On the Relation of Constraint Answer Set Programming Languages and Algorithms

Yuliya Lierler
Department of Computer Science
The University of Kentucky
329 Rose Street
Lexington, KY 40506
yuliya@cs.uky.edu

Abstract

Recently a logic programming language *AC* was proposed by Mellarkod et al. (2008) to integrate answer set programming (ASP) and constraint logic programming. Similarly, Gebser et al. (2009) proposed a CLINGCON language integrating ASP and finite domain constraints. These languages allow new efficient inference algorithms that combine traditional ASP procedures and other methods in constraint programming. In this paper we show that a transition system introduced by Nieuwenhuis et al. (2006) to model SAT solvers can be extended to model the "hybrid" ACSOLVER algorithm by Mellarkod et al. developed for *simple AC* programs and the CLINGCON algorithm by Gebser et al. for clingcon programs. We define *weakly-simple* programs and show how the introduced transition systems generalize the ACSOLVER and CLINGCON algorithms to such programs. Finally, we state the precise relation between *AC* and CLINGCON languages and the ACSOLVER and CLINGCON algorithms.

Introduction

Mellarkod et al. (2008) introduced a knowledge representation language *AC* extending the syntax and semantics of answer set programming with constraint processing features. The origins of their work go back to (Baselice, Bonatti, and Gelfond 2005). In a similar vein, Gebser et al. (2009) proposed a CLINGCON language integrating ASP and finite domain constraints. The *AC* and CLINGCON languages allow not only new modeling features but also novel computational methods that combine traditional ASP algorithms with constraint (logic) programming (CLP/CSP) algorithms. This combined approach opens new horizons for declarative programming applications. For instance, it allows us to reason about problems with variables whose values range over very large domains such as dynamic systems in real time. Mellarkod et al. 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 the ASP algorithm SMODELS (Niemelä and Simons 2000) and a "defined" part using CLP tools. Similarly, system CLINGCON (Gebser, Ostrowski, and Schaub 2009) takes advantage of answer set solver CLASP (Gebser et al. 2007) and constraint solver GECODE (`http://www.gecode.org`). It is intuitively clear that the *AC* and CLINGCON languages are related as well as the systems ACSOLVER and CLINGCON. This paper puts these relationships in precise mathematical terms.

We show that transition systems (graphs) introduced by Nieuwenhuis et al. (2006) to model and analyze SAT solvers can be adapted to describe constraint answer set solvers ACSOLVER and CLINGCON. By introducing such new transition systems we provide an alternative description of ACSOLVER and CLINGCON and also an alternative proof of their correctness. This abstract view on the systems allows us to state the relation between them in precise terms by studying the underlying graph representations. The ACSOLVER algorithm was proved to be correct for a class of "simple" programs. We define a more general class of weakly-simple programs and demonstrate how newly introduced transition systems immediately capture a class of algorithms for such programs and demonstrate their correctness.

This work clarifies and extends state of the art developments in the area of constraint answer set programming and we believe will promote further progress in the area.

We start by reviewing *AC* logic programs and a notion of an answer set for such programs. We introduce a new class of weakly-simple programs. We then review a transition system introduced by Lierler (2008) to model SMODELS. We extend this transition system to model the ACSOLVER algorithm and show how the newly defined graph can characterize the computation behind the system ACSOLVER. In the subsequent section we introduce the CLINGCON language and formally state its relation to the *AC* language. At last we define a graph suitable for modeling the system CLINGCON and state a formal result on the relation between the ACSOLVER and CLINGCON algorithms.

A preliminary report on some of the results of this paper has been presented at Workshop on Answer Set Programming and Other Computing Paradigms (Lierler and Zhang 2011).

Review: AC Logic Programs

A *sort* (*type*) is a non-empty countable collection of strings over some fixed alphabet. 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. 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 that belong to regular sort and objects that belong to constraint sorts; such predicates can be defined in terms of constraint, regular, and defined predicates;
- *Mixed* predicates denote relations between constants that belong to regular sort and objects that 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 (Kakas, Kowalski, and Toni 1992).

An atom formed by a regular predicate is called *regular*. Similarly for constraint, defined, and mixed atoms. We say that an atom is a *non-mixed atom* if it is regular, constraint, or defined.

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

$$a_0 \leftarrow a_1, \ldots, a_l, not\ a_{l+1}, \ldots, not\ a_m, \\ not\ not\ a_{m+1}, \ldots, not\ not\ a_n, \quad (1)$$

where a_0 is $\bot$ or a ground (non-constraint) atom, and each a_i $(1 \leq i \leq n)$ is a ground (non-constraint) atom. If $a_0 = \bot$, we often omit $\bot$ from the notation. This is a special case of programs with nested expressions (Lifschitz, Tang, and Turner 1999). We assume that the reader is familiar with the definition of an answer set of a logic program and refer to the paper by Lifschitz et al. (1999) for details. A *choice rule* construct $\{a\}$ (Niemelä and Simons 2000) of the LPARSE[1] language can be seen as an abbreviation for a rule $a \leftarrow not\ not\ a$ (Ferraris and Lifschitz 2005). We adopt this abbreviation in the rest of the paper.

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 non-mixed atom if a_0 is a defined atom,
- each a_i, $l+1 \leq i \leq m$, is a non-mixed atom,
- each a_i, $m+1 \leq i \leq n$, is a regular atom,
- $n = m$, if a_0 is a defined atom.

Rule (1) is called a *defined* rule if a_o is a defined atom. It is easy to see that defined rules of a program neither contain mixed atoms in its body nor contain doubly negated atoms ($not\ not\ a$). We assume that any mixed atom occurring in *AC* program is of the restricted form $m(\vec{r}, V)$, where $\vec{r}$ is a sequence of regular constants and V is a variable.

A part of the *AC* program Π that consists of defined rules is called a *defined* part denoted by Π_D. By Π_R we 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 $time = \{0..200\}$, $computer = \{1..2\}$; a mixed predicate $at(step, time)$, two regular predicates $occurs(action, step)$, on, and two defined predicates $okTime(time)$ and $okComp(computer, time)$. A sample *AC* program over Σ_1 follows

$$\begin{array}{l} okComp(1,T) \leftarrow T \leq 5, on \\ okComp(2,106) \leftarrow on \\ okTime(T) \leftarrow T \leq 10, okComp(1,T) \\ okTime(T) \leftarrow T \geq 100, okComp(2,T) \\ \leftarrow occurs(a,0), at(0,T),\ T \neq 1,\ not\ okTime(T) \\ occurs(a,0) \leftarrow \\ \{on\} \end{array} \quad (2)$$

The first four rules of the program form its defined part whereas the last three rules form Π_R.

Mellarkod et al. (Mellarkod, Gelfond, and Zhang 2008) considered programs of more sophisticated syntax than discussed here. For instance, in (Mellarkod, Gelfond, and Zhang 2008) classical negation may precede atoms in rules. Also signature Σ may contain variables of regular sort. Nevertheless, the *AC* language discussed here is sufficient to capture the class of programs covered by the ACSOLVER algorithm.

The expression a_0 is the *head* of a rule (1). If B denotes the *body* of (1), the right hand side of the arrow, 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 under single negation as failure, i.e., $B^{neg} = \{a_{l+1}, \ldots, a_m\}$. We frequently identify the body of (1) with the conjunction of its elements (in which not is replaced with the classical negation connective $\neg$):

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

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

$$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_l \vee a_{l+1} \vee \cdots \vee a_m \vee \neg a_{m+1} \vee \cdots \vee \neg a_n \quad (3)$$

(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.

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 $ground^*(\Pi)$ is obtained from $ground(\Pi)$ by

- dropping the rules where a constraint atom a occurs in B^{pos} (B^{neg}) and a is *false* (*true*, respectively) under the intended interpretation of its symbols,
- dropping all constraint literals from the remaining rules (here we use the term *literal* to refer to a and $not\ a$.)

It is easy to see that $ground^*(\Pi)$ is a regular program.

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

$$okTime(100) \leftarrow 100 > 100,\ okComp(2,100)$$
$$okTime(101) \leftarrow 101 > 100,\ okComp(2,101)$$

[1] http://www.tcs.hut.fi/Software/smodels/ .

then $ground^*(\Pi)$ is

$$okTime(101) \leftarrow \ okComp(2, 101).$$

We say that a sequence of (regular) constants $\vec{r}$ is *specified* by a mixed predicate m if $\vec{r}$ follows the sorts of the regular arguments of m. For instance, for program (2) 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 $\vec{r}$ of regular constants is *bound* in X by a (constraint) constant c w.r.t. predicate m if there is an atom $m(\vec{r}, c)$ in X. 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 (2) 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.

Definition 1 *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* $ground^*(\Pi) \cup M$.

For example, sets of atoms

$$\{at(0, 1),\ occurs(a, 0)\} \tag{4}$$

and

$$\{on,\ \ at(0, 0),\ \ occurs(a, 0),\\ okComp(1, 0), \ldots, okComp(1, 5),\ okComp(2, 106),\\ okTime(0), \ldots, okTime(5),\ okTime(106)\}$$

are answer sets of (2).

The definition of an answer set for *AC* programs presented here is different from the original definition in (Mellarkod, Gelfond, and Zhang 2008), but there is a close relation between them.

Proposition 1 *For an* AC *program* Π *over signature* Σ *such that* Π *contains no doubly negated atoms 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 (Mellarkod, Gelfond, and Zhang 2008)) of* Π.

Weakly-Simple *AC* Programs

The correctness of the ACSOLVER algorithm was shown for simple *AC* programs. We start this section by reviewing simple programs. We then define a more general class of programs called weakly-simple. In the next section we state correctness results for ACSOLVER-like algorithms for such programs.

We say that an *AC* program Π is *safe* (Mellarkod, Gelfond, and Zhang 2008) if every variable occurring in a non defined rule in Π also occurs in a mixed atom of this rule. An *AC* program Π is *super safe* if Π is *safe* and

1. if a mixed atom $m(\vec{c}, X)$ occurs in Π then a mixed atom $m(\vec{c}, X')$ does not occur in Π (where X and X' are distinct variable names),
2. if a mixed atom $m(\vec{c}, X)$ occurs in Π then neither a mixed atom $m'(\vec{c}\ ', X)$ such that $\vec{c} \neq \vec{c}\ '$ nor a mixed atom $m'(\vec{c}, X)$ 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 by a simple syntactic transformation. Lierler and Zhang (2011, Section 7) provide 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.

For any atom $p(\vec{t})$, by $p(\vec{t})^0$ 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
- for each rule (1) in Π has an edge from a_0^0 to a_i^0 where $1 \leq i \leq l$.

We say that an *AC* program Π is *weakly-simple* if

- it is super safe,
- all regular atoms occurring in Π_D also occur in Π_R, 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. For example, program (2) is weakly-simple but not simple.

Review: Abstract Smodels

Most state-of-the-art answer set solvers are based on algorithms closely related to the DPLL procedure (Davis, Logemann, and Loveland 1962). Nieuwenhuis et al. described DPLL by means of a transition system that can be viewed as an abstract framework underlying DPLL computation (Nieuwenhuis, Oliveras, and Tinelli 2006). Lierler (2008) proposed a similar framework, SM_Π, for specifying an answer set solver SMODELS. 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, one of the main building blocks of ACSOLVER. The presentation follows (Lierler 2008).

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

$$\emptyset,\ \ a,\ \ \neg a,\ \ a^\Delta,\ \ a\ \neg a,\ \ \bot,\ \ a\bot,\ \ \neg a\bot,\ \ a^\Delta\bot,\ \ a\ \neg a\bot.$$

We say that a state is inconsistent if either $\bot$ or two complementary literals occur in it. For example, 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 $\{a, \neg b\}^- = \{b\}$.

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. We assume that the reader is familiar with the definition of *unfounded* for the class of regular programs (Lee 2005).

By $Bodies(\Pi, a)$ we denote the set of the bodies of all rules of a regular program Π with the head a. We recall that a set U of atoms occurring in a regular program Π is *unfounded* (Van Gelder, Ross, and Schlipf 1991; Lee 2005) on a consistent set M of literals with respect to Π if for every $a \in U$ and every $B \in Bodies(\Pi, a)$, $M \models \overline{B}$ (where B is identified with the conjunction of its elements), or $U \cap B^{pos} \neq \emptyset$.

Each regular 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

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 $\begin{cases} M \text{ is inconsistent and different from } \bot, \\ M \text{ contains no decision literals} \end{cases}$
Backtrack:
$P\ l^{\Delta}\ Q \Longrightarrow P\ \bar{l}$ if $\begin{cases} P\ l^{\Delta}\ Q \text{ is inconsistent, and} \\ Q \text{ contains no decision literals} \end{cases}$
Unfounded:
$M \Longrightarrow M\ \neg a$ if $a \in U$ for a set U unfounded on M wrt Π

and the transition rules *All Rules Cancelled* and *Backchain True* whose details we omit in this review. A node is *terminal* in a graph if no edge leaves this node.

The graph SM_Π can be used for deciding whether a regular program Π has an answer set by constructing a path from $\emptyset$ to a terminal node. Following proposition serves as a proof of correctness and termination for any procedure that is captured by the graph SM_Π.

Proposition 2 *For any regular 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.*

Abstract ACSOLVER

In order to present the transition system suitable for capturing ACSOLVER we introduce several concepts.

Query, Extensions, and Consequences: Given an *AC* program Π and a set $\mathbf{p}$ of predicate symbols, a set X of atoms is a $\mathbf{p}$-*input answer set* (or an input answer set w.r.t. $\mathbf{p}$) of Π 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 $\mathbf{p}$. [2] 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 a $\mathbf{p}$-input answer set of a program $a(1) \leftarrow b(1)$. On the other hand, it is not an input answer set for the same program with respect to a set $\{a, b\}$.

For a set S of literals, by S_R, S_D, and S_C we denote the set of regular, defined, and constraint literals occurring in S respectively. By $S_{R,D}$ and $S_{D,C}$ we denote the unions $S_R \cup S_D$ and $S_D \cup S_C$ respectively. By $At(\Pi)$ we denote the set of atoms occurring in a program Π.

For an AC program Π, a (complete) query Q is a (complete) consistent set of literals over $At(\Pi_D)_R \cup At(\Pi_R)_{D,C}$. For a query Q of Π, a complete query E is a *satisfying extension* of Q w.r.t. Π if $Q \subseteq E$ and there is a (sort respecting) substitution γ of variables in E by ground terms so that the result of this substitution, $E\gamma$, satisfies the conditions

1. if a constraint literal $l \in E\gamma$ then l is true under the intended interpretation of its symbols, and
2. there is an input answer set A of Π_D w.r.t. defined predicates of Π such that $E\gamma^+_{R,D} \subseteq A$ and $E\gamma^-_{R,D} \cap A = \emptyset$.

We say that literal l is a consequence of Π and Q if for every satisfying extension E of Q w.r.t. Π, $l \in E$. By $Cons(\Pi, Q)$, we denote the set of all consequences of Π and Q. If there are no satisfying extensions of Q w.r.t. Π we identify $Cons(\Pi, Q)$ with the singleton $\{\bot\}$.

Let Π be (2) and Q be $\{okTime(T),\ T \neq 1\}$. A set

$$\{on,\ okTime(T),\ T \neq 1\}$$

forms a satisfying extension of Q w.r.t. Π. Indeed, consider a substitution $\{T/106\}$. This is the only satisfying extension of Q w.r.t. Π. Consequently, it forms $Cons(\Pi, Q)$. On the other hand, there are no satisfying extensions for a query $\{\neg on,\ okTime(T)\}$ so that $\{\bot\}$ corresponds to $Cons(\Pi, Q)$.

The graph *AC*$_\Pi$: For each constraint and defined atom A of signature Σ, select a new symbol A^ξ, called the *name* of A. By Σ^ξ we denote the signature obtained from Σ by adding all names A^ξ as additional regular predicate symbols (so that A^ξ itself is a regular atom).

For an *AC* program Π, by Π^ξ we denote a set of rules consisting of (i) choice rules $\{a^\xi\}$ for each constraint and defined atom a occurring in Π_R, and (ii) Π_R whose mixed atoms are dropped, and constraint and defined atoms are replaced by their names. Note that Π^ξ is a regular program.

For instance, let Π be (2) then Π^ξ consists of the rules

$$\begin{array}{l} \{T \neq 1^\xi\} \qquad \{okTime(T)^\xi\} \\ \leftarrow occurs(a, 0),\ T \neq 1^\xi,\ not\ okTime(T)^\xi \\ occurs(a, 0) \leftarrow \\ \{on\} \end{array} \tag{5}$$

For a set M of atoms over Σ^ξ, by $M^{\xi-}$ we denote a set of atoms over Σ by replacing each name A^ξ occurring in M with a corresponding atom A. For instance, $\{T \neq 1^\xi,\ okTime(T)^\xi\}^{\xi-}$ is $\{T \neq 1,\ okTime(T)\}$.

Let Π be an *AC* logic program. The nodes of the graph *AC*$_\Pi$ are the states relative to the set of atoms occurring in Π^ξ.

For a state M of *AC*$_\Pi$, by $query(M)$ we denote the largest subset of $M^{\xi-}$ over $At(\Pi_D)_R \cup At(\Pi_R)_{D,C}$. Let Π be (2)

[2] Intuitively set $\mathbf{p}$ denotes a set of intensional predicates (Ferraris et al. 2009). The concept of $\mathbf{p}$-input answer sets is closely related to "$\mathbf{p}$-stable models" in (Ferraris, Lee, and Lifschitz 2011).

and M be a state $occurs(a,0)\ \neg on^{\Delta}\ okTime(T)^{\xi\,\Delta}$ then $query(M)$ is $\{\neg on,\ okTime(T)\}$.

The edges of the graph AC_Π are described by the transition rules of SM_{Π^ξ} and the additional transition rule

Query Propagate:
$M \implies M\, l^{\xi}$ if $l \in Cons(\Pi, query(M))$,

where we abuse notation and identify $\bot^{\xi}$ with $\bot$ itself.

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

Proposition 3 *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* $\bot$*,* $(M^{\xi-})^+_R$ *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 3 shows that algorithms that find a path in the graph AC_Π from $\emptyset$ to a terminal node can be regarded as *AC* solvers for weakly-simple programs.

Let Π be an *AC* program (2). 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:

$$\emptyset \overset{\textit{Unit Propagate}}{\Longrightarrow} occurs(a,0) \overset{\textit{Decide}}{\Longrightarrow} occurs(a,0)\ \neg on^{\Delta} \overset{\textit{Decide}}{\Longrightarrow}$$
$$occurs(a,0)\ \neg on^{\Delta}\ (okTime(T)^{\xi})^{\Delta} \overset{\textit{Query Propagate}}{\Longrightarrow}$$
$$occurs(a,0)\ \neg on^{\Delta}\ (okTime(T)^{\xi})^{\Delta}\ \bot \overset{\textit{Backtrack}}{\Longrightarrow}$$
$$occurs(a,0)\ \neg on^{\Delta}\ \neg okTime(T)^{\xi} \overset{\textit{Unit Propagate}}{\Longrightarrow}$$
$$occurs(a,0)\ \neg on^{\Delta}\ \neg okTime(T)^{\xi}\ \neg T \neq 1^{\xi}$$

Since the last state in the path is terminal, Proposition 3 asserts that $occurs(a,0)$ is a set of all regular atoms in some answer set of Π. Indeed, recall answer set (4).

The 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 the graph's 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 follows a transition due to a rule in a state if a rule with higher priority is applicable. A strategy may also include restrictions on rule's applications.

We use this approach to describe the ACSOLVER algorithm (Mellarkod, Gelfond, and Zhang 2008, 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 Propagate* >> *Decide*.

Note that ACSOLVER also only follows a transition due to the rule *Query Propagate* if there are no satisfying extensions of $query(M)$ w.r.t. Π_D, i.e., $Cons(\Pi, query(M)) = \{\bot\}$.

Mellarkod et al. (2008) demonstrated the correctness of the ACSOLVER algorithm for the class of safe canonical programs by analyzing the properties of its pseudocode. Proposition 3 provides an alternative proof of correctness for this algorithm for a more general class of weakly-simple programs that relies on the transition system AC_Π. Furthermore, Proposition 3 encapsulates the proof of correctness for a class of algorithms that can be described using AC_Π. For instance, it immediately follows that the ACSOLVER algorithm modified to follow an arbitrary transition due to the rule *Query Propagate* is still correct.

The CLINGCON Language

Consider a subset of the *AC* language, denoted 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. The language of the constraint answer set solver CLINGCON defined in (Gebser, Ostrowski, and Schaub 2009) 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. For a signature Σ, a *clingcon variable* is an expression of the form $p(\vec{r})$, where p is a mixed predicate and $\vec{r}$ is a sequence of regular constants. For any clingcon variable $p(\vec{r})$, by $p(\vec{r})^0$ we denote its predicate symbol p and by $p(\vec{r})^s$ we denote its sequence of regular constants $\vec{r}$.

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

$$v_1 \circ \cdots \circ v_k \circ c_1 \circ \cdots \circ c_m\ \odot\ v_{k+1} \circ \cdots \circ v_l \circ c_{m+1} \circ \cdots \circ c_n, \quad (6)$$

where v_i is a clingcon variable; c_i is a constraint constant; $\circ$ is a primitive constraint operation; and $\odot$ is a primitive constraint relation.

A *clingcon program* is a finite set of rules of the form (1) where (i) a_0 is $\bot$ or a regular atom, (ii) each a_i, $1 \leq i \leq m$ is a regular atom or clingcon atom, and (iii) each a_i, $m{+}1 \leq i \leq n$ is a regular atom.

Any clingcon program Π can be rewritten in AC^- using a function ν that maps the set of clingcon variables occurring in Π to the set of distinct variables over Σ. For a clingcon variable v, v^ν denotes a variable assigned to v by ν.

For each occurrence of clingcon atom (6) in some rule r of Π (i) add a set of mixed atoms $v_i^0(v_i^s, v_i^\nu)$ for $1 \leq i \leq l$ to the body of r, and (ii) replace (6) in r by a constraint atom

$$v_1^\nu \circ \cdots \circ v_k^\nu \circ c_1 \circ \cdots \circ c_m\ \odot\ v_{k+1}^\nu \circ \cdots \circ v_l^\nu \circ c_{m+1} \circ \cdots \circ c_n.$$

We denote resulting AC^- program by $ac(\Pi)$.

For instance, let clingcon program Π over Σ_1 consist of a single rule

$$\leftarrow\ occurs(a,0),\ at(0) \neq 1.$$

Given ν that maps $at(0)$ to T, $ac(\Pi)$ has the form

$$\leftarrow\ occurs(a,0),\ at(0,T),\ T \neq 1.$$

Proposition 4 *For a clingcon program* Π *over signature* Σ*, a set* X *is a constraint answer set of* Π *according to the definition in (Gebser, Ostrowski, and Schaub 2009) iff there is a functional set* M *of ground mixed atoms of* Σ *such that* $X \cup M$ *is an answer set of* $ac(\Pi)$.

Note that $ac(\Pi)$ is a weakly-simple program (in fact, it is a simple program). It follows that a class of algorithms captured by the graph AC_Π is applicable to clingcon programs after minor syntactic transformations. Nevertheless the graph AC_Π is not suitable for describing the CLINGCON system. In the next section we present another graph suitable for this purpose.

The Basic CLINGCON Algorithm

The CLINGCON system is based on tight coupling of the answer set solver CLASP and the constraint solver GECODE. Recall that CLASP starts its computation by building a propositional formula called completion (Clark 1978) of a given program so that its propagation relies not only on the program but also on the completion. Furthermore, it implements such backtracking search techniques as backjumping, learning, forgetting, and restarts. Lierler and Truszczynski (2011) introduced the transition system $\text{SML(ASP)}_{F,\Pi}$ and demonstrated how it captures the CLASP algorithm. It turns out that $\text{SML(ASP)}_{F,\Pi}$ augmented with the transition rule *Query Propagate* is appropriate for describing CLINGCON. The graph $\text{SML(ASP)}_{F,\Pi}$ extends a simpler graph $\text{SM(ASP)}_{F,\Pi}$. These extensions are essential for capturing such advanced features of CLASP and CLINGCON as conflict-driven backjumping and learning. To simplify the presentation we review the graph $\text{SM(ASP)}_{F,\Pi}$ and show that augmenting it with the rule *Query Propagate* captures *basic* CLINGCON algorithm implementing a simple backtrack strategy in place of conflict-driven backjumping and learning. This abstract view on CLINGCON allows us to compare it to ACSOLVER in formal terms.

Abstract basic CLINGCON: We write $Head(\Pi)$ for the set of nonempty heads of rules in a program Π. For a clause $C = \neg a_1 \vee \ldots \vee \neg a_l \vee a_{l+1} \vee \ldots \vee a_m$ we write C^r to denote the rule

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

For a set F of clauses, we define $F^r = \{C^r \mid C \in F\}$. For a set A of atoms, by $\Pi(A)$ we denote a program Π extended with the rules $\{a\}$ for each atom $a \in A$.

The transition graph $\text{SM(ASP)}_{F,\Pi}$ for a set F of clauses and a regular program Π is defined as follows. The set of nodes of $\text{SM(ASP)}_{F,\Pi}$ consists of the states relative to $At(F \cup \Pi)$. There are five transition rules that characterize the edges of $\text{SM(ASP)}_{F,\Pi}$. The transition rules *Unit Propagate*, *Decide*, *Fail*, *Backtrack* of the graph $\text{SM}_{F^r \cup \Pi}$, and the transition rule

Unfounded':

$$M \implies M\ \neg a \text{ if } \begin{cases} a \in U \text{ for a set } U \text{ unfounded on } M \\ \text{w.r.t. } \Pi(At(F \cup \Pi) \setminus Head(\Pi)) \end{cases}$$

Lierler and Truszczynski (2011) demonstrated how $\text{SM(ASP)}_{ED\text{-}Comp(\Pi),\Pi}$ models *basic* CLASP (without conflict driven backjumping and learning) where *ED-Comp*(Π) denotes clausified completion with the use of auxiliary atoms.

We now define the graph $\text{CON}_{F,\Pi}$ for *AC* programs that extends $\text{SM(ASP)}_{F,\Pi}$ in a similar way as AC_Π extends SM_Π.

For an *AC* logic program Π and a set F of clauses, the nodes of $\text{CON}_{F,\Pi}$ are the states relative to the set $At(F \cup \Pi^\xi)$. The edges of $\text{CON}_{F,\Pi}$ are described by the transition rules of $\text{SM(ASP)}_{F,\Pi^\xi}$ and the transition rule *Query Propagate* of AC_Π.

For an *AC* program Π, a set F of clauses is Π-*safe* if

1. $F \models \neg a$, for every $a \in At(\Pi^\xi) \setminus Head(\Pi^\xi)$, and
2. for every answer set X of Π^ξ there is a model M of F such that $X = M^+ \cap Head(\Pi^\xi)$.

In fact, a set F of clauses is Π-safe if it is Π^ξ-safe according to the "safeness" definition given in (Lierler and Truszczynski 2011).

Proposition 5 *For any weakly-simple* AC *program* Π *and a* Π*-safe set* F *of clauses,*

(a) graph $\text{CON}_{F,\Pi}$ *is finite and acyclic,*

(b) for any terminal state M *of* $\text{CON}_{F,\Pi}$ *other than* $\bot$, $(M^{\xi-})^+_R \cap At(\Pi)$ *is a set of all regular atoms in some answer set of* Π*,*

(c) state $\bot$ *is reachable from* $\emptyset$ *in* $\text{CON}_{F,\Pi}$ *if and only if* Π *has no answer sets.*

The algorithm behind basic CLINGCON is modeled by means of the graph $\text{CON}_{ED\text{-}Comp(\Pi^\xi),\Pi}$ with the following priorities

Backtrack, *Fail* $>>$ *Unit Propagate* $>>$ *Unfounded* $>>$ *Query Propagate* $>>$ *Decide*.

Proposition 3 demonstrates that the CLINGCON algorithm is applicable to a broader class of weakly-simple *AC* programs.

Following concept helps us to formulate the relation between AC_Π and $\text{CON}_{F,\Pi}$ precisely. An edge $M \implies M'$ in the graph AC_Π ($\text{CON}_{F,\Pi}$) is *singular* if:

- the only transition rule justifying this edge is *Unfounded*, and
- some edge $M \implies M''$ can be justified by a transition rule other than *Unfounded* or *Decide*.

It is easy to see that due to priorities of ACSOLVER and CLINGCON, singular edges are inessential. We define AC^-_Π ($\text{CON}^-_{F,\Pi}$) as the graph obtained by removing all singular edges from AC_Π ($\text{CON}_{F,\Pi}$).

Proposition 6 *Let* $Comp(\Pi^\xi)$ *be completion clausified in a straightforward way by applying distributivity. For every* AC *program* Π*, the graphs* AC^-_Π *and* $\text{CON}^-_{Comp(\Pi^\xi),\Pi}$ *are equal.*

It follows that the graph $\text{CON}^-_{Comp(\Pi^\xi),\Pi}$ also provides an abstract model of ACSOLVER. Hence the difference between ACSOLVER and basic CLINGCON algorithms can be stated in terms of difference in Π-safe formulas $Comp(\Pi^\xi)$ and *ED-Comp*(Π^ξ) that they are applied to.

Conclusions

In this paper, we designed transition systems AC_Π and $\text{CON}_{F,\Pi}$ for describing algorithms for computing (subsets of) answer sets of *AC* programs. We used these graphs to specify the ACSOLVER and the basic CLINGCON algorithms. We demonstrated a formal relation between the *AC* and

CLINGCON languages and the algorithms behind ACSOLVER and 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 graphs AC_Π and $\text{CON}_{F,\Pi}$ offer a convenient tool to describe, compare, analyze, and prove correctness for a class of algorithms. In fact we formally show the relation between the subgraphs of AC_Π and $\text{CON}_{F,\Pi}$. Furthermore, the transition systems for ACSOLVER and CLINGCON result in new algorithms for solving a larger class of *AC* programs – weakly-simple programs introduced in this paper. Neither the ACSOLVER nor CLINGCON procedures, 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 Yuanlin Zhang, 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.

References

Baselice, S.; Bonatti, P. A.; and Gelfond, M. 2005. Towards an integration of answer set and constraint solving. In Gabbrielli, M., and Gupta, G., eds., *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, 52–66. Springer.

Clark, K. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logic and Data Bases*. New York: Plenum Press. 293–322.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem proving. *Communications of the ACM* 5(7):394–397.

Ferraris, P., and Lifschitz, V. 2005. Weight constraints as nested expressions. *Theory and Practice of Logic Programming* 5:45–74.

Ferraris, P.; Lee, J.; Lifschitz, V.; and Palla, R. 2009. Symmetric splitting in the general theory of stable models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 797–803.

Ferraris, P.; Lee, J.; and Lifschitz, V. 2011. Stable models and circumscription. *Artificial Intelligence* 175:236–263.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. Conflict-driven answer set solving. In *Proceedings of 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, 386–392. MIT Press.

Gebser, M.; Ostrowski, M.; and Schaub, T. 2009. Constraint answer set solving. In *Proceedings of 25th International Conference on Logic Programming (ICLP)*, 235–249. Springer.

Kakas, A.; Kowalski, R.; and Toni, F. 1992. Abductive logic programming. *Journal of Logic and Computation* 2(6):719–770.

Lee, J. 2005. A model-theoretic counterpart of loop formulas. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 503–508. Professional Book Center.

Lierler, Y., and Truszczynski, M. 2011. Transition systems for model generators — a unifying approach. *Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP'11) Special Issue* 11, issue 4-5.

Lierler, Y., and Zhang, Y. 2011. A transition system for AC language algorithms. In *Proceedings of ICLP Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP)*.

Lierler, Y. 2008. Abstract answer set solvers. In *Proceedings of International Conference on Logic Programming (ICLP)*, 377–391. Springer.

Lifschitz, V.; Tang, L. R.; and Turner, H. 1999. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence* 25:369–389.

Mellarkod, V. S.; Gelfond, M.; and Zhang, Y. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*.

Niemelä, I., and Simons, P. 2000. Extending the Smodels system with cardinality and weight constraints. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer. 491–521.

Nieuwenhuis, R.; Oliveras, A.; and Tinelli, C. 2006. 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.

Van Gelder, A.; Ross, K.; and Schlipf, J. 1991. The well-founded semantics for general logic programs. *Journal of ACM* 38(3):620–650.