

# Implementing Stable-Unstable Semantics with ASPTOOLS and Clingo

Tomi Janhunen<sup>[0000–0002–2029–7708]</sup>

Tampere University, Tampere, Finland  
Tomi.Janhunen@tuni.fi

**Abstract.** Normal logic programs subject to stable model semantics cover reasoning problems from the first level of polynomial time hierarchy (PH) in a natural way. Disjunctive programs reach one level beyond this, but the access to the underlying NP oracle(s) is somewhat implicit and available for the programmer using the so-called saturation technique. To address this shortcoming, stable-unstable semantics was proposed, making oracles explicit as subprograms having no stable models. If this idea is applied recursively, any level of PH can be reached with normal programs only, in analogy to quantified Boolean formulas (QBFs). However, for the moment, no native implementations of stable-unstable semantics have emerged except via translations toward QBFs. In this work, we alleviate this situation with a translation of (effectively) normal programs that combines a main program with any fixed number of oracles subject to stable-unstable semantics. The result is a disjunctive program that can be fed as input for answer set solvers supporting disjunctive programs. The idea is to hide saturation from the programmer altogether, although it is exploited by the translation internally. The translation of oracles is performed using translators and linkers from the ASPTOOLS collection while Clingo is used as the back-end solver.

## 1 Introduction

The semantics of answer set programming paradigm (see, e.g., [6, 22] for an overview) rests on the notion of *stable models* first proposed for *normal* logic programs (NLPs) [15] and later generalized for *disjunctive* logic programs (DLPs) [16]. The known complexity results [9, 30] indicate that NLPs subject to stable model semantics cover reasoning problems from the first level of polynomial time hierarchy (PH) in a natural way while DLPs reach one level beyond. In the latter case, however, the access to underlying NP oracle(s) is somewhat implicit and best understood via the so-called *saturation* technique from the original complexity result [9]. When using saturation, the programmer is confronted with the fact that an oracle must be essentially expressed as a Boolean satisfiability problem, which differs from NLPs with respect to both syntax and semantics (cf. Section 5). In spite of this mismatch, saturation has been successfully applied when expressing properties pertaining to the second-level of PH [10, 13], e.g., when using meta-programming techniques together with saturation.

The *stable-unstable semantics* [3] was proposed as a remedy to the problems identified above. The main ideas are (i) to use NLPs when encoding problems, (ii) to make a subprogram acting as an oracle explicit, and (iii) to change the mode of reasoning from stability to instability for the oracle.<sup>1</sup> If this idea is applied in a nested fashion by merging NLPs recursively as *combined programs*, any level of PH can be reached with NLPs only, in analogy to *quantified Boolean formulas* (QBFs). In a nutshell, according to the stable-unstable semantics, we seek a stable model  $M$  for the main NLP  $P$  such that the NLP  $Q$  acting as the oracle has no stable model  $N$  that agrees with  $M$  about the truth values of atoms shared by  $P$  and  $Q$ . In contrast with QBFs, this leaves the quantification of atoms implicit, i.e., the atoms of  $P$  are existentially quantified while the local atoms of  $Q$  are effectively universal. There are follow-up approaches [1, 11] that make the quantification of atoms explicit. Regardless of this objective, the semantics of quantified programs is still aligned with the stable-unstable semantics, see [1, Theorem 7] and [11, Appendix B] for details. For the purposes of this work, however, implicit quantification is very natural, since the quantification of atoms can be controlled in terms of `#show`-statements directly supported by Clingo.

For the moment, no native implementations of stable-unstable semantics have emerged except via translations toward QBFs [3, 11]. The goal of this work is to alleviate this situation with a translation of (effectively) normal programs that combines a main program  $P$  with any fixed number of oracle programs  $P_1, \dots, P_n$  subject to stable-unstable semantics. In this way, we facilitate the incorporation of several oracles although, in principle, they could be merged into a single oracle first. The result of the translation is a DLP that can be fed as input for answer set solvers supporting DLPs. Thus we are mainly concentrating on search problems that reside on the second level of polynomial hierarchy.

One central idea behind our approach is to hide saturation from the programmer altogether, even though it is exploited by the translation internally. The reason behind this is that encoding saturation is error-prone when using non-ground rules with first-order variables. To this end, consider positive rules

$$\mathbf{u} \mid \mathbf{p}_1(\mathbf{X}_1) \mid \dots \mid \mathbf{p}_k(\mathbf{X}_k) \leftarrow \mathbf{p}_{k+1}(\mathbf{X}_{k+1}), \dots, \mathbf{p}_{k+m}(\mathbf{X}_{k+m}), \quad (1)$$

$$\mathbf{d}_1(\mathbf{Y}_1), \dots, \mathbf{d}_n(\mathbf{Y}_n).$$

used to encode an oracle where the special atom  $\mathbf{u}$  denotes *unsatisfiability*,  $\mathbf{p}_i$ :s are application predicates subject to saturation, and  $\mathbf{d}_j$ :s are domain predicates. In general, their argument lists  $\mathbf{X}_i$ :s and  $\mathbf{Y}_j$ :s consist of first-order terms and the domain predicates in the rule body restrict the possible values of variables occurring in the rule. Modern grounders are also able to infer part of this domain information based on the occurrences of predicates elsewhere in a program. Now, the saturating rules  $\mathbf{p}_i(\mathbf{t}) \leftarrow \mathbf{u}$  should be generated for every *ground* (non-input) atom  $\mathbf{p}_i(\mathbf{t})$  appearing in the ground rules of the oracle (see Definition 8 for details). Overseeing this objective presumes an understanding of which ground rules are actually produced for the oracle and, therefore, it becomes inherently difficult to find non-ground counterparts for the saturating rules for individual

<sup>1</sup> In terms of QBFs, this amounts to treating a QBF  $\exists X \forall Y \phi$  as  $\exists X \neg \exists Y \neg \phi$ .

predicates  $p_i(\mathbf{X})$ . In the worst case, the only option is to accompany each rule (1) of the oracle with saturating rules of the forms  $p_i(\mathbf{X}_i) \leftarrow u, d_1(\mathbf{Y}_1), \dots, d_n(\mathbf{Y}_n)$  for every  $1 \leq i \leq k + m$ . The number of such rules may get high and it is an extra burden for the programmer to keep these rules in synchrony with (1) when the rules encoding the oracle are further elaborated.

Our implementation is based on translators and linkers available under the ASPTOOLS<sup>2</sup> collection. Moreover, we expect that the grounding component of Clingo, namely Gringo, is used for instantiation. Thus we can use any Clingo program as the main program, exploiting extended rule types, proper disjunctive rules, and optimization as needed. As regards oracles, the translation-based approach of [21] sets the limits for their support in contrast with main programs. Due to existing normalization tools [4, 5], aggregates can be used. However, the use of disjunction in rule heads is restricted, i.e., only head-cycle-free disjunctions can be tolerated, as they can be translated away. Finally, optimization does not make sense in the context of oracles — supposed to have no stable models.

The rest of this article is organized as follows. In Section 2, we recall the syntax and the semantics of logic programs, including stable-unstable semantics. An account of the modularity properties of stable models is given in Section 3. Then, we concentrate on translations required in the subsequent treatment of oracles, i.e., the translation of NLPs into propositional clauses in Section 4 and the saturation technique in Section 5. Then we are ready to present our saturation-based technique for linking a main program with oracle programs in Section 6. The details of the implementation, including a saturating translator UNSAT2LP, are presented in Section 7. Moreover, we illustrate practical modeling with stable-unstable semantics in terms of the *point of no return* problem [2] involving a non-trivial oracle which is challenging to encode in ASP directly. The paper is concluded by Section 8 including a plan for future work.

## 2 Preliminaries

In this section, we review the syntax and semantics of logic programs and, in particular, the fragments of normal and disjunctive programs in the propositional case. Thus, as regards syntax, a *logic program* is a set of *rules* of form<sup>3</sup>

$$a_1 \mid \dots \mid a_k \leftarrow b_1, \dots, b_n, \text{not } c_1, \dots, \text{not } c_m. \quad (2)$$

where  $a_1, \dots, a_k, b_1, \dots, b_n$ , and  $c_1, \dots, c_m$ , are (propositional) atoms and “not” denotes negation by default. *Literals* are either atoms “ $a$ ” or their negations “not  $a$ ”, also called *positive* and *negative* literals, respectively. Using shorthands  $A, B$ , and  $C$  for the sets of atoms involved in (2), the rule can be abbreviated as  $A \leftarrow B, \text{not } C$  where “not  $C$ ” stands for the set of negative conditions  $\{\text{not } c \mid c \in C\}$ . The intuition behind a rule is that some atom from the *head*  $A$  of the

<sup>2</sup> <https://github.com/asptools>

<sup>3</sup> The syntax of logic programs has been generalized, e.g., with choice, cardinality, and weight rules in [32], but such extensions can be translated back to normal rules [4].

rule can be inferred true whenever the *body* of the rule is satisfied, i.e., when all atoms of  $B$  are true and no atom of  $C$  is true by any other rules in the program.

A rule (2) is *proper disjunctive*, if  $k > 1$ , *normal*, if  $k = 1$ , and a *constraint* if  $k = 0$ . A rule (2) is a *fact*, if  $k > 0$ ,  $n = 0$ , and  $m = 0$  and then  $\leftarrow$  is typically omitted. A *normal (logic) program* (NLP) consists of normal rules only whereas a *disjunctive (logic) program* (DLP) allows for any number of head atoms in its rules. Additionally, a program is called *positive* if  $m = 0$  for all of its rules (2).

## 2.1 Minimal and Stable Models

Turning our attention to semantics, let  $\text{At}(P)$  denote the *signature* of a program  $P$ , i.e.,  $A \cup B \cup C \subseteq \text{At}(P)$  for every rule  $A \leftarrow B, \text{not } C$  of  $P$ .<sup>4</sup> The semantics of a *positive* DLP  $P$  is determined as follows. An *interpretation*  $I$  of  $P$  is simply any subset of  $\text{At}(P)$  considered to be true under  $I$ . A (positive) rule  $A \leftarrow B$  of  $P$  is satisfied in an interpretation  $I \subseteq \text{At}(P)$  of  $P$ , if  $B \subseteq I$  implies  $A \cap I \neq \emptyset$ . A *model* of  $P$  is an interpretation  $M \subseteq \text{At}(P)$  satisfying all rules of  $P$ . A model  $M$  of  $P$  is a (subset) *minimal model* of  $P$  if there is no other model  $M'$  of  $P$  such that  $M' \subset M$ . The set of minimal models of  $P$  is denoted by  $\text{MM}(P)$ .

By the definitions above, a positive DLP may have no minimal models ( $P_1 = \{a. \leftarrow a.\}$ ), a unique minimal model ( $P_2 = \{a \leftarrow a.\}$ ), or several minimal models ( $P_3 = \{a \mid b.\}$ ). A widely agreed semantics of NLPs and DLPs is given by their stable models [15, 16] based on the Gelfond-Lifschitz reduct of a logic program  $P$  with respect to a model candidate  $M$ . The reduced program is

$$P^M = \{(A \leftarrow B) \mid (A \leftarrow B, \text{not } C) \in P, A \neq \emptyset, B \subseteq M, \text{ and } C \cap M = \emptyset\}. \quad (3)$$

**Definition 1 (Gelfond and Lifschitz [15, 16]).** *A model  $M$  of a program  $P$  is stable if and only if  $M \in \text{MM}(P^M)$ .*

We let  $\text{SM}(P)$  denote the set of stable models of  $P$ . It should be noted that Definition 1 covers constraints (2) with  $k = 0$  by the requirement that  $M$  is a *model* of  $P$ , i.e., for every rule  $A \leftarrow B, \text{not } C$  of  $P$ ,  $B \subseteq M$  and  $C \cap M = \emptyset$  imply  $A \cap M \neq \emptyset$ , thus treating default negation in rule bodies classically.

## 2.2 Stable-Unstable Semantics

The original definition of stable-unstable semantics adds one subprogram as an oracle to the main program. To cater for more flexible use cases, we formulate a generalized definition with any fixed number  $n \geq 0$  of oracles. While there is no real reason to restrict the main program  $P$ , it is assumed that oracles  $P_i$  are effectively NLPs containing no rule  $a \leftarrow B, \text{not } C$  such that  $a \in \text{At}(P)$ . Thus oracles may not *define* any concepts for the main program, they simply receive some facts as input, relating to the stable models of the main program.

<sup>4</sup> Typically,  $\text{At}(P)$  is selected to be minimal in this respect, i.e., it only contains atoms that actually appear in  $P$ . But larger sets might be used, e.g., if  $P$  resulted from rewriting and certain atoms were removed, but the semantics of  $P$  is unaffected.

**Definition 2.** A model  $M$  of a program  $P$  is stable-unstable with respect to oracle programs  $P_1, \dots, P_n$  if and only if  $M \in \text{SM}(P)$  and for every oracle  $P_i$  with  $1 \leq i \leq n$ ,  $\text{SM}(P_i \cup \{a. \mid a \in M \cap \text{At}(P_i)\}) = \emptyset$ .

Note that if there is an (input) atom  $a \in \text{At}(P) \cap \text{At}(P_i)$  such that  $a \notin M$ , then  $a$  will remain false by default in the context of the oracle  $P_i$  whose rules may not have  $a$  as the head. Moreover, when  $n = 0$ , we obtain the standard stable semantics (cf. Definition 1) as a special case of Definition 2.

### 3 Modularity

In this section, we adopt the Gaifman-Shapiro-style module architecture of DLPs from [24]. The respective modularity properties of stable models enable the modular (de)composition of DLPs. Programs are encapsulated as follows.

**Definition 3.** A program module  $\Pi$  is a quadruple  $\langle P, I, O, H \rangle$  where

1.  $P$  is a logic program,
2.  $I$ ,  $O$ , and  $H$  are pairwise disjoint sets of input, output, and hidden atoms;
3.  $\text{At}(P) \subseteq \text{At}(\Pi) = I \cup O \cup H$ ; and
4. if  $A \neq \emptyset$  for some rule  $A \leftarrow B, \text{not } C$  of  $P$ , then  $A \cap (O \cup H) \neq \emptyset$ .

The program interface of a module  $\Pi$  splits the signature  $\text{At}(\Pi)$  in three disjoint parts that serve the following purposes. The *visible* part  $\text{At}_v(\Pi) = I \cup O$  of  $\text{At}(\Pi)$  can be accessed by other modules to supply input for  $\Pi$  or to utilize its output. The *input signature*  $I$  and the *output signature*  $O$  of  $\Pi$  are also denoted by  $\text{At}_i(\Pi)$  and  $\text{At}_o(\Pi)$ , respectively. The *hidden* atoms in the difference  $\text{At}_h(\Pi) = \text{At}(\Pi) \setminus \text{At}_v(\Pi) = H$  can be used to formalize some internal (auxiliary) concepts of  $\Pi$ . The fourth requirement of Definition 3 ensures that every rule with a non-empty head must mention at least one non-input atom from  $O \cup H$ . This is a particular relaxation for disjunctive rules—note that the head of a normal rule cannot be an input atom by this requirement, but the heads of disjunctive rules may refer to input atoms as well. Thus, every rule in a module  $\Pi$  must contribute to the *definition* of at least one atom in  $O \cup H$ .

*Example 1.* Consider a module  $\Pi$  having only one rule  $a \mid b \leftarrow \text{not } c$  such that  $I = \{b, c\}$ ,  $O = \{a\}$ , and  $H = \emptyset$ . Therefore, the overall signature  $\text{At}(\Pi) = \{a, b, c\}$ . The requirements of Definition 3 are met. The fourth one is satisfied because the head  $a \mid b$  mentions the output atom  $a$  besides the input atom  $b$ . ■

A module  $\Pi$  corresponds to a conventional logic program when  $\text{At}_i(\Pi) = \emptyset = \text{At}_h(\Pi)$  and then the semantics of the module  $\Pi = \langle P, \emptyset, O, \emptyset \rangle$  is given by  $\text{SM}(P) \subseteq \mathbf{2}^{\text{At}_o(\Pi)} = \mathbf{2}^O$  directly. Hidden atoms become only relevant, when we compare programs or modules with each other, e.g., using the notion of *visible equivalence* [17]. Then, the idea is that each stable model  $M$  is reduced to  $M \setminus H = M \cap (I \cup O)$ , i.e., hidden atoms are neglected in comparisons but they do not affect stability by any means. However, to cover input atoms, the definition of stable models must be generalized, e.g., according to [24].

**Definition 4.** Given a program module  $\Pi = \langle P, I, O, H \rangle$ , the reduct of  $P$  with respect to a set  $M \subseteq \text{At}(\Pi)$  and the input signature  $I$ , denoted by  $P^{M,I}$ , contains a positive disjunctive rule  $(A \setminus I) \leftarrow (B \setminus I)$  if and only if there is a rule  $A \leftarrow B, \text{not } C$  of  $P$  such that  $A \neq \emptyset$ ,  $A \cap I \cap M = \emptyset$ ,  $B \cap I \subseteq M$ , and  $M \cap C = \emptyset$ .

In analogy to (3), the reduct  $P^{M,I}$  evaluates all negative literals in rule bodies of  $P$  and, in addition, all input atoms appearing elsewhere in  $P$ . Intuitively, if the satisfaction of a rule  $A \leftarrow B, \text{not } C$  under  $M$  depends on the remaining atoms in the rule, the respective reduced rule  $(A \setminus I) \leftarrow (B \setminus I)$  is included in  $P^{M,I}$ . The head  $A \setminus I$  of any such rule is necessarily non-empty when  $A \neq \emptyset$  by Item 4 in Definition 3 and  $P^{M,I}$  is guaranteed to possess (minimal) classical models. Potential constraints of  $P$  with  $A = \emptyset$  are covered in analogy to Definition 1.

**Definition 5.** A model  $M \subseteq \text{At}(\Pi)$  of a program module  $\Pi = \langle P, I, O, H \rangle$ , is stable if and only if  $M \setminus I \in \text{MM}(P^{M,I})$ .

As for programs, we let  $\text{SM}(\Pi)$  denote the set of stable models of  $\Pi$ .

*Example 2.* The module  $\Pi$  from Example 1 has four stable models in total, i.e.,  $\text{SM}(\Pi)$  equals to  $\{\{a\}, \{b\}, \{c\}, \{b, c\}\}$ . To verify that  $M = \{a\}$  is indeed stable, we note that  $P^{M,I} = \{a.\}$  with a minimal model  $\{a\}$  and  $\{a\} \setminus \{b, c\} = \{a\}$ . ■

Inputs to modules can also be taken into account by other means: an input  $M \cap I$  defined by an interpretation  $M \subseteq \text{At}(\Pi)$  could be added to  $\Pi$  as a set of facts [25]. The other option is to amend modules with *input generators* [31] that can be used to capture stable models of modules using Definition 1 and the standard reduct (3). Unfortunately, stable models of program modules do not provide a fully *compositional* semantics for logic programs: taking simple unions of modules does not guarantee that the stable models of the union could be obtained as straightforward combinations of the stable models for the modules involved. Towards this goal, two modules  $\Pi_1$  and  $\Pi_2$  are eligible for composition only if their output signatures are disjoint and they *respect each other's hidden atoms*, i.e.,  $\text{At}_h(\Pi_1) \cap \text{At}(\Pi_2) = \emptyset$  and  $\text{At}_h(\Pi_2) \cap \text{At}(\Pi_1) = \emptyset$ .

**Definition 6 (Composition [24]).** The composition of logic program modules  $\Pi_1 = \langle P_1, I_1, O_1, H_1 \rangle$  and  $\Pi_2 = \langle P_2, I_2, O_2, H_2 \rangle$ , denoted by  $\Pi_1 \oplus \Pi_2$ , is

$$\langle P_1 \cup P_2, (I_1 \setminus O_2) \cup (I_2 \setminus O_1), O_1 \cup O_2, H_1 \cup H_2 \rangle \quad (4)$$

if  $O_1 \cap O_2 = \emptyset$  and  $\Pi_1$  and  $\Pi_2$  respect each other's hidden atoms.

As demonstrated in [24], the conditions of Definition 6 do not yet imply the desired relationship of stable models in general. The conditions can be suitably tightened using the *positive dependency graph* of the composition  $\Pi_1 \oplus \Pi_2$ . Generally speaking, the *positive dependency graph*  $\text{DG}^+(\Pi)$  associated with a program module  $\Pi = \langle P, I, O, H \rangle$  is the pair  $\langle O \cup H, \leq \rangle$  where  $b \leq a$  holds for any atoms  $a$  and  $b$  of  $O \cup H$  if there is a rule  $A \leftarrow B, \text{not } C$  of  $P$  such that  $a \in A$  and  $b \in B$ . A *strongly connected component* (SCC)  $S$  of  $\text{DG}^+(P)$  is a maximal set

$S \subseteq \text{At}(P)$  such that  $b \leq^* a$  holds for every  $a, b \in S$ , i.e., all atoms of  $S$  depend positively on each other. If the composition  $\Pi_1 \oplus \Pi_2$  is defined, the members of the composition are *mutually dependent* if and only if  $\text{DG}^+(\Pi_1 \oplus \Pi_2)$  has an SCC  $S$  such that  $S \cap \text{At}_o(\Pi_1) \neq \emptyset$  and  $S \cap \text{At}_o(\Pi_2) \neq \emptyset$ , i.e., the SCC in question is effectively *shared* by  $\Pi_1$  and  $\Pi_2$ . Then, following [24], the *join*  $\Pi_1 \sqcup \Pi_2$  of  $\Pi_1$  and  $\Pi_2$  is defined as  $\Pi_1 \oplus \Pi_2$ , provided  $\Pi_1 \oplus \Pi_2$  is defined and  $\Pi_1$  and  $\Pi_2$  are mutually *independent*. The key observation from the viewpoint of compositional semantics is that stable models do not tolerate positive recursion across module boundaries. Thus, independence leads to a natural relationship<sup>5</sup> between the sets of stable models  $\text{SM}(\Pi_1 \sqcup \Pi_2)$ ,  $\text{SM}(\Pi_1)$ , and  $\text{SM}(\Pi_2)$  as detailed below.

**Theorem 1 (Module Theorem [24]).** *If  $\Pi_1$  and  $\Pi_2$  are program modules such that  $\Pi_1 \sqcup \Pi_2$  is defined, then  $\text{SM}(\Pi_1 \sqcup \Pi_2) = \text{SM}(\Pi_1) \bowtie \text{SM}(\Pi_2)$ .*

In Theorem 1, the operation  $\bowtie$  denotes a *natural join of compatible* stable models, i.e.,  $M_1 \cup M_2$  belongs to  $\text{SM}(\Pi_1) \bowtie \text{SM}(\Pi_2)$  if and only if  $M_1 \in \text{SM}(\Pi_1)$ ,  $M_2 \in \text{SM}(\Pi_2)$ , and  $M_1 \cap \text{At}_v(\Pi_2) = M_2 \cap \text{At}_v(\Pi_1)$ . Theorem 1 is easily generalized for finite joins of modules: if  $\Pi_1 \sqcup \dots \sqcup \Pi_n = \bigsqcup_{i=1}^n \Pi_i$  is defined, then

$$\text{SM}(\bigsqcup_{i=1}^n \Pi_i) = \bowtie_{i=1}^n \text{SM}(\Pi_i). \quad (5)$$

*Example 3.* Let us consider modules  $\Pi_1 = \langle \{a \leftarrow \text{not } b.\}, \{b\}, \{a\}, \emptyset \rangle$ ,  $\Pi_2 = \langle \{b \leftarrow \text{not } c.\}, \{c\}, \{b\}, \emptyset \rangle$ , and  $\Pi_3 = \langle \{c \leftarrow \text{not } a.\}, \{a\}, \{c\}, \emptyset \rangle$ . The respective sets of stable models are  $\text{SM}(\Pi_1) = \{\{a\}, \{b\}\}$ ,  $\text{SM}(\Pi_2) = \{\{b\}, \{c\}\}$ , and  $\text{SM}(\Pi_3) = \{\{c\}, \{a\}\}$ . The joins between the three modules are well-defined, since the output signatures are disjoint, no atoms are hidden, and no positive recursion is involved. Thus, we obtain by Theorem 1 that

$$\text{SM}(\Pi_1 \sqcup \Pi_2) = \text{SM}(\Pi_1) \bowtie \text{SM}(\Pi_2) = \{\{a\}, \{b\}\} \bowtie \{\{b\}, \{c\}\} = \{\{a, c\}, \{b\}\}$$

where the compatibility of stable models depends on  $\text{At}_i(\Pi_1) = \{b\} = \text{At}_o(\Pi_2)$ . When incorporating  $\text{SM}(\Pi_3) = \{\{a\}, \{c\}\}$ , we observe no models compatible with the ones listed above, i.e.,  $\text{SM}(\bigsqcup_{i=1}^3 \Pi_i) = \bowtie_{i=1}^3 \text{SM}(\Pi_i) = \emptyset$ . ■

Finally, it is worth noting that oracles  $P_i$ , as detailed in Definition 2, can be viewed as modules  $\Pi_i = \langle P_i, I_i, \emptyset, H_i \rangle$  that can be composed/joined with the main module  $\Pi = \langle P, \emptyset, O, \emptyset \rangle$ . By hiding all non-input atoms in the oracle modules, they cannot interfere with the atoms of  $\Pi$  in any well-defined compositions. However, since the visible parts  $M \cap I_i$  of stable models  $M \in \text{SM}(\Pi_i)$  are essentially witnesses for rejecting particular stable models of the main module, the relationship behind stable-unstable semantics cannot be expressed with  $\bowtie$  directly. This goes back to insisting on instability (cf. Theorem 4 in Section 6).

<sup>5</sup> The respective property of propositional formulas  $\phi_1$  and  $\phi_2$  is formalized by  $\text{CM}(\phi_1 \wedge \phi_2) = \text{CM}(\phi_1) \bowtie \text{CM}(\phi_2)$  where  $\text{CM}(\phi) \subseteq \mathbf{2}^{\text{At}(\phi)}$  gives the classical models of  $\phi$ .

## 4 Translating NLPs into SAT

In the forthcoming translations, we need to express *normal* logic programs as sets of propositional clauses as an intermediary step. The goal of this section is to recollect some results in this respect. A *clause* is an expression of the form

$$a_1 \vee \dots \vee a_m \vee \neg b_1 \vee \dots \vee \neg b_n \quad (6)$$

where  $a_1, \dots, a_m$  and  $b_1, \dots, b_n$  are (propositional) atoms. Given a set of clauses  $S$ , we write  $\text{At}(S)$  for the signature of  $S$  in analogy to the signature of a logic program (cf. Section 2). In the same way, an *interpretation*  $I$  for  $S$  is any subset of  $\text{At}(S)$  so that an atom  $a \in \text{At}(S)$  is considered *true* if  $a \in I$  and *false*, otherwise. A clause  $C$  of form (6) is satisfied by  $I$ , denoted by  $I \models C$ , if and only if some  $a_i \in I$  or some  $b_j \notin I$ . An interpretation  $M \subseteq \text{At}(S)$  is a (*classical*) *model* of  $S$ , denoted by  $M \models S$ , if and only if  $M \models C$  for every clause  $C$  of  $S$ . Then, let  $\text{CM}(S) = \{M \subseteq \text{At}(S) \mid M \models S\}$ . The signature  $\text{At}(S)$  can be partitioned into  $\text{At}_i(S)$ ,  $\text{At}_o(S)$ , and  $\text{At}_h(S)$ , if we wish to treat  $S$  as a module  $\langle S, I, O, H \rangle$  [19] in analogy to Section 3, keeping the semantics  $\text{CM}(S)$  intact.

Due to the greater expressive power of NLPs—relating to both default negation and recursive definitions—the translations from NLPs to SAT incur at least some blow-up. If no auxiliary atoms are introduced, the translation based on *loop formulas* [28, 29] is deemed worst-case exponential [26]. However, if new atoms are allowed, polynomial transformations become feasible, e.g., quadratic [27] and even sub-quadratic [17]. We adopt loop formulas for a brief illustration but exploit the most compact translation in the actual implementation. But, in contrast with [29], we use *bd* as a *new name* for a rule body  $B$ , not  $C$ . This amounts to a Tseitin transformation [33] of rule bodies. These new atoms enable a linear translation for the first part of the translation (Items 1 and 2 below) that captures Clark’s *completion* [7] for the program. The last item is based on loops  $L \subseteq \text{At}(P)$  which are strongly connected in the same way as SCCs but not necessarily maximal as sets. Thus, an SCC  $S$  may induce several loops.

**Definition 7 ([29]).** *Given an NLP  $P$ , the translation  $\text{Tr}_{\text{SAT}}(P)$  contains for every  $a \in \text{At}(P)$  and the defining rules  $a \leftarrow B_i, \text{not } C_i$  of  $a$  in  $P$  with  $1 \leq i \leq k$ ,*

1. *clauses  $a \vee \neg bd_1, \dots, a \vee \neg bd_k$  and a clause  $\neg a \vee bd_1 \vee \dots \vee bd_k$ ,*
2. *for each body indexed by  $1 \leq i \leq k$ , a clause  $bd_i \vee \neg B \vee C$ ,<sup>6</sup>  
clauses  $\{\neg bd_i \vee b \mid b \in B_i\}$ , and clauses  $\{\neg bd_i \vee \neg c \mid c \in C_i\}$ ;*

*and for every loop  $\emptyset \subset L \subseteq \text{At}(P)$  and the related externally supporting rules  $a \leftarrow B_i, \text{not } C_i$  of  $P$  with a head  $a \in L$ , positive body  $B_i \cap L = \emptyset$ , and  $1 \leq i \leq k$ ,*

3. *clauses  $\{bd_1 \vee \dots \vee bd_k \vee \neg a \mid a \in L\}$ .*

Intuitively, the clauses of the first item express  $a \leftrightarrow bd_1 \vee \dots \vee bd_k$  for each atom  $a$  while the second item establishes equivalences  $bd_i \leftrightarrow B_i \wedge \neg C_i$  for each

<sup>6</sup> A set of literals is understood disjunctively as part of a disjunction.

rule body indexed by  $1 \leq i \leq k$ . If  $k = 1$  for an atom  $a$ , then we can forget about  $bd_1$  and encode  $a \leftrightarrow B_i \wedge \neg C_i$  directly with the respective clauses. Last, the clauses in the third item essentially express a loop formula  $\neg bd_1 \wedge \dots \wedge \neg bd_k \rightarrow \neg L$  falsifying all atoms of the loop  $L$  in case they lack *external support* altogether.

**Theorem 2 ([29]).** *Let  $P$  be an NLP and  $\text{Tr}_{\text{SAT}}(P)$  its translation into SAT.*

1. *If  $M \in \text{SM}(P)$ , then  $N \models \text{Tr}_{\text{SAT}}(P)$  for a unique truth assignment  $N = M \cup \{bd \mid (a \leftarrow B, \text{not } C) \in P, B \subseteq M, C \cap M = \emptyset, \text{ and } bd \text{ names } (B, \text{not } C)\}$ .*
2. *If  $N \models \text{Tr}_{\text{SAT}}(P)$ , then  $M = N \cap \text{At}(P) \in \text{SM}(P)$ .*

The translation is also applicable to modules  $\Pi = \langle P, I, O, H \rangle$  by neglecting input atoms having no defining rules in program  $P$ . The resulting SAT-module  $\langle \text{Tr}_{\text{SAT}}(P), I, O, H' \rangle$  extends  $H$  to  $H'$  with new names  $bd$  introduced by  $\text{Tr}_{\text{SAT}}(\cdot)$ .

*Example 4.* Consider a program module  $\Pi = \langle P, \{c\}, \{a, b\}, \emptyset \rangle$  based on:

$$a \leftarrow b, \text{not } c. \quad b \leftarrow a. \quad a \leftarrow c.$$

The module has two stable models  $\{\}$  and  $\{a, b, c\}$ . The translation into SAT is:

$$\begin{array}{llll} a \vee \neg bd_1, & a \vee \neg bd_2, & \neg a \vee bd_1 \vee bd_2, & [a \leftrightarrow bd_1 \vee bd_2] \\ bd_1 \vee \neg b \vee c, & \neg bd_1 \vee b, & \neg bd_1 \vee \neg c, & [bd_1 \leftrightarrow b \wedge \neg c] \\ bd_2 \vee \neg c, & \neg bd_2 \vee c, & & [bd_2 \leftrightarrow c] \\ b \vee \neg a, & \neg b \vee a, & & [b \leftrightarrow a] \\ bd_2 \vee \neg a, & bd_2 \vee \neg b, & & [\neg bd_2 \rightarrow \neg a \wedge \neg b] \end{array}$$

Since  $c$  is an input atom, it is only treated as a condition in rule bodies. The only non-trivial loop of  $\Pi$  is  $\{a, b\}$  that gives rise to the last two clauses of the translation. The resulting SAT-module  $\langle \text{Tr}_{\text{SAT}}(P), \{c\}, \{a, b\}, \{bd_1, bd_2\} \rangle$  has two satisfying assignments  $\{\}$  and  $\{a, b, c, bd_2\}$  that capture the stable models of  $\Pi$ . It is important to note that the last two clauses exclude the truth assignment  $\{a, b, bd_1\}$  which would suggest an extra (incorrect) stable model  $\{a, b\}$  for  $\Pi$ . ■

## 5 Saturation

Saturation was introduced in [9] when showing that the main decision problems related to DLPs are complete on the second level of PH. It offers a central primitive for changing the mode of reasoning from *unsatisfiability* to the existence of a stable model. Typically, saturation is used as an integral part of the main program, but the goal of this section is to extract respective subprograms as independent program modules with input interfaces. Therefore, it is assumed below that a SAT-module  $\langle S, I, O, H \rangle$  is provided as input. Given an interpretation  $N \subseteq \text{At}_i(S)$ , we write  $S|_N$  for a *partial evaluation* of  $S$  obtained by (i) removing  $C \in S$  if  $N \models l$  for some literal  $l \in C$  and (ii) removing from  $C \in S$  any literal  $l \in C$  such that  $N \not\models l$ . The translation aims to capture truth assignments  $N$  over the set of input atoms  $I$  that render  $S|_N$  inconsistent.

**Definition 8.** Given a SAT-module  $\langle S, I, O, H \rangle$  encapsulating a set of clauses  $S$ , the saturation translation  $\text{Tr}_{\text{UNSAT}}(S)$  contains

1. for every clause (6), a positive disjunctive rule  $u \mid a_1 \mid \dots \mid a_m \leftarrow b_1, \dots, b_n$ ;
2. for every atom  $a \in \text{At}_h(S) \cup \text{At}_o(S)$ , the saturating rule  $a \leftarrow u$ ; and
3. the rule  $u \leftarrow \text{not } u$

where  $u \notin \text{At}(S)$  is a new atom. Moreover, we set  $\text{At}_i(\text{Tr}_{\text{UNSAT}}(S)) = \text{At}_i(S)$ ,  $\text{At}_o(\text{Tr}_{\text{UNSAT}}(S)) = \emptyset$ ,  $\text{At}_h(\text{Tr}_{\text{UNSAT}}(S)) = \text{At}_o(S) \cup \text{At}_h(S)$ .

All atoms except input atoms are hidden in  $\text{Tr}_{\text{UNSAT}}(S)$  because their values are uninteresting (all true) under any stable model of the translation. The intuitive idea of  $\text{Tr}_{\text{UNSAT}}(S)$  is that if  $S|_N$  is satisfiable for an input interpretation  $N \subseteq \text{At}_i(S)$ , then the positive rules in  $\text{Tr}_{\text{UNSAT}}(S)$  have a  $\subseteq$ -minimal (classical) model  $M$  extending  $N$  with  $u \notin M$ . Then, the rule  $u \leftarrow \text{not } u$  prevents stability.

*Example 5.* Consider the following set  $S$  of clauses:

$$a \vee b, \quad \neg a \vee b, \quad \neg a \vee \neg b.$$

Assuming that  $b$  is the only input atom of  $S$ , we observe that  $S|_{\emptyset} = \{a, \neg a\}$  is unsatisfiable while  $S|_{\{b\}} = \{\neg a\}$  is satisfiable. The translation  $\text{Tr}_{\text{UNSAT}}(S)$ :

$$\begin{array}{lll} u \mid a \mid b. & u \mid b \leftarrow a. & u \leftarrow a, b. \\ a \leftarrow u. & u \leftarrow \text{not } u. & \end{array}$$

where  $b$  is treated as an input atom. The translation has a stable model  $\{a, u\}$  indicating that  $S|_{\emptyset}$  is unsatisfiable. Note how this stable model would be excluded if  $b \leftarrow u$  were added in the translation. However, if  $b$  is added as a fact, there is no way to derive  $a$  nor  $u$  being false by default. Then, the translation augmented by the fact  $b$  has no stable models, indicating that  $S|_{\{b\}}$  is satisfiable. ■

**Theorem 3.** Given a SAT-module  $\langle S, I, O, H \rangle$  and its translation as a program module  $\Pi = \langle P, I, \emptyset, O \cup H \rangle$  with  $P = \text{Tr}_{\text{UNSAT}}(S)$ :

1. If  $S|_N$  is unsatisfiable for an input interpretation  $N \subseteq I$ , then  $N \cup O \cup H \cup \{u\} \in \text{SM}(\Pi)$ .
2. If  $M \in \text{SM}(\Pi)$  and  $N = M \cap I$ , then  $M = N \cup O \cup H \cup \{u\}$  and  $S|_N$  is unsatisfiable.

*Proof.* (1.) Let  $S|_N$  be unsatisfiable for some  $N \subseteq I$  and let  $M = N \cup O \cup H \cup \{u\}$ . To show  $M \in \text{SM}(\text{Tr}_{\text{UNSAT}}(S))$ , we should establish that  $M \setminus I \in \text{MM}(P^{M,I})$ .

(i) Since  $u \in M$ , the rule  $u \leftarrow \text{not } u$  does not contribute to the reduct, but  $a \leftarrow u$  is included for every  $a \in O \cup H$ . The rule is satisfied by  $M \setminus I = O \cup H \cup \{u\}$ . Moreover, the reduct contains  $u \mid (A \setminus I) \leftarrow (B \setminus I)$  for every clause  $A \vee \neg B \in S$  such that  $A \cap N = \emptyset$  and  $B \cap I \subseteq N$ , i.e.,  $A \vee \neg B \in S|_N$ . This rule is trivially satisfied by  $M \setminus I$  containing  $u$ . Thus  $M \setminus I \models P^{M,I}$ . (ii) Suppose that  $M' \models P^{M,I}$  for some  $M' \subset M \setminus I$ . If  $u \notin M'$ , then  $M' \models S|_N$ , a contradiction. Thus  $u \in M'$  and since  $a \leftarrow u$  is in  $P^{M,I}$  for every  $a \in O \cup H$  and  $M' \models a \leftarrow u$ , it follows that  $M' = M \setminus I$ , a contradiction. Thus,  $M \setminus I \in \text{MM}(P^{M,I})$ .

(2.) Let  $M \in \text{SM}(\Pi)$  and  $N = M \cap I$ . Due to  $u \leftarrow \text{not } u$  in  $P$ ,  $u \in M$  is necessarily the case. Since  $a \leftarrow u$  is contained in  $P^{M,I}$  for every  $a \in O \cup H$  and  $M \models P^{M,I}$  it follows that  $O \cup H \subseteq M$  and  $M = N \cup O \cup H \cup \{u\}$ . Calculating as above, the reduct contains  $u \mid (A \setminus I) \leftarrow (B \setminus I)$  for every  $A \vee \neg B \in S|_N$ . Assuming that  $S|_N$  is satisfiable, gives us  $M' \models S|_N$  such that  $u \notin M'$ . It follows that  $M' \subset M$  and  $M' \models P^{M,I}$ , a contradiction. Thus  $S|_N$  is unsatisfiable.  $\square$

## 6 Capturing Stable-Unstable Semantics

The goal of this section is to define a translation  $\text{Tr}_{\text{ST-UNST}}(\Pi, \Pi_1, \dots, \Pi_n)$  that captures the stable-unstable semantics of a *main program module*  $\Pi$  combined with *oracle modules*  $\Pi_1, \dots, \Pi_n$ . The translation exploits the preceding translations devised in Sections 4 and 5 as well as modularity properties from Section 3. Therefore, we formulate the result for modules with proper interface definitions.

**Definition 9.** *Given a main program module  $\Pi = \langle P, I, O, H \rangle$  and oracle modules  $\Pi_1, \dots, \Pi_n$  encapsulating NLPs  $P_1, \dots, P_n$ , the stable-unstable translation*

$$\text{Tr}_{\text{ST-UNST}}(\Pi, \Pi_1, \dots, \Pi_n) = \Pi \sqcup \bigsqcup_{i=1}^n \text{Tr}_{\text{UNSAT}}(\text{Tr}_{\text{SAT}}(\Pi_i)). \quad (7)$$

Due to pairwise input-output relationships there are no mutual positive dependencies between the translation  $\text{Tr}_{\text{UNSAT}}(\text{Tr}_{\text{SAT}}(\Pi_i))$  of each oracle module  $\Pi_i$  and the main module  $\Pi$ . The same can be stated about the translations of any pair of oracles  $\Pi_i$  and  $\Pi_j$  with  $i < j$ , because only input atoms are made visible and we may assume without loss of generality that  $\text{At}_h(\Pi_i) \cap \text{At}_h(\Pi_j) = \emptyset$ , since hidden atoms can always be renamed apart. Moreover, the new atoms ( $u$ ) introduced by the translation  $\text{Tr}_{\text{UNSAT}}(\cdot)$  can be assumed distinct for the oracles  $\Pi_1, \dots, \Pi_n$ , say atoms  $u_1, \dots, u_n$ . Thus the joins in (7) are well-formed and the resulting signatures of the translation  $\Pi' = \text{Tr}_{\text{ST-UNST}}(\Pi, \Pi_1, \dots, \Pi_n)$  are

1.  $\text{At}_i(\Pi') = \text{At}_i(\Pi)$ ,
2.  $\text{At}_o(\Pi') = \text{At}_o(\Pi)$ , and
3.  $\text{At}_h(\Pi') = \text{At}_h(\Pi) \cup (\bigcup_{i=1}^n \text{At}(\text{Tr}_{\text{UNSAT}}(\text{Tr}_{\text{SAT}}(\Pi_i))) \setminus \text{At}_i(\Pi_i))$ .

**Theorem 4.** *For a main program module  $\Pi$ , the NLP oracle program modules  $\Pi_1, \dots, \Pi_n$  of  $\Pi$ , and their stable-unstable translation:*

1. *If  $\text{Tr}_{\text{ST-UNST}}(\Pi, \Pi_1, \dots, \Pi_n)$  has a stable model  $N$ , then  $M = N \cap \text{At}(\Pi)$  is stable-unstable model of  $\Pi$  with respect to oracles  $\Pi_1, \dots, \Pi_n$ .*
2. *If  $\Pi$  has a stable-unstable model  $M$  with respect to oracles  $\Pi_1, \dots, \Pi_n$ , then  $\text{Tr}_{\text{ST-UNST}}(\Pi, \Pi_1, \dots, \Pi_n)$  has a stable model  $N$  such that  $M = N \cap \text{At}(\Pi)$ .*

*Proof.* Since the joins in Definition 9 are well-formed, we may apply Theorem 1:

$$\text{SM}(\text{Tr}_{\text{ST-UNST}}(\Pi, \Pi_1, \dots, \Pi_n)) = \text{SM}(\Pi) \bowtie (\bigotimes_{i=1}^n \text{SM}(\text{Tr}_{\text{UNSAT}}(\text{Tr}_{\text{SAT}}(\Pi_i))))). \quad (8)$$

For brevity, let  $\Pi'$  stand for the entire translation  $\text{Tr}_{\text{ST-UNST}}(\Pi, \Pi_1, \dots, \Pi_n)$  and  $\Pi'_i$  for the translation  $\text{Tr}_{\text{UNSAT}}(\text{Tr}_{\text{SAT}}(\Pi_i))$  of each oracle  $\Pi_i$  with  $1 \leq i \leq n$ .

By the model correspondence (8) established above,  $N$  is a stable model of  $\Pi'$  if and only if  $M = N \cap \text{At}(\Pi) \in \text{SM}(\Pi)$  and  $N_i = N \cap \text{At}(\Pi'_i) \in \text{SM}(\Pi'_i)$  for each  $1 \leq i \leq n$ . By Theorem 3, this holds if and only if  $M \in \text{SM}(\Pi)$  and  $\text{Tr}_{\text{SAT}}(\Pi_i)|_{M_i}$  is unsatisfiable for each  $1 \leq i \leq n$  and the respective input  $M_i = N_i \cap \text{At}_i(\Pi_i)$ . By Theorem 2, this is equivalent to  $M \in \text{SM}(\Pi)$  and each oracle  $\Pi_i$  with  $1 \leq i \leq n$  having no stable models given the input  $M_i$ , i.e.,  $M$  is a stable-unstable model of  $\Pi$  with respect to  $\Pi_1, \dots, \Pi_n$ .  $\square$

Definition 9 and Theorem 4 characterize our method for computing stable-unstable models in the propositional case. Therefore, let us discuss how non-ground programs fit into this scenario. Given a set of non-ground rules  $P$ , we write  $\text{Gnd}(P)$  for the resulting ground program produced by a grounder such as Gringo in the Clingo system. Since  $\text{Gnd}(P)$  depends on the grounder, we leave its exact definition open and assume that the semantics of a non-ground program  $P$  is determined by  $\text{SM}(\text{Gnd}(P))$  where  $\text{Gnd}(P)$  is understood as a propositional program. The signature of  $\text{Gnd}(P)$  is also determined during the grounding phase, based on directives supplied by the programmer. Thus, for a non-ground main program  $P$  and each non-ground oracle  $P_i$  with  $1 \leq i \leq n$ , we effectively obtain the ground module  $\Pi = \langle \text{Gnd}(P), I, O, H \rangle$  and the ground oracle modules  $\Pi_i = \langle \text{Gnd}(P_i), I_i, \emptyset, H_i \rangle$  where  $1 \leq i \leq n$ . Then, stable-unstable models can be computed using the translation  $\text{Tr}_{\text{ST-UNST}}(\Pi, \Pi_1, \dots, \Pi_n)$  in (7).

## 7 Implementation and Practical Modeling

In what follows, we describe how our method for computing stable-unstable semantics can be realized in practice using tools available in the ASPTOOLS collection and Clasp as the the back-end solver. Finally, we illustrate practical modeling in terms of an application problem that is challenging to formalize if the goal is to represent the entire problem as a single DLP in Section 7.1. The performance of Clingo on the resulting stable-unstable encoding of the problem is screened in Section 7.2. Reflecting Definition 9, our implementation involves the following three steps for the *ground* modules  $\Pi$  and  $\Pi_1, \dots, \Pi_n$ :

1. translating each oracle module  $\Pi_i$  into SAT, i.e., the SAT module  $\text{Tr}_{\text{SAT}}(\Pi_i)$ ,
2. translating each  $\text{Tr}_{\text{SAT}}(\Pi_i)$  into a DLP module  $\text{Tr}_{\text{UNSAT}}(\text{Tr}_{\text{SAT}}(\Pi_i))$ , and
3. linking the parts of the translation (7) together.

*Translating Oracles into SAT.* The translation of oracles is based on translators in the LP2SAT family. These translators implement the more compact transformation described in [17], the one described in Section 4 is compatible up to forming the completion of the program. In addition, we deploy other tools in order to extend the applicability of our approach somewhat beyond the class of NLPs. In general, it is recommended to use a tool pipeline similar to those used in the latest ASP competitions. Brief descriptions of the tools follow. (i) Remove

invisible facts produced by the grounder using LPSTRIP. (ii) Make the symbol table of the program contiguous with LPCAT as described below. (iii) Unwind *head-cycle-free* (HCF) disjunctions by *shifting* [8] as implemented by LPSHIFT. (iv) Translate away aggregates [32] using LP2NORMAL2 [5]. (v) Instrument SCCs with additional rules that guarantee the acyclicity of support within components [12] by calling LP2ACYC. (vi) Produce the respective CNF using LP2SAT and its command-line option `-b` for a translation in line with [17].

*Saturation Transformation.* The compiler for the saturation transformation, called UNSAT2LP, is available in the ASPTOOLS collection [21]. The input of the compiler consists of a DIMACS file extended by the definitions of symbols in comments. The translators described in the preceding step produce these definitions automatically and they are crucial information for the linking phase. The compiler UNSAT2LP is directly based on Definition 8. The output is a DLP in the SMODELS format [18], supported by Clingo for backward compatibility.

*Linking.* Definition 6 provides the specification for a link editor called LPCAT [20]. Given ground program modules  $\Pi_1, \dots, \Pi_n$  as input, assuming that the join  $\Pi_1 \sqcup \dots \sqcup \Pi_n$  is defined, the tool can be used to safely compute their composition. In the output, every atom will have a unique number (i.e., index in the atom table) and atoms are numbered from 1 to  $n$  where  $n$  gives the number of atoms in the program. The stable models of the resulting ground program are then governed by (5) and they can be computed by invoking Clasp. Other disjunctive solvers can be potentially used, if the final ground program is translated back into symbolic form (e.g., using the program listing tool LPLIST from the ASPTOOLS collection) and parsed again.

## 7.1 Practical Modeling

Having described the steps of translation involved in our implementation, let us introduce one concrete encoding to demonstrate the use of tools in practice. In the sequel, we use the problem *point of no return* [2] for illustration. The problem was specifically designed to reside on the second level of PH and it requires the representation of an oracle which is non-trivial to encode via saturation directly, due to interlinked reachability and satisfiability conditions. Below we recall the problem, but by using clauses rather than formulas as labels for a digraph.

**Definition 10 (Point of No Return).** *Given a digraph  $G = \langle N, A \rangle$  where  $A \subseteq N^2$ , a start node  $s \in N$ , and a labeling function  $cl(\cdot)$  that maps each arc  $\langle n_1, n_2 \rangle \in A$  to a clause  $cl(n_1, n_2)$ , a point of no return is a node  $n \in N$  so that*

1. *there is a directed path  $s = n_1, n_2, \dots, n_k = n$  in  $G$ ,*
2. *the set of clauses  $S(s, n) = \{cl(n_1, n_2), \dots, cl(n_{k-1}, n_k)\}$  is satisfiable, and*
3. *there is **no** directed return path  $n = m_1, m_2, \dots, m_l = s$  in  $G$  such that the set of clauses  $S(s, n) \cup \{cl(m_1, m_2), \dots, cl(m_{l-1}, m_l)\}$  is satisfiable.*

**Listing 1.** Point of No Return: A Minimal Instance

% Assign clauses to arcs		% Arcs
lit(1,2,a).	lit(1,2,b).	% 1 == a b ==> 2
lit(2,3,n(a)).	lit(2,3,n(b)).	% 2 == -a -b ==> 3
lit(3,4,a).	lit(3,4,n(b)).	% 3 == a -b ==> 4
lit(3,1,n(a)).	lit(3,1,c).	% 3 == -a c ==> 1
lit(4,1,n(a)).	lit(4,1,b).	% 4 == -a b ==> 1

**Listing 2.** Point of No Return: Domain Declarations

% Identify atoms and literals	
literal(L) :- lit(_,_,L).	
negative(n(A)) :- literal(n(A)).	
atom(L) :- literal(L), not negative(L).	
% Determine arcs, nodes, and the start node	
arc(X,Y) :- lit(X,Y,_).	
node(X) :- arc(X,_).	node(Y) :- arc(_,Y).
start(N) :- node(N), N2 >= N: node(N2).	

Our ASP encoding of this problem is given as four Clingo code snippets: (i) an example of a problem instance, (ii) some joint domain definitions, (iii) the main program, and (iv) the oracle. In Listing 1, we describe a minimal problem instance using a predicate `lit/3` which associates for an arc  $\langle n_1, n_2 \rangle$  one literal  $l$  involved in the labeling clause  $cl(n_1, n_2)$  at a time. The function symbol `n/1` is used to express negative literals. Assuming that 1 is the start node, then the node 4 is a point of no return: the set of clauses  $\{a \vee b, \neg a \vee \neg b, a \vee \neg b\}$  is satisfiable while adding the final clause  $\neg a \vee b$  will make it necessarily unsatisfiable. The other nodes are not points of no return due to the short-cutting arc from 3 to 1 enforcing the clause  $\neg a \vee c$ . The rules in Listing 2 define some joint domains involved in the problem. The names of predicates `literal/1`, `atom/1`, `arc/2`, and `node/1` should be self-explanatory in this respect. Moreover, the predicate `start/1` picks the smallest node as the start node for the whole input graph.

The main program, as given by Listing 3, deploys *choice rules* [32] with *bounds on cardinality* for the sake of conciseness. The `path/2` predicate captures the path from the start node to a node acting as a candidate point of no return and eventually pointed out by predicate `ponr/1`. The recursive selection of the path is guided by the predicate `reach/1` formalizing reachability from the start node along the path. Finally, predicate `true/1` chooses a subset of atoms to be true and the satisfaction of the clauses along the chosen path is enforced by the constraint in the end. The encoding for the required oracle is quite similar as can be seen from Listing 4. At first, the input predicates are declared using choice rules. Only then are they made visible for translators. The choice of return path is

**Listing 3.** Point of No Return: Main Program

```

% Choose path and the point of no return
{ path(X,Y): arc(X,Y), not start(Y) } = 1 :- start(X).
{ path(X,Y): arc(X,Y), not start(Y) } <= 1 :- reach(X).

% The point of no return is the final node reached
reach(Y) :- path(X,Y).
ponr(X) :- reach(X), not path(X,Y): arc(X,Y).
:- not ponr(X): node(X).

% Check satisfiability along the chosen path
{ true(A) } :- atom(A).
true(n(A)) :- negative(n(A)), not true(A).
:- arc(X,Y), path(X,Y), not true(L): lit(X,Y,L).

```

analogous but formalized *backwards* from the start node toward the anticipated point of no return. The predicate `reach/1` can be reused here since it will not be visible to the main program. The final satisfiability check is quite the same except that the clauses along both chosen paths ought to be satisfied. Assuming that the portions of code from Listings 1–4 are stored in files `literals.lp`, `graph.lp`, `main.lp`, and `oracle.lp`, respectively, we may invoke the following shell commands to solve the problem with Gringo and Clasp:

```

$ gringo --output smodels literals.lp graph.lp main.lp > main.sm
$ gringo --output smodels literals.lp graph.lp oracle.lp \
  | lp2normal2 | lp2acyc | lp2sat -b | unsat2lp > oracle.sm
$ lpcat main.sm oracle.sm | clasp

```

For backward compatibility, we use Gringo’s option `--output smodels`. Notice how the instance and domain declarations are used when grounding the main program and the oracle in separation. Finally, they are linked together with LPCAT and fed as input for Clasp which accepts Smodels format as such.

## 7.2 Performance Analysis

To get an idea of the performance of our approach to implementing stable-unstable semantics, we carried out some preliminary experiments using the encodings from Listings 1–4. We used Gringo (v. 5.2.2) as the grounder and Clasp (v. 3.3.4) as the solver. All runs were executed on an Intel(R) Core i7-8750H CPU with a 2.20GHz clock rate under Linux operating system.

Since the existence of Hamiltonian paths in planar graphs has been previously investigated, we decided to generate such graphs using the PLANAR tool from the ASPTOOLS collection. The tool outputs a random planar graph with a given number of nodes. The graphs are directed and symmetric, i.e., arcs are provided in both directions. First, we check the performance of Clasp on unsatisfiable

**Listing 4.** Point of No Return: Oracle

```

% Input
{ path(X,Y) } :- arc(X,Y), not start(Y).
{ ponr(X) } :- node(X), not start(X).
#show path/2.
#show ponr/1.

% Choose return path
{ return(X,Y): arc(X,Y), not start(X) } = 1 :- reach(Y).
:- return(X,Y), path(X,Y).

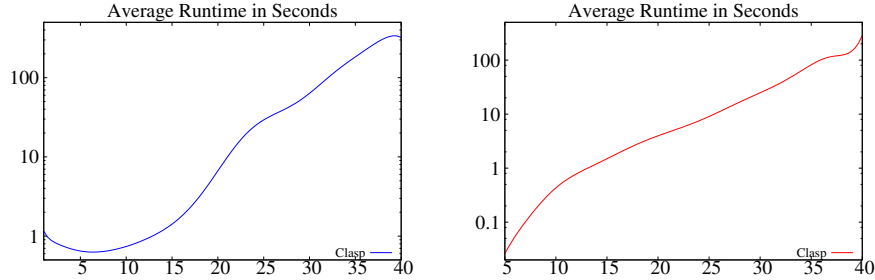
% Check that the point of no return is reached
reach(X) :- start(X).
reach(X) :- return(X,Y), not ponr(X).
:- ponr(X), not return(X,Y): arc(X,Y).

% Check satisfiability along both paths
{ true(A) } :- atom(A).
true(n(A)) :- negative(n(A)), not true(A).
:- arc(X,Y), path(X,Y), not true(L): lit(X,Y,L).
:- arc(X,Y), return(X,Y), not true(L): lit(X,Y,L).

```

instances obtained from planar graphs with  $n = 9 \dots 18$  nodes and roughly from 40 to 85 arcs. By mapping arcs to a fixed atom  $a$ , all paths are consistent and no points of no return are feasible. The runtimes vary from 0.70s to 16 000s and the growth is clearly exponential in  $n$  that we verified using a logarithmic plot.

Based on the preliminary screening, we pick  $n = 15$ , for which a runtime of 390 seconds is initially obtained, for further study. Next, we map the arcs of the planar graphs to random literals based on  $v$  different atoms. Both the atom and its polarity are selected uniformly. As a result, arcs labeled with opposite literals become mutually exclusive. On the one hand, this is a significant source of complexity (see [22] for an analogous restriction) but, on the other hand, makes points of no return existent. When the number of atoms  $v$  is increased, the resulting instances are expected to become more demanding. To see the effect, we generate ten instances for each  $v = 1 \dots 40$ . The runtime behavior is illustrated by the graph in Fig. 1 (left). The number of unsatisfiable instances starts to grow from  $v = 31$ , i.e., roughly 45% of the number of arcs when  $n = 15$ , and therefore, runtimes approach and settle around 400s as observed for unsatisfiable instances earlier. Next, we select  $v = n$  as the criterion for our final experiment and let  $n$  vary from 5 to 40 nodes. The runtimes are illustrated by the graph in Fig. 1 (right). By the logarithmic scale, runtimes tend to grow exponentially in  $n$ . The resulting instances are mostly satisfiable except for small values for  $n$  when finding a (satisfiable) path may become an obstacle (cf. Definition 10).



**Fig. 1.** Point of No Return: Runtime Scaling for Instances Based on Planar Graphs

## 8 Discussion and Conclusion

In this work, we propose an alternative way to implement stable-unstable semantics of (normal) logic programs. In contrast with related approaches based on meta-programming [10, 13] and translations toward QBFs [2, 11], we encode oracles as stand-alone (effectively normal) programs, ground and translate them separately, and finally link them with the ground main program for solving. This makes our approach highly modular and enables the separation of concerns in case of multiple oracles, thus generalizing stable-unstable semantics in the first place. We anticipate that the saturation step is less error-prone when outsourced for a translator, relieving the programmer from a potentially intricate task and enabling the testing of oracles in separation. Moreover, in contrast with [1, 11] our approach counts on implicit quantification as put forth in original stable-unstable semantics. When modeling with stable-unstable semantics, we essentially seek solutions to problems whose particular subproblems have no solutions. Finally, our preliminary performance analysis suggests that computing points of no return will provide a challenging benchmark for answer set solvers.

As regards future work, we note that it is possible to change the translation  $\text{Tr}_{\text{SAT}}(S)$  from NLPs to SAT very easily, e.g., for improving performance. Although our approach enables more comprehensive modeling based on stable-unstable semantics, we still call for native implementations that support stable-unstable semantics directly rather than through the stable semantics of DLPs. Such implementations are expected to mimic the design of GnT [23] with interacting solvers, but use conflict-driven nogood learning (CDNL) [14] instead of traditional branch-and-bound search. Moreover, if solvers are integrated with each other recursively, following the original idea of *combined programs* from [3], the levels beyond the second one in polynomial hierarchy can also be covered.

*Acknowledgments.* The author wishes to thank the anonymous referees for comments and suggestions for improvement. The author has been partially supported by the Academy of Finland projects ETAIROS (327352) and AI-ROT (335718).

## References

1. Amendola, G., Ricca, F., Truszczynski, M.: Beyond NP: Quantifying over answer sets. *Theory Pract. Log. Program.* **19**(5-6), 705–721 (2019)
2. Bogaerts, B., Janhunen, T., Tasharrofi, S.: Declarative solver development: Case studies. In: *KR 2016*. pp. 74–83. AAAI Press (2016)
3. Bogaerts, B., Janhunen, T., Tasharrofi, S.: Stable-unstable semantics: Beyond NP with normal logic programs. *Theory Pract. Log. Program.* **16**(5-6), 570–586 (2016)
4. Bomanson, J., Gebser, M., Janhunen, T.: Improving the normalization of weight rules in answer set programs. In: *Proceedings of JELIA 2014*. pp. 166–180. Springer (2014)
5. Bomanson, J., Janhunen, T., Niemelä, I.: Applying visible strong equivalence in answer-set program transformations. *ACM Trans. Comput. Log.* **21**(4), 33:1–33:41 (2020)
6. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. *Commun. ACM* **54**(12), 92–103 (2011)
7. Clark, K.: Negation as failure. In: *Logic and Data Bases*, pp. 293–322. Plenum Press (1978)
8. Dix, J., Gottlob, G., Marek, V.W.: Reducing disjunctive to non-disjunctive semantics by shift-operations. *Fundam. Informaticae* **28**(1-2), 87–100 (1996)
9. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence* **15**(3–4), 289–323 (1995)
10. Eiter, T., Polleres, A.: Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *Theory Pract. Log. Program.* **6**(1-2), 23–60 (2006)
11. Fandinno, J., Laferrière, F., Romero, J., Schaub, T., Son, T.C.: Planning with incomplete information in quantified answer set programming. *Theory Pract. Log. Program.* **21**(5), 663–679 (2021)
12. Gebser, M., Janhunen, T., Rintanen, J.: Answer set programming as SAT modulo acyclicity. In: *Proceedings of ECAI 2014*. pp. 351–356. IOS Press (2014)
13. Gebser, M., Kaminski, R., Schaub, T.: Complex optimization in answer set programming. *Theory Pract. Log. Program.* **11**(4-5), 821–839 (2011)
14. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. *Artif. Intell.* **187**, 52–89 (2012)
15. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: *Proceedings of ICLP*. pp. 1070–1080. MIT Press (1988)
16. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. *New Gener. Comput.* **9**(3/4), 365–386 (1991)
17. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. *J. Appl. Non Class. Logics* **16**(1-2), 35–86 (2006)
18. Janhunen, T.: Intermediate languages of ASP systems and tools. In: *Proceedings of SEA 2007, the 1st International Workshop on Software Engineering for Answer Set Programming*. pp. 12–25. University of Bath, Department of Computer Science, Report CSBU-2007-05 (2007)
19. Janhunen, T.: Modular equivalence in general. In: *Proceedings of ECAI 2008*. pp. 75–79 (2008)
20. Janhunen, T.: Modular construction of ground logic programs using LPCAT. In: *The 3rd International Workshop on Logic and Search (LaSh'10)* (2010)

21. Janhunen, T.: Cross-translating answer set programs using the ASPTOOLS collection. *Künstliche Intell.* **32**(2-3), 183–184 (2018)
22. Janhunen, T., Niemelä, I.: The answer set programming paradigm. *AI Mag.* **37**(3), 13–24 (2016)
23. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.: Unfolding partiality and disjunctions in stable model semantics. *ACM Trans. Comput. Log.* **7**(1), 1–37 (2006)
24. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. *J. Artif. Intell. Res.* **35**, 813–857 (2009)
25. Lierler, Y., Truszczynski, M.: On abstract modular inference systems and solvers. *Artif. Intell.* **236**, 65–89 (2016)
26. Lifschitz, V., Razborov, A.: Why are there so many loop formulas? *ACM Trans. Comput. Log.* **7**(2), 261–268 (2006)
27. Lin, F., Zhao, J.: On tight logic programs and yet another translation from normal logic programs to propositional logic. In: *Proceedings of IJCAI 2003*. pp. 853–858 (2003)
28. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. In: *Proceedings of AAAI 2002*. pp. 112–118 (2002)
29. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. *Artif. Intell.* **157**(1-2), 115–137 (2004)
30. Marek, V., Truszczyński, M.: Autoepistemic logic. *J. ACM* **38**(3), 588–619 (1991)
31. Oikarinen, E., Janhunen, T.: A translation-based approach to the verification of modular equivalence. *J. Log. Comput.* **19**(4), 591–613 (2009)
32. Simons, P., Niemelä, I., Sooinen, T.: Extending and implementing the stable model semantics. *Artificial Intelligence* **138**(1-2), 181–234 (2002)
33. Tseitin, G.: On the complexity of derivation in the propositional calculus. *Zapiski Nauchnykh Seminarov LOMI* **8**, 234–259 (1968)