2-2014

Relating Constraint Answer Set Programming Languages and Algorithms

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

Relating Constraint Answer Set Programming Languages and Algorithms

Yuliya Lierler

Department of Computer Science
The University of Nebraska at Omaha
6001 Dodge Street
Omaha, NE 68182

Abstract

Recently a logic programming language AC was proposed by Mellarkod et al. (2008) to integrate answer set programming and constraint logic programming. Soon after that, a CLINGCON language integrating answer set programming and finite domain constraints, as well as an EZCSP language integrating answer set programming and constraint logic programming were introduced. The development of these languages and systems constitutes the appearance of a new AI subarea called constraint answer set programming. All these languages have something in common. In particular, they aim at developing new efficient inference algorithms that combine traditional answer set programming procedures and other methods in constraint programming. Yet, the exact relation between the constraint answer set programming languages and the underlying systems is not well understood. In this paper we address this issue by formally stating the precise relation between several constraint answer set programming languages – AC, CLINGCON, EZCSP – as well as the underlying systems.

Keywords: (Constraint) answer set programming, constraint satisfaction processing, satisfiability modulo theories

1. Introduction

Constraint answer set programming (CASP) is a novel, promising direction of research whose roots can be traced back to propositional satisfiability (SAT). SAT solvers are efficient tools for solving Boolean constraint satisfaction problems that arise in different areas of computer science, including software and hardware verification. Answer set programming (ASP) extends computational methods of SAT using ideas from knowledge representation, logic programming, and nonmonotonic reasoning. As a declarative programming paradigm, it provides a rich, and yet simple modeling language that, among other features, incorporates recursive definitions. Satisfiability modulo theories (SMT) extends computational methods of SAT by integrating non-Boolean symbols defined via a background theory in other formalisms, such as first order theory or a constraint processing language. The key ideas behind such integration are that (a) some constraints are more naturally expressed by non-Boolean constructs and (b) computational methods developed in other areas of automated reasoning than SAT may complement its technology in an effective manner processing these constraints.

Constraint answer set programming draws on both of these extensions of SAT technology: it integrates answer set programming with constraint processing. This new area has already demonstrated promising results, including the development of the CASP solvers ACSOLVER [1] (Texas Tech University), CLINGCON[1] [2, 3] (Potsdam University, Germany), EZCSP[2] [4] (KODAK), IDP[3] [5] (KU Leuven). These systems provide new horizons to knowledge representation as a field by broadening the applicability of its computational tools. CASP not only provides new modeling features for answer set programming but also improves grounding and solving performance by delegating processing of constraints over large and possibly infinite domains to specialized systems. The origins of this work go back to [6, 7].

Drescher and Walsh [8, 9] (INCA, NICTA, Australia), Liu et al. [10] (MINGO, Aalto University, Finland) took an alternative approach to tackling CASP languages — a translational approach. In the former case, the CASP programs are translated into ASP programs (Drescher and Walsh proposed a number of translations). In the latter, the program is translated into integer linear programming formalism. The empirical results demonstrate that this is also a viable approach towards tackling CASP programs.

The general interest towards CASP paradigms illustrates the importance of developing synergistic approaches in the automated reasoning community. To do so effectively one requires a clear understanding of the important fea-

[1]`http://www.cs.uni-potsdam.de/clingcon/`

[2]`http://marcy.cjb.net/ezcsp/index.html`

[3]`http://dtai.cs.kuleuven.be/krr/software/idp`

tures of the CASP-like languages and underlying systems. Current CASP languages are based on the same principal ideas yet relating them is not a straightforward task. One difficulty lies in the fact that these languages are introduced together with a specific system architecture in mind that rely on various answer set programming, constraint satisfaction processing, constraint logic programming, and integer linear programming technologies. The syntactic differences stand in the way of clear understanding of the key features of the languages. For example, the only CASP language that was compared to its earlier sibling was the language EZCSP. Balduccini [4] formally stated that the EZCSP language is a special case of AC. Relating CASP systems formally is an even more complex task. The variations in underlying technologies complicate clear articulation of their similarities and differences. For instance, the main building blocks of the CASP solver ACSOLVER [1] are the ASP system SMODELS [11] and SICStus Prolog[4]. The technology behind CLINGCON [2, 3] is developed from the ASP solver CLASP [12] and the constraint solver GECODE [13]. In addition, the CASP solvers adopt different communication schemes among their heterogeneous solving components. For instance, the system EZCSP relies on *blackbox* integration of ASP and CSP tools in order to process the EZCSP language [4]. Systems ACSOLVER and CLINGCON promote tighter integration of multiple automated reasoning methods.

The broad attention to CASP suggests a need for a principled and general study of methods to develop unifying terminology and formalisms suitable to capture variants of the languages and solvers. This work can be seen as a step in this direction. First, it presents a formal account that illustrates a precise relationship between the languages of ACSOLVER, CLINGCON, and EZCSP. Second, it formally relates the systems that take a hybrid approach to solving in CASP. In particular, it accounts for systems ACSOLVER, CLINGCON, and EZCSP.

Usually backtrack search procedures (Davis-Putnam-Logemann-Loveland (DPLL)-like procedures [14]), the backbone of CASP computational methods are described in terms of pseudocode. In [15], the authors proposed an alternative approach to describing DPLL-like algorithms. They introduced an abstract graph-based framework that captures what the states of computation are and what transitions between states are allowed. This approach

[4]`http://www.sics.se/isl/sicstuswww/site/index.html`

allows us to model a DPLL-like algorithm by a mathematically simple and elegant object, a graph, rather than a collection of pseudocode statements. We develop a similar abstract framework for performing precise formal analysis on relating the constraint answer set solvers ACSOLVER, CLINGCON, and EZCSP. Furthermore, this framework allows an alternative proof of correctness of these systems. 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.

More on Related Work: Another direction of work related to the developments in CASP is research on HEX-programs [16]. These programs integrate logic programs under answer set semantics with external computation sources via *external atoms*. They were motivated by the need to interface ASP with external computation sources, for example, to allow the synergy of ASP and description logic computations within the context of the semantic web. CASP has a lot in common with HEX-programs. System DLVHEX[5] [17] computes models of such programs. It allows defining plug-ins for inference on external atoms and as such can be used as a general framework for developing CASP solvers (but it does not provide any specific computational mechanism by default).

Heterogeneous nonmonotonic multi-context systems [18] is another formalism related both to CASP and HEX-programs. CASP and HEX-programs can be seen as one of the possible incarnations of a special case of multi-context systems. Multi-context systems provide a more general formalism where "contexts" written in different logics relate with each other via bridge rules. Intuitively, CASP provides two contexts: one in the language of answer set programming and another one in the language of constraint programming. Yet, the bridge rules are of extremely simplistic nature in CASP, in particular, they relate atoms in a logic program to constraints of constraint processing.

Paper Structure: We start by reviewing AC programs introduced by Mellarkod et al. (2008) and the notion of an answer set for such programs. In the subsequent section we introduce the CLINGCON language and formally state its relation to the AC language. We then define a new class of weakly-simple programs and demonstrate that the ACSOLVER algorithm is applicable

[5]`http://www.kr.tuwien.ac.at/research/systems/dlvhex/`

also to such programs. We review a transition system introduced by Lierler (2008, 2011) 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. We define a graph suitable for modeling the system CLINGCON and state a formal result on the relation between the ACSOLVER and CLINGCON algorithms. At last we illustrate how the same graph may model the EZCSP system. The final section presents the proofs of the formal results stated in the paper.

A report on some of the results of this paper has been presented at [21] and [22]. This work extends earlier efforts by introducing a transition system that captures advanced CASP solvers CLINGCON and EZCSP featuring learning and backjumping. This paper also provides a complete account of proofs for the formal results.

2. Review: AC 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.* Each variable takes on values of a unique constraint sort. For example, let signature Σ_1 contain three regular sorts $step = \{0..1\}$, $action = \{a\}$, $fluent = \{f\}$; and two constraint sorts $time = \{0..200\}$, $computer = \{1..2\}$; variable T, T' of constraint sort $time$; and predicates

$$at(step, time) \;\; occurs(action, step) \;\; next(step, step) \;\; holds(fluent, step)$$
$$on \;\; okTime(time) \;\; okComp(computer, time).$$

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 literal is either an atom a or its negation $\neg a$. A constraint sort is often a large numerical set with primitive constraint relations (examples include arithmetic constraint relations like $\leq$).

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 constants 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 constants 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 [23].

For example, for signature Σ_1, we define $at(step, time)$ to be a mixed predicate; $occurs(action, step)$, on, $next(step, step)$, $holds(fluent, step)$ to be regular predicates; $okTime(time)$ and $okComp(computer, time)$ to be defined predicates.

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. For signature Σ_1, atoms $at(0, T)$ and $occurs(a, 1)$ are sample mixed and regular atoms respectively.

An *nested program* is a finite set of rules of the form

$$\begin{array}{r} a_0 \leftarrow a_1, \dots, a_l, not\ a_{l+1}, \dots, not\ a_m, \\ not\ not\ a_{m+1}, \dots, not\ not\ a_n, \end{array} \tag{1}$$

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 or symbols $\top$, $\bot$. If $a_0 = \bot$, we often omit $\bot$ from the notation. This is a special case of programs with nested expressions [24]. 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., $B^{pos} = \{a_1, \dots, a_l\}$; B^{neg} for the elements occurring under single negation as failure, i.e., $B^{neg} = \{a_{l+1}, \dots, a_m\}$; and B^{neg2} for the elements occurring under double negation as failure, i.e., $B^{neg2} = \{a_{m+1}, \dots, a_n\}$. We frequently identify the body of (1) with the conjunction of its elements (in which *not* is replaced with the classical negation connective $\neg$):

$$a_1 \wedge \dots \wedge a_l \wedge \neg a_{l+1} \wedge \dots \wedge \neg a_m \wedge \neg\neg a_{m+1} \wedge \dots \wedge \neg\neg a_n. \tag{2}$$

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

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

(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 the rules in Π.

We restate the definition of an answer set due to Lifschitz et al. (1999) for nested programs in a form convenient for our purposes. The *reduct* Π^X of a nested program Π with respect to set X of atoms is obtained from Π by deleting each rule (1) such that X does not satisfy its body (recall that we identify its body with (2)), and replacing each remaining rule (1) by $a_0 \leftarrow B^{pos}$ where B stands for the body of (1). A set X of atoms is an *answer set* of a nested program Π if it is minimal among sets of atoms satisfying $(\Pi^X)^{cl}$.

According to [25], a *choice rule* construct [11]

$$\{a\}$$

of the LPARSE[6] language can be seen as an abbreviation for a rule

$$a \leftarrow \ not\ not\ a.$$

We adopt this abbreviation in the rest of the paper. For a program that consists of this rule both $\emptyset$ and $\{a\}$ form its answer sets.

The rules and programs are called *regular* if the bodies of the rules do not contain symbols $\top$ or $\bot$.

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

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

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. This assumption does not impact applicability of the language but is made for the ease of the presentation.

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

For instance, a sample AC program over signature Σ_1 follows

$$\begin{array}{l}
okComp(1,T) \leftarrow T \leq 5, on. \\
okComp(2,106) \leftarrow on. \\
okTime(T) \leftarrow T \leq 10, okComp(1,T). \\
okTime(T) \leftarrow T \geq 100, okComp(2,T). \\
\leftarrow occurs(a,0), at(0,T),\ T \neq 1,\ not\ okTime(T). \\
\leftarrow occurs(a,0), at(0,T),\ T \geq 110. \\
\leftarrow occurs(a,1), at(1,T'),\ T' \geq 110. \\
holds(f,1) \leftarrow occurs(a,0), next(1,0). \\
next(1,0). \\
occurs(a,0). \\
\{on\}.
\end{array} \tag{4}$$

The implementation of such language requires the declaration of the signature Σ_1 itself. In syntax proposed by Mellarkod et al. (2008) we encode the declaration of Σ_1 as follows:

$$\begin{array}{l}
time(0..200). \\
computer(1..2). \\
step(0..1). \\
action(a). \\
fluent(f). \\
\#csort(time). \\
\#csort(computer). \\
\#mixed\ at(step, time). \\
\#regular\ occurs(action, step). \\
\#regular\ on. \\
\#defined\ okTime(time). \\
\#defined\ okComp(computer, time).
\end{array}$$

This sample program is inspired by Example 1 in [1] that encodes a small planning domain. It is well known that answer set programming provides a convenient language for encoding planning problems. Yet if in a problem actions have to be mapped to real time that is represented by a large integer domain, grounding becomes a bottleneck for answer set programming. Mellarkod et al. illustrated how AC language allows us to overcome this limitation in Example 1.

Mellarkod et al. [1] considered programs of different syntax than discussed here. For instance, in [1] 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.

2.1. Semantics of the AC *Language*

We define the semantics of AC programs by transforming a program into a nested program using grounding. For an AC program Π over signature Σ, by the set $ground(\Pi)$ we denote the set of all ground instances of the rules in Π. The set $ground^*(\Pi)$ is obtained from $ground(\Pi)$ by replacing a constraint atom a by $\top$ or $\bot$ if a is *true* or *false* respectively. A ground constraint atom evaluates to *true* or *false* under a standard interpretations of its symbols. For example, a constraint atom $1 = 1$ evaluates to *true*, whereas a constraint atom $1 \neq 1$ evaluates to *false*. It is easy to see that $ground^*(\Pi)$ is a nested program.

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

$$\begin{array}{l} okTime(100) \leftarrow 100 > 100,\ okComp(2,100). \\ okTime(101) \leftarrow 101 > 100,\ okComp(2,101). \\ \leftarrow occurs(a,0), at(0,101),\ 101 \neq 1,\ not\ okTime(101). \end{array}$$

then $ground^*(\Pi)$ is

$$\begin{array}{l} okTime(101) \leftarrow \bot,\ \ okComp(2,100). \\ okTime(101) \leftarrow \top,\ okComp(2,101). \\ \leftarrow occurs(a,0), at(0,101), \top,\ not\ okTime(101). \end{array}$$

If we define the semantics of an AC program as the semantics of a corresponding nested program $ground^*(\Pi)$ then mixed atoms will never be part of answer sets (indeed, mixed atoms never occur in the heads of the rules). This is different from an intended meaning of these atoms that suppose to "connect" the values of regular constants and their constraint counterpart. We now introduce notion of a functional set composed of mixed atoms that is crucial in defining answer sets of AC programs. 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 (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 $\vec{r}$ 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 (4) sets $\{at(0,1),\ at(1,1)\}$ and $\{at(0,2),\ at(1,1)\}$ are functional, whereas $\{at(0,1)\}$ and $\{at(0,1),\ at(0,2)\}$ are not functional sets.

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

$$\{at(0,1),\ at(1,1),\ occurs(a,0),\ next(1,0),\ holds(f,1)\} \tag{5}$$

and

$$\begin{array}{l}\{on,\ \ at(0,0),\ \ occurs(a,0),\ at(1,1),\ next(1,0),\ holds(f,1)\\ \ \ okComp(1,0),\dots,okComp(1,5),\ okComp(2,106),\\ \ \ okTime(0),\dots,okTime(5),\ okTime(106)\}\end{array}$$

are among answer sets of (4).

The definition of an answer set for AC programs presented here is different from the original definition in [1] (even when we restrict our attention to programs without doubly negated atoms), 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 [1]) of* Π*.*

3. The clingcon Language

Consider a subset of the AC language, denoted AC^-, so that any AC program without defined atoms is an AC^- program. The language of the constraint answer set solver clingcon defined in [2, 3][7] can be seen as a syntactic variant of the AC^- language.

[7]The system clingcon accepts programs of more general syntax than discussed in [2] (for instance, aggregates such as *#count* are allowed by clingcon).

We now review clingcon programs and show how they map into $\mathcal{AC}^{-}$ 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

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

where v_i is a clingcon variable; c_i is a constraint constant; $\circ$ are primitive constraint operations (note that $\circ$ denotes an occurrence of an operation so that a different operation can be used at each occurrence); 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, $1 \leq i \leq n$, is a regular or clingcon atom. The system CLINGCON accepts rules where a_0 is a clingcon atom but it should be seen as an abbreviation for the same rule with $\bot$ as the head and *not* a_0 occurring in the body.

Any clingcon program Π can be rewritten in $\mathcal{AC}^{-}$ 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

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

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

For instance, let clingcon program Π over Σ_1 without defined predicates consist of a single rule

$$\leftarrow \ occurs(a, 0),\ at(0) \geq 110. \tag{7}$$

Given ν that maps $at(0)$ to T', $ac(\Pi)$ has the form

$$\leftarrow \ occurs(a, 0),\ at(0, T'),\ T' \geq 110. \tag{8}$$

The following proposition makes the relation between a clingcon program and its $\mathcal{AC}^{-}$ counterpart precise.

Proposition 2. *For a clingcon program* Π *over signature* Σ*, a set* X *is a constraint answer set of* Π *according to the definition in [2, 3] iff there is a functional set* M *of ground mixed atoms of* Σ *such that* $X \cup M$ *is an answer set of* $ac(\Pi)$*.*

We demonstrated how any clingcon program can be seen as a program in the language of AC^{-}. There is an important class of AC programs called *safe*. In fact, Mellarkod et al. [1] only considered such programs in devising the algorithm for processing the AC programs. We will now illustrate that given any safe AC^{-} program, we can syntactically transform this program into a clingcon one. Thus the languages of AC^{-} and CLINGCON (for which the solving procedures have been proposed) are truly only syntactic variants of each other. We now define a two step transformation that first transforms a safe AC^{-} program into what we call a super-safe program and then into a corresponding clingcon program.

Rule (1) is called a *defined* rule if a_0 is a defined atom. We say that an AC program Π is *safe* [1] if every variable occurring in a non defined rule in Π also occurs in a mixed atom of this rule. In other words, any constraint variable occurring in a non defined rule is mapped to some sequence of regular constants $\vec{r}$ specified by a mixed predicate. Consider, a safe program (8). A constraint variable T' of sort *time* maps to a regular constant 0 of sort *step* specified by mixed predicate *at*.

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.

For example, program (4) is super-safe. On the other hand, if we replace the sixth rule of (4) by semantically the same rule (8) the program is not super-safe (but safe) as both conditions 1 and 2 are violated. Intuitively, super-safeness ensures that a constraint variable maps uniquely to some sequence of (regular) constants $\vec{r}$ specified by a specific mixed predicate m and the other way around. For instance, in program (4) constraint variable T of sort *time* corresponds to regular constant 0 of sort *step* specified by mixed predicate *at*, whereas constraint variable T' of sort *time* corresponds to constant 1 of sort *step* specified by *at*.

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

Proposition 3. *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*.*

Section 10 presents such a transformation.

For any super-safe AC^{-} program Π, by $con(\Pi)$ we denote a clingcon program constructed as follows: (i) all mixed atoms in Π are dropped, (ii) every constraint variable X is replaced by an expression $m(\vec{c})$ where $m(\vec{c}, X)$ is a mixed atom in Π in which X occurs (given the conditions of super-safeness we are guaranteed that there is a unique mixed atom of the form $m(\vec{c}, X)$ for each constraint variable X in Π).

For instance, let AC^{-} program Π over Σ_1 (without defined predicates) consist of a single rule (8). This program is clearly super-safe. The corresponding clingcon program $con(\Pi)$ is (7).

The following proposition makes the relation between a super-safe AC^{-} program and its clingcon counterpart precise.

Proposition 4. *For a super-safe program* Π *over signature* Σ*, there is a functional set* M *of ground mixed atoms of* Σ *such that* $X \cup M$ *is an answer set of* Π *iff a set* X *is a constraint answer set of* $con(\Pi)$ *according to the definition in [2, 3].*

In other words, Proposition 4 suggests that the languages AC^{-} and clingcon are syntactic variants of each other.

4. Weakly-Simple *AC* Programs

To the best of our knowledge system ACSOLVER was the first CASP solver implemented. The correctness of the ACSOLVER algorithm was shown for *simple AC* programs.[8] We start this section by reviewing simple programs. We then define a more general class of programs called weakly-simple. In Section 6 we present a generalization of the ACSOLVER algorithm and state its correctness for such programs.

[8]The ACSOLVER algorithm was proved to be correct for a class of "safe canonical" programs – a special case of simple programs. Any simple program may be converted to a canonical program by means of syntactic transformations discussed in [1].

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 program (4), the rules

$$\begin{array}{l} okComp(1,T) \leftarrow T \leq 5, on \\ okComp(2,106) \leftarrow on \\ okTime(T) \leftarrow T \leq 10, okComp(1,T) \\ okTime(T) \leftarrow T \geq 100, okComp(2,T) \end{array}$$

form its defined part whereas the other rules form Π_R.

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. In weakly-simple programs that we define here we first lift the restriction that a defined part of a program has a unique answer set. Second, weakly-simple programs allow regular atoms in defined rules under some syntactic conditions that we define by means of a predicate dependency graph. For any AC program Π, the *predicate dependency graph*[9] of Π is a directed graph that

- 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 $1 \leq i \leq l$.

Definition 2. *We say that an* AC *program* Π *is weakly-simple if*

- *it is super-safe,*
- *each strongly connected component of the predicate dependency graph of* Π *is a subset of either regular predicates of* Π *or defined predicates.*

It is easy to see that any simple program is also a weakly-simple program but not the other way around. For example, program (4) is weakly-simple but not simple since its defined part contains a regular atom *on*.

5. Abstract SMODELS

Most state-of-the-art answer set solvers are based on algorithms closely related to the DPLL procedure [14]. Nieuwenhuis et al. described DPLL by

[9]A similar definition of predicate dependency graph was given in [26] for programs of more general syntax.

means of a transition system that can be viewed as an abstract framework underlying DPLL computation [15]. Lierler (2008, 2011) proposed a similar framework, SM_Π, for specifying an answer set solver SMODELS following the lines of its pseudo-code description [27]. Our goal is to design such a 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 [19].

For a set σ of atoms, a *record* relative to σ is a list 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, e.g., a and $\neg a$. For example, states $a\;\neg a$ and $a\bot$ are inconsistent. Frequently, we identify a state M with a set of literals occurring in it possibly with the symbol $\bot$, ignoring both the annotations and the order between its elements. In some cases we identify a set of literals with a conjunction of its members, thus we can write $M \models \phi$ where M is a state and $\models$ is understood as a satisfiability relation. If neither a literal l nor its complement $\bar{l}$ occur in M, then l is *unassigned* by M. For a set M of literals, by M^+ and M^- we denote the set of atoms stemming from 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.

By $\mathit{Bodies}(\Pi, a)$ 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* [28, 29] on a consistent set M of literals with respect to Π if for every $a \in U$ and every $B \in \mathit{Bodies}(\Pi, a)$, $M \models \overline{B}$ (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 presented in Figure 1. A node is *terminal* in a graph if no edge leaves this node.

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:
$$M \implies \bot \text{ if } \begin{cases} M \text{ is inconsistent and different from } \bot, \\ M \text{ contains no decision literals} \end{cases}$$
Backtrack:
$$P\ l^{\Delta}\ Q \implies P\ \bar{l} \text{ if } \begin{cases} P\ l^{\Delta}\ Q \text{ is inconsistent, and} \\ Q \text{ contains no decision literals} \end{cases}$$
All Rules Cancelled:
$M \implies M\ \neg a$ if $\overline{B} \cap M \neq \emptyset$ for all $B \in \mathit{Bodies}(\Pi, a)$
Backchain True:
$$M \implies M\ l \text{ if } \begin{cases} a \leftarrow B \in \Pi,\ a \in M,\ l \in B, \\ \overline{B'} \cap M \neq \emptyset \text{ for all } B' \in \mathit{Bodies}(\Pi, a) \setminus B \end{cases}$$
Unfounded:
$M \implies M\ \neg a$ if $a \in U$ for a set U unfounded on M w.r.t. Π

Figure 1: The transition rules of the graph $\textsc{sm}_\Pi$.

The graph $\textsc{sm}_\Pi$ 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 $\textsc{sm}_\Pi$.

Proposition 5. *For any regular*[10] *program* Π*,*

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

6. Abstract ACSOLVER

6.1. Query, Extensions, and Consequences

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

Given an AC program Π and a set $\mathbf{p}$ of predicate symbols, a set X of atoms is a ***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 $\mathbf{p}$. 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 $\{a, b\}$ of predicate symbols. Intuitively set $\mathbf{p}$ denotes a set of intentional predicates [30]: The concept of $\mathbf{p}$-input answer sets is closely related to "$\mathbf{p}$-stable models" in [30].

Proposition 6. *For a nested program* Π*, a complete set* X *of literals, and a set* $\boldsymbol{p}$ *of predicate symbols such that predicate symbols occurring in the heads of* Π *form a subset of* $\boldsymbol{p}$*,* X^+ *is a* $\boldsymbol{p}$*-input answer set of* Π *iff* X *is a model of* $SM_{\boldsymbol{p}}[\Pi]$ *(i.e.,* $\boldsymbol{p}$*-stable model of* Π*).*

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

[10] In [19], only programs without doubly negated atoms were considered. Extension of the results to regular programs is straightforward.

the set of atoms occurring in a program Π. Recall that a *substitution* Θ is a finite set of the form

$$\{v_1/t_1, \ldots, v_n/t_n\}$$

where $v_1, \ldots, v_n$ are distinct variables and each t_i is a term other than v_i. Given a substitution Θ and a set X of literals, we write $X\Theta$ for the result of a substitution.

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 (4) 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 substitutions $\{T/106\}$. 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 $Cons(\Pi, Q)$.

6.2. The graph AC_Π

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.

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

Figure 2: The transition rule *Query Propagate*.

For instance, let Π be (4) then Π^{ξ} consists of the rules

$$\begin{array}{ll}
\{T \neq 1^{\xi}\}. \quad \{T \geq 110^{\xi}\}. \quad \{T' \geq 110\}. \quad \{okTime(T)^{\xi}\}. & \\
\leftarrow occurs(a,0),\ T \neq 1^{\xi},\ not\ okTime(T)^{\xi} & \\
\leftarrow occurs(a,0),\ T \geq 110^{\xi}. & \\
\leftarrow occurs(a,1),\ T' \geq 110^{\xi}. & \\
occurs(a,0). & (9) \\
holds(f,1) \leftarrow occurs(a,0), next(1,0). & \\
next(1,0). & \\
occurs(a,0). & \\
\{on\}. &
\end{array}$$

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, $\{okTime(T)^{\xi},\ T \neq 1^{\xi}\}^{\xi-}$ is $\{okTime(T),\ T \neq 1\}$.

Let Π be an AC program. The nodes of the graph AC_{Π} are the states relative to the set $At(\Pi^{\xi}) \cup At(\Pi_D)_R$ of atoms.

For a state M of AC_{Π}, by $query(M)$ we denote the largest subset of $M^{\xi-}$ over $At(\Pi_D)_R \cup At(\Pi_R)_{D,C}$. For example, for program (4) and the state M

$$occurs(a,0)\ \ \neg on^{\Delta}\ \ okTime(T)^{\xi^{\Delta}},$$

$query(M)$ is $\{\neg on,\ okTime(T)\}$.

The edges of the graph AC_{Π} are described by the transition rules of $\text{SM}_{\Pi^{\xi}}$ and the additional transition rule *Query Propagate* presented in Figure 2 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 7. *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 7 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 (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:

$$\begin{array}{l}
\emptyset \overset{\mathit{Unit\ Propagate}}{\Longrightarrow} \mathit{occurs}(a,0) \overset{\mathit{Decide}}{\Longrightarrow} \mathit{occurs}(a,0)\ \neg \mathit{on}^{\Delta} \overset{\mathit{Decide}}{\Longrightarrow} \\
\mathit{occurs}(a,0)\ \neg \mathit{on}^{\Delta}\ \mathit{okTime}(T)^{\xi\Delta} \overset{\mathit{Query\ Propagate}}{\Longrightarrow} \\
\mathit{occurs}(a,0)\ \neg \mathit{on}^{\Delta}\ \mathit{okTime}(T)^{\xi\Delta}\ \bot \overset{\mathit{Backtrack}}{\Longrightarrow} \\
\mathit{occurs}(a,0)\ \neg \mathit{on}^{\Delta}\ \neg \mathit{okTime}(T)^{\xi} \overset{\mathit{Unit\ Propagate}}{\Longrightarrow} \\
\mathit{occurs}(a,0)\ \neg \mathit{on}^{\Delta}\ \neg \mathit{okTime}(T)^{\xi}\ \neg T \neq 1^{\xi} \overset{\mathit{Unit\ Propagate}}{\Longrightarrow} \\
\mathit{occurs}(a,0)\ \neg \mathit{on}^{\Delta}\ \neg \mathit{okTime}(T)^{\xi}\ \neg T \neq 1^{\xi}\ \mathit{next}(1,0) \overset{\mathit{Unit\ Propagate}}{\Longrightarrow} \\
\mathit{occurs}(a,0)\ \neg \mathit{on}^{\Delta}\ \neg \mathit{okTime}(T)^{\xi}\ \neg T \neq 1^{\xi}\ \mathit{next}(1,0)\ \mathit{holds}(f,1)
\end{array} \tag{10}$$

Since the last state in the path is terminal, Proposition 7 asserts that

$$\{\mathit{occurs}(a,0)\ \mathit{next}(1,0)\ \mathit{holds}(f,1)\}$$

forms the set of all regular atoms in some answer set of Π. Indeed, recall answer set (5).

6.3. 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 [1, 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*$_\perp$ >> *Decide*,

where by *Query Propagate*$_\perp$ we denote a transition due to the rule *Query Propagate* if there are no satisfying extensions of $query(M)$ w.r.t. Π_D, i.e., $Cons(\Pi, query(M)) = \{\perp\}$. It is easy to show that Proposition 7 also holds for subgraphs of AC_Π that are constructed by dropping all other edges due to *Query Propagate* but *Query Propagate*$_\perp$. Let Π be an AC program (4). Path (10) in AC_Π does not comply with the priorities of the ACSOLVER algorithm. On the other hand, the path

$$\begin{array}{l}
\emptyset \overset{\textit{Unit Propagate}}{\Longrightarrow} occurs(a,0) \overset{\textit{Unit Propagate}}{\Longrightarrow} occurs(a,0)\ next(1,0) \overset{\textit{Unit Propagate}}{\Longrightarrow} \\
occurs(a,0)\ next(1,0)\ holds(f,1) \overset{\textit{Decide}}{\Longrightarrow} \\
occurs(a,0)\ next(1,0)\ holds(f,1)\ \neg on^{\Delta} \overset{\textit{Decide}}{\Longrightarrow} \\
occurs(a,0)\ next(1,0)\ holds(f,1)\ \neg on^{\Delta}\ okTime(T)^{\xi\Delta} \overset{\textit{Query Propagate}}{\Longrightarrow} \\
occurs(a,0)\ next(1,0)\ holds(f,1)\ \neg on^{\Delta}\ okTime(T)^{\xi\Delta}\ \perp \overset{\textit{Backtrack}}{\Longrightarrow} \\
occurs(a,0)\ next(1,0)\ holds(f,1)\ \neg on^{\Delta}\ \neg okTime(T)^{\xi} \overset{\textit{Unit Propagate}}{\Longrightarrow} \\
occurs(a,0)\ next(1,0)\ holds(f,1)\ \neg on^{\Delta}\ \neg okTime(T)^{\xi}\ \neg T \neq 1^{\xi}
\end{array} \tag{11}$$

is a valid path of ACSOLVER. Indeed, this path respects the fact that the transition rule *Unit Propagate* has a higher priority than *Decide*.

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 7 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 7 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 different priorities of transition rules is still correct.

Note that for a clingcon program Π, $ac(\Pi)$ 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 for this purpose.

7. Abstract CLINGCON

The CLINGCON system is based on tight coupling of the answer set solver CLASP and the constraint solver GECODE. The CLASP system starts its computation by building a propositional formula called completion [31] 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 $\text{SM(ASP)}_{F,\Pi}$ [32]. These extensions are essential for capturing such advanced features of CLASP and CLINGCON as conflict-driven backjumping and learning. In this section we start by reviewing the graph $\text{SM(ASP)}_{F,\Pi}$ and showing 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. We call new graph $\text{CON}_{F,\Pi}$. This abstract view on basic CLINGCON allows us to compare it to ACSOLVER in formal terms. To capture full CLINGCON algorithm we extend $\text{CON}_{F,\Pi}$ with the rules *Backjump* and *Learn* in a similar manner as the graph $\text{SM(ASP)}_{F,\Pi}$ was extended to the graph $\text{SML(ASP)}_{F,\Pi}$ in [32].

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*, *Unfounded'*:

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

Figure 3: The transition rule *Unfounded'*.

Backtrack of the graph $\textsc{sm}_{F^r \cup \Pi}$, and the transition rule *Unfounded'* presented in Figure 3

Lierler and Truszczynski (2011) demonstrated how $\textsc{sm}(\textsc{asp})_{ED\text{-}Comp(\Pi),\Pi}$ models *basic* CLASP (without conflict driven backjumping and learning) where $ED\text{-}Comp(\Pi)$ denotes clausified completion with the use of auxiliary atoms. Formula $ED\text{-}Comp(\Pi)$ exhibits an important property that Lierler and Truszczynski call Π-safe. We now extend this notion to AC programs. For an AC program Π, a set F of clauses is Π-*safe* if

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

We note that, a set F of clauses is Π-safe if it is Π^ξ-safe according to the "safeness" definition given in [32]. If all regular atoms in Π also occur in its regular part, formula $\Pi^{\xi^{cl}}$ is a straight-forward example of Π-safe formula. Under the same restriction, formulas $Comp(\Pi^\xi)$ and $ED\text{-}Comp(\Pi^\xi)$ are also Π-safe, where the former stands for the completion of Π^ξ whose formulas are clausified in a straightforward way by applying distributivity. We refer the reader to [32] for precise definitions of $Comp(\Pi)$ and $ED\text{-}Comp(\Pi)$ constructed from the Clark's completion of Π [31]. We call AC programs, which satisfy the restriction that all regular atoms in Π also occur in its regular part, *friendly.* This restriction is inessential as for such atoms adding simple constraints will turn a non friendly program into a friendly one.

We now define the graph $\textsc{con}_{F,\Pi}$ for AC programs that extends $\textsc{sm}(\textsc{asp})_{F,\Pi}$ in a similar way as AC_Π extends $\textsc{sm}_\Pi$.

For an AC program Π and a set F of clauses, the nodes of $\textsc{con}_{F,\Pi}$ are the states relative to the set $At(F \cup \Pi^\xi) \cup At(\Pi_D)_R$. The edges of $\textsc{con}_{F,\Pi}$ are described by the transition rules of $\textsc{sm}(\textsc{asp})_{F,\Pi^\xi}$ and the transition rule *Query Propagate* of AC_Π.

Proposition 8. *For any weakly-simple* AC *program* Π *and a* Π*-safe set* F *of clauses,*

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

(b) for any terminal state M *of* $\textsc{con}_{F,\Pi}$ *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* $\textsc{con}_{F,\Pi}$ *if and only if* Π *has no answer sets.*

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

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

Proposition 8 demonstrates that the basic CLINGCON algorithm is applicable not only to clingcon programs but also to a broader class of weakly-simple *AC* programs.

7.1. On the Relation of ACSOLVER *and Basic* CLINGCON

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

- the only transition rule justifying this edge is *Unfounded*, and
- some edge $M \Longrightarrow M''$ 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. Indeed, given that *Unfounded* is assigned lowest priority, the singular edges will never be followed as other transitions such as *Unit Propagate* are available (see the second condition of the definition of a singular edge).

We define AC^-_Π ($\textsc{con}^-_{F,\Pi}$) as the graph obtained by removing all singular edges from AC_Π ($\textsc{con}_{F,\Pi}$).

Proposition 9. *For a friendly* AC *program* Π*, the graphs* AC^-_Π *and* $\textsc{con}^-_{Comp(\Pi^\xi),\Pi}$ *are equal.*

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

7.2. The CONL$_{F,\Pi}$ *Graph*

The CLINGCON algorithm incorporates backjumping and learning, modern techniques of DPLL-like procedures. Nieuwenhuis et al. [15] provide comprehensive description of these techniques. We start by briefly describing the main ideas behind them. We then proceed to defining the graph CONL$_{F,\Pi}$ that will allow us to model the CLINGCON algorithm featuring backjumping and learning.

Consider a state

$$M_0 \; l_1^{\Delta} \ldots l_n^{\Delta} \; M_n, \tag{12}$$

where $l_1^{\Delta} \ldots l_n^{\Delta}$ are the only decision literals. We say that all the literals of each $l_i M_i$ belong to decision level i. Consider a state of the form (12) such that the transition rule *Backtrack* is applicable to it. It is easy to see that the rule *Backtrack* has an effect of backtracking from decision level n to level $n-1$. At times, it is safe to backtrack to a decision level prior to $n-1$. This process is called *backjumping.*

Learning technique is responsible for augmenting the database of given clauses or logic rules in the hope that newly acquired information is instrumental in future search. This technique proved to be of extreme importance to the success of modern SAT and ASP technology.

We now define the graph CONL$_{F,\Pi}$. To accommodate the fact that this graph has to capture learning, we introduce the notion of an augmented state that includes not only currently assigned literals but also of "learned information". Such learned information corresponds to newly derived constraints that become available for future propagations. In case of CLINGCON these constraints are represented as clauses. We say that a regular program Π entails a clause C when for each consistent and complete set M of literals, if M^+ is an answer set for Π, then $M \models C$. For instance, any regular program entails each rule (understood as a clause) occurring in it. For an AC program Π and a set F of clauses, an *augmented state* relative to F and Π is either a distinguished state $\bot$ or a pair $M||\Gamma$ where M is a record relative to the set $At(F \cup \Pi^{\xi}) \cup At(\Pi_D)_R$, and Γ is a set of clauses over $At(F \cup \Pi^{\xi}) \cup At(\Pi_D)_R$ such that $F \models \Gamma$ or $\Pi^{\xi} \models \Gamma$.

Unit Propagate Learn:

$$M||\Gamma \implies M\ l||\Gamma \text{ if } \begin{cases} C \vee l \in F \cup \Pi^{\xi^{cl}} \cup \Gamma \text{ and} \\ \overline{C} \subseteq M \end{cases}$$

Backjump:

$$P\ l^{\Delta}\ Q||\Gamma \implies P\ l'||\Gamma \text{ if } \begin{cases} P\ l^{\Delta}\ Q \text{ is inconsistent and} \\ F \models l' \vee \overline{P} \text{ or } \Pi^{\xi} \models l' \vee \overline{P} \end{cases}$$

Learn:

$$M||\Gamma \implies M||\ C,\ \Gamma \text{ if } \begin{cases} \text{every atom in } C \text{ occurs in } F \text{ or } \Pi^{\xi}, \\ F \models C \text{ or } \Pi^{\xi} \models C, \text{ and} \\ C \notin \Gamma. \end{cases}$$

Figure 4: The additional transition rules of the graph $\text{CONL}_{F,\Pi}$.

For an AC program Π and a set F of clauses, the nodes of $\text{CONL}_{F,\Pi}$ are the *augmented state* relative to F and Π. The rules *Decide*, *Unfounded*, and *Fail* of $\text{SM}(\text{ASP})_{F,\Pi}$ are extended to $\text{CONL}_{F,\Pi}$ as follows: $M||\Gamma \implies M'||\Gamma$ ($M||\Gamma \implies \bot$, respectively) is an edge in $\text{CONL}_{F,\Pi}$ justified by *Decide* or *Unfounded* (*Fail*, respectively) if and only if $M \implies M'$ ($M \implies \bot$) is an edge in $\text{SM}(\text{ASP})_{F,\Pi}$ justified by *Decide* or *Unfounded* (*Fail*, respectively). The other transition rules of $\text{CONL}_{F,\Pi}$ are presented in Figure 4. The transition rule *Backjump* describes the essence of backjumping procedure that replaces backtracking. The rule *Learn* captures the essence of learning in terms of hybrid constraint answer set solvers exemplified by CLINGCON. The rule *Unit Propagate Learn* is a modification of the transition rule *Unit Propagate* in $\text{CON}_{F,\Pi}$. This modification addresses the effect of learning. Indeed, this unit propagate derives atoms not only from given F and Π (that are identical in all sets of states in $\text{CONL}_{F,\Pi}$), but also from a set of learnt clauses Γ that depends on a particular state in $\text{CONL}_{F,\Pi}$. We refer to the transition rules *Unit Propagate Learn*, *Unfounded'*, *Backjump*, *Decide*, and *Fail* of the graph $\text{SML}(\text{ASP})_{F,\Pi}$ as *basic.* We say that a node in the graph is *semi-terminal* if no rule other than *Learn* is applicable to it. We omit the word "augmented" before "state" when this is clear from a context.

The graph $\text{CONL}_{F,\Pi}$ can be used for deciding whether a weakly-simple AC program Π has a model in the following sense.

Proposition 10. *For any weakly-simple* AC *program* Π *and a* Π*-safe set* F

of clauses,

(a) every path in $\textsc{conl}_{F,\Pi}$ *contains only finitely many edges justified by basic transition rules,*

(b) for any semi-terminal state $M||\Gamma$ *of* $\textsc{conl}_{F,\Pi}$ *reachable from* $\emptyset||\emptyset$, $(M^{\xi-})^{+}_{R} \cap At(\Pi)$ *is the set of all regular atoms in some answer set of* Π*,*

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

On the one hand, Proposition 10 (a) asserts that if we construct a path from $\emptyset||\emptyset$ so that basic transition rules periodically appear in it then some semi-terminal state is eventually reached. On the other hand, parts (b) and (c) of Proposition 10 assert that as soon as a semi-terminal state is reached the problem of deciding whether Π has an answer set is solved. In other words, Proposition 10 shows that the graph $\textsc{conl}_{F,\Pi}$ gives rise to a class of correct algorithms for computing answer sets for weakly-simple AC programs. It gives a proof of correctness to every CASP solver in this class and a proof of termination under the assumption that basic transition rules periodically appear in a path constructed from $\emptyset||\emptyset$.

Nieuwenhuis et al. 2006 proposed transition rules to model such techniques as forgetting and restarts. The graph $\textsc{conl}_{F,\Pi}$ can easily be extended with such rules.

7.3. CLINGCON *algorithm*

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

Backjump(*Learn*), *Fail* >> *Unit Propagate Learn* >> *Unfounded'* >>
Query Propagate >> *Decide.*

By *Backjump*(*Learn*) we denote the fact that learning occurs in CLINGCON every time backjump occurs. Proposition 10 demonstrates that the CLINGCON algorithm is applicable not only to clingcon programs but also to a broader class of weakly-simple AC programs.

8. The EZCSP Language and Algorithm

Balduccini [4] demonstrated that the EZCSP language may be seen as a subset of the AC language. In fact, it is a subset of the $AC^{\leftharpoonup}$ language. EZCSP restricts the $AC^{\leftharpoonup}$ by requiring that constraint atoms occur only in the rules whose head is symbol $\bot$. The EZCSP system is based on *loose* coupling of answer set solvers, e.g., SMODELS or CLASP, and constraint logic programming systems, e.g., SICStus Prolog. The system EZCSP treats a given program as a regular program (with slight modifications accounting for the special treatment of constraint atoms) and allows an answer set solver to find an answer set. This answer set is used to form a query that is then processed by a constraint logic programming system. This process may be repeated. To make these ideas precise we may model the system EZCSP that couples the answer set solver SMODELS and SICStus Prolog using the graph AC_Π. The EZCSP algorithm selects edges according to the priorities on the transition rules of AC_Π as follows:

$$\begin{array}{l} \textit{Backtrack}, \textit{Fail} >> \\ \textit{Unit Propagate}, \textit{All Rules Cancelled}, \textit{Backchain True} >> \\ \textit{Unfounded} >> \textit{Decide} >> \textit{Query Propagate}_\bot. \end{array}$$

Note that this set of priorities highlights the difference between ACSOLVER and EZCSP (based on SMODELS) as the difference in priorities on the transitions that the systems follow. Indeed, EZCSP always follows the transition *Decide* prior to exploring the transition due to *Query Propagate*.

Similarly, we can use the graph $\text{CONL}_{F,\Pi}$ to capture the EZCSP algorithm based on the answer set solver CLASP. It will demonstrate that the difference between CLINGCON and EZCSP (based on CLASP) roots in the same place as the difference between ACSOLVER and EZCSP. The EZCSP selects edges according to the priorities on the transition rules of $\text{CONL}_{\textit{ED-Comp}(\Pi^\xi),\Pi}$ as follows:

$$\begin{array}{l} \textit{Backjump}(\textit{Learn}), \textit{Fail} >> \\ \textit{Unit Propagate Learn}, >> \\ \textit{Unfounded'} >> \textit{Decide} >> \textit{Query Propagate}_\bot. \end{array}$$

EZCSP always follows the transition *Decide* prior to exploring the transition due to *Query Propagate*. Furthermore, unlike CLINGCON which allows *Query Propagate* to propagate new atoms EZCSP uses only limited version of this transition rule, in particular *Query Propagate*$_\bot$. In other words, EZCSP may only conclude that constraint atoms of a program are conflicting but not able to derive any inferences from it.

9. Conclusions, Discussions, and Future Work

We started this paper by listing a number of CASP languages and systems ACSOLVER, CLINGCON, EZCSP, IDP, INCA, and MINGO that recently have come into use as an attempt to broaden the applicability of automated reasoning methods. Distinguishing feature of such CASP solvers as ACSOLVER, CLINGCON, EZCSP is combining the inferences stemming from traditionally different research fields. This general interest towards hybrid solving illustrates the importance of developing synergistic approaches in the automated reasoning community. A clear picture of the distinguishing features of CASP-like languages and underlying systems is of importance in order to facilitate further developments of the field. This is the prime focus of this work that (a) formally states the relation between the CASP languages of ACSOLVER, CLINGCON, and EZCSP and (b) provides a systematic account on the algorithmic differences between the underlying solvers. For example, one take home lesson is that the languages of CLINGCON and EZCSP are the syntactic variants of AC^-. Another lesson is that despite all the technological differences in the newly developed solvers such as ACSOLVER, CLINGCON, and EZCSP they have a lot in common. This alludes to a possibility of creating a general-purpose platform that would assist the creation of new architectures of CASP-like technology.

To summarize the technical contributions of this paper: we demonstrated a formal relation between the AC and CLINGCON languages and the algorithms behind ACSOLVER and CLINGCON. We designed transition systems AC_Π and $\text{CONL}_{F,\Pi}$ for describing algorithms for computing (subsets of) answer sets of weakly-simple AC programs. We used these graphs to specify the ACSOLVER, CLINGCON and EZCSP algorithms. Compared with traditional pseudo-code descriptions 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 $\text{CON}_{F,\Pi}$, namely that the graphs AC^-_Π and $\text{CON}^-_{Comp(\Pi),\Pi}$ are equal. 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 systems, 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. Work by Balduccini et al. [33] is a step in that direction.

In the future we would like to uncover the precise relationship with IDP, and the translational solvers introduced in [8, 9, 10]. The IDP language builds on top of the formalism called PC(ID) [34] that is strongly related to logic programs under answer set semantics [32]. As of this point there is no formal account describing the insides of the IDP system supporting a CASP language. The translational solvers developed in [8, 9] rely on transformations from a CASP language to an ASP formalism. A graph underlying an ASP solver (for instance, $\text{SML}(\text{ASP})_{F,\Pi}$ for the case of CLASP [32]) applied to a transformation devised in [8, 9] can be used to characterize such a solver. Even though we introduced similar graphs, AC and CONL, for capturing hybrid solvers these graphs are not appropriate to formally compare translational and hybrid solvers. The transition rule *Query Propagate* of AC and CONL is too crude to capture the details of solving that occurs on the side of specialized solvers (such as GECODE or SICStus Prolog). To point at exact differences between a translational solver and its hybrid counterpart the computation done in a specialized solver has to be unfolded. The MINGO solver [10] implements CASP by translating its programs into integer linear programs. It is an interesting direction of research to study how the technology of integer linear programming compares to the technology of ASP and CASP.

Acknowledgments

We are grateful to Yuanlin Zhang, Daniel Bailey, Marcello Balduccini, Broes de Cat, Michael Gelfond, Vladimir Lifschitz, Max Ostrowski, Peter Schueller, and Miroslaw Truszczynski for useful discussions related to the topic of this work as well as their comments on earlier drafts of the paper. We are in debt to the reviewers of the paper for their valuable feedback that allowed to improve the presentation.

10. Appendix: Proofs of Formal Results

We start by introducing the necessary terminology used in [1].

A consistent set S of ground atoms over the signature Σ is called a *partial interpretation* of an AC program Π if it satisfies the following conditions:

1. A constraint atom $l \in S$ iff l is true under its intended interpretation;

2. The mixed atoms of S form the functional set w.r.t. the signature of Π.

The definition of semantics of AC program follows [1]. By replacing "nested program" with "ground AC program", the definition of reduct in Section 2 is trivially extended to ground AC programs. A partial interpretation S of Σ is an *acc-answer set* of an AC program Π if S is minimal (in the sense of set-theoretic inclusion) among the partial interpretations of Σ satisfying the rules of $(\mathit{ground}(\Pi) \cup M)^S$, where M is the set of all mixed atoms occurring in S. We note that this is a generalization of the answer set definition presented in [1] to the case of programs with doubly negated atoms.

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

Proof. Left-to-right: Let X be an answer set of Π. By definition, there is a functional set M of ground mixed atoms such that X is an answer set of $\mathit{ground}^*(\Pi) \cup M$, i.e., minimal among sets of atoms satisfying

$$((\mathit{ground}^*(\Pi) \cup M)^X)^{cl}. \tag{13}$$

Obviously, M forms the set of all mixed atoms occurring in X.

Let the set Y of atoms be any model of (13). It is easy to see that (i) $Y \cup T$ is a partial interpretation, and (ii) $Y \cup T$ satisfies

$$((\mathit{ground}(\Pi) \cup M)^{X \cup T})^{cl}. \tag{14}$$

Indeed, from the construction of (13) and (14) it immediately follows that we obtain (13) from (14) by replacing every atom from T with $\top$. Thus $Y \cup T$ satisfies (14) iff Y satisfies (13). It follows that $X \cup T$ is a minimal partial interpretation satisfying (14) (since X is minimal among sets of atoms satisfying (13)) and hence is an acc-answer set of Π.

Right-to-left. Let $X \cup T$ be an acc-answer set of Π. By M we denote the set of all mixed atoms occurring in X.

Let $Y \cup T$ be any partial interpretation satisfying (14). Using the argument from left-to-right direction we derive that Y is a model satisfying (13). From the fact that $X \cup T$ is a minimal partial interpretation satisfying (14) (i.e., it is an acc-answer set of Π) it follows that X is a minimal model satisfying (13). □

A *splitting set* [35] for a nested program Π is any set U of atoms such that, for every rule $r \in \Pi$, if $Head(r) \cap U$ then $At(r) \in U$. The set of rules $r \in \Pi$ such that $At(r) \in U$ is called a *bottom* of Π relative to the splitting set U and denoted by $b_U(\Pi)$. The set $\Pi \setminus b_U(\Pi)$ is the top of Π relative to U. By $e_U(\Pi, X)$, we denote the program consisting of all rules obtained from Π by replacing an atom a in U, if $a \in X$ with $\top$ and $\bot$ otherwise.

Proposition 11 (Splitting Set Theorem [35])**.** *Let U be a splitting set for a nested program Π. A set A of atoms is an answer set for Π iff $A = X \cup Y$ where X is an answer set for $b_U(\Pi)$ and Y is an answer set for $e_U(\Pi \setminus b_U(\Pi), X)$.*

For a set M of ground mixed atoms over Σ, by $\tilde{M}$ we denote the following set of atoms

$$\{m \mid m \text{ is a ground mixed atom over } \Sigma,\ m \notin M\},$$

i.e., the set consisting of all ground mixed atoms over Σ that are not in M.

Observation 1. *A set of atoms is an answer set of nested program Π if and only if it is an answer set of the regular program constructed from Π by*

- *dropping the rules where $\bot$ occurs in $B^{pos} \cup B^{neg2}$ and $\top$ occurs in B^{neg},*
- *dropping $\top$, not $\bot$, and not not $\top$ from the rest of the rules.*

This observation is due to equivalent transformations on programs with nested expressions [24].

Proposition 2. *For a clingcon program Π over signature Σ, a set X is a* constraint answer set *of Π according to the definition in [2, 3] iff there is a functional set M of ground mixed atoms of Σ such that $X \cup M$ is an answer set of $ac(\Pi)$.*

Proof. It is easy to see that $ac(\Pi)$ is a super-safe program (by its construction and function V definition).

Left-to-right: Let X be a *constraint answer set* of Π according to the definition in [2]. Then there is an *assignment* $A : V[\Pi] \to D[\Pi]$ such that X is an answer set of Π^A defined in [2], where $V[\Pi]$ denotes the set of all clingcon variables occurring in Π and $D[\Pi]$ is a set of constraint constants. For a clingcon variable c in Π, by $A(c)$ we denote a constraint constant assigned by A to c. Let us construct a substitution γ using A as follows: for each clingcon variable c in Π add $c^V/A(c)$ to γ.

For a clingcon variable $p(\vec{r})$ by $p(\vec{r})^M$ we denote a mixed atom $p(\vec{r}, p(\vec{r})^V)$. We say that such mixed atom is *matching* for $p(\vec{r})$. Let S denote the set of matching mixed atoms constructed from all the clingcon variables occurring in Π. It is easy to see that $S\gamma$ is a functional set of ground mixed atoms. Let M be $S\gamma$. We now show that $X \cup M$ is an answer set of $ac(\Pi)$. By the definition, $X \cup M$ is an answer set of $ac(\Pi)$ if $X \cup M$ is an answer set of

$$ground^*(ac(\Pi)) \cup M. \tag{15}$$

By Proposition 11, $X \cup M$ is an answer set of (15) iff X is an answer set of

$$ground^*(ac(\Pi)). \tag{16}$$

The transformation described in Observation 1 applied to (16) will result in Π^A. We are given that X is an answer set of Π^A. Consequently, it is an answer set of (16).

Right-to-left: Let M be a functional set of ground atoms of Σ and X be a set of ground atoms such that $X \cup M$ is an answer set of $ac(\Pi)$. Let $V[\Pi]$ denote the set of all clingcon variables occurring in Π. From the fact that M is a functional set it follows that for every variable $p(\vec{r}) \in V[\Pi]$ there is an atom of the form $p(\vec{r}, c)$ in M. We can construct an assignment A for $V[\Pi]$ as follows: for every variable $p(\vec{r}) \in V[\Pi]$, $A(p(\vec{r})) = c$ where $p(\vec{r}, c) \in M$.

By the definition of an answer set for an AC program, $X \cup M$ is an answer set of (15). Furthermore, by Proposition 11, X is an answer set of (16). From Π^A construction and it follows that the transformation described in Observation 1 applied to (16) will result in Π^A. We are given that X is an answer set of Π^A. Consequently, X is an answer set of Π^A and therefore a constraint answer set of Π. $\square$

Proposition 3. *For any safe* AC *program* Π, *there is a transformation on* Π *that produces a super-safe* AC *program which has same answer sets as* Π.

Proof. In this proof, $\vec{c}$ always denotes a sequence of regular constants, and m denotes a mixed predicate.

Let T denote the following transformation

```
For each sequence of constants $\vec{c}$ and each $m$ such that $\vec{c}$ is specified by $m$
    Associate a unique new variable with $\langle m, \vec{c} \rangle$ denoted by $Y_{\langle m,\vec{c}\rangle}$;
For each rule $r$ of $\Pi$
    Let $r'$ be the same as $r$;
        For each variable $X$ of $r$
            Let $m_1(\vec{c_1}, X), ...m_k(\vec{c_k}, X)$ be the mixed atoms of $r$
            Replace $m_i(\vec{c_i}, X)$ in $r'$ with $m_i(\vec{c_i}, Y_{\langle m_i,\vec{c_i}\rangle})$ for $i \in 1..k$;
            Add $Y_{\langle m_1,\vec{c_1}\rangle} = \ldots = Y_{\langle m_i,\vec{c_i}\rangle}$ to $r'$;
            Replace each occurrence of $X$ in $r'$ by $Y_{\langle m_1,\vec{c_1}\rangle}$.
```

Let Π' be the new program produced by the transformation T. It is easy to see that Π' is indeed a super safe program as a unique new variable is associated with every pair $\langle m, \vec{c}\rangle$ and Π' is constructed from Π to preserve the requirements of super-safe program.

To prove that Π and Π' have the same answer sets, let M be any functional set. It is easy to see that $\tilde{M}$ is a splitting set of *ground*$^*(\Pi) \cup M$ and *ground*$^*(\Pi') \cup M$, where the bottom of both programs is formed by empty set of rules. By Proposition 11, it follows that a set X of atoms is an answer set of *ground*$^*(\Pi) \cup M$ (thus, of Π) iff X is an answer set of

$$e_{\tilde{M}}(\mathit{ground}^*(\Pi) \cup M, X). \tag{17}$$

Similarly, X is an answer set of *ground*$^*(\Pi') \cup M$ (thus, of Π') iff it is an answer set of

$$e_{\tilde{M}}(\mathit{ground}^*(\Pi') \cup M, X). \tag{18}$$

It is easy to see that the transformation described in Observation 1 applied to (17) and (18) results in the same program. Thus any set X of atoms is an answer set of Π iff it is an answer set of Π'. $\square$

Proposition 4 is proved in the same spirit as Proposition 2.

We refer the reader to [30] for the review of operator SM but we state several formal results from [30] and [26] in the form convenient for our presentation.

Proposition 12 (Special case of Theorem 1 [30])**.** *For any nested program* Π *and a complete set* X *of literals over* $At(\Pi)$*, the following conditions are equivalent*

- X^+ *is an answer set of* Π

- *X is a $\boldsymbol{p}$-stable model [30], where $\boldsymbol{p}$ is the list of all predicate symbols in Π (i.e., X is a model of $SM_{\boldsymbol{p}}[\Pi]$)).*

Proposition 13 (Special case of Symmetric Splitting Theorem [26])**.** *Let Π and Π' be nested programs and let $\boldsymbol{p}$, $\boldsymbol{q}$ be disjoint tuples of distinct predicate symbols. If*

- *each strongly connected component of predicate dependency graph of $\Pi \cup \Pi'$ is a subset of $\boldsymbol{p}$ or a subset of $\boldsymbol{q}$,*
- *no atom with predicate symbols in $\boldsymbol{q}$ occurs in any head in Π,*
- *no atom with predicate symbols in $\boldsymbol{p}$ occurs in any head in Π',*

then $SM_{\boldsymbol{pq}}[\Pi \cup \Pi']$ is equivalent to $SM_{\boldsymbol{p}}[\Pi] \wedge SM_{\boldsymbol{q}}[\Pi']$.

For a set A of atoms by $pred(A)$ we denote the set of predicate symbols of atoms in A. We will sometime abuse the notation and use $pred(\Pi)$ to denote the set of predicate symbols occurring in program Π.

Proposition 6. *For a nested program Π, a complete set X of literals, and a set $\boldsymbol{p}$ of predicate symbols such that predicate symbols occurring in the heads of Π form a subset of $\boldsymbol{p}$ (i.e., $pred(Heads(\Pi)) \subseteq \boldsymbol{p}$), X^+ is a $\boldsymbol{p}$-input answer set of Π iff X is a model of $SM_{\boldsymbol{p}}[\Pi]$ (i.e., $\boldsymbol{p}$-stable model of Π).*

Proof. By the definition, X^+ is a $\mathbf{p}$-input answer set of Π iff X^+ is an answer set of $\Pi \cup X^+(\mathbf{p})$. By Proposition 12, X^+ is an answer set of $\Pi \cup X^+(\mathbf{p})$ iff X is a model of

$$SM_{pred(\Pi)\ pred(X^+(\mathbf{p}))}[\Pi \cup X^+(\mathbf{p})]. \tag{19}$$

From the fact that $pred(Heads(\Pi)) \subseteq \mathbf{p}$, it follows that

$$X^+(\mathbf{p}) \cap Heads(\Pi) = \emptyset,$$

so that atoms with predicate symbols different from $\mathbf{p}$ occur in the heads only of the facts of $X^+(\mathbf{p})$. Consequently, by Proposition 13, (20) is equivalent to

$$SM_{pred(\Pi)}[\Pi] \wedge SM_{pred(X^+(\mathbf{p}))}[X^+(\mathbf{p})]. \tag{20}$$

Obviously, set X is a model of the second conjunct of (20). We derive that X is a model of (20) iff X is a model of $SM_{pred(\Pi)}[\Pi]$. Recall that $pred(Heads(\Pi)) \subseteq \mathbf{p}$. By Proposition 13, the following expressions are equivalent

- $SM_{pred(\Pi)}[\Pi]$,
- $SM_{pred(Heads(\Pi))}[\Pi]$,
- $SM_{\mathbf{p}}[\Pi]$.

$\square$

Weakly-simple AC programs satisfy important syntactic properties that allow to characterize their answer sets by means of queries based on them. The proof of Proposition 7 relies on this alternative characterization that we make precise in Lemma 1. To state the lemma we introduce several concepts that we rely on in the proof.

For an AC program Π, we say that a query Q is based on Π if it is a complete and consistent set of literals over $At(\Pi_D)_R \cup At(\Pi_R)_{D,C}$. If Q is a satisfying extension of itself, then there is a substitution γ of variables in Q by ground terms such that the result, $Q\gamma$, satisfies the conditions 1 and 2 of the definition of satisfying extension. We say that $Q\gamma$ is an *interpretation* of Q. We call such queries *satisfiable* (i.e., queries that are satisfying extensions).

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

1. eliminating Π_D,
2. dropping removing each occurrence of a mixed atom,
3. replacing an atom a by $\top$ if $a \in Q_{D,C}$,
4. replacing an atom a by $\bot$ if $\neg a \in Q_{D,C}$,
5. 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 nested program. For instance, let Π be program (4). A query Q consisting of literals

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

is based on Π. Program $\Pi(Q)$ follows

$$\begin{array}{l} \leftarrow\ occurs(a,0),\ \top,\ not\ \top \\ occurs(a,0) \leftarrow \\ \{on\} \\ \leftarrow\ not\ \ on. \end{array}$$

Let Q_1 be a query based on Π consisting of literals

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

Program $\Pi(Q)$ follows

$$\begin{array}{l} \leftarrow\ occurs(a,0),\ \top,\ not\ \bot \\ occurs(a,0) \leftarrow \\ \{on\} \\ \leftarrow\ not\ \ on. \end{array}$$

An *expression* is a term, an atom, or a rule. Given a substitution Θ and an expression e, we write $e\Theta$ for the result of replacing v_i in e by t_i for every $i \leq n$.

Recall that 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_{Mix} we denote a set of mixed literals occurring in S.

We call a rule (1) with $a_0 = \bot$ a *constraint.*

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 some answer set of* Π *such that* X *is the set of all regular atoms in it.*

Proof. Left-to-right: Assume that Π has an answer set. Let S be a complete and consistent set of ground[11] literals over Σ such that S^+ is an answer set of Π. By the definition of an answer set, there exists a functional set M of ground mixed atoms over Σ such that S^+ is an answer set of the nested program $ground^*(\Pi) \cup M$.

It is sufficient to construct a satisfiable query based on Π such that $\Pi(Q)$ has an answer set. We start by constructing a query Q based on Π using S. Then we demonstrate that

(i) this Q is satisfiable w.r.t. Π, and

(ii) $\Pi(Q)$ has an answer set.

[11]We will sometimes omit the word "ground", when it is clear from the context.

Let Π_{Mix} be a subset of Π_R that consists of all rules in Π_R whose bodies contain mixed, defined, or constraint literals. Since Π is a weakly-simple program and every weakly-simple program is safe, it follows that each rule in Π_{Mix} contains a mixed atom. From the fact that M is a functional set of mixed atoms and Π is a weakly-simple program and hence super-safe (see conditions 1 and 2), it follows that for each rule r in Π_{Mix} there is a substitution Θ such that $r\Theta \in \mathit{ground}(\Pi)$ so that every mixed atom m occurring in $r\Theta$ also occurs in M; furthermore, there is no substitution Θ' different from Θ such that every mixed atom m occurring in $r\Theta'$ also occurs in M. Let us denote such rule $r\Theta$ that corresponds to r in $\mathit{ground}(\Pi)$ by $r(\mathit{ground}(\Pi), M)$.

We now construct a query Q as follows.

1. Let Q contain every regular literal in S_R whose atom occurs in Π_D,
2. For each rule r in Π_{Mix}, so that by Ξ we denote the substitution such that $r\Xi = r(\mathit{ground}(\Pi), M)$, let Q contain
   (a) defined atom d, if d occurs in r and $d\Xi \in S_D^+$,
   (b) defined literal $\neg d$, if atom d occurs in r and $d\Xi \in S_D^-$,
   (c) constraint atom c, if c occurs in r and $c\Xi$ is true under intended interpretation of its symbols
   (d) constraint literal $\neg c$, if c occurs in r and $c\Xi$ is false under intended interpretation of its symbols.

From the query Q construction (and the fact that Π_{Mix} consists of all rules in Π_R whose bodies contain mixed, defined, or constraint literals) it immediately follows that Q is based on Π. Let γ be the union of Ξ for each rule r in Π_{Mix} such that Ξ is the substitution so that $r\Xi = r(\mathit{ground}(\Pi), M)$. From the fact that Π is a weakly-simple program and hence super-safe and the choice of Ξ that relies on a functional set M of mixed atoms it follows that γ is also a substitution.

In order to demonstrate *(i)* and *(ii)* we introduce the following notation and state additional observations. By $R(\Pi)$, $D(\Pi)$, $Mix(\Pi)$ we denote the set of regular, defined, and mixed predicates of Π respectively. From Propositions 12, 13 and the fact that Π is a weakly-simple program it follows that S^+ is an answer set of Π iff S is a model of

$$SM_{R(\Pi)\ Mix(\Pi)}[\mathit{ground}^*(\Pi_R) \cup M] \wedge SM_{D(\Pi)}[\mathit{ground}^*(\Pi_D)]. \qquad (21)$$

Consequently, S is a model of

$$SM_{D(\Pi)}[\mathit{ground}^*(\Pi_D)].$$

By Proposition 6, S^+ is an input answer set of $ground^*(\Pi_D)$ w.r.t. $D(\Pi)$. Hence, S^+ is an input answer set of Π_D w.r.t. $D(\Pi)$. This constitutes observation (a).

Similarly, S is a model of

$$SM_{R(\Pi)\ Mix(\Pi)}[ground^*(\Pi_R) \cup M]. \tag{22}$$

Furthermore, from Proposition 13, (22), and the fact that mixed atoms appear in the heads only in facts in M, it follows that S is a model of

$$SM_{R(\Pi)}[ground^*(\Pi_R)].$$

By Proposition 6, S^+ is an input answer set of $ground^*(\Pi_R)$ w.r.t. $R(\Pi)$. This constitutes observation (b).

(i) We now show that $Q\gamma$ is a query interpretation for a query Q w.r.t. Π. From the Q construction (see conditions 2c and 2d) it follows that if a constraint literal $l \in Q\gamma$ then l is true under the intended interpretation of its symbols. It is left to show that there is an input answer set A of Π_D w.r.t. defined predicates of Π, so that $Q\gamma^+_{R,D} \subseteq A$ and $Q\gamma^-_{R,D} \cap A = \emptyset$: Let A be S^+ and recall observation (a) and the construction of Q (see conditions 2a and 2b).

(ii) We now show that $\Pi(Q)$ has an answer set. Recall observation (b). By the definition of an input answer set, S^+ is an answer set of

$$ground^*(\Pi_R) \cup S^+_{R(\Pi)}.$$

Recall that for a set X of atoms, by $X_{\mathbf{p}}$ we denote the set of atoms in X whose predicate symbols are different from the ones occurring in $\mathbf{p}$. It is easy to see that $S^+_{R(\Pi)} = S^+_D \cup S^+_{Mix}$. In other words, $ground^*(\Pi_R) \cup S^+_{R(\Pi)}$ is equal to

$$ground^*(\Pi_R) \cup S^+_D \cup S^+_{Mix}. \tag{23}$$

It is easy to see that $S^+_D \cup S^+_{Mix} \cup S^-_D \cup S^-_{Mix}$ is a splitting set of (23). By Proposition 11, $S^+ \setminus (S^+_D \cup S^+_{Mix})$ is an answer set of the program constructed from (23) by

- eliminating facts $S^+_D \cup S^+_{Mix}$,
- replacing atom a by $\top$ if $a \in S^+_D \cup S^+_{Mix}$,
- replacing atom a by $\bot$ if $a \in S^+_D \cup S^+_{Mix}$.

It is easy to see that the transformation described in Observation 1 applied to such program and to $\Pi(Q)$ without the constraints introduced in step 5 of $\Pi(Q)$ construction results in the same program. It is easy to see that these constraints are satisfied by $S^+ \setminus (S^+_D \cup S^+_{Mix})$. By the Theorem on Constraints [24] it follows that $S^+ \setminus (S^+_D \cup S^+_{Mix})$ is an answer set of $\Pi(Q)$. Consequently, it is an answer set of $\Pi(Q)$.

Right-to-left: Assume that there is a query Q based on Π such that Q is satisfiable w.r.t. Π and $\Pi(Q)$ has an answer set. We show that Π also has an answer set.

From the fact that Q is satisfiable it follows that there is an interpretation I for Q w.r.t. Π. By the definition of the interpretation, there is a substitution γ such that $Q\gamma$ is I itself.

By the safety condition on weakly-simple programs, any variable X that occurs in some rule in Π also occurs in some mixed atom in this rule. Let us define a substitution γ' as follows: it extends the substitution γ by X/v_X for each variable X such that in every rule that X occurs in, it occurs only once; v_X is an arbitrary constant of the same sort as X (recall that sorts are non empty). By γ' construction, every variable in $At(\Pi)_{Mix}$ also occurs in γ'. Let M be the ground set of mixed atoms $At(\Pi)_{Mix}\gamma'$ (i.e., for each atom $m \in At(\Pi)_{Mix}\gamma'$, $m\gamma' \in M$.) From the fact that Π is a super-safe program it follows that M is a functional set over Σ.

By the definition of an answer set of an AC program, to demonstrate that Π has an answer set, it is sufficient to show that

$$ground^*(\Pi) \cup M \tag{24}$$

has the answer set. It is obvious that $M \cup \tilde{M}$ is a splitting set of (24). By Proposition 11, (24) has an answer set iff a program constructed from (24) by dropping facts M, dropping the rules that contain mixed atoms in $\tilde{M}$, and removing atoms from M from the remaining rules. We denote this ground program by $[\Pi, M]$.

By Proposition 12, $[\Pi, M]$ has an answer set iff there is a model of

$$SM_{D(\Pi),R(\Pi)}[\ [\Pi, M]\ ]. \tag{25}$$

Indeed, $D(\Pi), R(\Pi)$ form the set of all predicate symbols of a program. By Proposition 13, (25) is equivalent to

$$SM_{R(\Pi)}[\ [\Pi, M]_R\ ] \wedge SM_{D(\Pi)}[\ [\Pi, M]_D\ ].$$

That can be equivalently rewritten as

$$SM_{R(\Pi)}[\ [\Pi, M]_R\ ] \wedge SM_{D(\Pi)}[ground^*(\Pi_D)]. \tag{26}$$

Consider a complete and consistent set Y of ground regular literals such that Y^+ is an answer set of $\Pi(Q)$. (By our assumption Y exists). By the condition 5 of $\Pi(Q)$ construction it follows that $I_R \subseteq Y$. By the Theorem on Constraints [24] it follows that Y^+ is also an answer set of the program $\Pi(Q)'$ constructed from $\Pi(Q)$ by dropping the constraints in $\Pi(Q)$ derived from the condition 5. We now note that the transformation described in Observation 1 applied to $\Pi(Q)'$ results in a program identical to $[\Pi, M]_R$. We derive that Y^+ is an answer set of $[\Pi, M]_R$.

From the fact that I is a query an interpretation, it follows that there is an input answer set A of Π_D w.r.t. $D(\Pi)$ such that (i) $Q_R^+ \subseteq A$, $Q_R^- \cap A = \emptyset$ (note that $Q_R = I_R$), and (ii) $I_D^+ \subseteq A$, $I_D^- \cap A = \emptyset$. Furthermore, by Proposition 11, the input answer set definition and the fact that $I_R \subseteq Y$, it follows that there is a complete and consistent set Z of ground non-constraint literals over Σ such that $Y \subseteq Z$, $I_D \subseteq Z$, and Z^+ is an input answer set of Π_D w.r.t. $D(\Pi)$. By Proposition 6, Z is a model of

$$SM_{D(\Pi)}[ground^*(\Pi_D)]. \tag{27}$$

From the fact that Y^+ is an answer set of $[\Pi, M]_R$ and Proposition 12, it follows that Y is a model of

$$SM_{R(\Pi)}[\ [\Pi, M]_R\ ]. \tag{28}$$

Since $Y \subseteq Z$, Z is a model of (28) also.

From (27) and (28) we derive that Z is a model of (26). Consequently Π has an answer set. □

We now establish the relation between answer sets of Π^ξ and $\Pi(Q)$. For a set M of atoms over Σ, by M^ξ we denote a set of atoms over Σ^ξ by replacing each constraint and defined literal A occurring in M with a corresponding name A^ξ. For instance, $\{T \neq 1,\ acceptTime(T)\}^\xi$ is

$$\{T \neq 1^\xi,\ acceptTime(T)^\xi\}.$$

Lemma 2. *For a weakly-simple* AC *program* Π*, a query* Q *based on* Π*, and a set* X *of atoms over* $At(\Pi)_R$ *so that* $Q_R^+ \subseteq X$*,* X *is an answer set of* $\Pi(Q)$ *iff* $X \cup (Q_{D,C}^+)^\xi$ *is an answer set of* Π^ξ.

Proof. By $\Pi(Q)'$ we denote a program constructed from $\Pi(Q)$ by dropping the constraints in $\Pi(Q)$ derived from the condition 5. It is easy to see that set $(Q^+_{D,C})^\xi \cup (Q^-_{D,C})^\xi$ is a splitting set of Π^ξ. The bottom part of Π^ξ (relevant to this splitting set) consists of choice rules for each atom occurring in $(Q_{D,C})^\xi$. It follows that $(Q^+_{D,C})^\xi$ is an answer set of the bottom. Let U denote $(Q^+_{D,C})^\xi \cup (Q^-_{D,C})^\xi$. Note that $e_U(\Pi^\xi \setminus b_U(\Pi^\xi), (Q^+_{D,C})^\xi)$ coincides with $\Pi(Q)'$.

Left-to-right: Let X be an answer set of $\Pi(Q)$. By the Theorem on Constraints [24] it follows that X is also an answer set of the program $\Pi(Q)'$. By Proposition 11, $X \cup (Q^+_{D,C})^\xi$ is an answer set of Π^ξ.

Right-to-left: Let $X \cup (Q^+_{D,C})^\xi$ be an answer set of Π^ξ. By Proposition 11, it follows that X is an answer set of $\Pi(Q)'$. From the fact that $Q^+_R \subseteq X$ the Theorem on Constraints [24] it follows that X is also an answer set for $\Pi(Q)$. □

Proposition 7. *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.*

Proof. Part (a) is proved as in the proof of Proposition 1 in [19].
(b) Let M be a terminal state. Recall that $\textsc{sm}_{\Pi^\xi}$ is a subgraph of AC_Π. From Proposition 5, it follows that M^+ is an answer set of Π^ξ. It is obvious that $query(M)$ forms a query based on Π, and $query(M)^+_R \subseteq (M^{\xi-})^+_R$. By Lemma 2, it follows that $(M^{\xi-})^+_R$ is an answer set of $\Pi(query(M))$. Furthermore, since *Query Propagate* is not applicable we conclude that

$$Cons(\Pi_D, query(M)) \subseteq M^{\xi-}$$

and therefore it is different from $\{\bot\}$. Consequently, $query(M)$ is a satisfiable query. By Lemma 1, $(M^{\xi-})^+_R$ is a set of all regular atoms in some answer set of Π.
(c) Left-to-right: Since $\bot$ is reachable from $\emptyset$, there is an inconsistent state M without decision literals such that there exists a path from $\emptyset$ to M and M has the form:

Case 1. M is of the form $l_1\ \dots l_n \perp \dots$. From Lemma 5 in [19], it follows that any answer set of Π^ξ satisfies $l_1\ \dots l_n$. On the other hand, $\perp$ appears in M due to the application of the transition rule *Query Propagate* so that $Cons(\Pi_D, query(l_1\ \dots l_n)) = \{\perp\}$. In other words there exists no satisfying extension of $query(l_1\ \dots l_n)$ w.r.t. Π_D. From Lemmas 2 and 1, it follows that Π has no answer sets.

Case 2. M is of the form $l_1\ \dots\ l_n$ where each l_i is an atom. From Lemma 5 in [19], it follows that any answer set of Π^ξ satisfies $l_1\ \dots l_n$. Since $l_1\ \dots\ l_n$ is inconsistent we conclude that Π^ξ has no answer sets. From Lemmas 2 and 1, it follows that Π has no answer sets.

Right-to-left: From (a) it follows that there is a path from $\emptyset$ to some terminal state. By (b), this state cannot be different from $\perp$, because Π has no answer sets. □

Proposition 14. *[32, Proposition 5] Let* Π *be a regular program. For every* Π*-safe [32] set* F *of clauses, a set* X *of atoms is an answer set of* Π *if and only if* $X = M^+ \cap At(\Pi)$*, for some model* M *of* $[F, \Pi]$.

Proposition 15. *[32, Proposition 7] For any SM(ASP) theory* $[F, \Pi]$,

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

(b) for any terminal state M *of* $\text{SM(ASP)}_{F,\Pi}$ *other than* $\perp$, M *is a model of* $[F, \Pi]$,

(c) state $\perp$ *is reachable from* $\emptyset$ *in* $\text{SM(ASP)}_{F,\Pi}$ *if and only if* $[F, \Pi]$ *has no models.*

Proposition 8. *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* $\perp$, $(M^{\xi-})^+_R \cap At(\Pi)$ *is a set of all regular atoms in some answer set of* Π,

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

Proof. The proof of this proposition follows the lines of the proof of Proposition 7 relying on Propositions 14 and 15. □

Proposition 9. *For a friendly* AC *program* Π*, the graphs* AC^-_Π *and* $\textsc{con}^-_{Comp(\Pi^\xi),\Pi}$ *are equal.*

Proof. The proof of this proposition immediately follows Proposition 8 in [32] and the fact that graphs

$$AC^-_\Pi \text{ and } \textsc{con}^-_{Comp(\Pi),\Pi}$$

differ from

$$\textsc{sm}^-_\Pi \text{ and } \textsc{sm}(\textsc{asp})^-_{Comp(\Pi),\Pi}$$

graphs, respectively, by the same transition rule *Query Propagate.* □

Proof of Proposition 10 follows the lines of the proof of Proposition 7, Proposition 14, and Proposition 9 in [32].

References

[1] V. S. Mellarkod, M. Gelfond, Y. Zhang, Integrating answer set programming and constraint logic programming, Annals of Mathematics and Artificial Intelligence 53 (2008) 251–287.

[2] M. Gebser, M. Ostrowski, T. Schaub, Constraint answer set solving, in: Proceedings of 25th International Conference on Logic Programming (ICLP), Springer, 2009, pp. 235–249.

[3] M. Ostrowski, T. Schaub, Asp modulo csp: The clingcon system, Theory and Practice of Logic programming (TPLP) 12 (2012) 485–503.

[4] M. Balduccini, Representing constraint satisfaction problems in answer set programming, in: Proceedings of ICLP Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP), `https://www.mat.unical.it/ASPOCP09/`, 2009.

[5] J. Wittocx, M. Mariën, M. Denecker, The IDP system: a model expansion system for an extension of classical logic, in: Proceedings of Workshop on Logic and Search, Computation of Structures from Declarative Descriptions (LaSh), electronic, 2008, pp. 153–165. Available at `https://lirias.kuleuven.be/bitstream/123456789/229814/1/lash08.pdf`.

[6] I. Elkabani, E. Pontelli, T. C. Son, Smodels with clp and its applications: A simple and effective approach to aggregates in asp, in: B. Demoen, V. Lifschitz (Eds.), ICLP, volume 3132 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 73–89.

[7] S. Baselice, P. A. Bonatti, M. Gelfond, Towards an integration of answer set and constraint solving, in: M. Gabbrielli, G. Gupta (Eds.), ICLP, volume 3668 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 52–66.

[8] C. Drescher, T. Walsh, Translation-based constraint answer set solving, in: Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI), AAAI Press, 2011, pp. 2596–2601.

[9] C. Drescher, T. Walsh, A translational approach to constraint answer set solving, Theory and Practice of Logic programming (TPLP) 10 (2011) 465–480.

[10] G. Liu, T. Janhunen, I. Niemelä, Answer set programming via mixed integer programming, in: Principles of Knowledge Representation and Reasoning: Proceedings of the 13th International Conference, AAAI Press, 2012, pp. 32–42.

[11] I. Niemelä, P. Simons, Extending the Smodels system with cardinality and weight constraints, in: J. Minker (Ed.), Logic-Based Artificial Intelligence, Kluwer, 2000, pp. 491–521.

[12] M. Gebser, B. Kaufmann, A. Neumann, T. Schaub, Conflict-driven answer set solving, in: Proceedings of 20th International Joint Conference on Artificial Intelligence (IJCAI'07), MIT Press, 2007, pp. 386–392.

[13] C. Schulte, P. J. Stuckey, Efficient constraint propagation engines, Transactions on Programming Languages and Systems (2008).

[14] M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, Communications of the ACM 5(7) (1962) 394–397.

[15] R. Nieuwenhuis, A. Oliveras, C. Tinelli, Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T), Journal of the ACM 53(6) (2006) 937–977.

[16] T. Eiter, G. Brewka, M. Dao-Tran, M. Fink, G. Ianni, T. Krennwallner, Combining Nonmonotonic Knowledge Bases with External Sources, in: S. Ghilardi, R. Sebastiani (Eds.), 7th International Symposium on Frontiers of Combining Systems (FroCos 2009), volume 5749 of *LNAI*, Springer, 2009, pp. 18–42.

[17] T. Eiter, G. Ianni, R. Schindlauer, H. Tompits, A uniform integration of higher-order reasoning and external evaluations in answer set programming, in: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), Professional Book Center, 2005, pp. 90–96.

[18] G. Brewka, T. Eiter, Equilibria in heterogeneous nonmonotonic multi-context systems, in: Proceedings of National conference on Artificial Intelligence (AAAI), AAAI Press, 2007, pp. 385–390.

[19] Y. Lierler, Abstract answer set solvers, in: Proceedings of International Conference on Logic Programming (ICLP), Springer, 2008, pp. 377–391.

[20] Y. Lierler, Abstract answer set solvers with backjumping and learning, Theory and Practice of Logic Programming 11 (2011) 135–169.

[21] Y. Lierler, Y. Zhang, A transition system for AC language algorithms, in: Proceedings of ICLP Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP), `http://www.dbai.tuwien.ac.at/proj/aspocp11/accepted.html`, 2011.

[22] Y. Lierler, On the relation of constraint answer set programming languages and algorithms, in: Proceedings of the AAAI Conference on Artificial Intelligence, MIT Press, 2012.

[23] A. Kakas, R. Kowalski, F. Toni, Abductive logic programming, Journal of Logic and Computation 2 (1992) 719–770.

[24] V. Lifschitz, L. R. Tang, H. Turner, Nested expressions in logic programs, Annals of Mathematics and Artificial Intelligence 25 (1999) 369–389.

[25] P. Ferraris, V. Lifschitz, Weight constraints as nested expressions, Theory and Practice of Logic Programming 5 (2005) 45–74.

[26] P. Ferraris, J. Lee, V. Lifschitz, R. Palla, Symmetric splitting in the general theory of stable models, in: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), IJCAI press, 2009, pp. 797–803.

[27] P. Simons, Extending the stable model semantics with more expressive rules, in: Logic Programming and Non-monotonic Reasoning: Proceedings Fifth Int'l Conf. (Lecture Notes in Artificial Intelligence 1730), Springer, 1999, pp. 305–316.

[28] A. Van Gelder, K. Ross, J. Schlipf, The well-founded semantics for general logic programs, Journal of ACM 38 (1991) 620–650.

[29] J. Lee, A model-theoretic counterpart of loop formulas, in: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), Professional Book Center, 2005, pp. 503–508.

[30] P. Ferraris, J. Lee, V. Lifschitz, Stable models and circumscription, Artificial Intelligence 175 (2011) 236–263.

[31] K. Clark, Negation as failure, in: H. Gallaire, J. Minker (Eds.), Logic and Data Bases, Plenum Press, New York, 1978, pp. 293–322.

[32] Y. Lierler, M. Truszczynski, 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 (2011) 629–646.

[33] M. Balduccini, Y. Lierler, P. Schueller, Prolog and asp inference under one roof, in: Proceedings of 12th International Conference on Logic Programming and Nonmonotonic Reasoning, Springer, 2013, pp. 148–160.

[34] M. Mariën, J. Wittocx, M. Denecker, M. Bruynooghe, SAT(ID): Satisfiability of propositional logic extended with inductive definitions, in: Theory and Applications of Satisfiability Testing, 11th International Conference (SAT), Springer, 2008, pp. 211–224.

[35] S. T. Erdoğan, V. Lifschitz, Definitions in answer set programming, in: V. Lifschitz, I. Niemelä (Eds.), Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), Springer, 2004, pp. 114–126.