Bogaerts, Bart; Janhunen, Tomi; Tasharrofi, Shahab

Stable-unstable semantics

Published in:
Theory and Practice of Logic Programming

DOI:
10.1017/S1471068416000387

Published: 01/09/2016

Document Version
Peer reviewed version

Please cite the original version:
Bogaerts, B., Janhunen, T., & Tasharrofi, S. (2016). Stable-unstable semantics: Beyond NP with normal logic programs. Theory and Practice of Logic Programming, 16(5-6), 570-586.
https://doi.org/10.1017/S1471068416000387

Stable-Unstable Semantics: Beyond NP with Normal Logic Programs

BART BOGAERTS, TOMI JANHUNEN and SHAHAB TASHARROFI

Helsinki Institute for Information Technology HIIT∗ Department of Computer Science
Aalto University, FI-00076 AALTO, Finland
(*e-mail:* firstname.lastname@aalto.fi)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Standard answer set programming (ASP) targets at solving search problems from the first level of the polynomial time hierarchy (PH). Tackling search problems beyond NP using ASP is less straightforward. The class of disjunctive logic programs offers the most prominent way of reaching the second level of the PH, but encoding respective hard problems as disjunctive programs typically requires sophisticated techniques such as saturation or meta-interpretation. The application of such techniques easily leads to encodings that are inaccessible to non-experts. Furthermore, while disjunctive ASP solvers often rely on calls to a (co-)NP oracle, it may be difficult to detect from the input program where the oracle is being accessed. In other formalisms, such as Quantified Boolean Formulas (QBFs), the interface to the underlying oracle is more transparent as it is explicitly recorded in the quantifier prefix of a formula. On the other hand, ASP has advantages over QBFs from the modeling perspective. The rich high-level languages such as ASP-Core-2 offer a wide variety of primitives that enable concise and natural encodings of search problems. In this paper, we present a novel logic programming–based modeling paradigm that combines the best features of ASP and QBFs. We develop so-called *combined logic programs* in which oracles are directly cast as (normal) logic programs themselves. Recursive incarnations of this construction enable logic programming on arbitrarily high levels of the PH. We develop a proof-of-concept implementation for our new paradigm. This paper is under consideration for acceptance in TPLP.

KEYWORDS: disjunctive logic programming, polynomial hierarchy, quantified Boolean formulas

1 Introduction

With the launch of the idea that stable models (Gelfond and Lifschitz 1988) of a logic program can be used to encode search problems, a new programming paradigm, called Answer Set Programming (ASP) was born (Marek and Truszczyński 1999; Niemelä 1999; Lifschitz 1999). Nowadays, the fact that normal logic programs can effectively encode NP-complete decision and function problems is exploited in applications in many different domains such as robotics (Andres et al. 2015), machine learning (Janhunen et al. 2015; Bruynooghe et al. 2015), phylogenetic inference (Koponen et al. 2015; Brooks et al. 2007), product configuration (Tiihonen et al. 2003), decision support for the Space Shuttle (Nogueira et al. 2001), e-Tourism (Ricca et al. 2010), and knowledge management (Grasso et al. 2009).

∗ The support from the Finnish Center of Excellence in Computational Inference Research (COIN) funded by the Academy of Finland (under grant #251170) is gratefully acknowledged.

Tackling search problems beyond NP with ASP requires one to use more expressive logic programs than the normal ones. To this end, the class of disjunctive programs (Gelfond and Lifschitz 1991) is the most prominent candidate. As shown by Eiter and Gottlob (1995), the main decision problems associated to disjunctive programs are Σ_2^P- and Π_2^P-complete, depending on the reasoning mode, i.e., *credulous* vs. *cautious* reasoning. But when it comes to applications, one encounters disjunctive encodings less frequently than encodings as normal logic programs. This is also witnessed by the benchmark problems submitted to ASP competitions (Calimeri et al. 2016). Such a state of affairs is not due to a lack of application problems since many complete problems from the second level of the PH are known. Neither is it due to a lack of implementations, since state-of-the-art ASP solvers such as DLV (Leone et al. 2006) and CLASP (Drescher et al. 2008; Gebser et al. 2015) offer a seamless support for disjunctive programs.

An explanation for the imbalance identified above can be found in the essentials of disjunctive logic programming when formalizing problems from the second level of the PH. There are results (Ben-Eliyahu and Dechter 1994) showing that such programs must involve *head cycles*, i.e., cyclic positive dependencies established by the rules of the program that intertwine with the disjunctions in the program. Such dependencies may render disjunctive programs hard to understand and to maintain. Moreover, the existing generic encodings of complete problems from the second level of the PH as disjunctive programs are based on sophisticated *saturation* (Eiter and Gottlob 1995) or *meta-interpretation* (Gebser et al. 2011) techniques, which may turn an encoding inaccessible to a non-expert. Eiter and Polleres (2006) identify the limitations of subprograms that act as (co-)NP-oracles and are embedded in disjunctive programs using the saturation technique. Summarizing our observations, the access to the underlying oracle is somewhat cumbersome and difficult to detect from a given disjunctive program. Interestingly, the oracle is better visible in native implementations of disjunctive logic programs (Janhunen et al. 2006; Drescher et al. 2008) where two ASP solvers cooperate: one is responsible for *generating* model candidates and the other for *testing* the minimality of candidates. In such an architecture, a successful minimality test amounts to showing that a certain subprogram has no stable models.

In other formalisms, the second level of the PH is reached differently. For instance, *quantified Boolean formulas* (QBFs) (Stockmeyer and Meyer 1973), record the interface between existentially and universally quantified subtheories, intuitively corresponding to the generating and testing programs mentioned above, explicitly in the quantifier prefix of the theory. From a modelling perspective, on one hand, QBFs support the natural formalization of subproblems as subtheories and the quantifications introduced for variables essentially identify the oracles involved. On the other hand, logic programs also have some advantages over QBFs. Most prominently, they allow for the natural encodings of *inductive definitions*, not to forget about *default negation*, *aggregates* and *first-order features* available in logic programming. The rich high-level modelling languages such as ASP-Core-2 (Calimeri et al. 2013) offer a wide variety of primitives that are not available for QBFs and require substantial elaboration if expressed as part of a QBF.

In this paper, we present a novel logic programming–based modeling paradigm that combines the best features of ASP and QBF. We introduce the notion of a *combined logic program* which explicitly integrates a normal logic program as an oracle to another program. The semantics of combined programs is formalized as *stable-unstable models* whose roots can be recognized from earlier work of Eiter and Polleres (2006). Our design directly reflects the generate-and-test methodology discussed above, enabling one to encode problems up to the second level of the PH. Compared to disjunctive programs, our approach is thus closer to QBFs and if the same

design is applied recursively, our new formalism can be adapted to tackle problems arbitrarily high in the PH, in analogy to QBFs. We develop a proof-of-concept solver for our new formalism on top of the recently introduced solver SAT-TO-SAT (Janhunen et al. 2016), which is based on an architecture of two interacting, conflict-driven clause learning (CDCL) SAT solvers. The solver capable of searching for stable-unstable models is obtained using the methodology of Bogaerts et al. (2016a), who automatically translate a second-order specification, combined with data that represents the involved ground programs in a reified form, into a SAT-TO-SAT specification. The details of the solver architecture are hidden from the user so that a user experience similar to native ASP solvers is obtained, where the user inputs two logic programs in a familiar syntax and the solver produces answer sets.

The rest of this paper is structured as follows. In Section 2, we discuss related work in more detail. We recall some basic notions of logic programs in Section 3. Afterwards, in Section 4, we present our new logic programming methodology. We illustrate how it can be used to tackle some problems from the second level of the PH in Section 5. In Section 6, we show how our new formalism can be implemented on top of SAT-TO-SAT. We show how our formalism naturally extends beyond the second level of the PH in Section 7 and conclude the paper in Section 8.

2 Related Work

A fundamental technique to encode Σ_2^P-complete problems as disjunctive programs is known as *saturation*. The technique goes back to the Σ_2^P-completeness proof for the existence of stable models in the case of disjunctive programs (Eiter and Gottlob 1995). Although saturation can be applied in a very systematic fashion to some programs of interest, Eiter and Polleres (2006) identify the impossibility of having negation as a central limitation of oracles encoded by saturation, rendering the oracle call to a bare minimality check rather than showing that an oracle program has no stable models. This limitation can be partially circumvented using *meta-interpretation* (Eiter and Polleres 2006; Gebser et al. 2011), but these techniques do not necessarily decrease the *conceptual complexity* of disjunctive programming from the user's perspective.

The approach of Eiter and Polleres (2006) is perhaps most closely related to our work. They present a transformation of two *head-cycle free* (HCF) disjunctive logic programs $(\mathscr{P}_g, \mathscr{P}_t)$, where $\mathscr{P}_g$ and $\mathscr{P}_t$ form the *generating* and *testing* programs, into a disjunctive program $\mathscr{P}_c$. In our terminology, the *stable-unstable* models of the *combined program* $(\mathscr{P}_g, \mathscr{P}_t)$ are in one-to-one correspondence with the stable models of $\mathscr{P}_c$. Thus, their approach is based on essentially the same base definition. However, their transformation counts on meta-interpretation and $\mathscr{P}_c$ is encoded as a disjunctive meta program to capture the intended semantics of $(\mathscr{P}_g, \mathscr{P}_t)$. A similar meta-encoding can be obtained using the approach of Gebser et al. (2011), but stable-unstable semantics is not explicit in their work. Since these meta programming approaches use disjunctive logic programs as the back end formalism, they are inherently confined to the second level of the PH. Our approach, on the other hand, easily generalizes for the classes of the entire PH, as to be shown in Section 7. Moreover, when Eiter and Polleres (2006) translate $(\mathscr{P}_g, \mathscr{P}_t)$ into a disjunctive logic program the essential structural distinction between $\mathscr{P}_g$ and $\mathscr{P}_t$ is lost. Many disjunctive answer set solvers (Janhunen et al. 2006; Drescher et al. 2008) try to recover this interface due to their internal data structures. In our approach, the generate-and-test structure of the original problem is explicitly present in the input presented to the solver.

While meta programming can be viewed as a front-end to disjunctive logic programming, the goal of our work is to foster the idea of generate-and-test programs as a basis for a logic program-

ming methodology that complexity-wise covers the entire PH. In this paper, we present a proof-of-concept implementation based on the recursive SAT-TO-SAT solver architecture (Janhunen et al. 2016; Bogaerts et al. 2016b). It is reasonable to expect that such an architecture can be realized in the future using native ASP solvers as building blocks, too, thus eliminating the need for second-order interpretation.

Another formalization of a similar idea was worked out by Eiter et al. (1997), based on the theory of generalized quantifiers (Mostowski 1957; Lindström 1966). The semantics we propose for combined logic programs can be obtained as a special case of a (stratified) logic program with generalized quantifiers (Eiter et al. 1997). One important difference is that in our approach, the interaction between the two programs is fixed: one program serves as generator and the second as a tester program. The approach of Eiter et al. (1997) is more general in the sense that it allows for other types of interaction as well. The price to pay for this generality is that the interaction between programs needs to be specified explicitly by users, resulting in a more error-prone modelling process. Moreover, in our approach, the input expected from the user is a set of source files in a familiar syntax (ASP-Core-2), requiring no syntactic extension for quantification.

3 Preliminaries: Logic Programming

In this section, we recall some preliminaries from logic programming. The new semantics is only formulated for propositional programs but, in practice, the users are not expected to write propositional programs. Instead, they are supposed to use grounders, such as the state-of-the-art grounder GRINGO, to transform first-order programs to propositional ones.

A *vocabulary* is a set of symbols, also called *atoms*; vocabularies are denoted by σ, τ. A *literal* is an atom or its negation. A *logic program* $\mathscr{P}$ over vocabulary σ is a set of *rules* r of form

$$h_1 \vee \cdots \vee h_l \leftarrow a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m. \tag{1}$$

where h_i's, a_i's, and b_i's are atoms in σ. We call $h_1 \vee \cdots \vee h_l$ the *head* of r, denoted $head(r)$, and $a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m$ the *body* of r, denoted $body(r)$. A program is *normal* (resp. *positive*) if $l = 1$ (resp. $m = 0$) for all rules in $\mathscr{P}$. If $n = m = 0$, we simply write $h_1 \vee \cdots \vee h_l$.

An interpretation I of a vocabulary σ is a subset of σ. An interpretation I is a *model* of a logic program $\mathscr{P}$ if, for all rules r in $\mathscr{P}$, whenever $body(r)$ is satisfied by I, so is $head(r)$. The *reduct* of $\mathscr{P}$ with respect to I, denoted $\mathscr{P}^I$, is the program that consists of rules $h_1 \vee \cdots \vee h_l \leftarrow a_1 \wedge \cdots \wedge a_n$ for all rules of the form (1) in $\mathscr{P}$ such that $b_i \notin I$ for all i. An interpretation I is a *stable model* of $\mathscr{P}$ if it is a $\subseteq$-minimal model of $\mathscr{P}^I$ (Gelfond and Lifschitz 1988).

Parameterized logic programs have been implicitly present in the literature for a long time, by assigning a meaning to *intensional* databases. They have been made explicit in various forms (Gelfond and Przymusinska 1996; Oikarinen and Janhunen 2006; Denecker and Vennekens 2007; Denecker et al. 2012). We briefly recall the basics. Assume that $\tau \subseteq \sigma$ and $\mathscr{P}$ is a logic program over σ such that no atoms from τ occur in the head of a rule in $\mathscr{P}$. We call I a *parameterized stable model* of $\mathscr{P}$ with respect to *parameters* τ if I is a stable model of $\mathscr{P} \cup (I \cap \tau)$. Parameters τ are also known as *external*, *open*, or *input atoms*.

From time to time, we use syntactic extensions such as choice rules, constraints, and cardinality atoms in this paper. A *cardinality atom* $m \leq \#\{l_1, \ldots, l_n\} \leq k$ (with $l_1, \ldots, l_n$ being literals and $m, k \in \mathbb{N}$) is satisfied by I if $m \leq \#\{i \mid l_i \in I\} \leq k$. A *choice rule* is a rule with a cardinality atom in the head. A *constraint* is a rule with an empty head. An interpretation I satisfies a constraint c if it does not satisfy $body(c)$. These language constructs can all be translated to normal rules

(Bomanson and Janhunen 2013). We also sometimes use the colon syntax $H : L$ for conditional literals as a way to succinctly specify a set of literals in the body of a rule or in a cardinality atom (Gebser et al. 2015).

4 Stable-Unstable Semantics

The design goal of our new formalism is to isolate the logic program that is acting as an oracle for another program. Thus, we would like to find a stable model I for a program while showing the *non-existence* of stable models for the oracle program given I. Following this intuition, we formalize the pair $(\mathscr{P}_g, \mathscr{P}_t)$ of a *generating* program $\mathscr{P}_g$ and a *testing* program $\mathscr{P}_t$ as follows.[1]

Definition 4.1 (*Combined logic program*)
A *combined logic program* is pair $(\mathscr{P}_g, \mathscr{P}_t)$ of normal logic programs $\mathscr{P}_g$ and $\mathscr{P}_t$ with vocabularies σ_g and σ_t such that $\mathscr{P}_g$ is parameterized by $\tau_g \subseteq \sigma_g$ and $\mathscr{P}_t$ is parameterized by $\sigma_g \cap \sigma_t$.

The vocabulary of the program $(\mathscr{P}_g, \mathscr{P}_t)$ is σ_g; it consists of all symbols that are "visible" to the outside. Symbols in $\sigma_t \setminus \sigma_g$ are considered to be *quantified internally*. The use of normal programs in the definition of combined logic programs, or *combined programs* for short, is a design decision aiming at programs that are easily understandable (compared to, for instance, disjunctive programs with head-cycles). In principle, our theory also works when replacing normal programs with another class of programs. Our next objective is to define the semantics of combined programs which should not be a surprise given the above intuitions.

Definition 4.2 (*Stable-unstable model*)
Given a combined program $(\mathscr{P}_g, \mathscr{P}_t)$ with vocabularies σ_g and σ_t, a σ_g-interpretation I is a *stable-unstable model* of $(\mathscr{P}_g, \mathscr{P}_t)$ if the following two conditions hold:

1. I is a parameterized stable model of $\mathscr{P}_g$ with respect to τ_g (the parameters of $\mathscr{P}_g$) and
2. there is no parameterized stable model J of $\mathscr{P}_t$ that coincides with I on $\sigma_t \cap \sigma_g$ (i.e., such that $I \cap \sigma_t = J \cap \sigma_g$).

The fact that a σ_g-interpretation I is a stable-unstable model of $(\mathscr{P}_g, \mathscr{P}_t)$ is denoted $I \models_{su} (\mathscr{P}_g, \mathscr{P}_t)$. Note that the testing program stands for the *non-existence* of stable models. If $\sigma_g \cap \sigma_t \neq \emptyset$, the programs truly interact. Otherwise, we call $(\mathscr{P}_g, \mathscr{P}_t)$ *independent*.

Example 4.3
Let $\mathscr{P}_1 = \{0 \leq \#\{c\} \leq 1.\ \leftarrow c \wedge d.\ \leftarrow \neg c \wedge b.\}$ and $\mathscr{P}_2 = \{0 \leq \#\{a\} \leq 1.\ b \leftarrow a.\}$ where $\mathscr{P}_1$ has vocabulary $\sigma_1 = \{c, b, d\}$ and parameters $\tau_1 = \{b, d\}$, and $\mathscr{P}_2$ has vocabulary $\sigma_2 = \{a, b, d\}$ and parameters $\tau_2 = \{d\}$. The stable models of $\mathscr{P}_1$ and $\mathscr{P}_2$ are, respectively, $\{\{d\}, \{b, c\}, \{\}, \{c\}\}$ and $\{\{d, a, b\}, \{d\}, \{a, b\}, \{\}\}$. Notice that $\tau_1 = \sigma_1 \cap \sigma_2$. The combined program $(\mathscr{P}_2, \mathscr{P}_1)$ has parameters τ_2 and has only one stable-unstable model $\{d, a, b\}$ since all other stable models of $\mathscr{P}_2$ coincide with a stable model of $\mathscr{P}_1$ on τ_1.

Theorem 4.4
Deciding the existence of a stable-unstable model for a *finite* combined program $(\mathscr{P}_g, \mathscr{P}_t)$ is Σ_2^P-complete in general, and D^P-complete for *independent* combined programs.

[1] The terminology goes back to GNT, one of the early solvers developed for disjunctive programs (Janhunen et al. 2006).

Proof
The theorem is a straightforward consequence of known complexity results. The membership in Σ_2^P follows directly from the definition of Σ_2^P and the fact that deciding whether a normal logic program has a stable model is NP-complete (Marek and Truszczyński 1999). For hardness in the general case, we recall that Janhunen et al. (2006) have shown that any disjunctive logic program $\mathscr{P}$ can be represented as a pair of normal programs $(\mathscr{P}_g, \mathscr{P}_t)$ whose stable-unstable models essentially capture the stable models of $\mathscr{P}$.

In the case of an independent input $(\mathscr{P}_g, \mathscr{P}_t)$, the decision problem conjoins an NP-complete problem (showing that $\mathscr{P}_g$ has a stable model) and a co-NP-complete problem (showing that $\mathscr{P}_t$ has no stable models). Thus, membership in D^p is immediate. The hardness is implied by Niemelä's reduction (1999) that translates a set of clauses C into a normal logic program $N(C)$, when applied to instances of the D^P-complete SAT-UNSAT problem. □

Example 4.5
Any $\exists\forall$QBF of the form $\exists\vec{x}\forall\vec{y} : \varphi$ with φ a Boolean formula in DNF can be encoded as a combined program as follows. Let $\mathscr{P}_g$ be a logic program that expresses the choice of a truth value for every variable x in $\vec{x}$ using two normal rules $x \leftarrow \neg x'$ and $x' \leftarrow \neg x$ where x' is new. Also, let $\mathscr{P}_t$ be a logic program that similarly chooses truth values for every y in $\vec{y}$ and contains for each conjunction $l_1 \wedge \cdots \wedge l_n$ in the DNF φ a rule $\mathsf{sat} \leftarrow l_1 \wedge \cdots \wedge l_n$ where sat is a new atom that is true if φ is satisfied. Moreover, let $\mathscr{P}_t$ have the rule $\mathsf{fail} \leftarrow \neg\mathsf{fail} \wedge \mathsf{sat}$. This rule enforces that sat must be false in models of $\mathscr{P}_t$. As such $\mathscr{P}_t$ corresponds to the sentence $\exists\vec{y} : \neg\varphi$. Since $\neg\exists\vec{y} : \neg\varphi \equiv \forall\vec{y} : \varphi$, we thus find that $\exists\vec{x}\forall\vec{y} : \varphi$ is valid iff $(\mathscr{P}_g, \mathscr{P}_t)$ has a stable-unstable model.

It follows from Theorem 4.4 that the theoretical expressiveness of combined programs equals that of $\exists\forall$QBFs. There are, however, several reasons why one would prefer combined programs. Firstly, logic programs are equipped with rich, high-level, first-order modeling languages. Secondly, logic programs allow for natural encodings of *inductive definitions*. These reasons are comparable to the advantages of logic programs on the first level of the hierarchy in contrast with pure SAT. For instance, the former can naturally express reachability in digraphs, while the latter requires a non-trivial encoding, which is non-linear in the size of the input graph. The advantage of combined programs over $\exists\forall$QBFs is analogous when solving problems on the second level. The expressive power of inductive definitions and the high-level modeling language are available both in $\mathscr{P}_g$ and in $\mathscr{P}_t$. We exploit this when presenting examples in the next section.

5 Applications

The goal of this section is to present some applications of stable-unstable programming. We will focus on *modelling* aspects, i.e., how certain application problems can be represented. The programs to be presented are non-ground (and may also use some constructs present in ASP-Core-2, such as arithmetic) while the stable-unstable semantics was formulated for ground programs only. However, in practice, input programs are first grounded and thus covered by the propositional semantics. Hence, the user has all high-level primitives of ASP at his/her disposal.

5.1 Winning Strategies for Parity Games

Parity games, to be detailed below, have been studied intensively in computer aided verification since they correspond to model checking problems in the μ-calculus. We show how to represent parity game instances as combined programs. A parity game consists of a finite graph

$G = (V; A, v_0, V_\exists, V_\forall, \Omega)$, where V is a set of nodes, A a set of arcs, $v_0 \in V$ an initial node, $V_\exists$ and $V_\forall$ partition V into two subsets, respectively owned by an existential and a universal player, and $\Omega : V \to \mathbb{N}$ assigns a priority to each node. All nodes are assumed to have at least one outgoing arc. A *play* in a parity game is an infinite path in G starting from v_0. We denote such a play by a function $\pi : \mathbb{N} \to V$. A play π is generated by setting $\pi(0) = v_0$ and, at each step i, asking the player who owns node $\pi(i)$ to choose a following node $\pi(i+1)$ such that $(\pi(i), \pi(i+1)) \in A$. The existential player wins if $\min\{\Omega(v) \mid v \text{ appears infinitely often in } \pi\}$ is an even number. Otherwise, the universal player wins. A *strategy* σ_x for a player $x \in \{\exists, \forall\}$ is a function that takes a finite path $(v_0, v_1, \cdots, v_n)$ in G with $v_n \in V_x$ and returns a node v_{n+1} such that $(v_n, v_{n+1}) \in A$. A play π conforms to σ_x if, whenever $\pi(n) \in V_x$, it holds that $\sigma_x(\pi(0), \pi(1), \cdots, \pi(n)) = \pi(n+1)$. A strategy σ_x is a *winning strategy* for x if x wins all plays that conform to σ_x. A strategy σ_x is called *positional* if $\sigma(v_0, v_1, \cdots, v_n)$ only depends on v_n. Two important properties of parity games are that (i) exactly one player has a winning strategy and (ii) a player has a winning strategy if and only if it has a positional winning strategy (Emerson and Jutla 1991).

Using the above properties, we provide an intuitive axiomatization $(\mathscr{P}_g, \mathscr{P}_t)$ to capture winning strategies of the existential player in a given parity game. The generator program $\mathscr{P}_g$ is simple: it guesses a (positional) strategy (called eStrategy) for player $\exists$. The test program is more involved. It guesses a positional strategy (called uStrategy) for player $\forall$ and accepts uStrategy if it wins against eStrategy. To perform the acceptance test, we define the set inf of nodes that appear infinitely often on the unique play that conforms to both strategies. We reject uStrategy if the minimum priority of nodes in inf is an even number. Hence, $(\mathscr{P}_g, \mathscr{P}_t)$ has a stable-unstable model if $\mathscr{P}_g$ can find a positional strategy σ for the existential player such that $\mathscr{P}_t$ cannot find any positional strategy to defeat σ. The entire programs can be found below.

$$\mathscr{P}_g = \left\{ \; 1 \leq \#\{\mathsf{eStrategy}(X,Y) : \mathsf{arc}(X,Y)\} \leq 1 \leftarrow \mathsf{existNode}(X). \; \right\}$$

$$\mathscr{P}_t = \left\{ \begin{array}{l} 1 \leq \#\{\mathsf{uStrategy}(X,Y) : \mathsf{arc}(X,Y)\} \leq 1 \leftarrow \mathsf{univNode}(X). \\ \mathsf{next}(X,Y) \leftarrow \mathsf{eStrategy}(X,Y). \\ \mathsf{next}(X,Y) \leftarrow \mathsf{uStrategy}(X,Y). \\ \mathsf{r}(v_0). \qquad \mathsf{r}(Y) \leftarrow \mathsf{r}(X) \wedge \mathsf{next}(X,Y). \\ \mathsf{inf}(v_0) \leftarrow \mathsf{next}(X,v_0) \wedge \mathsf{r}(X). \\ \mathsf{inf}(X) \leftarrow \mathsf{next}(Y,X) \wedge \mathsf{next}(Z,X) \wedge \mathsf{r}(Y) \wedge \mathsf{r}(Z) \wedge Y \neq Z. \\ \mathsf{inf}(Y) \leftarrow \mathsf{inf}(X) \wedge \mathsf{next}(X,Y). \\ \mathsf{infNum}(N) \leftarrow \mathsf{omega}(X,N) \wedge \mathsf{inf}(X). \\ \mathsf{num}(N) \leftarrow \mathsf{omega}(X,N). \\ \mathsf{minNum}(N) \leftarrow \mathsf{num}(N) \wedge N \leq M : \mathsf{num}(M). \\ \mathsf{nextNum}(N,M) \leftarrow \mathsf{num}(N) \wedge \mathsf{num}(M) \wedge M \leq P : \mathsf{num}(P) : N < P. \\ \mathsf{nonMin}(M) \leftarrow \mathsf{infNum}(N) \wedge \mathsf{nextNum}(N,M). \\ \mathsf{nonMin}(M) \leftarrow \mathsf{nonMin}(N) \wedge \mathsf{nextNum}(N,M). \\ \mathsf{min}(N) \leftarrow \mathsf{infNum}(N) \wedge \neg\mathsf{nonMin}(N). \\ \leftarrow \mathsf{min}(N) \wedge N \equiv 0 \; (\mathrm{mod}2). \end{array} \right\}$$

Deciding if a parity game has a winning strategy for the existential player has been encoded in difference logic and in SAT (Heljanko et al. 2012). We see two reasons why our encoding as a combined program can still be of interest. First, it is an intuitive encoding that corresponds directly to the problem definition. Second, to the best of our knowledge, it is the first encoding whose size is linear in the size of the graph, i.e., $\mathscr{O}(|V| + |A|)$. The existing difference logic encoding has size $\mathscr{O}(|V|^2 + |A|)$ and the existing SAT encoding (which is developed on top of the difference logic encoding) has size $\mathscr{O}(|V|^2 \times \log|V| + |A|)$ (Heljanko et al. 2012).

5.2 Conformant Planning

(Classical) planning is the task of generating a plan (i.e., a sequence of actions) that realizes a certain goal given a complete description of the world. *Conformant planning* is the task of generating a plan that reaches a given goal given a partial description of the world (certain facts about the initial state and/or actions' effects are unknown). In this section, we focus on *deterministic* conformant planning problems: problems where the state of the world at any time is completely determined by the initial state and the actions taken. It is well-known that deciding if a conformant plan exists is a Σ_2^P-complete decision problem.

To encode conformant planning problems in our formalism, we assume a vocabulary $\sigma = \sigma_a \cup \sigma_w \cup \sigma_i$ is given. Here, σ_a, σ_w and σ_i represent a sequence of actions, the state of the world over time, and the initial state of the world, respectively. We also assume that σ_w contains an atom goal with intended interpretation that the goal of the planning problem is reached at some time. Furthermore, we assume that σ_i is partitioned in σ_{unc} and σ_c, where σ_{unc} are the atoms subject to uncertainty (to which our plan should be conformant). Let $\mathscr{P}_{ca}$ be a logic program containing a rule $0 \leq \#\{\mathsf{a}\} \leq 1$ for each $\mathsf{a} \in \sigma_a$. Intuitively, the program $\mathscr{P}_{ca}$ *guesses* a sequence of actions. Similarly, let us introduce a program $\mathscr{P}_{unc}$ containing a rule $0 \leq \#\{\mathsf{u}\} \leq 1$ for each $\mathsf{u} \in \sigma_{unc}$. Furthermore, we assume the availability of a program $\mathscr{P}_w$ that defines the atoms in σ_w (including goal) deterministically in terms of σ_a and σ_i. Also, let $\mathscr{P}_{pa}$ be a program that contains a rule $\mathsf{fail} \leftarrow \mathsf{a} \wedge \neg \mathsf{p}$ for each $\mathsf{a} \in \sigma_a$, $\mathsf{p} \in \sigma_w$ such that p is a precondition of a. With these building blocks, we can easily encode conformant planning as a combined program

$$(\mathscr{P}_{ca}, \mathscr{P}_w \cup \mathscr{P}_{pa} \cup \mathscr{P}_{unc} \cup \{\leftarrow \mathsf{goal} \wedge \neg \mathsf{fail}\}).$$

This program is parameterized by σ_c. To see that it encodes the conformant planning problem, we notice that stable-unstable models of this program are stable models of $\mathscr{P}_{ca}$, i.e., sequences of actions. Furthermore, models of the testing program are interpretations of the atoms in σ_{unc} such that in this world, either one of the preconditions on the actions is not satisfied or the goal is not reached. I.e., models of the testing program amount to showing that the sequence of actions is *not* a conformant plan. The stable-unstable semantics dictates that there can be no such counterexample.

In the above, we described $\mathscr{P}_w$ and $\mathscr{P}_{pa}$ only informally since these components have already been worked out in the literature. More precisely, many classical planning encodings use exactly those components, combining them to a program of the form

$$\mathscr{P} = \mathscr{P}_{ca} \cup \mathscr{P}_w \cup \mathscr{P}_{pa} \cup \{\leftarrow \neg \mathsf{goal}. \quad \leftarrow \mathsf{fail}.\}.$$

These components (or very similar) are used for instance by Lifschitz (1999), Leone et al. (2001), and by Bogaerts et al. (2014). This illustrates that our encoding of conformant planning stays very close to the existing encodings of classical planning problems in ASP. On the other hand, native conformant planning encodings in ASP are often based on saturation (Leone et al. 2001). After applying saturation, it is very hard to spot the original components.

5.3 Points of No Return: A Generic Problem Combining Logic and Graphs

We now present a generic problem that connects graphs with logic. Let $G = (V, A, s)$ be a directed multi-graph: V is a set of nodes, $s \in V$ is an initial node and A is a set of arcs labeled with Boolean formulas. We use $a : u \xrightarrow{\phi} v$ to denote that a is an arc from u to v labeled with ϕ. There may be

multiple arcs between u and v with different labels. We call a node $v \in V$ a *point of no return* if (i) G contains a path $s = v_0 \xrightarrow{\phi_1} v_1 \xrightarrow{\phi_2} \ldots \xrightarrow{\phi_n} v_n = v$ such that $\phi_1 \wedge \ldots \wedge \phi_n$ is satisfiable and (ii) the preceding path in G cannot be extended with a path $v = v_n \xrightarrow{\phi_{n+1}} v_{n+1} \xrightarrow{\phi_{n+2}} \ldots \xrightarrow{\phi_{n+m}} v_{n+m} = s$ such that $\phi_1 \wedge \cdots \wedge \phi_n \wedge \phi_{n+1} \wedge \ldots \wedge \phi_{n+m}$ is satisfiable. Thus, points of no return are nodes v that can be reached from s in a way that makes s unreachable from v (i.e., reaching s back from v would violate a constraint of the path from s to v).

Proposition 5.1

Given a finite labeled graph $G = (V, A, s)$ as above and a node $v \in V$, it is a Σ_2^P-complete problem to decide if v is a point of no return.

Proof

Membership in Σ_2^P is obvious. We present a reduction from $\exists\forall$QBF to support hardness. Consider an $\exists\forall$QBF formula $\exists x_1 \cdots \exists x_n \forall y_1 \cdots \forall y_m \phi$. This formula is equivalent to

$$\exists x_1 \cdots \exists x_n \neg \exists y_1 \cdots \exists y_m \neg \phi.$$

Now, construct a graph G with nodes $v_0, v_1, \ldots, v_n, v_{n+1}, \ldots, v_{n+m+1}$ and following labeled arcs:

$$\begin{array}{ll} v_{i-1} \xrightarrow{x_i} v_i \text{ and } v_{i-1} \xrightarrow{\neg x_i} v_i & (\text{for } 1 \leq i \leq n), \\ v_{n+j} \xrightarrow{y_j} v_{n+j+1} \text{ and } v_{n+j} \xrightarrow{\neg y_j} v_{n+j+1} & (\text{for } 1 \leq j \leq m), \\ v_n \xrightarrow{\neg \phi} v_{n+1} \text{ and } v_{n+m+1} \xrightarrow{\top} v_0. & \end{array}$$

Observe that, setting $s = v_0$ and $v = v_n$, we have that v is a point of no return if and only if $\exists x_1 \cdots \exists x_n \forall y_1 \cdots \forall y_m \phi$ is valid. □

To model the problem of checking whether a node is point of no return as a combined program, we assume that each arc is labeled by a *literal* and that there is at most one arc between every two nodes. Our programs easily generalize to the general case. To allow for multiple arcs between two nodes, it suffices to introduce explicit identifiers for arcs. To allow more complex labeling formulas, we can introduce Tseitin predicates for subformulas and use standard meta-interpreter approaches to model the truth of such a formula; see for instance (Gebser et al. 2011, Section 3).

We use unary predicates init and ponr to respectively interpret the initial node s and the point of no return v. Herbrand functions pos and neg map atoms (represented as constants) to literals. The predicate $\mathsf{arc}(X,Y,L)$ holds if there is an arc between nodes X and Y labeled with literal L. In $\mathscr{P}_g$ (and $\mathscr{P}_t$), we use predicates pick_g (and pick_t) such that $\mathsf{pick}_g(X,Y)$ ($\mathsf{pick}_t(X,Y)$) holds if the arc from X to Y is chosen in the path $v_0 \to \cdots \to v_n$ (the path $v_n \to \cdots \to v_{n+m}$ respectively). The programs contain constraints ensuring that the selected edges indeed form paths from s to v (respectively from v to s), using an additional predicate r_g (r_t) and that the formulas associated to the respective paths are satisfiable. Thus, $\mathscr{P}_g$ encodes that there exists a path from s to v and $\mathscr{P}_t$ encodes that this path can be extended to a cycle back to s. As such, the combined program indeed models that v is a point of no return. The entire combined program can be found below.

$$\mathscr{P}_g = \left\{\begin{array}{l} 0 \leq \#\{\mathsf{pick}_g(X,Y)\} \leq 1 \leftarrow \mathsf{arc}(X,Y,L). \\ \leftarrow \mathsf{pick}_g(X,Y) \wedge \mathsf{pick}_g(X',Y') \\ \quad \wedge \mathsf{arc}(X,Y,\mathsf{pos}(A)) \wedge \mathsf{arc}(X',Y',\mathsf{neg}(A)). \\ \mathsf{r}_g(X) \leftarrow \mathsf{init}(X). \\ \mathsf{r}_g(Y) \leftarrow \mathsf{r}_g(X) \wedge \mathsf{pick}_g(X,Y). \\ \leftarrow \neg\mathsf{r}_g(X) \wedge \mathsf{pick}_g(X,Y). \\ \leftarrow \mathsf{ponr}(X) \wedge \neg\mathsf{r}_g(X). \\ \leftarrow \mathsf{ponr}(X) \wedge \mathsf{pick}_g(X,Y). \\ \leftarrow \mathsf{pick}_g(X,Y) \wedge \mathsf{pick}_g(X,Z) \wedge Y \neq Z. \\ \leftarrow \mathsf{pick}_g(X,Y) \wedge \mathsf{pick}_g(Z,Y) \wedge X \neq Z. \end{array}\right\}$$

$$\mathscr{P}_t = \left\{\begin{array}{l} 0 \leq \#\{\mathsf{pick}_t(X,Y)\} \leq 1 \leftarrow \mathsf{arc}(X,Y,L). \\ \mathsf{pick}(X,Y) \leftarrow \mathsf{pick}_t(X,Y). \\ \mathsf{pick}(X,Y) \leftarrow \mathsf{pick}_g(X,Y). \\ \leftarrow \mathsf{pick}(X,Y) \wedge \mathsf{pick}(X',Y') \wedge \\ \quad \mathsf{arc}(X,Y,\mathsf{pos}(A)) \wedge \mathsf{arc}(X',Y',\mathsf{neg}(A)). \\ \mathsf{r}_t(X) \leftarrow \mathsf{ponr}(X). \\ \mathsf{r}_t(Y) \leftarrow \mathsf{r}_t(X) \wedge \mathsf{pick}_t(X,Y). \\ \leftarrow \neg\mathsf{r}_t(X) \wedge \mathsf{pick}_t(X,Y). \\ \leftarrow \mathsf{init}(X) \wedge \neg\mathsf{r}_t(X). \\ \leftarrow \mathsf{init}(X) \wedge \mathsf{pick}_t(X,Y). \\ \leftarrow \mathsf{pick}_t(X,Y) \wedge \mathsf{pick}_t(X,Z) \wedge Y \neq Z. \\ \leftarrow \mathsf{pick}_t(X,Y) \wedge \mathsf{pick}_t(Z,Y) \wedge X \neq Z. \end{array}\right\}$$

6 Implementation

Next, we present a prototype implementation of a solver for the stable-unstable semantics.

6.1 Preliminaries: SAT-TO-SAT

We assume familiarity with the basics of second-order logic (SO). Our implementation is based on a recently introduced solver, called SAT-TO-SAT (Janhunen et al. 2016). The SAT-TO-SAT architecture combines multiple SAT solvers to tackle problems from any level of the PH, essentially acting like a QBF solver (Bogaerts et al. 2016b). We do not give details on the inner workings of SAT-TO-SAT, but rather refer the reader to the original papers for details. What matters for the current paper is that Bogaerts et al. (2016a) presented a high-level (second-order) interface to SAT-TO-SAT. The idea is that in order to obtain a solver for a new paradigm, it suffices to give a second-order theory that *describes* the semantics of the formalism declaratively. Bogaerts et al. showed, e.g., how to obtain a solver for (disjunctive) logic programming using this idea.

Following Bogaerts et al. (2016a), we describe a logic program by means of predicates r, a, p, h, pb and nb with intended interpretation that $\mathsf{r}(R)$ holds for all rules R, $\mathsf{a}(A)$ holds for all atoms A, $\mathsf{p}(A)$ holds for all parameters, $\mathsf{h}(R,H)$ means that H is an atom in the head of rule R, $\mathsf{pb}(R,A)$ that A is a positive literal in the body of R and $\mathsf{nb}(R,B)$ that B is the atom of a negative literal in the body of R. With this vocabulary, augmented with a predicate i with intended meaning that $\mathsf{i}(A)$ holds for all atoms A true in some interpretation, we describe the parameterized stable semantics for disjunctive logic programs with the theory T_{SM}:

$$\left\{\begin{array}{l} \forall A : \mathsf{i}(A) \Rightarrow \mathsf{a}(A). \\ \forall R : \mathsf{r}(R) \Rightarrow \big((\forall A : \mathsf{pb}(R,A) \Rightarrow \mathsf{i}(A)) \wedge (\forall B : \mathsf{nb}(R,B) \Rightarrow \neg\mathsf{i}(B)) \Rightarrow \\ \qquad\qquad \exists H : \mathsf{h}(R,H) \wedge \mathsf{i}(H)\big). \\ \neg\exists \mathsf{i}' : \\ \quad (\forall A : \mathsf{i}'(A) \Rightarrow \mathsf{i}(A)) \wedge (\exists A : \mathsf{i}(A) \wedge \neg\mathsf{i}'(A)) \wedge (\forall A : \mathsf{p}(A) \Rightarrow (\mathsf{i}'(A) \Leftrightarrow \mathsf{i}(A))) \wedge \\ \quad \forall R : \mathsf{r}(R) \Rightarrow \big((\forall A : \mathsf{pb}(R,A) \Rightarrow \mathsf{i}'(A)) \wedge \\ \qquad\qquad (\forall B : \mathsf{nb}(R,B) \Rightarrow \neg\mathsf{i}(B)) \Rightarrow \exists H : \mathsf{h}(R,H) \wedge \mathsf{i}'(H)\big). \end{array}\right\}$$

The first part of this theory expresses that i is interpreted as a model of $\mathscr{P}$: the constraint

$\mathsf{i}(A) \Rightarrow \mathsf{a}(A)$ expresses that the interpretation is a subset of the vocabulary and the second constraint expresses that whenever the body of a rule is satisfied in i, so is at least one of its head atoms. The constraint $\neg\exists \mathsf{i}' \ldots$ expresses that I is $\subseteq$-minimal: there cannot be an interpretation $\mathsf{i}' \subsetneq \mathsf{i}$ that agrees with i on the parameters and that is a model of the reduct of $\mathscr{P}$ with respect to i. In other words, whenever i' satisfies all positive literals in the body of a rule R and i satisfies all negative literals in the body of R, i' must also satisfy some atom in the head of R.

Theorem 6.1 (*Theorem 4.1 of (Bogaerts et al. 2016a)*)
Let $\mathscr{P}$ be a (disjunctive) logic program and I an interpretation that interprets $\{\mathsf{a}, \mathsf{r}, \mathsf{p}, \mathsf{pb}, \mathsf{nb}, \mathsf{h}\}$ according to $\mathscr{P}$. Then, $I \models T_{SM}$ if and only if i^I is a parameterized stable model of $\mathscr{P}$.

From Theorem 6.1, it follows that feeding T_{SM} to SAT-TO-SAT results in a solver for disjunctive logic programs. The same theory also works for normal logic programs.

6.2 An Implementation on Top of SAT-TO-SAT

In order to obtain a solver for our new paradigm in the spirit of Bogaerts et al. (2016a), we need to provide a second order specification of our semantics. A first observation is that we can reuse the theory T_{SM} from the previous section, both to enforce that I is a stable model of $\mathscr{P}_g$ and that there exists no stable model of $\mathscr{P}_t$ that coincides with I on the shared vocabulary. When translating the definition of stable-unstable models to second-order logic, we obtain the following theory

$$T_{SU} = \left\{ \begin{array}{l} T_{SM}[\mathsf{r}/\mathsf{r}_g, \mathsf{a}/\mathsf{a}_g, \mathsf{p}/\mathsf{p}_g, \mathsf{h}/\mathsf{h}_g, \mathsf{pb}/\mathsf{pb}_g, \mathsf{nb}/\mathsf{nb}_g]. \\ \neg\exists \mathsf{i}_t : T_{SM}[\mathsf{r}/\mathsf{r}_t, \mathsf{a}/\mathsf{a}_t, \mathsf{h}/\mathsf{h}_t, \mathsf{pb}/\mathsf{pb}_t, \mathsf{nb}/\mathsf{nb}_t, \mathsf{i}/\mathsf{i}_t, \mathsf{p}/\mathsf{p}_t] \\ \qquad\qquad \wedge (\forall A : \mathsf{a}_g(A) \wedge \mathsf{a}_t(A) \Rightarrow (\mathsf{i}(A) \Leftrightarrow \mathsf{i}_t(A))). \end{array} \right\},$$

where $T_{SM}[\mathsf{r}/\mathsf{r}_g]$ abbreviates a second-order theory obtained from T_{SM} by replacing all free occurrences of r by r_g.

Theorem 6.2
Let $(\mathscr{P}_g, \mathscr{P}_t)$ be a combined logic program and I an interpretation that interprets $\{\mathsf{a}_g, \mathsf{r}_g, \mathsf{p}_g, \mathsf{pb}_g, \mathsf{nb}_g, \mathsf{h}_g\}$ according to $\mathscr{P}_g$ and $\{\mathsf{a}_t, \mathsf{r}_t, \mathsf{p}_t, \mathsf{pb}_t, \mathsf{nb}_t, \mathsf{h}_t\}$ according to $\mathscr{P}_t$. Then, $I \models T_{SU}$ if and only if i^I is a stable-unstable model of $(\mathscr{P}_g, \mathscr{P}_t)$.

Proof
Theorem 6.1 ensures that the first sentence of this theory is equivalent with the condition of i^I being a stable model of $\mathscr{P}_g$. Also, the second sentence states that one cannot have an interpretation i_t that coincides with i^I on shared atoms (those that are in both a_g and a_t) and is a stable model of $\mathscr{P}_t$. This is exactly the definition of the stable-unstable semantics. □

Providing an ASCII representation of T_{SU} to the second-order interface of SAT-TO-SAT immediately results in a solver that generates stable-unstable models of a combined logic program. Our implementation, which is available online[2], consists only of the second-order theory above and some marshaling (to support ASP-Core-2 format and to exploit the symbol table to identify which atoms from different programs are actually the same). The overall workflow of our tool is as follows. We take, as input, three logic programs: $\mathscr{P}_g$ (a non-ground generate program), $\mathscr{P}_t$ (a non-ground test program) and $\mathscr{P}_i$ (an instance). We then use GRINGO (Gebser et al. 2007)

[2] `http://research.ics.aalto.fi/software/sat/sat-to-sat/so2grounder.shtml`.

to ground $\mathscr{P}_g \cup \mathscr{P}_i$ and $\mathscr{P}_t \cup \mathscr{P}_i$. Next, we interpret a_x, r_x, p_x, pb_x, nb_x and h_x (for $x \in \{g,t\}$) according to the reified representation of the two resulting ground programs. Such an interpretation is fed to SAT-TO-SAT along with the ASCII representation of T_{SU}; SAT-TO-SAT uses these to compute stable-unstable models of the original combined program $(\mathscr{P}_g \cup \mathscr{P}_i, \mathscr{P}_t \cup \mathscr{P}_i)$.

The implementation described above is proof-of-concept by nature and we plan to implement this technique natively on top of the CLASP solver (Drescher et al. 2008; Gebser et al. 2015). In spite of its prototypical nature, the current implementation is based on a state-of-the-art architecture shared by many QBF solvers and thus expected to perform reasonably well. This is especially the case when we go beyond the complexity class Σ_2^P in the next section.

7 Beyond Σ_2^P with Normal Logic Programs

In this section, we show how the ideas of this paper generalize to capture the entire PH. To this end, the definition of a combined logic program is turned into a recursive definition of *k*-combined programs where the parameter $k \geq 1$ reflects the *depth* of the combination.

Definition 7.1 (*k-combined program*)
1. For $k = 1$, a 1-*combined program* is defined as a normal program $\mathscr{P}$ over a vocabulary σ, parameterized by a vocabulary $\tau \subseteq \sigma$.
2. For $k > 1$, a *k-combined* program is a pair $(\mathscr{P}, \mathscr{C})$ where $\mathscr{P}$ is a normal program over a vocabulary σ, parameterized by a vocabulary $\tau \subseteq \sigma$ and $\mathscr{C}$ is a $(k-1)$-combined program over a vocabulary σ', parameterized by $\sigma \cap \sigma'$.

Note that *combined programs* (Definition 4.1) directly correspond to *k*-combined programs with $k = 2$. Similarly, the semantics of *k*-combined programs also directly generalizes Definition 4.2:

Definition 7.2 (*Stable-unstable models for k-combined programs*)
A stable model I of $\mathscr{P}$ is also called a *stable-unstable* model of a 1-combined program $\mathscr{P}$. Let $(\mathscr{P}, \mathscr{C})$ be a *k*-combined program with $k > 1$ over a vocabulary σ, parameterized by $\tau \subseteq \sigma$, where $\mathscr{C}$ has vocabulary σ'. A σ-interpretation I is a *stable-unstable model* of $(\mathscr{P}, \mathscr{C})$, if
1. I is a parameterized stable model of $\mathscr{P}$ and
2. there is no stable-unstable model J of $\mathscr{C}$ such that $I \cap \sigma' = J \cap \sigma$.

Example 7.3 (*Example 4.3 continued*)
Consider program $\mathscr{P}_3 = \{e \leftarrow e.\ d \leftarrow e.\}$ over vocabulary $\sigma_3 = \{d, e\}$. Program $\mathscr{P}_3$ has one stable model, namely $\emptyset$. This model is also a stable-unstable model of the 3-combined program $(\mathscr{P}_3, (\mathscr{P}_2, \mathscr{P}_1))$ since it does not coincide with a stable-unstable model of $(\mathscr{P}_2, \mathscr{P}_1)$ on $\sigma_3 \cap \sigma_2 = \{d\}$.

The complexity of deciding whether a *k*-combined program $(\mathscr{P}, \mathscr{C})$ has a stable-unstable model depends on the depth *k* of the combination.

Theorem 7.4
It is Σ_k^P-complete to decide if a finite *k*-combined program has a stable-unstable model.

Proof sketch.
The case $k = 1$ follows from the results of Marek and Truszczyński (1999) and Theorem 4.4 corresponds to $k = 2$. Using either one as the base case, it can be proven inductively that the decision problem in question is NP-complete assuming the availability of an oracle from the class Σ^P_{k-1}, effectively a $(k-1)$-combined program in our constructions. Thus, steps in recursion depth match with the levels of the PH (in analogy to the number of quantifier alternations in QBFs). □

8 Conclusion

In this paper, we propose *combined logic programs* subject to the *stable-unstable semantics* as an alternative paradigm to disjunctive logic programs for programming on the second level of the polynomial hierarchy. We deploy *normal* logic programs as the base syntax for combined programs, but other equally complex classes can be exploited analogously. Our methodology surpasses the need for saturation and meta-interpretation techniques that have previously been used to encode oracles within disjunctive logic programs. The use of the new paradigm is illustrated in terms of application problems and we also present a proof-of-concept implementation on top of the solver SAT-TO-SAT. Moreover, we show how combined programs provide a gateway to programming on any level k of the polynomial hierarchy with normal logic programs using the idea of recursive combination to depth k. In this sense, our formalism can be seen as a hybrid between QBFs and logic programs, combining desirable features from both.

References

ANDRES, B., RAJARATNAM, D., SABUNCU, O., AND SCHAUB, T. 2015. Integrating ASP into ROS for reasoning in robots. In *Proceedings of the 13th International Conference on Logic Programming and Non-monotonic Reasoning, LPNMR 2015*. Lecture Notes in Computer Science, vol. 9345. Springer, Lexington, Kentucky, USA, 69–82.

BEN-ELIYAHU, R. AND DECHTER, R. 1994. Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell. 12*, 1–2, 53–87.

BOGAERTS, B., JANHUNEN, T., AND TASHARROFI, S. 2016a. Declarative solver development: Case studies. In *Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning, KR 2016*. AAAI Press, Cape Town, South Africa, 74–83.

BOGAERTS, B., JANHUNEN, T., AND TASHARROFI, S. 2016b. Solving QBF instances with nested SAT solvers. In *Proceedings of the AAAI-16 Workshop on Beyond NP*. AAAI Press, Phoenix, Arizona, 307–313.

BOGAERTS, B., JANSEN, J., BRUYNOOGHE, M., DE CAT, B., VENNEKENS, J., AND DENECKER, M. 2014. Simulating dynamic systems using linear time calculus theories. *TPLP 14*, 4–5 (7), 477–492.

BOMANSON, J. AND JANHUNEN, T. 2013. Normalizing cardinality rules using merging and sorting constructions. In *Proceedings of the 12th International Conference on Logic Programming and Non-monotonic Reasoning, LPNMR 2013*. Lecture Notes in Computer Science, vol. 8148. Springer, Corunna, Spain, 187–199.

BROOKS, D. R., ERDEM, E., ERDOGAN, S. T., MINETT, J. W., AND RINGE, D. 2007. Inferring phylogenetic trees using answer set programming. *J. Autom. Reasoning 39*, 4, 471–511.

BRUYNOOGHE, M., BLOCKEEL, H., BOGAERTS, B., DE CAT, B., DE POOTER, S., JANSEN, J., LABARRE, A., RAMON, J., DENECKER, M., AND VERWER, S. 2015. Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with IDP3. *TPLP 15*, 6 (November), 783–817.

CALIMERI, F., FABER, W., GEBSER, M., IANNI, G., KAMINSKI, R., KRENNWALLNER, T., LEONE, N., RICCA, F., AND SCHAUB, T. 2013. ASP-Core-2 input language format. Tech. rep., ASP Standardization Working Group.

CALIMERI, F., GEBSER, M., MARATEA, M., AND RICCA, F. 2016. Design and results of the fifth answer set programming competition. *Artif. Intell. 231*, 151–181.

DENECKER, M., LIERLER, Y., TRUSZCZYŃSKI, M., AND VENNEKENS, J. 2012. A Tarskian informal semantics for answer set programming. In *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012*. LIPIcs, vol. 17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Budapest, Hungary, 277–289.

DENECKER, M. AND VENNEKENS, J. 2007. Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In *Proceedings of the 9th International Conference on Logic Programming and Non-monotonic Reasoning, LPNMR 2007*. Lecture Notes in Computer Science, vol. 4483. Springer, Tempe, Arizona, USA, 84–96.

DRESCHER, C., GEBSER, M., GROTE, T., KAUFMANN, B., KÖNIG, A., OSTROWSKI, M., AND SCHAUB, T. 2008. Conflict-driven disjunctive answer set solving. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning, KR 2008*. AAAI Press, Sydney, Australia, 422–432.

EITER, T. AND GOTTLOB, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell. 15*, 3-4, 289–323.

EITER, T., GOTTLOB, G., AND VEITH, H. 1997. Modular logic programming and generalized quantifiers. In *Proceedings of the 4th International Conference on Logic Programming and Non-monotonic Reasoning, LPNMR 1997*. Lecture Notes in Computer Science, vol. 1265. Springer, Dagstuhl Castle, Germany, 289–308.

EITER, T. AND POLLERES, A. 2006. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *TPLP 6*, 1-2, 23–60.

EMERSON, E. A. AND JUTLA, C. S. 1991. Tree automata, mu-calculus and determinacy. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, FOCS 1991*. IEEE Computer Society, San Juan, Puerto Rico, 368–377.

GEBSER, M., HARRISON, A., KAMINSKI, R., LIFSCHITZ, V., AND SCHAUB, T. 2015. Abstract gringo. *TPLP 15*, 4-5, 449–463.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., ROMERO, J., AND SCHAUB, T. 2015. Progress in clasp series 3. In *Proceedings of the 13th International Conference on Logic Programming and Non-monotonic Reasoning, LPNMR 2015*. Lecture Notes in Computer Science, vol. 9345. Springer, Lexington, Kentucky, USA, 368–383.

GEBSER, M., KAMINSKI, R., AND SCHAUB, T. 2011. Complex optimization in answer set programming. *TPLP 11*, 4-5, 821–839.

GEBSER, M., SCHAUB, T., AND THIELE, S. 2007. GrinGo: A new grounder for Answer Set Programming. In *Proceedings of the 9th International Conference on Logic Programming and Non-monotonic Reasoning, LPNMR 2007*. Lecture Notes in Computer Science, vol. 4483. Springer, Tempe, Arizona, USA, 266–271.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming, ICLP 1988*. MIT Press, Seattle, Washington, USA, 1070–1080.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 3/4, 365–385.

GELFOND, M. AND PRZYMUSINSKA, H. 1996. Towards a theory of elaboration tolerance: Logic programming approach. *Int. J. of Soft. Eng. and Knowl. Eng. 6*, 1, 89–112.

GRASSO, G., IIRITANO, S., LEONE, N., AND RICCA, F. 2009. Some DLV applications for knowledge management. In *Proceedings of the 10th International Conference on Logic Programming and Non-monotonic Reasoning, LPNMR 2009*. Lecture Notes in Computer Science, vol. 5753. Springer, Potsdam, Germany, 591–597.

HELJANKO, K., KEINÄNEN, M., LANGE, M., AND NIEMELÄ, I. 2012. Solving parity games by a reduction to SAT. *J. Comput. Syst. Sci. 78*, 2, 430–440.

JANHUNEN, T., GEBSER, M., RINTANEN, J., NYMAN, H., PENSAR, J., AND CORANDER, J. 2015. Learning discrete decomposable graphical models via constraint optimization. *Statistics and Computing*. Advance access.

JANHUNEN, T., NIEMELÄ, I., SEIPEL, D., SIMONS, P., AND YOU, J. 2006. Unfolding partiality and disjunctions in stable model semantics. *ACM Trans. Comput. Log. 7*, 1, 1–37.

JANHUNEN, T., TASHARROFI, S., AND TERNOVSKA, E. 2016. SAT-TO-SAT: Declarative extension of SAT solvers with new propagators. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence, AAAI 2016*. AAAI Press, Phoenix, Arizona, USA, 978–984.

KOPONEN, L., OIKARINEN, E., JANHUNEN, T., AND SÄILÄ, L. 2015. Optimizing phylogenetic supertrees using answer set programming. *TPLP 15*, 4-5, 604–619.

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., ROSATI, R., AND SCARCELLO, F. 2001. Enhancing answer set planning. In *Proceedings of the IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information*. Seattle, Washington, USA.

LIFSCHITZ, V. 1999. Answer set planning. In *Proceedings of the 16th International Conference on Logic Programming, ICLP 1999*. MIT Press, Las Cruses, New Mexico, USA, 23–37.

LINDSTRÖM, P. 1966. First order predicate logic with generalized quantifiers. *Theoria 32*, 3, 186–195.

MAREK, V. AND TRUSZCZYŃSKI, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*. Springer-Verlag, Berlin Heidelberg, 375–398.

MOSTOWSKI, A. 1957. On a generalization of quantifiers. *Fundamenta Mathematicae 44*, 1, 12–36.

NIEMELÄ, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell. 25*, 3-4, 241–273.

NOGUEIRA, M., BALDUCCINI, M., GELFOND, M., WATSON, R., AND BARRY, M. 2001. An A-Prolog decision support system for the space shuttle. In *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages, PADL 2001*. Lecture Notes in Computer Science, vol. 1990. Springer, Las Vegas, Nevada, USA, 169–183.

OIKARINEN, E. AND JANHUNEN, T. 2006. Modular equivalence for normal logic programs. In *Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2006*. IOS Press, Riva del Garda, Italy, 412–416.

RICCA, F., DIMASI, A., GRASSO, G., IELPA, S. M., IIRITANO, S., MANNA, M., AND LEONE, N. 2010. A logic-based system for e-tourism. *Fundam. Inform. 105*, 1-2, 35–55.

STOCKMEYER, L. J. AND MEYER, A. R. 1973. Word problems requiring exponential time: Preliminary report. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, STOC 1973*. ACM, Austin, Texas, USA, 1–9.

TIIHONEN, J., SOININEN, T., NIEMELÄ, I., AND SULONEN, R. 2003. A practical tool for mass-customising configurable products. In *Proceedings of the 14th International Conference on Engineering Design, ICED 2003*. Design Society, Stockholm, 1290–1299.