2011

A transition system for AC language algorithms

Yuliya Lierler
University of Nebraska at Omaha, ylierler@unomaha.edu

Yaunlin Zhang
Texas Tech University

A transition system for AC language algorithms

Yuliya Lierler[1] and Yuanlin Zhang[2]

[1] University of Kentucky
[2] Texas Tech University

Abstract. Recently a logic programming language AC was proposed by Mellarkod et al. (2008) to integrate answer set programming (ASP) and constraint logic programming. In a similar vein, Gebser et al. (2009) proposed a CLINGCON language integrating ASP and finite domain constraints. A distinguishing feature of these languages is their capacity to allow new efficient inference algorithms that combine traditional ASP procedures and other efficient methods in constraint programming. In this paper we show that a transition system introduced by Nieuwenhuis et al. (2006) can be extended to model the "hybrid" ACSOLVER algorithm, by Mellarkod et al., designed for processing a class of *simple* AC programs. We also define a new class of *weakly-simple* programs and show how the introduced transition system describes a class of algorithms for such programs. Finally, we demonstrate that any CLINGCON program can be seen as an AC program.

1 Introduction

Mellarkod et al. [11] introduced a knowledge representation language AC extending the syntax and semantics of answer set programming (ASP) with constraint processing features. The AC language allows not only new modeling features but also a novel computational methods that combine traditional ASP algorithms with constraint logic programming (CLP) algorithms. Mellarkod et al. [11] presented a "hybrid" ACSOLVER system for finding answer sets of AC programs that combines both ASP and CLP computational tools. The key feature of this system is that it processes a "regular" part of a given program using ASP technology and a "defined" part using CLP tools.

In this paper we show that transition systems introduced by Nieuwenhuis, Oliveras, and Tinelli [13] to model and analyze SAT solvers can be adapted to model such a system as ACSOLVER. The SMODELS algorithm [12] is a classical method for computing answer sets of a program that is also a basis of the ASP search procedure implemented in ACSOLVER. Lierler [9] introduced a transition system (graph), called SM_Π, to model this algorithm. In fact, the graph SM_Π captures a class of algorithms that also includes SMODELS. It has been shown, for instance in [12] and [9], that transition systems are well-suited for proving correctness and analyzing algorithms. In this paper we extend the system SM_Π so that it may capture a class of hybrid ACSOLVER-like algorithms. We call a new transition system AC_Π. This allows us to provide an alternative description of the ACSOLVER system and an alternative proof of its correctness.

The ACSOLVER algorithm was proved to be correct for a class of "simple" programs [11]. We define a more general class of weakly-simple programs and demonstrate how the graph AC_Π immediately provides the means for describing a class of algorithms for such programs and demonstrating their correctness.

We also formally relate the language implemented in the constraint answer set solver CLINGCON [6]with the language of ACSOLVER. It turns out that any CLINGCON program can be mapped in a straightforward manner into an ACSOLVER program. We discuss the possibility of capturing the CLINGCON algorithm by a transition system.

The structure of the paper is as follows. We start by reviewing AC logic programs, notion of an answer set for such programs, and a class of simple programs. We then review a transition system introduced by Lierler [9] to model the SMODELS algorithm. In Section 4, we extend this transition system to model the computation of the ACSOLVER algorithm. Section 5 shows how the newly defined graph can be used to characterize the computation behind the system ACSOLVER. In the subsequent section we generalize the concept of simple programs to weakly-simple and show how the formal results demonstrating the correctness of the ACSOLVER algorithm on simple programs extend to weakly-simple programs. At last we outline some of the concepts and ideas needed for proving major statements in this paper. In an appendix we conclude with a discussion on the relation between the CLINGCON and ACSOLVER languages.

2 Review: AC Logic Programs

A *sort* (*type*) is a non-empty countable collection of strings over some fixed alphabet. Strings of a sort S will be referred to as *constants* of S. A signature Σ is a collection of sorts, properly typed predicate symbols, constants, and variables. Sorts of Σ are divided into *regular* and *constraint* sorts. *All variables in Σ are of a constraint sort.* A term of Σ is either a constant or a variable. An atom is of the form $p(t_1, \ldots, t_n)$ where p is an n-ary predicate symbol, and $t_1, ..., t_n$ are terms of the proper sorts. Normally, a constraint sort is often a large numerical set with primitive constraint relations.

The partitioning of sorts induces a partition of predicates of the AC language. *Regular predicates* denote relations among constants of regular sorts; *constraint predicates* denote primitive constraint relations on constraint sorts. *Defined predicates* denote relations between constants which belong to regular sort and objects which belong to constraint sorts. They are defined in terms of constraint, regular, and defined predicates. *Mixed* predicates denote relations between constants which belong to regular sort and objects which belong to constraint sorts. Mixed predicates are not defined by the rules of a program and are similar to abducible relations of abductive logic programming.

An atom formed by a regular predicate is called *regular*. Similarly for constraint, defined, and mixed atoms. We assume that any mixed atom is of the following restricted form – $m(\boldsymbol{r}, X)$, where $\boldsymbol{r}$ is a sequence of regular constants and X is a variable.

2.1 Regular Programs

A *regular program* is a finite set of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_l, not\ a_{l+1}, \ldots, not\ a_m, \tag{1}$$

where

- a_0 is $\bot$ or a regular atom, and
- each a_i $(1 \leq i \leq m)$ is $\bot$, $\top$ or a regular atom. [3]

The expression a_0 is the *head* of the rule. If $a_0 = \bot$, we often omit $\bot$ from the notation. If B denotes the *body*

$$a_1, \ldots, a_l, not\ a_{l+1}, \ldots, not\ a_m \tag{2}$$

of (1), we write B^{pos} for the elements occurring in the *positive* part of the body, i.e., $B^{pos} = \{a_1, \ldots, a_l\}$ and B^{neg} for the elements occurring in the *negative* part of the body, i.e., $B^{neg} = \{a_{l+1}, \ldots, a_m\}$.

For a regular program Π by Π^* we denote the program constructed from Π by dropping the rules where $\bot$ in B^{pos} or $\top$ in B^{neg}; dropping $\top$ and $not\ \bot$ from the remaining rules.

For instance, let Π_1 be a regular program

$$\begin{array}{l} a \leftarrow \bot,\ not\ b \\ b \leftarrow c, \top,\ not\ \top \\ d \leftarrow \top,\ not\ a, \end{array}$$

then Π_1^* is a program $d \leftarrow\ not\ a$.

A set X of atoms is an answer set for a regular program Π if X is an answer set in the sense of [7] for Π^*. For instance, $\{d\}$ is the only answer set of Π_1.

2.2 *AC* Programs

An *(*AC*) logic program* is a finite set of rules of the form (1) where

- a_0 is $\bot$ or a regular or defined atom,
- each a_i, $1 \leq i \leq l$, is an arbitrary atom if a_0 is $\bot$ or a regular atom
- each a_i, $1 \leq i \leq l$, is a regular, defined, or constraint atom if a_0 is a defined atom
- each a_i, $l+1 \leq i \leq m$, is a regular, defined, or constraint atom.

Note that this definition restricts the occurrence of mixed and constraint atoms: mixed atoms may occur only in the positive part of the body of a rule whose head is either a regular atom or $\bot$; constraint atoms may occur only in the body of a rule.

[3] In the paper, we do not use the term *literal* to refer to *not a*. We reserve the term *literal* exclusively for expressions of the form a and $\neg a$.

We frequently identify the body (2) of a rule (1) with the conjunction of its elements (in which *not* is replaced with the classical negation connective $\neg$):

$$a_1 \wedge \cdots \wedge a_l \wedge \neg a_{l+1} \wedge \cdots \wedge \neg a_m.$$

Similarly, we often interpret a rule (1) as a clause

$$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_l \vee a_{l+1} \vee \cdots \vee a_m \tag{3}$$

(in the case when $a_0 = \bot$ in (1) a_0 is absent in (3)). Given a program Π, we write Π^{cl} for the set of clauses (3) corresponding to all rules in Π.

Rule (1) is called a *defined* rule if a_o is a defined atom. A part of the AC program Π that consists of such rules is called a *defined* part that we denote by Π_D. By Π_R we will denote a non-defined part of Π, i.e., $\Pi \setminus \Pi_D$.

For instance, let signature Σ_1 contain two regular sorts $step = \{0\}$, $action = \{a\}$ and two constraint sorts $time = 0..200$, $machine = 1..2$; a mixed predicate $at(step, time)$, a regular predicate $occurs(action, step)$, and two defined predicates $acceptableTime(time)$ and $machineAvailable(machine, time)$. A sample AC program over Σ_1 follows

$$\begin{array}{l} machineAvailable(1, T) \leftarrow T \leq 5 \\ machineAvailable(2, 106) \leftarrow \\ acceptableTime(T) \leftarrow T \leq 10, machineAvailable(1, T) \\ acceptableTime(T) \leftarrow T \geq 100, machineAvailable(2, T) \\ \leftarrow occurs(a, 0), at(0, T),\ T \neq 1,\ not\ acceptableTime(T) \\ occurs(a, 0) \leftarrow \end{array} \tag{4}$$

The first four rules of the program form its defined part.

For an AC program Π over signature Σ, by the set $ground(\Pi)$ we denote the set of all ground instances of all rules in Π. The set $ground^{\top,\bot}(\Pi)$ is called a *basic ground instantiation* of Π and obtained from $ground(\Pi)$ by replacing each constraint atom occurring in $ground(\Pi)$ by

- $\top$ if it is *true* under the intended interpretation of its symbols, and
- $\bot$ if it is *false* under the intended interpretation of its symbols.

For instance, let $ground(\Pi)$ consist of two rules

$$\begin{array}{l} acceptableTime(100) \leftarrow 100 > 100,\ machineAvailable(2, 100) \\ acceptableTime(101) \leftarrow 101 > 100,\ machineAvailable(2, 101) \end{array}$$

then $ground^{\top,\bot}(\Pi)$ is

$$\begin{array}{l} acceptableTime(100) \leftarrow \bot,\ machineAvailable(2, 100) \\ acceptableTime(101) \leftarrow \top,\ machineAvailable(2, 101) \end{array}$$

We say that a sequence of (regular) constants $\boldsymbol{r}$ is *specified* by a mixed predicate m if $\boldsymbol{r}$ follows the sorts of the regular arguments of m. For instance, for program (4) a sequence 0 of constants (of type *step*) is the only sequence

specified by mixed predicate at. For a set X of atoms, we say that a sequence $\boldsymbol{r}$ of regular constants is *bound* in X by a (constraint) constant c w.r.t. predicate m if there is an atom $m(\boldsymbol{r}, c)$ in X.

We say that a set M of ground mixed atoms is *functional* over the underlying signature if for every mixed predicate m, every sequence of regular constants specified by m is bound in M by a unique constraint constant w.r.t. m. For instance, for the signature of program (4) sets $\{at(0,1)\}$ and $\{at(0,2)\}$ are functional whereas $\{at(0,1)\ at(0,2)\}$ is not a functional set because 0 is bound in M by two different constants 1 and 2 w.r.t. at.

For an AC program Π, a set X of atoms is called an *answer set* of Π if there is a functional set M of ground mixed atoms of Σ such that X is an answer set of $ground^{\top,\bot}(\Pi) \cup M$.

For example, the set of atoms

$$\{at(0,0),\ \ occurs(a,0),\\ machineAvailable(1,0), \ldots, machineAvailable(1,5),\ machineAvailable(2,106),\\ acceptableTime(0), \ldots, acceptableTime(5),\ acceptableTime(106)\}$$

is an answer set of (4).

The definition of an answer set for AC programs presented here is different from the original definition given in [11]. It is easy to see that there is a close relation between the answer sets defined in this paper and the answer sets given in [11].

Proposition 1. *For an* AC *program Π over signature Σ and the set S of all true ground constraint literals over Σ, X is an answer set of Π if and only if $X \cup S$ is an answer set (in the sense of [11]) of Π.*

Mellarkod et al. [11] considered programs of more sophisticated syntax than discussed here. For instance, in [11] classical negation may precede atoms in rules. Also signature Σ may contain variables of regular sort. On the other hand, the ACSOLVER algorithm introduced in [11] for processing AC programs considers programs of more restricted class than defined in this section. Since the primary goal of this paper is modeling the ACSOLVER procedure, we restricted our review only to the special case of AC programs relevant to ACSOLVER.

2.3 Simple AC Logic Programs

In [11] a class of simple AC programs was defined. The correctness of the ACSOLVER algorithm was shown for such programs. To be more precise, the ACSOLVER algorithm was proved to be correct for a class of "safe canonical" programs. Canonical programs are a special case of simple programs so that any simple program may be converted to a canonical program by means of syntactic transformations given in [11]. In this section we review simple programs.

We say that an AC program Π is *safe* [11] if every variable occurring in a non defined rule in Π also occurs in a mixed atom of this rule. We say that an AC program Π is *super safe* if Π is *safe* and

1. if a mixed atom $m(\boldsymbol{c}, X)$ occurs in Π then a mixed atom $m(\boldsymbol{c}, X')$ does not occur in Π (where X and X' are distinct variable names),
2. if a mixed atom $m(\boldsymbol{c}, X)$ occurs in Π then neither a mixed atom $m'(\boldsymbol{c}', X)$ such that $\boldsymbol{c} \neq \boldsymbol{c}'$ nor a mixed atom $m'(\boldsymbol{c}, X)$ such that $m \neq m'$ occurs in Π.

We note that any safe AC program Π may be converted to a super safe program so that the resulting program has the same answer sets:

Proposition 2. *For any safe* AC *program Π, there is a transformation on Π that produces a super safe* AC *program which has the same answer sets as Π.*

In Section 7 we present such a transformation.

We say that an AC program Π is *simple* if it is super safe, and its defined part contains no regular atoms and has a unique answer set. [4] For instance, program (4) is a simple program.

3 Review: Abstract Smodels

Most state-of-the-art answer set solvers are based on algorithms closely related to the DPLL procedure [2]. Nieuwenhuis et al. described DPLL by means of a transition system that can be viewed as an abstract framework underlying DPLL computation [13]. Lierler [9] proposed a similar framework, SM_Π, for specifying an answer set solver SMODELS [12]. Our goal is to design a similar framework for describing an algorithm behind ACSOLVER. As a step in this direction we review the graph SM_Π that underlines an algorithm of SMODELS (and also ACSOLVER) for a special case of AC programs that contain only regular atoms (we call such programs – *r-programs*). The presentation follows Lierler [9].

For a set σ of atoms, a *record* relative to σ is an ordered set M of literals over σ, some possibly annotated by Δ, which marks them as *decision* literals. A *state* relative to σ is a record relative to σ possibly preceding symbol $\bot$. For instance, some states relative to a singleton set $\{a\}$ of atoms are

$$\emptyset, \quad a, \quad \neg a, \quad a^{\Delta}, \quad a\ \neg a, \quad \bot, \quad a\bot, \quad \neg a\bot, \quad a^{\Delta}\bot, \quad a\ \neg a\bot$$

We say that a state is inconsistent if either $\bot$ or two complementary literals occur in the state. For instance, states $a\ \neg a$ and $a\bot$ are inconsistent. Frequently, we consider a state M as a set of literals possibly with the symbol $\bot$, ignoring both the annotations and the order between its elements. If neither a literal l nor its complement occur in M, then l is *unassigned* by M. For a set M of literals, by M^+ and M^- we denote the set of positive and negative literals in M respectively. For instance, $\{a, \neg b\}^+ = \{a\}$ and $\{a, \neg b\}^- = \{b\}$.

[4] An original definition of a simple program given in [11] is less restrictive, i.e., such program is not required to be super safe. On the one hand, the "super safe" restriction is important for correctness results stated later in the paper. On the other hand, it is not essential: (i) any AC program may be transformed into a safe program and (ii) any safe program may be converted to super safe program by Proposition 2.

If C is a disjunction (conjunction) of literals then by $\overline{C}$ we understand the conjunction (disjunction) of the complements of the literals occurring in C. In some situations, we will identify disjunctions and conjunctions of literals with the sets of these literals.

By $Bodies(\Pi, a)$ we denote the set of the bodies of all rules of an AC program Π with the head a. We recall that a set U of atoms occurring in an r-program Π is *unfounded* [14] on a consistent set M of literals w.r.t. Π if for every $a \in U$ and every $B \in Bodies(\Pi, a)$, $M \models \overline{B}$ or $U \cap B^{pos} \neq \emptyset$.

Each r-program Π determines its *Smodels graph* SM_Π. The set of nodes of SM_Π consists of the states relative to the set of atoms occurring in Π. The edges of the graph SM_Π are specified by the transition rules presented in Figure 1.

Unit Propagate:	$M \implies M\ l$	if $C \vee l \in \Pi^{cl}$ and $\overline{C} \subseteq M$
Decide:	$M \implies M\ l^{\Delta}$	if l is unassigned by M
Fail:	$M \implies \bot$	if M is inconsistent, and M contains no decision literals
Backtrack:	$P\ l^{\Delta}\ Q \implies P\ \bar{l}$	if $P\ l^{\Delta}\ Q$ is inconsistent, and Q contains no decision literals
All Rules Cancelled:	$M \implies M\ \neg a$	if $\overline{B} \cap M \neq \emptyset$ for all $B \in Bodies(\Pi, a)$
Backchain True:	$M \implies M\ l$	if $a \leftarrow B \in \Pi,\ a \in M,\ l \in B$, $\overline{B'} \cap M \neq \emptyset$ for all $B' \in Bodies(\Pi, a) \setminus B$
Unfounded:	$M \implies M\ \neg a$	if M is consistent, and $a \in U$ for a set U unfounded on M w.r.t. Π

Fig. 1. The transition rules of the graph SM_Π.

A node is *terminal* in a graph if no edge leaves this node.

The graph SM_Π can be used for deciding whether an r-program Π has an answer set by constructing a path from $\emptyset$ to a terminal node.

Proposition 3 (Proposition 3 in [9]). *For any r-program Π,*

(a) graph SM_Π is finite and acyclic,
(b) for any terminal state M of SM_Π other than $\bot$, M^+ is an answer set of Π,
(c) state $\bot$ is reachable from $\emptyset$ in SM_Π if and only if Π has no answer sets.

4 Abstract ACSOLVER

In order to present the transition system suitable for capturing the ACSOLVER algorithm we first need to introduce a few concepts used in the description of this system, i.e., queries and query satisfiability.

Query, Query Interpretation, and Satisfiability: Given an AC program Π and a set $\mathbf{p}$ of predicate symbols, a set X of atoms is an *input answer set* of Π w.r.t. $\mathbf{p}$ if X is an answer set of $\Pi \cup X(\mathbf{p})$ where by $X(\mathbf{p})$ we denote the set of atoms in X whose predicate symbols are different from the ones occurring in $\mathbf{p}$[5]. For instance, let X be a set $\{a(1), b(1)\}$ of atoms and let $\mathbf{p}$ be a set $\{a\}$ of predicates, then $X(\mathbf{p})$ is $\{b(1)\}$. The set X is an input answer set of a program $a(1) \leftarrow b(1)$ w.r.t. $\mathbf{p}$. On the other hand it is not an input answer set for the same program with respect to a set $\{a, b\}$ of predicates.

For a set S of literals, by S_R we denote the set of regular literals occurring in S.

A *query* is a set of defined, regular, and constraint literals. A consistent set I of ground literals over signature Σ is called a *query interpretation* for a query Q w.r.t. an AC program Π when

1. there is a (sort respecting) substitution γ of variables in Σ by ground terms such that the result, $Q\gamma$, of this substitution is I itself,
2. if a constraint literal $l \in I$ then l is true under the intended interpretation of its symbols, and
3. there is an input answer set A of Π_D w.r.t. defined predicates of Π such that $Q_R^+ \subseteq A$, $Q_R^- \cap A = \emptyset$, and
   - if $a \in I$ and a is a defined atom then $a \in A$,
   - if $\neg a \in I$ and a is a defined atom then $a \notin A$.

We say that a query Q is *satisfiable* w.r.t. an AC program Π if there exists a query interpretation for Q w.r.t. Π.

For example, $\{acceptableTime(2),\ \ 2 \neq 1\}$ is a query interpretation for a query $\{acceptableTime(T),\ \ T \neq 1\}$ w.r.t. program (4). It is clear that $\{acceptableTime(T),\ \ T \neq 1\}$ is a satisfiable query.

The graph AC_Π: We say that a set U of regular atoms occurring in an AC program Π consisting of the rules of the form (1) is *unfounded* on a consistent set M of literals w.r.t. Π if for every $a \in U$ and every $B \in Bodies(\Pi, a)$, $M \models \overline{B}$ or $U \cap B^{pos} \neq \emptyset$.

For an AC program Π, by Π_R^m we denote a program Π_R whose mixed atoms are dropped. For instance, let Π be (4), then Π_R consists of the rules

$$\leftarrow occurs(a, 0), at(0, T),\ T \neq 1,\ not\ acceptableTime(T)$$
$$occurs(a, 0) \leftarrow$$

[5] Intuitively set $\mathbf{p}$ denotes a set of so called intensional predicates [4]. Also, the concept of an input answer set w.r.t. $\mathbf{p}$ is closely related to the concept of "$\mathbf{p}$-stable model" in [3]. We make this claim precise in Section 7. It is also related to the concept of an input answer set introduced in [10].

and Π^m_R consists of the rules

$$\leftarrow occurs(a,0),\ T \neq 1,\ not\ acceptableTime(T)$$
$$occurs(a,0) \leftarrow$$

For a set S of literals by $S_{D,C}$ we denote the set of defined and constraint literals occurring in S. By $|S|$ we denote the set of atoms occurring in S (either positively or negatively). For instance, $|\{a, \neg b\}| = \{a, b\}$.

For a state M, by $query(M)$ we denote the set $M_{D,C} \cup M'$ where M' is the largest subset of M_R such that $|M'|$ is a subset of regular atoms occurring in Π_D. It is easy to see that for simple programs, $query(M) = M_{D,C}$.

Let Π be an AC logic program. The nodes of AC_Π are the states relative to the set of atoms occurring in Π^m_R. The edges of the graph AC_Π are described by the transition rules of $\text{SM}_{\Pi^m_R}$ and the additional transition rule

Query Propagation: $M \implies M\ \bot$ if $query(M)$ is unsatisfiable w.r.t. Π_D

The graph AC_Π can be used for deciding whether a simple AC program Π has an answer set by constructing a path from $\emptyset$ to a terminal node:

Proposition 4. *For any simple* AC *program* Π*,*

(a) graph AC_Π *is finite and acyclic,*
(b) for any terminal state M *of* AC_Π *other than* $\bot$*,* M^+_R *is a set of all regular atoms in some answer set of* Π*,*
(c) state $\bot$ *is reachable from* $\emptyset$ *in* AC_Π *if and only if* Π *has no answer sets.*

Proposition 4 shows that algorithms that correctly find a path in the graph AC_Π from $\emptyset$ to a terminal node can be regarded as AC solvers for simple programs. It also provides a proof of correctness for every AC solver that can be shown to work in this way. In the next section we use the AC_Π graph to describe such solver, i.e., ACSOLVER.

For instance, let Π be an AC program (4). Here is a path in AC_Π with every edge annotated by the name of a transition rule that justifies the presence of this edge in the graph:

$$\emptyset \implies^{Unit\ Propagate} occurs(a,0) \implies^{Decide} occurs(a,0)\ \ T \neq 1^\Delta$$
$$\implies^{Unit\ Propagate} occurs(a,0)\ \ T \neq 1^\Delta\ \ acceptableTime(T).$$

Since the last state in the path is terminal, Proposition 4 asserts that $occurs(a,0)$ is a set of all regular atoms in some answer set of Π.

5 ACSOLVER Algorithm

We can view a path in the graph AC_Π as a description of a process of search for a set of regular atoms in some answer set of Π by applying transition rules. Therefore, we can characterize an algorithm of a solver that utilizes the transition rules of AC_Π by describing a strategy for choosing a path in this graph. A

strategy can be based, in particular, on assigning priorities to transition rules of AC_{Π}, so that a solver never applies a rule in a state if a rule with higher priority is applicable to the same state. A strategy may also include restrictions on how a rule is applied.

We use this approach to describe the ACSOLVER algorithm [11, Fig.1]. The ACSOLVER selects edges according to the priorities on the transition rules of the graph AC_{Π} as follows:

Backtrack, *Fail* >> *Unit Propagate*, *All Rules Cancelled*, *Backchain True* >> *Unfounded* >> *Query Propagation* >> *Decide* (*if unassigned literal* l *is regular*).

Note that ACSOLVER only follows a transition due to the rule *Decide* where unassigned literal l is regular.

We recall that the ACSOLVER algorithm is applicable to simple AC programs only. Its implementation consists of two interacting parts: an SMODELS like algorithm and a CLP solver [8]. The former makes decisions and computes consequences (by applying the transition rules *Unit Propagate*, *All Rules Cancelled*, and *Backchain True*) and unfounded sets (by applying *Unfounded*). In the process, queries consisting of defined and constraint literals are created. The satisfiability of queries is checked using a CLP solver that is based on a depth first search on a derivation tree constructed for the queries in terms of Π_D [8]. We note that the use of the CLP solver in the implementation of the ACSOLVER algorithm puts additional restrictions on the program Π_D. In particular, this program should be acyclic [1, Definition 1.4, Corollary 4.3].

Mellarkod et al. [11] demonstrated the correctness of the ACSOLVER algorithm by analyzing the properties of its pseudocode. Proposition 4 provides an alternative proof of correctness to the ACSOLVER algorithm that relies on the transition system AC_{Π} suitable for describing ACSOLVER. Proposition 4 encapsulates the proof of correctness for a class of algorithms that can be described using AC_{Π}. Therefore, for instance, it immediately follows that the ACSOLVER algorithm modified in a way that it may follow an arbitrary transition due to the rule *Decide* is still correct.

6 Weakly-simple AC Logic Programs

In this section we define a more general class of programs than simple and call them weakly-simple programs. Proposition 4 still holds when we replace "simple" there by "weakly-simple". This finding immediately makes any algorithm based on the graph AC_{Π} applicable to a more general class of programs.

For any atom $p(\boldsymbol{t})$, by $p(\boldsymbol{t})^0$ we denote its predicate symbol p. For any AC program Π, the *predicate dependency graph* of Π is the directed graph that (i) has all predicates occurring in Π as its vertices, and (ii) for each rule (1) in Π has an edge from a_0^0 to a_i^0 where $1 \leq i \leq l$. A similar definition of predicate dependency graph was given in [4] for programs of more general syntax.

We say that an AC program Π is *weakly-simple* if it is super safe and each strongly connected component of the predicate dependency graph of Π is a

subset of either regular predicates of Π or defined predicates of Π. It is easy to see that any simple program is also a weakly-simple program but not the other way around.

Let Σ_2 extend the signature Σ_1 of program (4) by a regular sort $power = \{on, off\}$ and a regular predicate $switch(power)$. The following weakly-simple program over Σ_2 results from modifying the first two rules of simple program (4) and adding a fact $switch(off)$:

$$\begin{array}{l} machineAvailable(1,T) \leftarrow T \leq 5, switch(on) \\ machineAvailable(2,106) \leftarrow switch(on) \\ acceptableTime(T) \leftarrow T \leq 10, machineAvailable(1,T) \\ acceptableTime(T) \leftarrow T \geq 100, machineAvailable(2,T) \\ \leftarrow occurs(a,0), at(0,T),\ T \neq 1,\ not\ acceptableTime(T) \\ occurs(a,0) \leftarrow \\ switch(off) \leftarrow \end{array} \tag{5}$$

Finally, Proposition 4', obtained from Proposition 4 by replacing "simple" with "weakly-simple", holds. Consequently any algorithm based on the graph AC_Π may be applied to program (5). As for an implementation of such an algorithm, since Π_D contains regular atoms, a classical CLP solver is not directly applicable to evaluate $query(M)$ in the *Query Propagation* transition rule. To overcome this issue, one way is to apply this rule only when every regular atom of Π_D occurs in some literal of $query(M)$. In such case, we can reduce Π_D to defined rules without regular atoms using the following transformation. First, every regular atom a in Π_D is replaced by $\top$ if $a \in query(M)$ and by $\bot$ if $\neg a \in query(M)$. We denote the resulting program by Π'_D. We construct a program $(\Pi'_D)^*$ from Π'_D by dropping the rules where $\bot$ ($\top$) occurs in B^{pos} (B^{neg}) and dropping $\top$ and $not\bot$ from the remaining rules. Under a condition that the program $(\Pi'_D)^*$ is acyclic [1, Definition 1.4], a CLP solver can test the satisfiability of the query $query(M)$ in terms of $(\Pi'_D)^*$.

7 Outlines of Some Proofs

Proposition 2 *For any safe* AC *program Π, there is a transformation on Π that produces a super safe* AC *program which has the same answer sets as Π.*

Let T denote the following transformation on an AC program

For each sequence $\boldsymbol{c}$ of constants and each m such that $\boldsymbol{c}$ is specified by m
  Associate a unique new variable with $< m, \boldsymbol{c} >$;
For each rule r of Π
  Let r' be the same as r;
    For each variable X of r
      Let $m_1(\boldsymbol{c}_1, X), ...m_k(\boldsymbol{c}_k, X)$ be the mixed atoms of r and
        Y_i be the unique new variable associated with $< m_i, \boldsymbol{c_i} >$ for $i \in 1..k$;
      Replace $m_i(\boldsymbol{c_i}, X)$ in r' with $m_i(\boldsymbol{c_i}, Y_i)$ for $i \in 1..k$;
      Add $Y_1 = Y_2 = \ldots = Y_i$ to the body of r';
      Replace each occurrence of X in r' by Y_1.

The key to the proof of Proposition 2 is to demonstrate that a program Π' produced by the transformation T from a safe AC program Π has the same answer sets as Π and is super safe.

Proposition 4′ *For any weakly-simple* AC *program* Π*,*

(a) graph AC_Π *is finite and acyclic,*
(b) for any terminal state M *of* AC_Π *other than* $\bot$*,* M_R^+ *is a set of all regular atoms in some answer set of* Π*,*
(c) state $\bot$ *is reachable from* $\emptyset$ *in* AC_Π *if and only if* Π *has no answer sets.*

The proof of Proposition 4' relies on an alternative characterization of the answer sets of weakly-simple programs that we state as Lemma 1 in this section. To state the lemma we will introduce several concepts.

For an AC program Π, by $atoms_R(\Pi)$ we denote the set of regular atoms occurring in Π. By $atoms_{D,C}(\Pi)$ we denote the set of defined and constraint atoms occurring in Π. We say that a query Q is *based* on an AC program Π if $|Q_{D,C}| = atoms_{D,C}(\Pi_R)$, and $|Q_R| = atoms_R(\Pi_D)$.

For instance, for program (4) there are four queries based on Π:

$$\{acceptableTime(T),\ \ T \neq 1\}\quad \{acceptableTime(T),\ \ \neg T \neq 1\}$$
$$\{\neg acceptableTime(T),\ \ T \neq 1\}\ \{\neg acceptableTime(T),\ \ \neg T \neq 1\}$$

For a query Q and an AC program Π, by $\Pi(Q)$ we denote a program constructed from Π by

1. eliminating Π_D,
2. replacing each occurrence of a mixed atom by $\top$,
3. replacing each a_i $(1 \leq i \leq m)$ in (1) such that a_i or $\neg a_i$ is in $Q_{D,C}$ by $\top$ if $a_i \in Q$, and $\bot$ if $\neg a_i \in Q$.
4. for each regular literal $l \in Q_R$ adding a rule
   - $\leftarrow$ *not* l if l is an atom, and
   - $\leftarrow$ a if l is a literal $\neg a$.

It is easy to see that for a query Q based on an AC program Π, $\Pi(Q)$ is a regular program. For instance, let Π be program (4) and Q be $\{acceptableTime(T),\ T \neq 1\}$, then $\Pi(Q)$ consists of two rules

$$\leftarrow occurs(a,0), \top,\ \top,\ not\ \top$$
$$occurs(a,0) \leftarrow$$

Weakly-simple AC programs satisfy important syntactic properties that allow to characterize their answer sets by means of queries based on them. The lemma below makes this claim precise. This lemma is also a key to proving Proposition 4'.

Lemma 1. *For a weakly-simple* AC *program* Π*,* Π *has an answer set iff there is a query* Q *based on* Π *such that* Q *is satisfiable w.r.t.* Π *and* $\Pi(Q)$ *has an answer set. Furthermore, if* I *is a query interpretation for* Q *w.r.t.* Π *and* X *is an answer set of* $\Pi(Q)$ *then* $X \cup I_D^+$ *is a subset of an answer set of* Π*.*

The proof of this lemma heavily relies on the (symmetric) Splitting Theorem in [4] and the following relation between an input answer set of a regular program Π and a **p**-stable model [3]:

Proposition 5. *For a regular logic program Π, a complete set X of literals, and a set* **p** *of predicate symbols such that $pred(Heads(\Pi)) \subseteq \mathbf{p}$, X^+ is an input answer set of Π iff X is a model of $SM_{\mathbf{p}}[\Pi]$ (i.e.,* **p***-stable model of Π).*

Proposition 4' follows immediately from Lemma 1 and the following result:

Lemma 2. *For any weakly-simple* AC *program Π,*

(a) graph AC_Π is finite and acyclic,
(b) for any terminal state M of AC_Π other than $\bot$, $query(M)$ is a satisfiable query based on Π and M_R^+ is an answer set of $\Pi(query(M))$,
(c) state $\bot$ is reachable from $\emptyset$ in AC_Π if and only if Π has no answer sets.

The proof of this lemma is similar in its structure to the proof of Proposition 3 given in [9].

8 Appendix: CLINGCON Algorithm

Let us consider a subset of the AC language, AC^-, so that any AC program without defined atoms is an AC^- program. It is easy to see that for any AC^- program, its defined part is empty so that any super safe AC^- program is simple.

The language of the constraint answer set solver CLINGCON defined in [6] is strongly related to the AC language.[6] In fact, it can be seen as a syntactic variant of the AC^- language.

We now review the CLINGCON programs and show how they map into AC^- programs.

We say that an atom is a *clingcon atom* over (sorted) signature Σ if it has the following form

$$\begin{array}{l} p_1(\boldsymbol{r_1}) \oplus \cdots \oplus p_k(\boldsymbol{r_k}) \oplus c_{k+1} \oplus \cdots \oplus c_m \quad \odot \\ p_{m+1}(\boldsymbol{r}_{m+1}) \oplus \cdots \oplus p_l(\boldsymbol{r_l}) \oplus c_{l+1} \oplus \cdots \oplus c_n, \end{array} \tag{6}$$

where p_i is a mixed predicate and $\boldsymbol{r_i}$ is a vector of regular constants; c_i is a constraint constant; $\oplus$ is a primitive constraint operation; and $\odot$ is a primitive constraint relation. We call expressions of the form $p_i(\boldsymbol{r_i})$ *clingcon variables*.

A *clingcon program* is a finite set of rules of the form (1) where

- a_0 is $\bot$ or a regular atom, and
- each a_i, $1 \leq i \leq m$ is a regular atom or clingcon atom.

Any clingcon program Π can be rewritten as an AC^- program using a function V that maps the set of clingcon variables occurring in Π to the set of distinct variables over Σ. For a clingcon variable c, c^V denotes a variable assigned to c by V. For each occurrence of clingcon atom (6) in some rule r of Π

[6] The system CLINGCON accepts programs of more general syntax than discussed in [6] (for instance, aggregates such as *#count* are allowed by CLINGCON).

– add a set of mixed atoms $p_1(\boldsymbol{r_1}, p_1(\boldsymbol{r_1})^V), \ldots, p_l(\boldsymbol{r_l}, p_l(\boldsymbol{r_l})^V)$ to the body of r,
– replace (6) in r by a constraint atom

$$p_1(\boldsymbol{r_1})^V \oplus \cdots \oplus p_k(\boldsymbol{r_k})^V \oplus c_{k+1} \oplus \cdots \oplus c_m \quad \odot \\ p_{m+1}(\boldsymbol{r_{m+1}})^V \oplus \cdots \oplus p_l(\boldsymbol{r_l})^V \oplus c_{l+1} \oplus \cdots \oplus c_n.$$

We denote resulting AC^{-} program by $ac(\Pi)$. It is easy to see that $ac(\Pi)$ is super safe program.

Proposition 6. *For a clingcon program Π over signature Σ, a set X is a constraint answer set of Π according to the definition in [6] iff there is a functional set M of ground mixed atoms of Σ such that $X \cup M$ is an answer set of $ac(\Pi)$.*

Based on Proposition 6 and the fact that for a clingcon program Π, $ac(\Pi)$ is a super-safe AC^{-} program it follows that a class of algorithms captured by the graph AC_Π is applicable to clingcon programs. Nevertheless the graph AC_Π is not suitable for describing the CLINGCON system. This system is based on tight coupling of the answer set solver CLASP [5] and the constraint (CSP) solver GECODE[7]. The CLINGCON starts its computation by building the "completion" of a clingcon program so that its propagation relies not only on the program but also on the program's propositional formula counterpart. Furthermore, it implements such backtracking search techniques such as backjumping, learning, forgetting, and restarts. Lierler and Truszczynski [10] introduced the transition system $\text{SML(ASP)}_{F,\Pi}$ and demonstrated how it can be used to capture the computation of CLASP. The graph $\text{SML(ASP)}_{F,\Pi}$ extended with the transition rule *Query Propagation* (in a similar manner as the graph SM_Π was extended with *Query Propagation* in this paper) is appropriate for describing the system CLINGCON. Furthermore, $\text{SML(ASP)}_{F,\Pi}$ extended with the transition rule *Query Propagation* would provide a generic description of a class of algorithms implementing backjumping and learning for the case of weakly-simple AC programs. The detailed description of this graph is out of the scope of this paper.

9 Conclusions

In this paper, we designed a transition system AC_Π that is well-suited for describing an algorithm behind the system ACSOLVER. The language defined by Gebser et al. [6] for the system CLINGCON is formally related to a subset, AC^{-}, of AC language (see appendix). As a result, the transition system AC_Π is also applicable to a class of algorithms for the CLINGCON language. (Although, it is not adequate to specify the procedure implemented in the system CLINGCON.) Compared with traditional pseudo-code description of algorithms, transition systems use a more uniform (i.e., graph based) language and offer more modular proofs. The graph AC_Π offers a convenient tool to describe, compare, analyze, and prove correctness for a class of algorithms. Furthermore, the transition system for ACSOLVER results in new algorithms for solving a larger class of AC

[7] `http://www.gecode.org` .

programs – weakly-simple programs. Neither the ACSOLVER nor CLINGCON procedures presented in [11] and [6], respectively, can deal with such programs. In the future we will consider ways to use current ASP/CLP technologies to design a solver for weakly-simple programs.

Acknowledgments

We are grateful to Michael Gelfond, Vladimir Lifschitz, and Miroslaw Truszczynski for useful discussions related to the topic of this work. Yuliya Lierler was supported by a CRA/NSF 2010 Computing Innovation Fellowship, and Yuanlin Zhang was partially supported by NSF under grant IIS-1018031.

References

1. Apt, K., Bezem, M.: Acyclic programs. New Generation Computing 9, 335–363 (1991)
2. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM 5(7), 394–397 (1962)
3. Ferraris, P., Lee, J., Lifschitz, V.: Stable models and circumscription. Artificial Intelligence 175, 236–263 (2011)
4. Ferraris, P., Lee, J., Lifschitz, V., Palla, R.: Symmetric splitting in the general theory of stable models. In: IJCAI. pp. 797–803 (2009)
5. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI. pp. 386–392 (2007)
6. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: International Conference on Logic Programming (ICLP). pp. 235–249 (2009)
7. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: International Logic Programming Conference and Symposium (1988)
8. Jaffar, J., Maher, M.J.: Constraint Logic Programming. Journal of Logic Programming 19/20, 503–581 (1994)
9. Lierler, Y.: Abstract answer set solvers. In: International Conference on Logic Programming (ICLP) (2008)
10. Lierler, Y., Truszczynski, M.: Transition systems for model generators — a unifying approach. In: International Conference on Logic Programming (ICLP) (2011)
11. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Ann of Math and Artif Intell (2008)
12. Niemelä, I., Simons, P.: Extending the Smodels system with cardinality and weight constraints. In: Logic-Based Artificial Intelligence, pp. 491–521 (2000)
13. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
14. Van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of ACM 38(3), 620–650 (1991)