Proceedings of the 22nd International Conference on Principles of Knowledge Representation and Reasoning
KR in the Wild

Model Checker For Recursive Aggregates

Mario Alviano , Carmine Dodaro , Salvatore Fiorentino
University of Calabria, Italy
{mario.alviano, carmine.dodaro, salvatore.fiorentino}@unical.it

Abstract

Model checking for disjunctive logic programs is a co-NP-complete task for which two main state-of-the-art approaches exist: one based on the unsatisfiability of SAT formulas derived from program reducts, and another based on unfounded sets. Although both are effective and efficient, their original formulations do not support recursive aggregates. This paper extends previous work by tackling the stability check problem in the presence of such aggregates. We generalize the reduct-based approach to operate over pseudo-Boolean theories rather than SAT encodings, yielding more compact and efficient formulations. In parallel, we extend the unfounded-set-based approach to incorporate aggregates, integrating both strategies as propagators within the ASP solver CLINGO. Additionally, we introduce partial stability checks to enable incremental or approximate verification of model stability. Our empirical evaluation demonstrates that these novel strategies not only preserve correctness but also substantially improve the efficiency of model checking for disjunctive programs with aggregates.

1 Introduction

Answer Set Programming (ASP) is a well-known declarative programming paradigm (Baral 2010; Eiter, Gottlob, and Mannila 1997; Gelfond and Lifschitz 1991). The idea of ASP is to represent a given computational problem by means of a logic program whose stable models (or answer sets) correspond to the desired solutions, and then to use an ASP solver to actually compute the stable models. The core language of propositional ASP, which features disjunction in rule heads and nonmonotonic negation in rule bodies, can be used to solve all problems at the second level of the polynomial hierarchy (Eiter, Gottlob, and Mannila 1997). Despite the intrinsic complexity of computing a stable model of a disjunctive program, many efficient disjunctive ASP systems have been developed, and the results of the latest ASP Competition series (Alviano et al. 2013a; Calimeri, Ianni, and Ricca 2014) witness the continuous improvements achieved in this research field. The first ASP solver able to evaluate disjunctive programs was DLV (Leone et al. 2006), followed by GNT (Janhunen and Niemelä 2004) and CMODELS (Lierler and Maratea 2004). At present, CLINGO (Gebser, Kaufmann, and Schaub 2012; Gebser, Kaufmann, and Schaub 2013) sets the standard for state-of-the-art ASP solvers, combining expressive language support with efficient solving techniques.

ASP solvers usually compute stable models of the input program by coupling two modules: the model generator and the model checker. The first generates stable model candidates, whose stability is subsequently verified by the second module. The model generator performs an NP-complete task, usually solved with techniques from SAT solving, such as CDCL (Marques Silva and Sakallah 1999) and clause learning (Zhang et al. 2001). Stable model checking is a co-NP-complete task in the case of disjunctive logic programs (Eiter, Gottlob, and Mannila 1997; Koch, Leone, and Pfeifer 2003). The first implementations of this task were founded on checking the unsatisfiability of a propositional formula derived from the reduct of the input program with respect to the candidate model (Koch, Leone, and Pfeifer 2003; Lierler 2005). Although the construction of such a formula is polynomial (Koch, Leone, and Pfeifer 2003), its practical cost is often higher than the unsatisfiability check itself. Moreover, this traditional reduct-based approach requires rebuilding a different formula for each candidate model, with no opportunity to reuse learned clauses or preprocessing, thus leading to a performance bottleneck. To overcome this, an alternative strategy based on unfounded sets was implemented in CLASP (Gebser, Kaufmann, and Schaub 2013), part of CLINGO (Gebser et al. 2016), which allows the reuse of a single propositional formula for multiple stability checks through literal assumptions (Eén and Sörensson 2003). This approach not only avoids repeated formula construction but also permits clause learning across checks, significantly improving practical performance.

Recent previous work (Alviano et al. 2019) showed that the reduct-based method can be reimagined to exploit literal assumptions as well. This resulted in a novel strategy, implemented in WASP (Alviano et al. 2013b; Alviano et al. 2019), where a single formula can be used throughout the solving process while maintaining the semantic foundations of the reduct-based characterization. Preliminary experiments confirmed that this approach outperforms traditional reduct-based checkers and is already competitive with the unfounded-set-based checker in CLINGO. This paper builds upon that foundation and introduces several new contributions. We extend the reduct-based approach to support programs with recursive aggregates, a common

feature in modern ASP applications. The extension targets pseudo-Boolean encodings instead of pure SAT formulas, which provides a more natural and efficient representation for aggregate expressions. We also generalize the unfounded-set-based strategy to handle aggregates and implement both techniques as propagators in CLINGO, integrating them tightly into the solving workflow. Finally, we introduce the notion of partial stability checks, which enable early and incremental validation of model stability, thereby reducing solver workload. Additionally, we revise the reduct-based approach to eliminate the need for generating a new clause to enforce the subset condition at each check, replacing it with a fixed constraint that works in combination with assumption literals. Empirical results show that these extended strategies provide tangible improvements over the state-of-the-art, demonstrating that both reduct- and unfounded-set-based approaches can effectively scale to richer program classes involving disjunction and aggregates.

2 Preliminaries

Syntax and semantics of Pseudo-Boolean constraints (Aavani 2011) and propositional ASP are briefly introduced in this section. (We refer Alviano, Faber, and Gebser (2015) for details on ASP with aggregates.)

2.1 Pseudo-Boolean Constraints

Syntax. Let $\mathscr{A}$ be a fixed, countable set of (Boolean) variables, or (propositional) atoms, including $\bot$. A *literal* ℓ is either an atom a, or its negation $\neg a$. Let $\top$ be syntactic sugar for $\neg\bot$. A *(Pseudo-Boolean) constraint* is of the form

$$w_1 \cdot \ell_1 + \cdots + w_n \cdot \ell_n \odot k \tag{1}$$

where $n \geq 1$, $w_1, \ldots, w_n, k$ are integers, $\ell_1, \ldots, \ell_n$ are literals, and $\odot \in \{<, \leq, \geq, >, =, \neq\}$. A *cardinality constraint* is a constraint of the form (1) with $w_1 = \cdots = w_n = 1$. A cardinality constraint of the form (1) with $k = 1$ and $\odot$ being $\geq$ is also called *clause* and possibly denoted as $\{\ell_1, \ldots, \ell_n\}$. A *theory* φ is a set of constraints. Abusing of notation, the set of atoms occurring in φ is denoted by $\mathscr{A}(\varphi)$ (or simply $\mathscr{A}$ if φ is clear from the context).

Semantics. An interpretation I is a set of atoms in $\mathscr{A} \setminus \{\bot\}$. Intuitively, atoms in I are true, and those in $\mathscr{A} \setminus I$ are false. Relation $\models$ is inductively defined as follows: for $a \in \mathscr{A}$, $I \models a$ if $a \in I$, and $I \models \neg a$ if $a \notin I$; for a constraint c of the form (1), $I \models c$ if $\left(\sum_{i=1..n:\ I \models \ell_i} w_i\right) \odot k$; for a theory φ, $I \models \varphi$ if $I \models c$ for all $c \in \varphi$. If $I \models \varphi$ then I is a *model* of φ, I *satisfies* φ, and φ is true w.r.t. I. If $I \not\models \varphi$ then I is not a model of φ, I *violates* φ, and φ is false w.r.t. I. Similar for literals, and constraints. A theory φ is *satisfiable* if there is an interpretation I such that $I \models \varphi$; otherwise, φ is *unsatisfiable*.

Example 1. *Consider the following theory* φ*:*

$$\left\{ \begin{array}{l} 4 \cdot \neg a + 2 \cdot b + 3 \cdot c \geq 4, \\ \{b, \neg a\}, \quad \{c, \neg b\}, \quad \{c, \neg d\}, \quad \{d, \neg c\} \end{array} \right\}$$

Theory φ *is satisfiable:* $I = \{a, b, c, d\}$ *is a model of* φ. $\lhd$

2.2 Answer Set Programming

Syntax. Let $\sim$ denote *negation as failure*. A $\sim$*-literal* is either an atom (a positive literal), or an atom preceded by $\sim$ (a negative literal). Let $\top$ be syntactic sugar for $\sim\bot$. An *aggregate literal*, or simply aggregate, is of the form

$$\text{SUM}[w_1 : b_1, \ldots, w_n : b_n] \odot k \tag{2}$$

where $n \geq 0$, $w_1, \ldots, w_n, k$ are integers, $b_1, \ldots, b_n$ are atoms, and $\odot \in \{<, \leq, \geq, >, =, \neq\}$. (Note that $[w_1 : b_1, \ldots, w_n : b_n]$ is a multiset.)

A *literal* is either a $\sim$-literal, or an aggregate. A disjunctive logic program Π is a finite set of rules of the following form:

$$a_1 \mid \cdots \mid a_n \leftarrow \ell_1, \ldots, \ell_m \tag{3}$$

where $n \geq 1$, $m \geq 0$, $a_1, \ldots, a_n$ are atoms in $\mathscr{A}$, and $\ell_1, \ldots, \ell_m$ are literals. For a rule r of the form (3), set $\{a_1 \ldots a_n\}$ is called *head* of r, and denoted $H(r)$; conjunction $\ell_1, \ldots, \ell_m$ is named *body* of r, and denoted $B(r)$; the sets of aggregates, and positive and negative $\sim$ literals in $B(r)$ are denoted $B^{\Sigma}(r)$, $B^{+}(r)$ and $B^{-}(r)$, respectively. A *constraint* is a rule r such that $H(r) = \{\bot\}$. Abusing of notation, the set of atoms occurring in a program Π is denoted by $\mathscr{A}(\Pi)$ (or simply $\mathscr{A}$ if Π is clear from the context).

Semantics. An interpretation I is a set of atoms in $\mathscr{A} \setminus \{\bot\}$. Relation $\models$ is extended as follows: for a negative $\sim$ literal $\sim a$, $I \models \sim a$ if $I \not\models a$; for an aggregate A of the form (2), $I \models A$ if $\left(\sum_{i=1..n:\ I \models b_i} w_i\right) \odot k$; for a conjunction $\ell_1, \ldots, \ell_n$ ($n \geq 0$) of literals, $I \models \ell_1, \ldots, \ell_n$ if $I \models \ell_i$ for all $i \in [1..n]$; for a rule r, $I \models r$ if $H(r) \cap I \neq \emptyset$ whenever $I \models B(r)$; for a program Π, $I \models \Pi$ if $I \models r$ for all $r \in \Pi$. The definition of stable model is based on a notion of program reduct: Let Π be a disjunctive logic program, and I an interpretation. The reduct of Π w.r.t. I, denoted Π^I, is obtained from Π by deleting each rule r such that $I \not\models B(r)$, and fixing the interpretation of negative literals by replacing them with $\top$ or $\bot$ depending on whether they are respectively true or false w.r.t. I. An interpretation I is a stable model of Π if $I \models \Pi^I$, and there is no $J \subset I$ such that $J \models \Pi^I$. Let $SM(\Pi)$ denote the set of stable models of Π. Program Π is *coherent* if $SM(\Pi) \neq \emptyset$; otherwise, Π is *incoherent*.

Example 2. *Let us consider program* Π_1 *by Alviano, Faber, and Gebser (2015), addressing the* Generalized Subset Sum *problem for* $\exists x_1 x_2 \forall y_1 y_2 (1 \cdot x_1 + 2 \cdot x_2 + 2 \cdot y_1 + 3 \cdot y_2 \neq 5)$*:*

$$\begin{array}{l} x_1 \mid nx_1 \quad x_2 \mid nx_2 \quad y_1 \leftarrow un \quad y_2 \leftarrow un \quad \bot \leftarrow \sim un \\ un \leftarrow \text{SUM}[1 : x_1, 2 : x_2, 2 : y_1, 3 : y_2] \neq 5 \end{array}$$

The idea is to enforce truth of y_1, y_2 *due to* un*, whose stability depends on the universal truth of the inequality (once* x_1, x_2 *are fixed).* $I_1 = \{x_1, nx_2, y_1, y_2, un\}$ *is a stable model of* Π_1*. Indeed, no* $J \subset I_1$ *is a model of the reduct* $\Pi_1^{I_1}$*:*

$$\begin{array}{l} x_1 \mid nx_1 \quad x_2 \mid nx_2 \quad y_1 \leftarrow u \quad y_2 \leftarrow u \\ un \leftarrow \text{SUM}[1 : x_1, 2 : x_2, 2 : y_1, 3 : y_2] \neq 5 \end{array} \quad \lhd$$

In the following we assume that all aggregate literals of the form (2) are normalized to have $\odot$ fixed to $\geq$ by using the strong equivalences (A)–(D) by Alviano, Faber, and Gebser (2015).

Example 3. *The (normalized) program* Π_N

$$
\begin{array}{ll}
x_1 \mid nx_1 & x_2 \mid nx_2 \quad y_1 \leftarrow un \quad y_2 \leftarrow un \quad \bot \leftarrow \sim un \\
un \leftarrow s & s \leftarrow \text{SUM}[1:x_1, 2:x_2, 2:y_1, 3:y_2] \geq 6 \\
ns \leftarrow \sim s & s \leftarrow \text{SUM}[1:x_1^F, 2:x_2^F, 2:y_1^F, 3:y_2^F] \geq 4 \\
p \mid p^F \leftarrow \sim ns & \forall p \in \{x_1, x_2, y_1, y_2\} \\
p^F \leftarrow \sim p & \forall p \in \{x_1, x_2, y_1, y_2\} \\
p^F \leftarrow s & \forall p \in \{x_1, x_2, y_1, y_2\}
\end{array}
$$

(obtained from Π_1 *in Example 2) has stable model* $I_N = I_1 \cup \{s, x_1^F, x_2^F, y_1^F, y_2^F\}$. *Indeed, in the reduct* $\Pi_N^{I_N}$ *atoms* x_1, x_2^F *are forced to be true (and consequently* nx_1, x_2 *are false). Hence, the two aggregates are true (and s would be enforced) if* $2 \cdot y_1 + 3 \cdot y_2 \geq 5$ *or* $2 \cdot y_1^F + 3 \cdot y_2^F \geq 4$*, respectively. Given that at least one of* y_1 *and* y_1^F*, and at least one of* y_2 *and* y_2^F *must be true , we have that at least one of the aggregates (and s) must be true.Given that s is true then all literals* $x_1^F, x_2^F, x_3^F, x_4^F$ *and* y_1, y_2, un *are forced to be true.* ◁

2.3 Aggregate-free Stability Check

The stability check problem can be stated as follows:

> Given an ASP program Π and an interpretation I, determine whether I is a stable model of Π.

In the aggregate-free case, the problem was addressed by encoding the program reduct Π^I as a propositional theory $basic(\Pi, I)$ that also excludes I and its supersets.

Definition 1 (Basic Reduct-Based Stability Check; Koch, Leone, and Pfeifer (2003))**.** *Let* Π *be a program with no aggregates, and I be an interpretation. The theory* $basic(\Pi, I)$ *is defined as follows:*

$$basic(\Pi, I) := reduct(\Pi, I) \cup \{subsetClause(I)\}; \quad (4)$$

$$reduct(\Pi, I) := \{reduct(r, I) \mid r \in \Pi^I\}; \quad (5)$$

$$reduct(r, I) := (H(r) \cap I) \cup \{\neg b \mid b \in B^+(r)\}; \quad (6)$$

$$subsetClause(I) := \{\neg a \mid a \in I\}. \quad (7)$$

The reduct-based stability check can be improved by making most of the generated clauses independent from the candidate model I.

Definition 2 (Advanced Reduct-Based Stab. Check; Alviano et al. (2019))**.** *Let* Π *be a program with no aggregates, and I be an interpretation. The theory* $adv(\Pi, I)$ *is defined as*

$$adv(\Pi, I) := reduct(\Pi) \cup fix(I) \cup \{subsetClause(I)\}; \quad (8)$$

$$reduct(\Pi) := \{reduct(r) \mid r \in \Pi\}; \quad (9)$$

$$reduct(r) := H(r) \cup \{\neg b \mid b \in B^+(r)\} \cup \{b' \mid b \in B^-(r)\}; \quad (10)$$

$$fix(I) := \{\{\neg a\} \mid a \in \mathscr{A} \setminus I\} \cup \{\{b'\} \mid b \in I\} \cup \{\{\neg b'\} \mid b \in \mathscr{A} \setminus I\}; Ma \quad (11)$$

where each b' *is a fresh atom (associated with b), i.e., an atom not occurring in* Π.

In (10), negative literals are encoded using fresh atoms that are fixed by (11) according to the interpretation I, as required by the definition of program reduct Π^I. Note that simplifying $reduct(\Pi)$ by means of the unary clauses in $fix(I)$ would result in the theory $reduct(\Pi, I)$.

Proposition 1 (Theorem 4.2 of Koch, Leone, and Pfeifer (2003) and Theorem 1 of Alviano et al. (2019))**.** *Let* Π *be a program with no aggregates, and I be an interpretation. The following statements are equivalent: 1)* $I \in SM(\Pi)$*; 2)* $I \models \Pi$*, and* $basic(\Pi, I)$ *is unsatisfiable; 3)* $I \models \Pi$*, and* $adv(\Pi, I)$ *is unsatisfiable.*

Stable models can be also characterized in terms of *unfounded sets*, yielding an alternative stability check. A nonempty set X of atoms is an unfounded set for Π with respect to I if each $r \in \Pi$ with $H(r) \cap X \neq \emptyset$ satisfies $I \not\models B(r)$, or $B^+(r) \cap X \neq \emptyset$, or $(H(r) \setminus X) \cap I \neq \emptyset$. I is a stable model of Π if and only if $I \models \Pi$ and no unfounded set X is such that $X \cap I \neq \emptyset$.

Definition 3 (Quadratic Unfounded-Based Stability Check; Gebser, Kaufmann, and Schaub (2013))**.** *Let* Π *be a program with no aggregates, and I be an interpretation. The theory* $unf^2(\Pi, I)$ *is defined as follows:*

$$unf^2(\Pi, I) := \mathscr{U}(\Pi) \cup \mathscr{H}(\Pi) \cup \{c(\Pi)\} \cup fix'(I) \quad (12)$$

$$\mathscr{U}(\Pi) := \{\mathscr{U}(r, a) \mid r \in \Pi, a \in H(r)\}; \quad (13)$$

$$\mathscr{U}(r, a) := \{\neg u_a\} \cup \{\neg b \mid b \in B^+(r)\} \cup \{b \mid b \in B^-(r)\} \cup \{u_b \mid b \in B^+(r)\} \cup \{h_b \mid b \in H(r) \setminus \{a\}\} \quad (14)$$

$$\mathscr{H}(\Pi) := \bigcup_{b \in \mathscr{A}} \mathscr{H}(b); \quad (15)$$

$$\mathscr{H}(b) := \{\{\neg h_b, b\}, \{\neg h_b, \neg u_b\}, \{h_b, \neg b, u_b\}\}; \quad (16)$$

$$c(\Pi) := \{u_b \mid b \in \mathscr{A}\} \quad (17)$$

$$fix'(I) := \{\{a\} \mid a \in I\} \cup \{\{\neg u_b\} \mid b \notin I\} \cup \{\{\neg b\} \mid b \notin I\} \quad (18)$$

where each u_b *and* h_b *is fresh and associated with* $b \in \mathscr{A}$.

Intuitively, u_b is true if and only if b is not false and unfounded, and h_b is true if and only if b is true and founded. Thus, (14) encodes the unfounded condition for rule r and its head atom a. The unfounded-based stability check can be improved by making the size of the checking formula linear with respect to the size of Π.

Definition 4 (Linear Unfounded-Based Stability Check; Alviano et al. (2019))**.** *Let* Π *be a program with no aggregates, and I be an interpretation. The theory* $unf^1(\Pi, I)$ *is defined as follows:*

$$unf^1(\Pi, I) := \mathscr{U}'(\Pi) \cup \mathscr{H}(\Pi) \cup \{c(\Pi)\} \cup fix'(I) \quad (19)$$

$$\mathscr{U}'(\Pi) := \{\mathscr{U}'(r) \mid r \in \Pi\} \cup \{\mathscr{U}'(r, a) \mid r \in \Pi, a \in H(r)\}; \quad (20)$$

$$\mathscr{U}'(r) := \{\neg u_r\} \cup \{\neg b \mid b \in B^+(r)\} \cup \{b \mid b \in B^-(r)\} \cup \{u_b \mid b \in B^+(r)\} \cup \{h_a \mid a \in H(r)\}; \quad (21)$$

$$\mathscr{U}'(r, a) := \{\neg u_a, u_r\} \quad (22)$$

where each u_r *is a fresh atom (associated with r).*

Intuitively, u_r is true if and only if some of the non-false head atoms of r are unfounded. Thus, (21) compactly encodes the unfounded condition for all non-false head atoms

of r by exploiting the fact that h_a and u_a cannot be both true due to (16).

Proposition 2 (Theorem 3 of Gebser, Kaufmann, and Schaub (2013) and Theorem 2 of Alviano et al. (2019))**.** *Let Π be a program with no aggregates, and I be an interpretation. The following statements are equivalent: 1) $I \in SM(\Pi)$; 2) $I \models \Pi$, and $unf^2(\Pi, I)$ is unsatisfiable; 3) $I \models \Pi$, and $unf^1(\Pi, I)$ is unsatisfiable.*

3 Generalized Stability Check

Definitions for both unfounded and reduct based check are extendend in this section.

3.1 Reduct-Based Stability Check

We extend Definitions 1–2 to programs with aggregates. To this aim, we have to encode (normalized) aggregates in the theories $basic(\Pi, I)$ and $adv(\Pi, I)$. In particular, equations (5)–(6) and (9)–(10) must be revised accordingly.

Definition 5 (Generalized Basic Reduct-Based Stability Check)**.** *Let Π be a program with normalized aggregates $A_1, \ldots, A_m$ ($m \geq 0$), and I be an interpretation. The theory $basic^*(\Pi, I)$ is defined as follows:*

$$basic^*(\Pi, I) := reduct^*(\Pi, I) \cup \{subsetClause(I)\}; \quad (23)$$

$$reduct^*(\Pi, I) := \{reduct^*(r, I) \mid r \in \Pi^I\} \cup \{reduct^*(A_i, I) \mid i \in [1..m]\}; \quad (24)$$

$$reduct^*(r, I) := reduct(r, I) \cup \{\neg aggr_i \mid i \in [1..m],\ A_i \in B^{\Sigma}(r)\}; \quad (25)$$

$$reduct^*(A_i, I) := \left\{ \begin{array}{l} k \cdot \neg aggr_i + w_1 b_1 + \cdots + w_n b_n \geq k \\ -\omega \cdot aggr_i + w_1 b_1 + \cdots + w_n b_n < k \end{array} \right\}; \quad (26)$$

where each $aggr_i$ is a fresh atom (associated with A_i), A_i has the form (2) *with $\odot$ being $\geq$, $\omega = |k| + 1 + \sum_{i=1}^{n} |w_i|$.*

Intuitively, each aggregate literal A_i is replaced by the fresh atom $aggr_i$ in the clause (25), and $aggr_i$ is made equisatisfiable to the aggregate by (26). In fact, note that exactly one of the two pseudo-Boolean constraints in (26) is satisfied by $aggr_i$ alone, and the other constraint enforces the semantics of $aggr_i$. The next step is to make most of the theory independent from I.

Definition 6 (Generalized Advanced Reduct-Based Stability Check)**.** *Let Π be a program with normalized aggregates $A_1, \ldots, A_m$ ($m \geq 0$), and I be an interpretation. The theory $adv^*(\Pi, I)$ is defined as follows:*

$$adv^*(\Pi, I) := reduct^*(\Pi) \cup fix(I) \cup \{subsetClause(I)\}; \quad (27)$$

$$reduct^*(\Pi) := \{reduct^*(r) \mid r \in \Pi\} \cup \{reduct^*(A_i) \mid i \in [1..m]\}; \quad (28)$$

$$reduct^*(r) := reduct(r) \cup \{\neg aggr_i \mid i \in [1..m],\ A_i \in B^{\Sigma}(r)\}; \quad (29)$$

$$reduct^*(A_i) := \left\{ \begin{array}{l} k \cdot \neg aggr_i + w_1 b_1 + \cdots + w_n b_n \geq k \\ -\omega \cdot aggr_i + w_1 b_1 + \cdots + w_n b_n < k \end{array} \right\}; \quad (30)$$

where each $aggr_i$ is a fresh atom (associated with A_i), A_i has the form (2) *with $\odot$ being $\geq$, and $\omega = |k| + 1 + \sum_{i=1}^{n} |w_i|$.*

Simplifying $reduct^*(\Pi)$ by means of the unary clauses in $fix(I)$ would result in the theory $reduct^*(\Pi, I)$.

Theorem 1. *Let Π be a program with normalized aggregates $A_1, \ldots, A_m$ ($m \geq 0$), and I be an interpretation. The following statements are equivalent: 1) $I \in SM(\Pi)$; 2) $I \models \Pi$, and $basic^*(\Pi, I)$ is unsatisfiable; 3) $I \models \Pi$, and $adv^*(\Pi, I)$ is unsatisfiable.*

Proof. Let $J|_{\Pi}$ (resp. $J|_{\varphi}$) be $J \cap \mathscr{A}(\Pi)$ (resp. $J \cap \mathscr{A}(\varphi)$). For any $J \subseteq \mathscr{A}(\Pi)$, let $ext(I, J)$ be $J \cup \{aggr_i \mid i \in [1..m],\ J \models A_i\} \cup \{b' \mid b \in I\} \cup \{\neg b' \mid b \in \mathscr{A} \setminus I\}$. For every $i \in [1..m]$, we have that (i) $J \subseteq \mathscr{A}(basic^*(\Pi, I))$ and $J \models reduct^*(A_i, I)$ imply $aggr_i \in J$ if and only if $J \models A_i$; (ii) $J \subseteq \mathscr{A}(adv^*(\Pi, I))$ and $J \models reduct^*(A_i)$ imply $aggr_i \in J$ if and only if $J \models A_i$.
$(1 \Rightarrow 2)$ $I \models \Pi$ is immediate by the definition of stable model. We shall show that no $J \subseteq \mathscr{A}(basic^*(\Pi, I))$ is such that $J \models basic^*(\Pi, I)$. If $J|_{\Pi} \not\subset I$, $J \not\models subsetClause(I)$, and we are done. If $J|_{\Pi} \subset I$, from the definition of stable model we have $J|_{\Pi} \not\models \Pi^I$, i.e., there is a rule $r^I \in \Pi^I$ obtained from $r \in \Pi$ such that $J \not\models r^I$. Hence, $I \models B(r)$, $H(r^I) \cap J = \emptyset$, $J \models B^+(r^I) \cup B^{\Sigma}(r^I)$. Combining with (i), we have $J \not\models reduct^*(A_i, I)$ or $J \not\models reduct^*(r^I, I)$, and we are done.
$(2 \Rightarrow 3)$ If $J \models basic^*(\Pi, I)$, then $ext(I, J|_{\Pi}) \models adv^*(\Pi, I)$. Indeed, $ext(I, J|_{\Pi}) \models fix(I) \cup \{subsetClause(I)\}$ by construction, and $ext(I, J|_{\Pi}) \models reduct^*(\Pi)$ because $reduct^*(\Pi)$ simplified by means of the unary clauses in $fix(I)$ is $reduct^*(\Pi, I)$.
$(3 \Rightarrow 1)$ For every $J \subset I$, $ext(I, J) \not\models adv^*(\Pi, I)$ by assumption. By construction, $ext(I, J) \models fix(I) \cup \{subsetClause(I)\}$ and $ext(I, J) \models reduct^*(A_i)$ for all $i \in [1..m]$, so there exists $r \in \Pi$ such that $ext(I, J) \not\models reduct^*(r)$; moreover, we can apply (ii) and conclude that $I \models B(r)$, $J \cap H(r) = \emptyset$, and $J \cap B^+(r) \cup B^{\Sigma}(r)$. We conclude that $J \not\models \Pi^I$, and we are done. □

Example 4. *For Π_N of Example 3, $reduct^*(\Pi_N)$ is*

$$\{\{x_1, nx_1\}, \{x_2, nx_2\}, \{y_1, \neg un\}, \{y_2, \neg un\}, \{un, \neg s\}, \{ns, s'\}, \{s, \neg aggr_1\}, \{s, \neg aggr_2\}\}$$
$$\cup \bigcup_{p \in \{x_1, x_2, y_1, y_2\}} \{\{p, p^F, ns'\}, \{p^F, p'\}, \{p^F, \neg s\}\}$$
$$\cup \left\{ \begin{array}{l} 6 \cdot \neg aggr_1 + 1 \cdot x_1 + 2 \cdot x_2 + 2 \cdot y_1 + 3 \cdot y_2 \geq 6 \\ -14 \cdot aggr_1 + 1 \cdot x_1 + 2 \cdot x_2 + 2 \cdot y_1 + 3 \cdot y_2 < 6 \end{array} \right\}$$
$$\cup \left\{ \begin{array}{l} 4 \cdot \neg aggr_2 + 1 \cdot x_1^F + 2 \cdot x_2^F + 2 \cdot y_1^F + 3 \cdot y_2^F \geq 4 \\ -12 \cdot aggr_2 + 1 \cdot x_1^F + 2 \cdot x_2^F + 2 \cdot y_1^F + 3 \cdot y_2^F < 4 \end{array} \right\}$$

Moreover, $subsetClause(I_N) = \{\neg x_1, \neg nx_2, \neg y_1, \neg y_2, \neg un, \neg s, \neg x_1^F, \neg x_2^F, \neg y_1^F, \neg y_2^F\}$, and $fix(I_N)$ comprises singletons for $\{\neg nx_1, \neg x_2, \neg ns\} \cup \{x_1', nx_2', y_1', y_2', un', s', x_1^{F\prime}, x_2^{F\prime}, y_1^{F\prime}, y_2^{F\prime}\} \cup \{\neg nx_1', \neg x_2', \neg ns'\}$. It can be checked that $adv^*(\Pi_N, I_N)$ (i.e., the theory comprising all previous formulas) has no model, and in fact I_N is a stable model of Π_N. For a different interpretation $I' = \{x_1, x_2, y_1, y_2, un, s, x_1^F, x_2^F, y_1^F, y_2^F\}$, we have $subsetClause(I') = \{\neg x_1, \neg x_2, \neg y_1, \neg y_2, \neg un, \neg s, \neg x_1^F, \neg x_2^F, \neg y_1^F, \neg y_2^F\}$, and $fix(I')$ comprises singletons for

$\{\neg nx_1, \neg nx_2, \neg ns\} \cup \{x'_1, x'_2, y'_1, y'_2, un', s', x_1^{F'}, x_2^{F'}, y_1^{F'}, y_2^{F'}\} \cup \{\neg nx'_1, \neg nx'_2, \neg ns'\}$. *It turns out that* $adv^*(\Pi_N, I')$ *has model* $\{x'_1, x'_2, y'_1, y'_2, un', s', x_1^{F'}, x_2^{F'}, y_1^{F'}, y_2^{F'}, x_1, x_2, y_1, y_2^F\}$, *and in fact* I' *is not a stable model of* Π_N. ◁

3.2 Unfounded-Based Stability Check

In this section, we generalize Definition 4 to programs with aggregates. First of all, we adapt the definition of unfounded set by Alviano et al. (2011) (Definition 1) to our notation and to the fragment of disjunctive programs with normalized aggregates. A nonempty set X of atoms is an unfounded set for Π with respect to I if each $r \in \Pi$ with $H(r) \cap X \neq \emptyset$ satisfies $I \not\models B(r)$, or $B^+(r) \cap X \neq \emptyset$, or for an aggregate $A \in B^\Sigma(r)$ of the form (2), $\left(\sum_{i=1..n:\ (I \setminus X) \models b_i} w_i\right) < k$, or $(H(r) \setminus X) \cap I \neq \emptyset$. Then, we have to modify theory $unf^1(\Pi, I)$ to handle (normalized) aggregates.

Definition 7 (Generalized Advanced Unfounded-Based Stability Check)**.** *Let* Π *be a program with normalized aggregates* $A_1, \ldots, A_m$ *(*$m \geq 0$*), and* I *be an interpretation. The theory* $unf^\Sigma(\Pi, I)$ *is defined as follows:*

$$unf^\Sigma(\Pi, I) := \mathscr{U}^*(\Pi) \cup \mathscr{H}(\Pi) \cup \{c(\Pi)\} \cup fix'(I) \quad (31)$$

$$\mathscr{U}^*(\Pi) := \{\mathscr{U}^*(r) \mid r \in \Pi\} \cup \{\mathscr{U}'(r,a) \mid r \in \Pi, a \in H(r)\} \cup \{\mathscr{U}^*(A_i) \mid i \in [1..m]\}; \quad (32)$$

$$\mathscr{U}^*(r) := \mathscr{U}'(r) \cup \{\neg aggr_i \mid i \in [1..m],\ A_i \in B^\Sigma(r)\}; \quad (33)$$

$$\mathscr{U}^*(A_i) := \left\{ \begin{array}{l} k \cdot \neg aggr_i + w_1 h_{b_1} + \cdots + w_n h_{b_n} \geq k \\ -\omega \cdot aggr_i + w_1 h_{b_1} + \cdots + w_n h_{b_n} < k \end{array} \right\}; \quad (34)$$

where each $aggr_i$ *is a fresh atom (associated with* A_i*),* A_i *has the form* (2) *with* $\odot$ *being* $\geq$*, and* $\omega = 1 + |k| + \sum_{i=1}^n |w_i|$.

The idea behind $\mathscr{U}^*(A_i)$ is as follows. On one hand if $aggr_i$ is true we want to ensure that the aggregate is founded; in the first inequality $\neg aggr_i$ disappears leaving the aggregate A_i where positive atoms are replaced with the foundeness atom, that is, the sum of founded atoms and true negative atoms has to be greater or equal than k, hence, A_i is founded. Meanwhile the second inequality is trivially satisfied. On the other hand the same reasoning applies, if $aggr_i$ is false we want to ensure that the aggregate is unfounded.

Unlike before, the first inequality is trivially satisfied, and the second one is simplified to the sum of founded atoms to be less than k, that is, A_i is unfounded.

Theorem 2. *Let* Π *be a program with normalized aggregates* $A_1, \ldots, A_m$ *(*$m \geq 0$*), and* I *be an interpretation. The following statements are equivalent: 1)* $I \in SM(\Pi)$*; 2)* $I \models \Pi$*, and* $unf^\Sigma(\Pi, I)$ *is unsatisfiable.*

Proof. Let $I \models \Pi$. First of all, (16) enforces exactly one of h_b, u_b when $b \in I$ (and $\neg h_b$ when $b \notin I$). Let $J_u \subseteq \mathscr{A}(unf^\Sigma(\Pi, I))$, $X(J_u) = \{a \in \mathscr{A} \mid u_a \in J_u\}$. For any normalized aggregate $A_i := \text{SUM}[w_1 : b_1, \ldots, w_n : b_n] \geq k$ we have (i) $J_u \models \mathscr{U}^*(A_i)$ implies $aggr_i \in J_u$ if and only if $\sum_{i=1..m, h_{b_i} \in J_u} w_i \geq k$.

$(1 \Rightarrow 2)$ We prove the contrapositive. Let J_u be such that $J_u \models unf^\Sigma(\Pi, I)$. Let $J := I \setminus X(J_u)$ (equivalent to $\{b \in \mathscr{A} \mid h_b \in J_u\}$). Since $J_u \models c(\Pi)$ and $J_u \models fix'(I)$, we have that $X(J_u) \neq \emptyset$, and therefore $J \subset I$. We shall show that $J \models \Pi^I$ holds. To this aim, let $r^I \in \Pi^I$ (obtained from $r \in \Pi$ with $I \models B(r)$) be such that $J \models B^+(r)$ and $J \cap H(r) = \emptyset$ (otherwise, it is trivial). In this case, $\emptyset \neq I \cap H(r) \subseteq X(J_u)$. Hence, for any $a \in I \cap H(r)$, we combine $u_a \in J_u$ with $J_u \models \mathscr{U}'(r,a)$ to conclude $u_r \in J_u$. Moreover, since $J_u \models \mathscr{U}^*(r)$, $u_r \in J_u$, $I \models B(r)$ and $J \models B^+(r^I)$, there must be some $A_i \in B^\Sigma(r)$ such that $I \models A_i$ but $aggr_i \notin J_u$ holds. By applying (i) we have $\sum_{i=1..m, h_{b_i} \in J_u} w_i < k$, i.e., $J \not\models A_i$.

$(2 \Rightarrow 1)$ We prove the contrapositive. Let $J \subset I$ be such that $J \models \Pi^I$. Let $X = I \setminus J$ and $J_u = I \cup \{u_a \mid a \in X\} \cup \{h_a \mid a \in J\} \cup \{u_r \mid r \in \Pi, H(r) \cap X \neq 0\} \cup \{u_r \mid r \in \Pi, I \not\models B(r)\} \cup \{aggr_i \mid i \in [1 \ldots m], J \models A_i\}$. By construction $J_u \models \mathscr{H}(\Pi) \cup \{c(\Pi)\} \cup fix'(I)$. It remains to show that $J_u \models \mathscr{U}^*(\Pi)$. Actually, for all $i \in [1..m]$, $J_u \models \mathscr{U}^*(A_i)$ also holds by construction. Also $J_u \models \mathscr{U}^*(r,a)$ is easy to see, for all $r \in \Pi$ and $a \in H(r)$. Similarly, $J_u \models \mathscr{U}^*(r)$ holds for all $r \in \Pi$ with $I \not\models B(r)$: there is $b \in B^+(r) \setminus I$ (and $\neg b \in \mathscr{U}^*(r)$), or $b \in B^-(r) \cap I$ (and $b \in \mathscr{U}^*(r)$), or $A_i \in B^\Sigma(r)$ with $I \not\models A_i$ (and therefore $aggr_i \notin J_u$ because $J \not\models A_i$ due to monotonicity of A_i). We are left to show $J_u \models \mathscr{U}^*(r)$ when $I \models B(r)$. Let us consider the case $u_r \in J_u$, $J \models B^+(r)$ and $H(r) \cap J = \emptyset$ (otherwise, it is trivial). Since $J \models \Pi^I$ by assumption, there must be $A_i \in B^\Sigma(r)$ such that $J \not\models A_i$. In this case, $aggr_i \notin J_u$ by construction, and we are done. □

Example 5 (Continuing Example 4)**.** *Theory* $unf^\Sigma(\Pi_N, \cdot)$ *contains the following constraints associated with* A_1, A_2*:*

$$\begin{array}{r} 6 \cdot \neg aggr_1 + 1 \cdot h_{x_1} + 2 \cdot h_{x_2} + 2 \cdot h_{y_1} + 3 \cdot h_{y_2} \geq 6 \\ -14 \cdot aggr_1 + 1 \cdot h_{x_1} + 2 \cdot h_{x_2} + 2 \cdot h_{y_1} + 3 \cdot h_{y_2} < 6 \\ 4 \cdot \neg aggr_2 + 1 \cdot h^F_{x_1} + 2 \cdot h^F_{x_2} + 2 \cdot h^F_{y_1} + 3 \cdot h^F_{y_2} \geq 4 \\ -12 \cdot aggr_2 + 1 \cdot h^F_{x_1} + 2 \cdot h^F_{x_2} + 2 \cdot h^F_{y_1} + 3 \cdot h^F_{y_2} < 4 \end{array}$$

It can be checked that $unf^\Sigma(\Pi_N, I_N)$ *is unsatisfiable, while* $unf^\Sigma(\Pi_N, I')$ *has model* $I' \cup \{h_{x_1}, h_{x_2}, h_{y_1}, u_{y_2}, u_{un}, u_s, u_{x_1^F}, u_{x_2^F}, u_{y_1^F}, h_{y_2^F}\}$. *Indeed,* $\{y_2$*, un, s,* x_1^F*,* x_2^F*,* $y_1^F\}$ *is an unfounded set for* Π_N *with respect to* I'. ◁

4 Implementation

Modern ASP solvers are based on the *Conflict-Driven Clause Learning* (CDCL) algorithm (Marques Silva and Sakallah 1999; Gebser, Kaufmann, and Schaub 2012), a SAT-inspired technique that incrementally builds a stable model candidate through decisions and propagations, learning from conflicts along the way. The CDCL solver begins by simplifying the program and then selects unassigned atoms, assigning truth values and applying *propagators* to infer further values deterministically. When a complete and consistent assignment is reached, it is checked for stability. If a conflict arises, a *nogood* (conflict clause) is learned to avoid revisiting the same state. If no backtracking is possible, the solver concludes that the program has no stable models. We refer the reader to (Dodaro 2024) for an in-depth treatment of CDCL in ASP. In our work, we build on

this foundation and integrate the proposed stability checks into the CDCL solving process using custom *propagators* in CLINGO (Gebser et al. 2016).

4.1 Stability Checks via Propagators

The stability check is implemented as a *custom propagator* that is invoked when the current assignment is either complete (full check) or sufficiently informative (partial check). Multiple rewriting strategies are supported, and the propagator activates the appropriate variant depending on the selected configuration.

Full Stability Checks. In the baseline (*basic*) rewriting, the checking formula is constructed from scratch at each check based on the input program and the current model candidate. This requires rebuilding the formula repeatedly and is impractical for large-scale problems. More efficient rewritings—such as (8), (13), (20), and the new strategies introduced in this work—enable *reuse* of the checking formula by isolating the model-dependent parts into small sets of *unary clauses*: $\mathit{fix}(I)$ and $\mathit{fix}'(I)$. These clauses can be injected using *assumption literals*, a feature of modern CDCL solvers that avoids reconstructing the theory while enabling clause learning.

Subset Condition Rewriting. In the reduct-based strategy, the condition enforcing that no strict subset of the candidate is also a model is traditionally encoded using a fresh atom per check. We improve this by using a *fixed rewriting*:

$$\begin{aligned}\mathit{subsetClause}(\Pi) := \{\{\neg a_{sub} \mid a \in \mathscr{A}(\Pi)\}\} \cup \\ \{\{a_{sub}, a\} \mid a \in \mathscr{A}(\Pi)\}. \end{aligned} \tag{35}$$

Then, for each candidate I, we simply add unary clauses $\{\{a_{sub}\} \mid a \in \mathscr{A} \setminus I\}$ to $\mathit{fix}(I)$. This design avoids the introduction of fresh atoms and enables full reuse of the underlying theory and learned nogoods.

Use of Strongly Connected Components. To improve performance, stability checks are performed *per component* of the program's positive dependency graph, as suggested by Alviano, Dodaro, and Ricca (2013). For each strongly connected component (SCC), the induced subprogram is extracted and handled independently. This modular treatment reduces unnecessary checks for unaffected components and improves solver performance.

4.2 Partial Stability Checks

We extend the implementation to support *partial stability checks*, which trigger before a complete model has been constructed. These anticipate conflicts early, prune the search space, and accelerate backtracking. The rewritings are adapted to handle *partial assignments* A as shown next, where T, F, and U to denote the sets of true, false, and undefined atoms in A, respectively:

- Reduct-based check (for SCC C):

$$\begin{aligned}\mathit{fix}(A) := \{\{\neg a\} \mid a \in F\} \cup \{\{a\} \mid a \in (U \cap C) \cup (T \setminus C)\} \cup \\ \{\{b'\} \mid b \in T \cap C\} \cup \{\{\neg b'\} \mid b \in (U \cap C) \cup F\} \cup \\ \{\{a_{sub}\} \mid a \in (U \cap C) \cup F\}.\end{aligned}$$

- Unfounded-based check (for SCC C):

$$\begin{aligned}\mathit{fix}'(A) :=& \{\{\neg u_p\} \mid p \in F \cap C\} \cup \{\{\neg p\} \mid p \in F\} \\ & \cup \{\{p\} \mid p \in T\}\end{aligned}$$

Because partial checks can incur non-trivial cost, we introduce two heuristic thresholds. The first one is (i) *pr*: the *changing rate*, i.e., the minimum fraction of atoms that must have changed since the last check. The second one is (ii) *pi*: the *internal threshold*, i.e., the minimum fraction of true atoms in a component required to perform a check. As shown in Section 5.2, optimal results are often achieved when $pi = 0$ for unfounded-based checks.

4.3 Learning from Failed Checks

When a stability check fails, the cause is an *unfounded set* (Leone, Rullo, and Scarcello 1997). Our implementation extracts a nogood that prevents the solver from revisiting the same unstable configuration.

Given a failed atom $p \in X$, we define:

$$R := \{r \mid H(r) \cap X \neq \emptyset,\ B^+(r) \cap X = \emptyset\}$$

For each $r \in R$, we identify the first literal $\ell_r \notin X$ (by syntactic order). For aggregates, we define a minimal set $S \subseteq F$ such that flipping literals outside S does not restore the truth of the aggregate. The resulting nogood is:

$$B(c) = \{p\} \cup \{\ell_r \mid r \in R\} \cup S$$

This prevents the solver from revisiting the same unfounded set and accelerates convergence to stable models.

4.4 Integration in CLINGO

The implementation is built using the C++ API of CLINGO, which supports custom propagators. However, certain limitations exist: (i) The *preprocessed program* is not exposed, so custom propagators must work on the raw input; (ii) CLINGO introduces *auxiliary atoms* for internal rule simplifications and better nogood learning, but these are not accessible from the API. Moreover, to avoid interference with our implementation, we disable built-in unfounded set checking via `--no-ufs-check`. Our own propagators take over stability validation, both full and partial. In Section 5, we describe how we address these integration constraints and evaluate the impact of our design choices. The implementation is publicly available as supplementary material.

5 Experiments

In this section, we compare the performance of the different approaches presented in this paper on a set of publicly-available benchmark instances. In particular, we compare the standard version of CLINGO, which implements a rewriting based on unfounded sets where aggregates are compiled

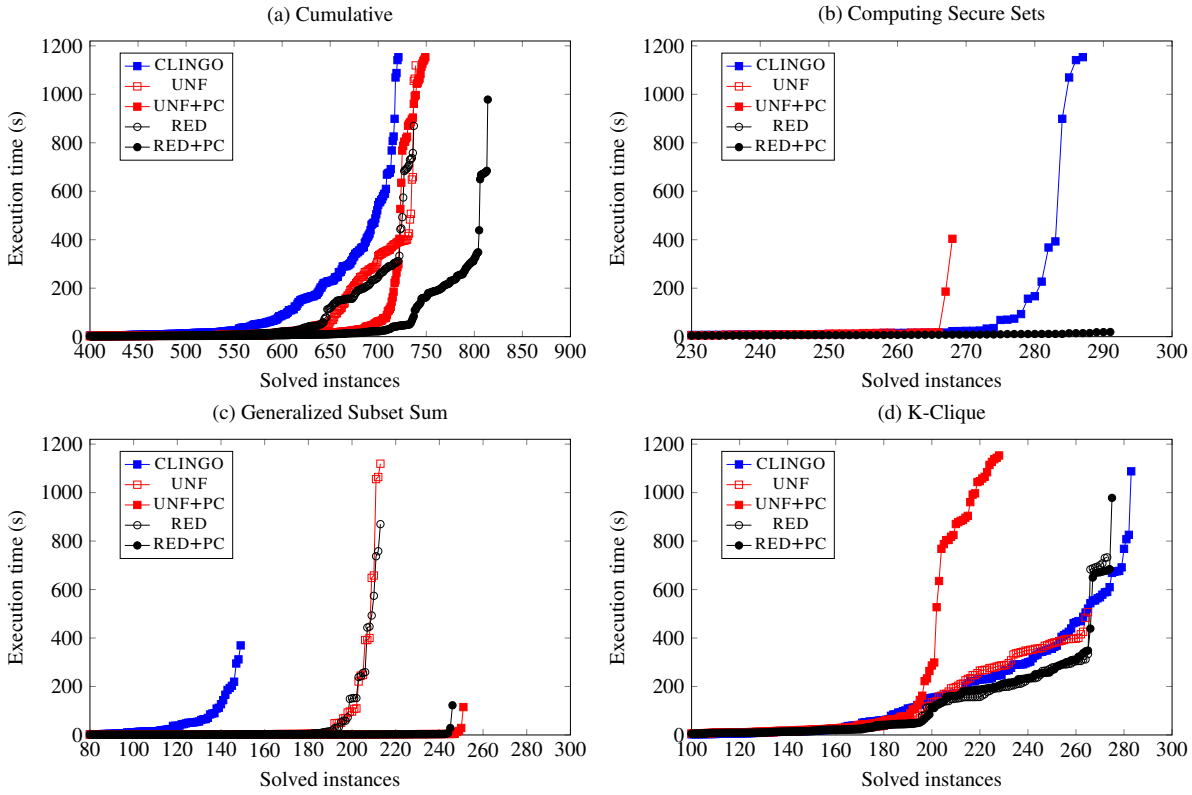

Figure 1: Cactus plots comparing the performance of all configurations overall and across the three benchmarks. Each graph shows the number of instances solved (y-axis) within a given time limit (x-axis). The top-left plot reports cumulative results over all instances, while the others refer to individual benchmarks: CSS (top-right), GSS (bottom-left), and KC (bottom-right).

	CSS		KC	
	Orig	**New**	**Orig**	**New**
CLINGO	288	288	**285**	284
UNF	260	260	262	**266**
UNF+PC	260	260	**196**	195
RED	250	250	262	**274**
RED+PC	260	**295**	196	**197**

Table 1: Number of solved instances for the original (**Orig**) and modified encodings (**New**), where the latter includes auxiliary atoms in rule bodies with two or more literals.

into SAT, against our custom implementations described throughout the paper. We consider the following variants of our approach. UNF: our implementation using the rewriting technique based on unfounded sets without partial checks; UNF+PC: our implementation using the rewriting technique based on unfounded sets with partial checks; RED: our implementation using the rewriting technique based on the reduct without partial checks; RED+PC: our implementation using the rewriting technique based on the reduct with partial checks. As for benchmarks, we consider *Computing Secure Sets* (CSS) Abseher et al. (2020), *Generalized Subset Sum* (GSS) Berman et al. (2002), and *K-Clique* (KC) Marx

	pr	*pi*	CSS	GSS	KC	Total
UNF+PC	0	0.25	260	245	195	700
	0.01	0.30	260	**254**	228	742
	0.025	0.20	260	**254**	**230**	744
	0.05	0.10	260	252	228	740
	0.05	0.25	260	252	228	740
	0.10	0	**269**	252	229	**750**
RED+PC	0	0.25	**295**	243	197	735
	0.01	0.30	292	**247**	**276**	**815**
	0.025	0.20	289	**247**	**276**	812
	0.05	0.10	277	**247**	**276**	800
	0.05	0.25	283	**247**	**276**	806
	0.10	0	279	**247**	**276**	802

Table 2: Effect of parameter configurations on solver performance. Each row reports the number of instances solved for a given setting of changing rate (*pr*) and internal (*pi*) thresholds, across the three benchmarks.

(2011). Each benchmark consists of 300 instances.

The experiments were run on a system with 2.30GHz Intel(R) Xeon(R) Gold 5118 CPU and 512GB of RAM with Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-137-generic x86_64). Time and memory limits were set to 1200 seconds and 8 GB, respectively.

Our experimental analysis follows three main directions. First, we evaluate the impact of missing the auxiliary atoms introduced by CLINGO and discussed in Section 4.4. To simulate their behavior, we manually modified the original encodings by introducing an auxiliary atom in every rule body containing two or more literals. This transformation ensures that, for each rule, either its body consists of a single literal, or its head atom appears in exactly one rule of the program. Second, we investigate the effect of the parameters that control the activation and frequency of partial checks. This analysis allows us to assess how the balance between early pruning and computational overhead influences performance. Finally, we evaluate the performance improvement achieved by our implementations compared to the baseline version of CLINGO.

5.1 Impact of the Auxiliary Atoms

Regarding the first direction of analysis, the results are reported in Table 1. As a first observation, we note that the performance of the baseline CLINGO solver is basically unaffected by the transformation of the encoding. This outcome is expected, as CLINGO internally applies similar simplifications during preprocessing. The minimal differences observed (e.g., a variation of one instance) can be attributed to marginal heuristic effects, such as changes in grounding order or atom naming.

More substantial effects are visible in our implementations. In the case of the CSS benchmark, we observe that the number of solved instances remains unchanged for most configurations, except for RED+PC, which shows a substantial improvement, solving 295 instances compared to 260 with the original encoding. This result confirms that the introduction of auxiliary atoms can enhance performance, especially when combined with partial checks.

For the KC benchmark, the impact of the transformation is more clear. Indeed, even if UNF+PC shows a slight decline (from 196 to 195), all the others benefit from the modified encoding (with the maximum improvment obtained by RED which solves 12 more instances). These results indicate that the effect of the transformation can vary depending on the structural properties of the benchmark instances, but overall, the auxiliary atoms tend to improve the overall performance.

Based on the results discussed above, in the following, we use the original encoding for the baseline CLINGO solver, while for our custom implementations, we employ the modified encoding that includes auxiliary atoms.

5.2 Impact of Parameters for Partial Checks

The second direction of our analysis focuses on evaluating the effect of parameter tuning on the performance of configurations that employ partial checks. Specifically, we study the interaction between two parameters, i.e., the changing rate (pr) and the internal (pi) thresholds. The results are reported in Table 2. We analyze both the configurations UNF+PC and RED+PC. For both, the choice of parameters significantly influences overall performance. In the case of UNF+PC, the best performance is achieved with $pr = 0.10$ and $pi = 0$, leading to a total of 750 solved instances, a 50-instance improvement over the baseline. Interestingly, while the CSS count increases to 269, the GSS and KC benchmarks also experience slight improvements. A similar trend is observed for RED+PC. The best setting is obtained with $pr = 0.01$ and $pi = 0.30$, which results in a total of 815 solved instances, i.e., an improvement of 80 over the baseline setting with $pr = 0$ and $pi = 0.25$. Notably, while performance on the CSS benchmark slightly decreases, the gain on KC is substantial, increasing from 197 to 276 solved instances.

In the following, for UNF+PC, and RED+PC we refer to the configuration with the best performance overall (i.e., 0.10 and 0 for UNF+PC, and 0.01 and 0.30 for RED+PC).

5.3 Overall Performance Comparison

The third direction of our analysis focuses on evaluating the overall performance of the proposed implementations across the three benchmark domains. The results are summarized in Figure 1, where shows a cactus plot for each benchmark, as well as the cumulative one including all the 900 instances.

As first observation, the cumulative results (Figure 1a) confirm the best performance of RED+PC, which solves the highest number of instances (815 out of 900) with the lowest average execution time outperforming all other alternatives. Overall, all customized propagators improve the performance of the plain CLINGO, as UNF, UNF+PC, and RED solve 18, 28, and 16 more instances, respectively.

For the CSS benchmark (Figure 1b), CLINGO and RED+PC obtain the best performance overall, solving 288 and 292 instances, respectively. Moreover, RED+PC is consistently faster, with lower average execution times per instance and a flatter cactus curve, indicating better scalability. Interestingly, it is possible to observe that in this benchmark, partial checks play an important role as RED+PC solves 42 more instances than RED, and UNF+PC solves 9 more instances than UNF.

On the GSS benchmark (Figure 1c), the differences are more pronounced. While the baseline CLINGO solves fewer instances (150), UNF+PC and RED+PC show significant gains in both number of solved instances and average solving time, as they solve 252 and 247 instances, respectively. In particular, UNF+PC emerges as the most efficient configuration being slightly faster than RED+PC. These results further highlight the importance of partial checks, which once again prove to be crucial for achieving better performance.

Finally, on the KC benchmark (Figure 1d), CLINGO outperforms all other approaches solving 284 instances, overall, followed by RED+PC which solves 276 instances. Interestingly, in this benchmark, partial checks do not appear to be effective as, for UNF, they lead to a degradation in performance, and for RED, the improvement is marginal.

5.4 Discussion

Overall, the experimental results demonstrate that the proposed configurations lead to a good improvement upon the state-of-the-art solver CLINGO. Across the tested benchmarks, our novel approaches achieve good performance both in terms of the number of solved instances and execution time. An additional observation emerging from the analysis is the general effectiveness of partial checks. In

most configurations, the inclusion of partial checks leads to substantial performance gains, confirming their usefulness in guiding the search and pruning unstable regions of the search space early. As a further analysis, we evaluated the performance of an idealized strategy that, for each instance, selects the best-performing configuration among those presented in Section 5.2. This *virtual best solver* solves 855 out of 900 instances, corresponding to approximately 95% of the benchmark set. When also including the plain CLINGO solver in the pool of selectable configurations, the total number of solved instances increases to 869, corresponding to 96.5%. These results suggest that a portfolio-based approach (Maratea, Pulina, and Ricca 2014; Hoos, Lindauer, and Schaub 2014), with an automatic configuration selector, could improve robustness by choosing the best solver variant on a per-instance basis.

6 Related Work

Stable model checking is a central task in disjunctive ASP and has been addressed by several strategies since the earliest ASP systems. A classical method is based on the reduct of a disjunctive logic program with respect to a candidate interpretation, and amounts to verifying the unsatisfiability of the resulting theory. This idea, stemming from the Gelfond-Lifschitz semantics (Gelfond and Lifschitz 1991), was concretely implemented in the early ASP solvers DLV (Koch, Leone, and Pfeifer 2003), CMODELS (Lierler and Maratea 2004; Lierler 2005), and the first version of WASP (Alviano et al. 2013b). These systems convert the reduct to a propositional formula and use SAT solving to test for inconsistency.

However, a major drawback of these approaches lies in their inefficiency: a distinct propositional theory must be generated for each candidate model. Given that the number of candidates may grow exponentially, and no clause reuse is possible, the overall performance is limited by the repeated overhead of theory construction. A different strategy was introduced in GNT (Janhunen and Niemelä 2004), an extension of SMODELS (Simons, Niemelä, and Soininen 2002) capable of handling disjunctive rules. GNT avoids theory conversion entirely by performing stable model checks using SMODELS directly, testing coherence of logic programs rather than propositional encodings. However, a new logic program is constructed for each check, hence facing the same limitation of other early ASP solvers.

A major breakthrough came with the ASP solver CLASP (Gebser, Kaufmann, and Schaub 2012; Gebser, Kaufmann, and Schaub 2013), which employs an unfounded-set-based characterization of stable models and avoids reconstructing the program for each check. Instead, it uses literal assumptions (Eén and Sörensson 2003) to inject candidate models into a fixed propositional theory. Unfounded sets are encoded via nogoods, a concept from constraint programming, resulting in a method that enables clause reuse, conflict-driven learning, and stability checks for partial interpretations, making it the *de facto* standard in practical solvers.

Despite its efficiency, the unfounded-set-based approach differs semantically and structurally from the reduct-based tradition. It is also inherently quadratic in the size of the program due to the number of nogoods generated, while the reduct-based strategy maintains linearity in theory size and retains a closer semantic alignment with the original ASP formalism. Recently, Alviano et al. (2019) revisited the reduct-based approach and introduced a variant that also leverages literal assumptions, overcoming the reuse limitation and enabling clause learning, and also revisited the unfounded-based approach to introduce a linear number of nogoods. These approaches were implemented in WASP, and empirical results showed it to be more efficient than traditional reduct-based checkers and competitive with CLINGO. Concerning recursive aggregates and model checker, CLINGO handles them at the implementation level; however, we have not found any publication in the literature that explicitly details the employed method. Based on source code inspection and informal discussions with one of the system's authors, it appears that CLINGO addresses recursion involving aggregates by rewriting them into a SAT formula. This approach differs from the one proposed in this paper, which relies on pseudo-Boolean constraints. Furthermore, CLINGO supports only the quadratic variant of the approach based on unfounded sets.

The present work advances this line of research along several dimensions: *(i)* It extends both reduct- and unfounded-set-based stability checking to recursive aggregates, a feature absent from prior model checking studies. *(ii)* It introduces pseudo-Boolean encodings for reducts instead of pure SAT, enabling compact and expressive representation of aggregates. *(iii)* It integrates these strategies as propagators in CLINGO, offering native support and flexibility within a widely adopted solver. *(iv)* It introduces partial stability checks into the reduct-based paradigm, a feature previously available only in the unfounded-set-based stability check. To our knowledge, this is the first work to support stable model checking with aggregates using both reduct- and unfounded-set-based approaches, and to validate their effectiveness within modern ASP solvers.

7 Conclusions

Stability checking is a fundamental task in ASP, particularly for disjunctive logic programs. Two main strategies have been developed to address it: one based on the reduct of the program and its encoding into a propositional theory, and another based on an alternative characterization of stable models in terms of unfounded sets. Both approaches have been refined over time to support efficient implementations, enabling practical solvers to perform stability checks effectively during model search. This paper extends both reduct- and unfounded-set-based stability checking to programs with recursive aggregates, combining pseudo-Boolean encodings with propagator-based implementations in CLINGO. We also introduce partial stability checks for the reduct-based approach, enabling early conflict detection during solving. Empirical results confirm that these techniques improve performance and scale to richer ASP constructs. These contributions demonstrate that both semantic strategies remain viable and efficient for advanced ASP tasks, paving the way for integration into optimization and portfolio-based solving frameworks.

Acknowledgments

This work was supported by the Italian Ministry of University and Research (MUR) under PRIN project PRODE "Probabilistic declarative process mining", CUP H53D23003420006, under PNRR project FAIR "Future AI Research", CUP H23C22000860006, under PNRR project Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006, and under PNRR project SERICS "SEcurity and RIghts in the CyberSpace", CUP H73C22000880001; by the Italian Ministry of Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by the Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); under PN RIC project ASVIN "Assistente Virtuale Intelligente di Negozio" (CUP B29J24000200005); and by the LAIA lab (part of the SILA labs). Mario Alviano is a member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

Aavani, A. 2011. Translating pseudo-boolean constraints into CNF. In *Proc. of SAT*, volume 6695 of *Lecture Notes in Computer Science*, 357–359. Springer.

Abseher, M.; Bliem, B.; Charwat, G.; Dusberger, F.; and Woltran, S. 2020. Computing secure sets in graphs using answer set programming. *J. Log. Comput.* 30(4):837–862.

Alviano, M.; Calimeri, F.; Faber, W.; Leone, N.; and Perri, S. 2011. Unfounded sets and well-founded semantics of answer set programs with aggregates. *J. Artif. Intell. Res.* 42:487–527.

Alviano, M.; Calimeri, F.; Charwat, G.; Dao-Tran, M.; Dodaro, C.; Ianni, G.; Krennwallner, T.; Kronegger, M.; Oetsch, J.; Pfandler, A.; Pührer, J.; Redl, C.; Ricca, F.; Schneider, P.; Schwengerer, M.; Spendier, L. K.; Wallner, J. P.; and Xiao, G. 2013a. The fourth answer set programming competition: Preliminary report. In *Proc. of LPNMR*, volume 8148 of *Lecture Notes in Computer Science*, 42–53. Springer.

Alviano, M.; Dodaro, C.; Faber, W.; Leone, N.; and Ricca, F. 2013b. WASP: A native ASP solver based on constraint learning. In *Proc. of LPNMR*, volume 8148 of *Lecture Notes in Computer Science*, 54–66. Springer.

Alviano, M.; Amendola, G.; Dodaro, C.; Leone, N.; Maratea, M.; and Ricca, F. 2019. Evaluation of disjunctive programs in WASP. In *Proc. of LPNMR*, volume 11481 of *Lecture Notes in Computer Science*, 241–255. Springer.

Alviano, M.; Dodaro, C.; and Ricca, F. 2013. Comparing alternative solutions for unfounded set propagation in ASP. In *Proc. of AI*IA*, volume 8249 of *Lecture Notes in Computer Science*, 1–12. Springer.

Alviano, M.; Faber, W.; and Gebser, M. 2015. Rewriting recursive aggregates in answer set programming: back to monotonicity. *Theory Pract. Log. Program.* 15(4-5):559–573.

Baral, C. 2010. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Berman, P.; Karpinski, M.; Larmore, L. L.; Plandowski, W.; and Rytter, W. 2002. On the complexity of pattern matching for highly compressed two-dimensional texts. *J. Comput. Syst. Sci.* 65(2):332–350.

Calimeri, F.; Ianni, G.; and Ricca, F. 2014. The third open answer set programming competition. *Theory Pract. Log. Program.* 14(1):117–135.

Dodaro, C. 2024. Design and implementation of modern CDCL ASP solvers. *Intelligenza Artificiale* 18(2):239–259.

Eén, N., and Sörensson, N. 2003. Temporal induction by incremental SAT solving. In *Proc. of BMC@CAV*, volume 89 of *Electronic Notes in Theoretical Computer Science*, 543–560. Elsevier.

Eiter, T.; Gottlob, G.; and Mannila, H. 1997. Disjunctive datalog. *ACM Trans. Database Syst.* 22(3):364–418.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Wanko, P. 2016. Theory solving made easy with clingo 5. In *Proc. of ICLP TCs*, volume 52 of *OASIcs*, 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Gebser, M.; Kaufmann, B.; and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.* 187:52–89.

Gebser, M.; Kaufmann, B.; and Schaub, T. 2013. Advanced conflict-driven disjunctive answer set solving. In *Proc. of IJCAI*, 912–918. IJCAI/AAAI.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Gener. Comput.* 9(3/4):365–386.

Hoos, H. H.; Lindauer, M.; and Schaub, T. 2014. claspfolio 2: Advances in algorithm selection for answer set programming. *Theory Pract. Log. Program.* 14(4-5):569–585.

Janhunen, T., and Niemelä, I. 2004. GNT - A solver for disjunctive logic programs. In *Proc. of LPNMR*, volume 2923 of *Lecture Notes in Computer Science*, 331–335. Springer.

Koch, C.; Leone, N.; and Pfeifer, G. 2003. Enhancing disjunctive logic programming systems by SAT checkers. *Artif. Intell.* 151(1-2):177–212.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.* 7(3):499–562.

Leone, N.; Rullo, P.; and Scarcello, F. 1997. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Inf. Comput.* 135(2):69–112.

Lierler, Y., and Maratea, M. 2004. Cmodels-2: Sat-based answer set solver enhanced to non-tight programs. In *Proc. of LPNMR*, volume 2923 of *Lecture Notes in Computer Science*, 346–350. Springer.

Lierler, Y. 2005. cmodels - sat-based disjunctive answer set solver. In *Proc. of LPNMR*, volume 3662 of *Lecture Notes in Computer Science*, 447–451. Springer.

Maratea, M.; Pulina, L.; and Ricca, F. 2014. A multi-engine approach to answer-set programming. *Theory Pract. Log. Program.* 14(6):841–868.

Marques Silva, J. P., and Sakallah, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers* 48(5):506–521.

Marx, D. 2011. Complexity of clique coloring and related problems. *Theor. Comput. Sci.* 412(29):3487–3500.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artif. Intell.* 138(1-2):181–234.

Zhang, L.; Madigan, C. F.; Moskewicz, M. W.; and Malik, S. 2001. Efficient conflict driven learning in boolean satisfiability solver. In *Proc. of ICCAD*, 279–285. IEEE Computer Society.