFS}(Q, A)$. As we will see next, it is possible to take advantage of the relevant modules already computed $P_0, ..., P_n$ to find the relevant modules of $R_{WFS}(Q, A)$, and thus optimizing the implementation.
From the definition 2 given in the previous sections, and from the fact that MM^r has the property of relevance, we can formulate the following definition for MM^r

Definition 5. *Let P be a normal program, then*

$$MM^r(P) = MM^r_c(freeTaut(P) \cup \{x \leftarrow x : x \in \mathcal{L}_P \setminus H(P)\})$$

Where the recursive definition of MM^r_c is

- *If P is of order 0, $MM^r_c(P) = MM(P)$.*
- *If P is of order $n > 0$, then*

$$MM^r_c(P) = \bigcup_{M \in MM^r_c(P_0 \cup ... \cup P_i)} \{M\} * MM^r_c(R_{WFS}(Q, A))$$

where $i \in \{0, ..., n\}$, $Q = P \setminus (P_0 \cup ... \cup P_i)$ y $A = \langle M, \mathcal{L}_{P_0 \cup ... \cup P_i} \setminus M \rangle$.

If we take $i = n - 1$ we get

$$MM^r_c(P) = \bigcup_{M \in MM^r_c(P \setminus P_n)} \{M\} * MM^r_c(R_{WFS}(P_n, \langle M, \mathcal{L}_{P \setminus P_n} \setminus M \rangle))$$

To apply the reductions it is not necessary to consider the atoms which are not in $\mathcal{L}_{P_n}$, so this equation becomes

$$MM^r_c(P) = \bigcup_{M \in MM^r_c(P \setminus P_n)} \{M\} * MM^r_c(R_{WFS}(P_n, \langle M \cap \mathcal{L}_{P_n}, (\mathcal{L}_{P \setminus P_n} \setminus M) \cap \mathcal{L}_{P_n} \rangle))$$

When translating this last equation into an iterative form, we get an iterative definition for $MM^r_c(P)$

Definition 6. *Let P be a normal program of order n, we define $MM^r_{c,i}$ as follows*

$$MM^r_{c,0} = MM(P_0)$$

$$MM^r_{c,1} = \bigcup_{M \in MM^r_{c,0}} \{M\} * MM^r_c(R_{WFS}(P_1, \langle M \cap \mathcal{L}_{P_1}, (\mathcal{L}_{P_0} \setminus M) \cap \mathcal{L}_{P_1} \rangle))$$

$$MM^r_{c,2} = \bigcup_{M \in MM^r_{c,1}} \{M\} * MM^r_c(R_{WFS}(P_2, \langle M \cap \mathcal{L}_{P_2}, (\mathcal{L}_{P_0 \cup P_1} \setminus M) \cap \mathcal{L}_{P_2} \rangle))$$

...

$$MM^r_{c,n} = \bigcup_{M \in MM^r_{c,n-1}} \{M\} * MM^r_c(R_{WFS}(P_n, \langle M \cap \mathcal{L}_{P_n}, (\mathcal{L}_{P_0 \cup, \ldots, \cup P_{n-1}} \setminus M) \cap \mathcal{L}_{P_n} \rangle))$$

$$MM^r_c(P) = MM^r_{c,n}$$

This definition gives the procedure we use to compute MM^r_c.

5.2 Implementation

Given a normal program P_T, the computation of $MM^r(P_T)$ can be outlined as follows:

1. Compute $P = freeTaut(P_T) \cup \{x \leftarrow x : x \in \mathcal{L}_{P_T} \setminus H(P)\}$.
2. Compute the relevant modules of P.
   (a) Construct the graph of dependencies G of P.
   (b) Find the strongly connected components of G.
   (c) Compute the relevant modules $P_0, ..., P_n$ of P according to the strongly connected components of G.
3. Use the procedure given by the definition 6 to compute $MM^r_c(P)$. For $i = 0$ to n we have to do the following
   (a) Compute the reduction

$$RED = R_{WFS}(P_i, \langle M \cap \mathcal{L}_{P_i}, (\mathcal{L}_{P_0 \cup, \ldots, \cup P_{i-1}} \setminus M) \cap \mathcal{L}_{P_i} \rangle)$$

   When $i = 0$, $RED = P_0$.
   (b) Compute the relevant modules of RED.
   (c) Compute minimal models of RED when RED is of order 0.
   (d) Recursively compute $MM^r_c(RED)$.

To remove the tautologies from P we use a simple algorithm that takes each rule and removes those that are tautologies. To compute the relevant modules of P' we base on the well known Kosaraju's algorithm [5] to find the strongly connected components of the graph of dependencies G of P'. A strongly connected C component of G is a maximum set of atoms such that each pair of atoms in C is mutually dependent. This algorithm gives a set $C_0, ..., C_n$ of strongly connected components of G such that for any pair of components C_i, C_j such that $i > j$, none of the atoms in C_j depends on an atom in C_i. We take advantage of this sequence of strongly connected components to compute the relevant modules of P. See the algorithm $create\_modules(Module\ P)$.

The algorithm $three\_in\_one(Module\ P_i)$ computes

$$RED = R_{WFS}(P_i, \langle M \cap \mathcal{L}_{P_i}, (\mathcal{L}_{P_0 \cup, \ldots, \cup P_{i-1}} \setminus M) \cap \mathcal{L}_{P_i} \rangle)$$

As we have said, in order to apply the reductions, we have to replace some atoms by 0 or 1. We associate a variable $state(a)$ to each atom a, it indicates the value that "replaces" a:

- $state(a) = one$ if a is to be replaced by 1.

- $state(a) = zero$ if a is to be replaced by 0.
- $state(a) = none$ if a is not to be replaced.

For optimization purpose, the algorithm $three\_in\_one(Module\ P_i)$ implements an heuristic that may save some computation in some cases. While computing RED, a rule r may be removed such that the order of the atom $H(r)$ is the same than the order of an atom $a \in B(r)$. When this happens, the dependence relation between $H(r)$ and a may be removed, it may cause the graph of dependencies of RED to have more than one strongly connected components, and it may cause the order of RED be bigger than 0. When one of those rules is removed, we can not assure that RED is of order bigger that 0, but when none of these rules is remove, we can prove that RED is of order 0. The algorithm $three\_in\_one(Module\ P_i)$ returns $true$ if one of the rules that may affect the dependency relations was removed, and $false$ if none of those rules were removed. After computing RED, the algorithm $three\_in\_one(Module\ P_i)$ constructs the graph of dependencies of RED.

The function $stratify(P_i)$ computes the relevant modules of P_i using the algorithms $three\_in\_one(P_i)$ and $create\_modules(P_i)$. Then returns $true$ if and only if the resulting program is of order bigger than zero.

To compute the minimal models of a program P, we use the algorithm $next\_minimal(P)$ which is based on MINISAT [8], each time $next\_minimal(P)$ is called, it tries to compute a minimal model of P different than the already computed, if another minimal model of P was found, returns $true$, otherwise discards the SAT solver and returns $false$.

Before explaining the main algorithms that we use to compute $MM_c^r(P)$, we explain some notation used. We associate a sequence (a list) of relevant modules $submodules(P)$ to P. Let $P_0, ..., P_n$ be the relevant modules of P. If $n > 0$, $submodules(P)$ is the sequence of relevant modules $P_0, ..., P_n$. If $n = 0$, $submodules(P)$ has no elements. We define the following operations over the elements of $submodules(P)$: $next(P_i) = P_{i+1}$ if $0 \leq i < n$, $back(P_i) = P_{i+1}$ if $0 \leq i < n$, and $next(P_n) = back(P_0) = null$.

To compute $MM_c^r(P')$, we use two algorithms, the algorithm $first\_EMM(P)$ computes one model of $MM_c^r(P')$. After calling $first\_EMM(P)$ we use the backtracking algorithm $next\_EMM(P)$ to compute more stratified minimal models. Let $submodules(P')$ be the list of relevant modules to which P belongs. When $next\_EMM(back(P))$ returns $false$, it means that P' has no more stratified minimal models, in this case $next\_EMM(P)$ also return $false$. If $next\_EMM(back(P))$ returns $false$, the algorithm $reset\_module(P)$ (not presented in this paper) is used to reset P and leave it as it was when initialized by $creates\_modules(P')$. After the backtracking we start again by calling to $first\_EMM(P)$.

Finally the algorithm $all\_EMM(PMM)$ shows how to put $first\_EMM(...)$ and $next\_EMM(...)$ together to compute $MM^r(PMM)$.

In table 1, it is shown the time it took to the solver to find the stratified minimal models of some randomly generated programs of 500000 rules, the first column shows the number of atoms(divided by 10^4), in the second, the average cardinality of $B^+(r) \cup B^-(r)$, then the initial number or modules, the time to find the first model, and the time between the subsequent models. The performance tests were executed in a Linux PC, with Pentium IV processor, 2.8Ghz and 512Mb RAM.

Table 1. Performance of the MM^r solver

$N_a/10^4$	$size/n_{rules}$	n	t	t_{next}
7	2	1	8.35	.3
15	2	339	9.5	.45
23	2	7046	8.72	.192
10	3	1	11.45	.41
16	3	26	11.50	.44
23	3	5314	10.20	.11
15	4	2	13.93	.44
20	4	650	12.81	.32
23	4	238	10.97	.001
7	5	1	15.5	.34
12	5	1	16.7	.39
17	5	3	16.02	.43

In table 1, it is shown the time it took to the solver to find the stratified minimal models of some randomly generated programs of 500000 rules, the first column shows the number of atoms(divided by 10^4), in the second, the average cardinality of $B^+(r) \cup B^-(r)$, then the initial number or modules, the time to find the first model, and the time between the subsequent models.

The interested reader can download our actual version of the MM^r solver from: **http://www.lsi.upc.edu/∼jcnieves/software/MMr.tar**
Also, one can find in **http://www.lsi.upc.edu/∼jcnieves/software//MMr-examples.tar** some illustrative examples.

6 Conclusions

Since extension-based argumentation reasoning was introduced, it was shown that one can perform practical argumentation reasoning by considering logic programming tools [7]. One of the main issue in argumentation community is the definition of argumentation tools able to perform reasoning by considering well-accepted argumentation semantics. One of the possible reasoning of the lack of real practical argumentation systems is that the well accepted argumentation semantics as the preferred semantics and CF2 are hard computable.

In this paper, we have introduced a solver for the stratified minimal model semantics. We have shown that the stratified minimal model semantics is practical enough for performing argumentation reasoning based on extension-based argumentation style. An interesting property of the stratified minimal model semantics is it can characterize a argumentation semantics called CF2. CF2 is an promising argumentation semantics able to overcome some of unexpected behaviors of argumentation semantics based on admissible sets [2, 1].

As we seen in Table 1, the current version of our stratified minimal models semantics' solver is quite efficient. Hence, we argue that our actual prototype can be considered as a candidate tool for building argumentation systems which could perform reasoning based on MM^r and of course CF2. It is worth mentioning, that to the best of our knowledge there is not an open implementation of CF2.

Acknowledgement

This research has been partially supported by the EC founded project ALIVE (FP7-IST-215890). The views expressed in this paper are not necessarily those of the ALIVE consortium.

References

1. P. Baroni and M. Giacomin. On principle-based evaluation of extension-based argumentation semantics. *Artificial Intelligence.*, 171(10-15):675–700, 2007.
2. P. Baroni, M. Giacomin, and G. Guida. SCC-recursiveness: a general schema for argumentation semantics. *Artificial Intelligence*, 168:162–210, October 2005.
3. T. J. M. Bench-Capon and P. E. Dunne. Argumentation in artificial intelligence. *Artificial Intelligence*, 171(10-15):619–641, 2007.
4. J. L. Carballido, J. C. Nieves, and M. Osorio. Inferring Preferred Extensions by Pstable Semantics. *Iberoamerican Journal of Artificial Intelligence (Inteligencia Artificial) ISSN: 1137-3601*, 13(41):38–53, 2009 (doi: 10.4114/ia.v13i41.1029).
5. T. H. Cormen, C. E. Leiserson, R. L. Riverst, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.
6. J. Dix, M. Osorio, and C. Zepeda. A General Theory of Confluent Rewriting Systems for Logic Programming and its applications. *Annals of Pure and Applied Logic*, 108(1–3):153–188, 2001.
7. P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–358, 1995.
8. N. Een and N. Sorensson. An Extensible SAT-Solver. In *SAT-2003*, 2003.
9. A. V. Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
10. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *5th Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
11. J. C. Nieves. *Modeling arguments and uncertain information — A non-monotonic reasoning approach*. PhD thesis, Software Department (LSI), Technical University of Catalonia, 2008.
12. J. C. Nieves and M. Osorio. A General Schema For Generating Argumentation Semantics From Logic Programming Semantics. Research Report LSI-08-32-R, Technical University of Catalonia, Software Department (LSI), http://www.lsi.upc.edu/dept/techreps/buscar.php, 2008.
13. J. C. Nieves, M. Osorio, and U. Cortés. Preferred Extensions as Stable Models. *Theory and Practice of Logic Programming*, 8(4):527–543, July 2008.
14. J. C. Nieves, M. Osorio, U. Cortés, I. Olmos, and J. A. Gonzalez. Defining new argumentation-based semantics by minimal models. In *Seventh Mexican International Conference on Computer Science (ENC 2006)*, pages 210–220. IEEE Computer Science Press, September 2006.

15. J. C. Nieves, M. Osorio, and C. Zepeda. Expressing Extension-Based Semantics based on Stratified Minimal Models. In H. Ono, M. Kanazawa, and R. de Queiroz, editors, *Proceedings of WoLLIC 2009, Tokyo, Japan*, volume 5514 of *FoLLI-LNAI subseries*, pages 305–319. Springer Verlag, 2009.

16. M. Osorio, J. A. Navarro, J. R. Arrazola, and V. Borja. Logics with Common Weak Completions. *Journal of Logic and Computation*, 16(6):867–890, 2006.

17. H. Prakken and G. A. W. Vreeswijk. Logics for defeasible argumentation. In D. Gabbay and F. Günthner, editors, *Handbook of Philosophical Logic*, volume 4, pages 219–318. Kluwer Academic Publishers, Dordrecht/Boston/London, second edition, 2002.

Appendix A: Algorithms

In this appendix, we make a detailed presentation of the functions that are relevant in the implementation of the MM^r solver.

Algorithm 1 function $next\_minimal(Module\ P)$

```
Require: Module $P$
  {$P.S$ is the MINISAT solver associated to $P$}
  if $P.S = null$ then
    create a new solver $P.S$ with the rules in $P$
  end if
  if $P.S.solve()$ {A new minimal model is computed} then
    for all $a \in \mathcal{L}_P$ that are in the new model generated $P.S.model$ do
      set $state(a) = one$
    end for
    for all $a \in \mathcal{L}_P$ that are not in $P.S.model$ do
      set $state(a) = zero$
    end for
    add to $P.S$ a clause with the atoms in the minimal model generated but negated
    return $true$
  else
    set $P.S = null$
    return $false$
  end if
```

Algorithm 2 function $create\_modules(Module\ P, Graph\ G))$

```
Require: Module $P$
  array of integer $B[n]$
  Obtain the sequence $C = C_0, ..., C_n$ of relevant modules of $G$ using the Kosaraju's algorithm
  for all $a \in \mathcal{L}_P$, set $ord(a) = -1$
  {we write $C[i]$ to refer to the i-th strongly connected component of the sequence $C$, $|C[i]|$ as
  the number of element of $C$, $ord(a)$ is the order of the atom $a$}
  initialize the elements of $B$ to $-1$
  for $i = 0$ to $n$ do
    for all $a \in C[i]$, initialize $ord(a) = B[i] + 1$
    for all $a \in C[i]$ do
      for all $b$ that depends immediately on $a$ (the edge $(a, b)$ is in $G$) do
        set $B[k] = B[i] + 1$, $k$ is such that $b \in C[k]$
      end for
    end for
  end for
  Let $m = max\{ord(a) : a \in \mathcal{L}_P\}$ {$m$ is the order of $P$}
  Create $m$ sets of rules $P_0, ..., P_m$ {these will be the relevant modules of $P$}
  for all $a \in \mathcal{L}_P$ do
    add the rules whose head is $a$ to $P_{ord(a)}$
  end for
```

Algorithm 3 function $three\_in\_one(Module\ P)$

```
Require: Module $P$
  Graph $G${empty graph}
  bool $strat\_affected = false$ {heuristic variable }
  for all $A$ in $HEAD(P)${Apply the reductions} do
    for all $r \in P$ such that $head(r) = A$ do
      for all $a \in B(r)$ do
        if $state(a) = one$ then
          if $a \in B^+(r)$ then
            remove $a$ from $B^+(r))$
          else
            remove $r$ from $P)$
          end if
        else
          if $state(a) = zero$ then
            if $a \in B^+(r)$ then
              remove $r$ from $P)$
            else
              remove $a$ from $B^-(r))$
            end if
          end if
        end if
      end for
      if $r$ was removed in the loop above then
        if there is an atom $b \in B(r)$ with the same order than $H(r)$ then
          set $strat\_affected = true$
        end if
      end if
    end for
  end for
  $WFS(P)${Apply the transformations of the $CS$ system}
  if it was removed a rule by the function $WFS(P)$ then
    set $strat\_affected = true$
  end if
  create in $G$ a graph of dependencies from the rules remaining in $P$.
  return $strat\_affected$
```

Algorithm 4 function $stratify(ModuleP)$

```
Require: Module $P$
  if $three\_in\_one(P)$ then
    $create\_modules(P, G)${G is the graph of dependencies created in $three\_in\_one$}
    if $|subcomponents(P)| > 1$ then
      return true
    end if
    set $P =$ the first element of $submodules(P)$
    $subcomponents(P).clear()$
  end if
  return false
```

Algorithm 5 function $first\_EMM(Module\ P)$

```
Require: Module $P$
  if not $stratify(P)$ then
    $next\_minimal(P)$
  else
    set $Q =$the first element of $submodules(P)$
    $next\_minimal(Q)$
    set $Q = next(Q)$
    while $Q \neq null$ do
      $first\_EMM(Q)$
      set $Q = next(Q)$
    end while
  end if
  return
```

Algorithm 6 function $next\_EMM(Module\ P)$

```
Require: Module $P$
  if $|subcomponent(P)| = 0$ then
    if $next\_minimal(P)$ then
      return true
    end if
  else
    set $Q =$the last element of $subcomponents(P)$
    if $next\_EMM(Q)$ then
      return true
    end if
  end if
  {backtracks iff $back(P) \neq null$}
  if $back(P) = null$ or $not\ next\_EMM(back(P))$ then
    return false
  end if
  $reset\_module(P)$
  $first\_EMM(P)$
  return true
```

Algorithm 7 function $all\_EMM(Program\ PMM)$

```
Require: Program $PMM$
  Let $M$ be a empty set of models
  from $PMM$ create its graph of dependencies $G$
  $create\_modules(PMM, G)$
  $first\_EMM(PMM)$
  add to $M$ the model computed
  while $next\_EMM(PMM)$ do
    add to $M$ the model computed
  end while
  return $M$
```