Cooperating Answer Set Programming

Davy Van Nieuwenborgh[1,⋆], Stijn Heymans[2], and Dirk Vermeir[1]

[1] Dept. of Computer Science
Vrije Universiteit Brussel, VUB
Pleinlaan 2, B1050 Brussels, Belgium
{dvnieuwe, dvermeir}@vub.ac.be
[2] Digital Enterprise Research Institute (DERI)
University of Innsbruck, Austria
stijn.heymans@deri.org

Abstract. We present a formalism for logic program cooperation based on the answer set semantics. The system consists of independent logic programs that are connected via a sequential communication channel. When presented with an input set of literals from its predecessor, a logic program computes its output as an answer set of itself, enriched with the input.

It turns out that the communication strategy makes the system quite expressive: essentially a sequence of a fixed number of programs n captures the complexity class Σ_n^P, i.e. the n-th level of the polynomial hierarchy. On the other hand, unbounded sequences capture the polynomial hierarchy $\mathcal{PH}$. These results make the formalism suitable for complex applications such as hierarchical decision making and preference-based diagnosis on ordered theories. In addition, such systems can be realized by implementing an appropriate control strategy on top of existing solvers such as DLV or SMODELS, possibly in a distributed environment.

1 Introduction

In *answer set programming* (see e.g. [2]) a logic program is used to describe the requirements, that must be fulfilled by the solutions to a problem. The models (answer sets) of the program, usually defined through (a variant of) the stable model semantics [18], then correspond to the solutions of the problem. This technique has been successfully applied in problem areas such as planning [20], configuration and verification [23], diagnosis [9], ...

In this paper we use the answer set semantics to formalize a framework in which programs cooperate to obtain a solution that is acceptable to all and cannot unilaterally be improved upon. E.g., when a company has to make up an emergency evacuation plan for a building, one of the employees will make up a strategy that could be implemented for that building. However, as she is probably not aware of all current regulations about such strategies, her solution is forwarded to the emergency services, e.g. the police or the fire brigade, who will try to improve her plan so it conforms to all legal requirements. This adapted, legal version of the received starting plan is

⋆ Supported by the Flemish Fund for Scientific Research (FWO-Vlaanderen).

S. Etalle and M. Truszczyński (Eds.): ICLP 2006, LNCS 4079, pp. 226–241, 2006.

then send back to the employee who verifies its feasibility. If the verification fails, the communication starts all over again by the employee sending a new possible plan to the emergency services. In the other case, i.e. the adapted plan is successfully verified by the employee, it is presented to the firm's management which will try to improve it to obtain e.g. a cheaper one. Again, this cheaper alternative is sent back to the emergency services for verification, and eventually also to the employee, to check its feasibility.

We develop a framework of cooperating programs that is capable of modeling hierarchical decision problems like the one above. To this end, we consider a sequence of programs $\langle P_i \rangle_{i=1,\ldots n}$. Intuitively, a program P_i communicates the solutions it finds acceptable to the next program P_{i+1} in the hierarchy. For such a P_i-acceptable solution S, the program P_{i+1} computes a number of solutions that it thinks improve on S. If one of these P_{i+1} improvements S' of S is also acceptable to P_i, i.e. S' can be successfully verified by P_i, the original S is rejected as an acceptable solution by the program P_{i+1}. On the other hand, if P_{i+1} has no improvements for S, or none of them are also acceptable to P_i, S is accepted by P_{i+1}. It follows that a solution that is acceptable to all programs must have been proposed by the starting program P_1.

It turns out that such sequences of programs are rather expressive. More specifically, we show not only that arbitrary complete problems of the polynomial hierarchy can be solved by such systems, but that such systems can capture the complete polynomial hierarchy, the latter making them suitable for complex applications.

Problems located at the first level of the polynomial hierarchy can be directly solved using answer set solvers such as DLV [16] or SMODELS [22]. On the second level, only DLV is left to perform the job directly. However, by using a "guess and check" fixpoint procedure, SMODELS can indirectly be used to solve problems at the second level [4, 15]. Beyond the second level, there are still some interesting problems. E.g., the most expressive forms of diagnostic reasoning, i.e. subset-minimal diagnosis on disjunctive system descriptions [13] or preference-based diagnosis on ordered theories [27], are located at the third level of the polynomial hierarchy, as are programs that support sequences of weak constraints[1] on disjunctive programs. For these problems, and problems located even higher in the polynomial hierarchy, the framework presented in this paper provides a means to effectively compute solutions for such problems, using SMODELS or DLV for each program in the sequence to compute better solutions. E.g., to solve the problems mentioned before on the third level, it suffices to write three well-chosen programs and to set up an appropriate control structure implementing the communication protocol sketched above.

The remainder of the paper is organized as follows. In Section 2, we review the answer set semantics and present the definitions for cooperating program systems. Further, we illustrate how such systems can be used to elegantly express common problems. Section 3 discusses the complexity and expressiveness of the proposed semantics, while Section 4 compares it with related approaches from the literature. Finally, we conclude and give some directions for further research in Section 5. Due to space restrictions, proofs have been omitted, but they can be found in [25].

[1] A weak constraint is a constraint that is "desirable" but may be violated if there are no other options.

2 Cooperating Programs

We give some preliminaries concerning the answer set semantics for logic programs [2]. A *literal* is an atom a or a negated atom $\neg a$. For a set of literals X, we use $\neg X$ to denote $\{\neg l \mid l \in X\}$ where $\neg\neg a = a$. When $X \cap \neg X = \emptyset$ we say X is *consistent*. An *extended literal* is a literal or a *naf-literal* of the form *not* l where l is a literal. The latter form denotes negation as failure. For a set of extended literals Y, we use Y^- to denote the set of ordinary literals underlying the naf-literals in Y, i.e. $Y^- = \{l \mid \mathit{not}\ l \in Y\}$. Further, we use *not* X to denote the set $\{\mathit{not}\ l \mid l \in X\}$. An extended literal l is true w.r.t. X, denoted $X \models l$ if $l \in X$ in case l is ordinary, or $a \notin X$ if $l = \mathit{not}\ a$ for some ordinary literal a. As usual, $X \models Y$ iff $\forall l \in Y \cdot X \models l$.

A *rule* is of the form $\alpha \leftarrow \beta$ where[2] α is a finite set of literals, β is a finite set of extended literals and $|\alpha| \leq 1$. Thus the *head* of a rule is either an atom or empty. A countable set of rules is called a *(logic) program*. The *Herbrand base* $\mathcal{B}_P$ of a program P contains all atoms appearing in P. The set of all literals that can be formed with the atoms in P, denoted by $\mathcal{L}_P$, is defined by $\mathcal{L}_P = \mathcal{B}_P \cup \neg\mathcal{B}_P$. Any consistent subset $I \subseteq \mathcal{L}_P$ is called an *interpretation* of P.

A rule $r = \alpha \leftarrow \beta$ is *satisfied* by an interpretation I, denoted $I \models r$, if $I \models \alpha$ and $\alpha \neq \emptyset$, whenever $I \models \beta$, i.e. if r is *applicable* ($I \models \beta$), then it must be *applied* ($I \models \alpha \cup \beta \wedge \alpha \neq \emptyset$). Note that this implies that a *constraint*, i.e. a rule with empty head ($\alpha = \emptyset$), can only be satisfied if it is not applicable ($I \not\models \beta$). For a program P, an interpretation I is called a *model* of P if $\forall r \in P \cdot I \models r$, i.e. I satisfies all rules in P. It is a minimal model of P if there is no model J of P such that $J \subset I$.

A *simple program* is a program without negation as failure. For simple programs P, we define an *answer set* of P as a minimal model of P. On the other hand, for a program P, i.e. a program containing negation as failure, we define the *GL-reduct* [18] for P w.r.t. I, denoted P^I, as the program consisting of those rules $\alpha \leftarrow (\beta \backslash \mathit{not}\ \beta^-)$ where $\alpha \leftarrow \beta$ is in P and $I \models \mathit{not}\ \beta^-$. Note that all rules in P^I are free from negation as failure, i.e. P^I is a simple program. An interpretation I is then an *answer set* of P iff I is a minimal model of the GL-reduct P^I.

Example 1. Consider the following program P about diabetes.

$$\begin{array}{ccc} \mathit{diabetes} \leftarrow & \mathit{thirsty} \leftarrow & \neg\mathit{sugar} \leftarrow \mathit{diabetes} \\ \mathit{cola\_light} \leftarrow \mathit{thirsty}, \mathit{not\ cola} & \mathit{cola} \leftarrow \mathit{thirsty}, \mathit{not}\ \neg\mathit{sugar}, \mathit{not\ cola\_light} & \end{array}$$

One can check that P has $I = \{\mathit{diabetes}, \mathit{thirsty}, \neg\mathit{sugar}, \mathit{cola\_light}\}$ as its single answer set. Indeed, the rule $\mathit{cola} \leftarrow \mathit{thirsty}, \mathit{not}\ \neg\mathit{sugar}, \mathit{not\ cola\_light}$ is removed to obtain the reduct P^I of P as $\neg\mathit{sugar} \in I$, i.e. the rule can never become applicable. Further, the rule $\mathit{cola\_light} \leftarrow \mathit{thirsty}, \mathit{not\ cola}$ is kept as $\mathit{cola\_light} \leftarrow \mathit{thirsty}$ in P^I. Clearly, the reduct P^I so obtained has I as its minimal model.

In the present framework, it is assumed that all programs "communicate using the same language", i.e. the Herbrand bases of the programs are all subsets of some set of atoms $\mathcal{PL}$ (and $\mathcal{L}_{\mathcal{PL}} = \mathcal{PL} \cup \neg\mathcal{PL}$). Because programs will receive input from other programs

[2] As usual, we assume that programs have already been grounded.

that influence their reasoning, we do not want any unintentional implicit interferences between the input of the program and the produced output. E.g., a program should be able to compute for an input containing a, an output containing $\neg a$ or containing neither a nor $\neg a$. For this purpose, we will also use a mirror language $\mathcal{PL}'$ of $\mathcal{PL}$, where we use $l' \in \mathcal{L}_{\mathcal{PL}'}$ to denote the mirror version of a literal $l \in \mathcal{L}_{\mathcal{PL}}$ and we have that $l'' = l \in \mathcal{L}_{\mathcal{PL}}$. The notation is extended to sets, i.e. $X' = \{l' \mid l \in X\}$.

Intuitively, a program will receive input in the language $\mathcal{L}_{\mathcal{PL}}$, do some reasoning with a program over $\mathcal{L}_{\mathcal{PL}} \cup \mathcal{L}_{\mathcal{PL}'}$ and it will only communicate the part over $\mathcal{L}_{\mathcal{PL}'}$ to the other programs, i.e. an input literal $l \in \mathcal{L}_{\mathcal{PL}}$ can only appear in the output as $l' \in \mathcal{L}_{\mathcal{PL}'}$ if the program explicitly provides rules for this purpose.

Definition 1. *For a language $\mathcal{PL}$, a **cooperating program** P is a program such that $\mathcal{B}_P \subseteq \mathcal{PL} \cup \mathcal{PL}'$. For such a program P and a set of literals $I \subseteq \mathcal{L}_{\mathcal{PL}}$, called the **input**, we use $P(I)$ to denote the program $P \cup \{l \leftarrow \mid l \in I\}$.*

*An interpretation $S \subseteq \mathcal{L}_{\mathcal{PL}}$ is called an **output** w.r.t. the input I, or an **improvement** by P of I, iff there exists an answer set M of $P(I)$ such that $S = (M \cap \mathcal{L}_{\mathcal{PL}'})'$.*

We use $\mathcal{AS}(P, I)$ to denote the set of all outputs of P w.r.t. input I.

Example 2. Take $\mathcal{PL} = \{sugar, cola, cola\_light, hypoglycemia, diabetes, thirsty\}$ and consider the following program P, where we use the notation **keep** $\{a_1, \ldots, a_n\}$, to denote the set of rules $\{a'_i \leftarrow a_i \mid 1 \leq i \leq n\}$, i.e. to denote that part of the input that can be literally copied to the output,

$$\begin{aligned} \textbf{keep}\ \{&thirsty, hypoglycemia, diabetes\} \\ cola\_light' &\leftarrow thirsty, \mathit{not}\ sugar', \mathit{not}\ cola' \\ cola' &\leftarrow thirsty, \mathit{not}\ \neg sugar', \mathit{not}\ cola\_light' \\ \neg sugar' &\leftarrow diabetes, \mathit{not}\ hypoglycemia \\ sugar' &\leftarrow hypoglycemia \end{aligned}$$

Intuitively, the above program only copies the part of the input concerning hypoglycemia, diabetes and thirsty, because these are the only possible non-critical input literals. Other possible input literals, like e.g. $cola$ or $cola\_light$, will be recomputed in function of the availability (or not) of certain input literals.

Let $I_1 = \emptyset$, $I_2 = \{thirsty, hypoglycemia\}$ and $I_3 = \{thirsty, diabetes, cola\}$ be three inputs. One can check that P has only one output $S_1 = \emptyset$ w.r.t. I_1. For both I_2 and I_3 there is an improvement $S_2 = I_2 \cup \{sugar, cola\}$ and $S_3 = \{thirsty, diabetes, \neg sugar, cola\_light\}$, respectively. Note the necessity of the mirror language to obtain the latter result.

A single cooperating program is not a very powerful instrument. However, connecting a number of such programs together reveals their real capabilities. To keep things simple we will use, in what follows, cooperating program systems of linearly connected programs.

Formally, a cooperating program system is a linear sequence of cooperating programs $P_1, \ldots, P_n$, where P_1 is the source program, i.e. the program that starts all communication. Solutions for such systems are inductively defined by the notion of acceptance. Intuitively, a solution S is accepted by the source program P_1 if it recognizes S

as an improvement of the empty input; and a successor program P_i, $1 < i \leq n$, accepts S if it has no improvement on S that can be verified by the previous program P_{i-1} to be acceptable to it.

Definition 2. *Let $\mathcal{PL}$ be a language. A **cooperating program system** is a sequence $\langle P_i \rangle_{i=1,\ldots n}$ of n cooperating programs over $\mathcal{PL}$.*

*The set of **acceptable interpretations** $\mathcal{AC}(P_i)$ for a cooperating program P_i, $1 \leq i \leq n$, is inductively defined as follows:*

- $\mathcal{AC}(P_1) = \mathcal{AS}(P_1, \emptyset)$
- *for $i > 1$,*

$$\mathcal{AC}(P_i) = \{S \in \mathcal{AC}(P_{i-1}) \mid \forall T \in \mathcal{AS}(P_i, S) \cdot T \neq S \Rightarrow T \notin \mathcal{AC}(P_{i-1})\}$$

*An interpretation $S \subseteq \mathcal{L}_{\mathcal{PL}}$ that is acceptable to P_n, i.e. $S \in \mathcal{AC}(P_n)$, is called a **global answer set** for the cooperating program system.*

Note that the above definition allows for an output interpretation S of P_{i-1} to be accepted by a program P_i even if $P_i \cup S$ has no answer sets. This fits the intuition that the answer sets of $P_i \cup S$ are to be considered as improvements upon S. Hence, if P_i cannot be used to provide such an improvement, $P_i \cup S$ should not have any answer sets, and S should be accepted by P_i.

Example 3. Consider the job selection procedure of a company. The first cooperating program P_1 establishes the possible profiles of the applicants together with a rule stating the belief that inexperienced employees are ambitious. Thus, each answer set[3] of the program below corresponds with a possible applicant's profile.

$$male' \oplus female' \leftarrow \qquad old' \oplus young' \leftarrow$$
$$experienced' \oplus inexperienced' \leftarrow \qquad ambitious' \leftarrow inexperienced'$$

The decision on which applicant gets the job goes through a chain of decision makers. First, the human resources department constructs a cooperating program P_2 that implements company policy which stipulates that experienced persons should be preferred upon inexperienced ones. Therefore, the program passes through all of its input, except when it encounters a profile containing *inexperienced*, which it changes to *experienced*, intuitively implementing that an applicant with the same profile but *experienced* instead of *inexperienced*, would be preferable. Further, as we tend to prefer *experienced* people, for which nothing about being *ambitious* is known, we do not have any rule in P_2 containing $ambitious$, such that the literal is dropped from the input if present.

$$\textbf{keep}\ \{male, female, old, young, experienced\}$$
$$experienced' \leftarrow inexperienced$$

On the next level of the decision chain, the financial department reviews the remaining candidates. As young and inexperienced persons tend to cost less, it has a strong desire to hire such candidates, which is implemented in the following cooperating program P_3.

[3] In the rest of the paper we will use rules of the form $a \oplus b \leftarrow$ to denote the set of rules $\{a \leftarrow \mathit{not}\ b\ ;\ b \leftarrow \mathit{not}\ a\}$.

$$
\begin{aligned}
\textbf{keep}\ \{&male, female, young, inexperienced\}\\
ambitious' &\leftarrow inexperienced'\\
inexperienced' &\leftarrow young, experienced\\
young' &\leftarrow old, inexperienced\\
young' &\leftarrow old, experienced, \textit{not}\ old'\\
old' &\leftarrow old, experienced, \textit{not}\ young'\\
inexperienced' &\leftarrow old, experienced, \textit{not}\ experienced'\\
experienced' &\leftarrow old, experienced, \textit{not}\ inexperienced'\\
&\leftarrow old', experienced'
\end{aligned}
$$

Intuitively, this program handles the four possible cases: when the input profile is from a young and inexperienced person, the input is passed without modification indicating that this cannot be improved upon. On the other hand, if only one of the properties is not as desired, e.g. *young* and *experienced*, then the only improvement would be a profile containing both *young* and *inexperienced*. Finally, a profile containing *old* and *experienced* has three possible improvements: the last 5 rules ensure that the improvements proposed by P_3 will contain *young* or *inexperienced*, or both.

Finally, the management has the final call in the selection procedure. As the current team of employees is largely male, the management prefers the new worker to be a woman, as described by the next program P_4, which is similar to P_2.

$$
\begin{aligned}
&\textbf{keep}\ \{female, old, young, experienced, inexperienced, ambitious\}\\
&female' \leftarrow male
\end{aligned}
$$

One can check that P_1 has eight answer sets (improvements on $\emptyset$), that are thus acceptable to it. However, only four of these are acceptable to P_2, i.e.

$$
\begin{aligned}
M_1 &= \{experienced, male, young\}\ ,\\
M_2 &= \{experienced, male, old\}\ ,\\
M_3 &= \{experienced, female, young\}\ ,\\
M_4 &= \{experienced, female, old\}\ ,
\end{aligned}
$$

which fits the company policy to drop inexperienced ambitious people. E.g., feeding $M_5 = \{inexperienced, female, young, ambitious\}$ as input to P_2 yields one answer set M_3, which is also acceptable to P_1 making M_5 unacceptable for P_2. Similarly, when P_3 is taken into account, only M_1 and M_3 are acceptable. Considering the last program P_4 yields a single global answer set, i.e. M_3, which fits our intuition that, if possible, a woman should get the job.

Note that rearranging the programs gives, in general, different results. E.g., interchanging P_2 with P_3 yields M_5 as the only global answer set.

3 Complexity

We briefly recall some relevant notions of complexity theory (see e.g. [2] for a nice introduction). The class $\mathcal{P}$ ($\mathcal{NP}$) represents the problems that are deterministically (nondeterministically) decidable in polynomial time, while $co\mathcal{NP}$ contains the problems

whose complements are in $\mathcal{NP}$. The polynomial hierarchy, denoted $\mathcal{PH}$, is made up of three classes of problems, i.e. Δ_k^P, Σ_k^P and Π_k^P, $k \geq 0$, which are defined as follows:

1. $\Delta_0^P = \Sigma_0^P = \Pi_0^P = \mathcal{P}$; and
2. $\Delta_{k+1}^P = \mathcal{P}^{\Sigma_k^P}$, $\Sigma_{k+1}^P = \mathcal{NP}^{\Sigma_k^P}$, $\Pi_{k+1}^P = co\Sigma_{k+1}^P$.

The class $\mathcal{P}^{\Sigma_k^P}$ ($\mathcal{NP}^{\Sigma_k^P}$) represents the problems decidable in deterministic (nondeterministic) polynomial time using an oracle for problems in Σ_k^P. The class $\mathcal{PH}$ is defined by $\mathcal{PH} = \bigcup_{k=0}^{\infty} \Sigma_k^P$. Finally, the class $PSPACE$ contains the problems that can be solved deterministically by using a polynomial amount of memory and unlimited time.

To prove hardness for the above complexity classes[4], we will use validity checking of quantified boolean formulas. A quantified boolean formula (QBF) is an expression of the form $Q_1 X_1 Q_2 X_2 \ldots Q_k X_k \cdot G$, where $k \geq 1$, G is a Boolean expression over the atoms of the pairwise nonempty disjoint sets of variables $X_1, \ldots, X_k$ and the Q_i's, for $i = 1, \ldots, k$ are alternating quantifiers from $\{\exists, \forall\}$. When $Q_1 = \exists$, the QBF is k-existential, when $Q_1 = \forall$ we say it is k-universal. We use $QBF_{k,\exists}$ ($QBF_{k,\forall}$) to denote the set of all valid k-existential (k-universal) QBFs. Deciding, for a given k-existential (k-universal) QBF ϕ, whether $\phi \in QBF_{k,\exists}$ ($\phi \in QBF_{k,\forall}$) is a Σ_k^P-complete (Π_k^P-complete) problem. When we drop the bound k on the number of quantifiers, i.e. considering $QBF_\exists = \bigcup_{i \in \mathbb{N}} QBF_{i,\exists}$, we have a hard problem for $PSPACE$.

The following results shed some light on the complexity of the global answer set semantics for linear combinations of cooperating programs.

First, we consider the case where the length of the sequence of cooperating programs is fixed by some number n.

Theorem 1. *Given a cooperating program system $\langle P_i \rangle_{i=1,\ldots,n}$, with n fixed, and a literal $l \in \mathcal{L}_{\mathcal{PL}}$, the problem of deciding whether there exists a global answer set containing l is Σ_n^P-complete. On the other hand, deciding whether every global answer set contains l is Π_n^P-complete.*

Proof Sketch. Membership Σ_n^P: It is shown, by induction, in [25] that checking whether an interpretation $S \subseteq \mathcal{L}_{\mathcal{PL}}$ is not acceptable to P_n, i.e. $S \notin \mathcal{AC}(P_n)$, is in Σ_{n-1}^P. The main result follows by

- guessing an interpretation $S \subseteq \mathcal{L}_{\mathcal{PL}}$ such that $S \ni l$; and
- checking that it is not the case that $S \notin \mathcal{AC}(P_n)$.

As the latter is in Σ_{n-1}^P, the problem itself can be done by an $\mathcal{NP}^{\Sigma_{n-1}^P}$ algorithm, i.e. the problem is in Σ_n^P.

Hardness Σ_n^P: To prove hardness, we provide a reduction of deciding validity of QBFs by means of a cooperating program system. Let $\phi = \exists X_1 \forall X_2 \ldots Q X_n \cdot G \in QBF_{n,\exists}$, where $Q = \forall$ if n is even and $Q = \exists$ otherwise. We assume, without loss of generality [24], that G is in disjunctive normal form, i.e. $G = \vee_{c \in C} c$ where C is a set of sets of literals over $X_1 \cup \ldots \cup X_n$ and each $c \in C$ has to be interpreted as a conjunction.

[4] Note that this does not hold for the class $\mathcal{PH}$ for which no complete, and thus hard, problem is known unless $\mathcal{P} = \mathcal{NP}$.

In what follows, we use P^i to denote the set rules

- **keep** $\{x\ ,\ \neg x \mid x \in X_j \wedge 1 \leq j < i\}$,
- $\{x' \leftarrow \mathit{not}\ \neg x'\ ;\ \neg x' \leftarrow \mathit{not}\ x' \mid x \in X_j \wedge i \leq j \leq n\}$, and
- $\{sat' \leftarrow c' \mid c \in C\}$.

Further, we use $P^i_\forall$ and $P^i_\exists$ to denote the programs $P^i_\forall = P^i \cup \{ \leftarrow sat'\ ;\ \leftarrow \mathit{not}\ sat\}$ and $P^i_\exists = P^i \cup \{ \leftarrow \mathit{not}\ sat'\ ;\ \leftarrow sat\}$ respectively.

The cooperating program system $\langle P_i \rangle_{i=1,\ldots,n}$ corresponding to ϕ is defined as:

- P_1 contains the rules $\{x' \leftarrow \mathit{not}\ \neg x'\ ;\ \neg x' \leftarrow \mathit{not}\ x' \mid x \in X_j \wedge 1 \leq j \leq n\}$ and $\{sat' \leftarrow c' \mid c \in C\}$;
- if n is even, then $P_i = P^{n+2-i}_\forall$ when i even and $P_i = P^{n+2-i}_\exists$ when $i > 1$ odd;
- if n is odd, then $P_i = P^{n+2-i}_\exists$ when i even and $P_i = P^{n+2-i}_\forall$ when $i > 1$ odd.

Obviously, the above construction can be done in polynomial time. Intuitively, P_1 has answer sets for every possible combination of the X_i's and if such a combination makes G valid, then the corresponding answer set also contains the atom sat. The intuition behind the program $P^i_\forall$ is that it tries to disprove, for the received input, the validity of the corresponding $\forall$, i.e. for a given input combination over the X_j's making G satisfied, the program $P^i_\forall$ will try to find a combination, keeping the X_j's with $j < i$ fixed, making G false. On the other hand, the program $P^i_\exists$ will try to prove the validity of the corresponding $\exists$, i.e. for a given combination making G false it will try to compute a combination, keeping the X_j's with $j < i$ fixed, making G satisfied.

Instead of giving the formal proof for the above construction, we give a feel on how the construction works by means of an example and refer the reader to [25] for the actual proof.

Consider

$$\phi = \exists x \cdot \forall y \cdot \exists z \cdot (x \wedge \neg y \wedge z) \vee (y \wedge \neg z)\ .$$

The cooperating program P_1 contains the rules

$$\begin{array}{llll} x' \leftarrow \mathit{not}\ \neg x' & \neg x' \leftarrow \mathit{not}\ x' & y' \leftarrow \mathit{not}\ \neg y' & \neg y' \leftarrow \mathit{not}\ y' \\ z' \leftarrow \mathit{not}\ \neg z' & \neg z' \leftarrow \mathit{not}\ z' & sat' \leftarrow x', \neg y', z' & sat' \leftarrow y', \neg z' \end{array}$$

We have 8 possible outputs for $P_1(\emptyset)$, i.e. $I_1 = \{x, y, z\}$, $I_2 = \{x, y, \neg z, sat\}$, $I_3 = \{x, \neg y, z, sat\}$, $I_4 = \{x, \neg y, \neg z\}$, $I_5 = \{\neg x, y, z\}$, $I_6 = \{\neg x, y, \neg z, sat\}$, $I_7 = \{\neg x, \neg y, z\}$ and $I_8 = \{\neg x, \neg y, \neg z\}$. Clearly, these are all acceptable interpretations for P_1.

The second cooperating program P_2 is defined by $P^3_\exists$ and thus contains the rules

$$\begin{array}{lllll} \textbf{keep}(\{x, \neg x, y, \neg y\}) \leftarrow & z' \leftarrow \mathit{not}\ \neg z' & \neg z' \leftarrow \mathit{not}\ z' & sat' \leftarrow x', \neg y', z' \\ & \leftarrow sat & \leftarrow \mathit{not}\ sat' & sat' \leftarrow y', \neg z' \end{array}$$

Feeding I_1 to P_2 yields I_2 as the single output. As I_2 is an acceptable interpretation to P_1, I_1 cannot be acceptable to P_2, i.e. $I_1 \notin \mathcal{AC}(P_2)$. On the other hand, for the input I_2, the program P_2 has no outputs, as the input contains sat, which makes the constraint $\leftarrow sat$ unsatisfied. As a result, I_2 is acceptable to P_2, i.e. $I_2 \in \mathcal{AC}(P_2)$. In case of the

input I_7, P_2 is not able to derive sat' with the given input, yielding that $\leftarrow$ *not* sat' can never be satisfied and thus P_2 will not produce any outputs for I_7, again yielding that I_7 will be acceptable to P_2.

One can check in similar ways that $\mathcal{AC}(P_2)$ contains 5 interpretations, i.e. $\mathcal{AC}(P_2) = \{I_2, I_3, I_6, I_7, I_8\}$. It is not difficult to see that for each of these acceptable solutions it holds that $\exists z \cdot (x \wedge \neg y \wedge z) \vee (y \wedge \neg z)$ when x and y are taken as in the interpretation iff the literal sat is contained in that interpretation.

The third and final cooperating program P_3 is given by $P^2_\forall$ and contains the rules

$$\textbf{keep}(\{x, \neg x\}) \leftarrow$$
$$\begin{array}{llll} y' \leftarrow \textit{not } \neg y' & \neg y' \leftarrow \textit{not } y' & \leftarrow \textit{not } sat & sat' \leftarrow x', \neg y', z' \\ z' \leftarrow \textit{not } \neg z' & \neg z' \leftarrow \textit{not } z' & \leftarrow sat' & sat' \leftarrow y', \neg z' \end{array}$$

When providing P_3 with the input I_2, we have two outputs, i.e. I_1 and I_4. However, neither $I_1 \in \mathcal{AC}(P_2)$ nor $I_2 \in \mathcal{AC}(P_2)$, yielding that I_2 is an acceptable solution to P_3. Intuitively, P_3 accepts the input I_2 as it cannot disprove $\forall y \cdot \exists z \cdot (x \wedge \neg y \wedge z) \vee (y \wedge \neg z)$ for the chosen truth value of x in I_2. In a similar way one can check that also $I_3 \in \mathcal{AC}(P_3)$.

On the other hand, feeding P_3 with I_6, we get the outputs $\{I_5, I_7, I_8\}$. This time, both $I_7 \in \mathcal{AC}(P_2)$ and $I_8 \in \mathcal{AC}(P_2)$, implying that I_6 is not acceptable to P_3. Further, using I_7 or I_8 as an input to P_3, results in no outputs, making them both acceptable to P_3. As a result, $\mathcal{AC}(P_3) = \{I_1, I_3, I_7, I_8\}$, which are also the global answer sets of the system.

Now, one can check that for each global answer set in $\mathcal{AC}(P_3)$ it holds that $\forall y \cdot \exists z \cdot (x \wedge \neg y \wedge z) \vee (y \wedge \neg z)$ for x taken as in the interpretation iff the literal sat is contained in that global answer set. From this it follows that ϕ is valid iff there exists a global answer set $I \in \mathcal{AC}(P_3)$ such that $sat \in I$. In our example, I_2 is such a global answer set and one can check that ϕ holds when we assume x is true.

Π^P_n-completeness: To show this result, we consider in [25] the complement decision problem and show that it is Σ^P_n-complete, from which the result follows. □

While the previous result handles the cases where the number of programs in the sequence is fixed, we can generalize the results to arbitrary sequences.

Theorem 2. *Given a cooperating program system $\langle P_i \rangle_{i=1,\ldots,n \in \mathbb{N}}$ and a literal $l \in \mathcal{L}_{\mathcal{PL}}$, the problem of deciding whether there exists a global answer set containing l is PSPACE-complete.*

Proof Sketch. Membership PSPACE: Intuitively, each program in the sequence needs the space to represent a single answer set, while the system itself needs the space to represent a global answer set. Now, the algorithm will place a possible solution in the latter allocated space, and will use the former allocated space to check acceptability for the different programs in the sequence. Thus, an algorithm for a sequence of n programs, needs maximum $n + 1$ times the space to represent an answer set, which is clearly polynomial in space, from which membership to $PSPACE$ follows.

Hardness PSPACE: Clearly, the hardness proof of Theorem 1 can be generalized to validity checking of arbitrary quantified boolean formula, from which hardness readily follows. □

While the previous results describe the complexity of reasoning with the presented framework, they don't give a clear picture on the expressiveness of the system, i.e. whether each problem that belongs to a certain complexity class can be expressed in the framework. The reason therefore is that a formalism F being complete for a particular class only implies that each instance of a problem in that class can be reduced in polynomial time to an instance of F such that the yes/no answer is preserved. However, completeness does not imply that the polynomial time reduction itself from an instance of the problem to an instance in F is expressible in F[5].

In this context, one says that a formalism *captures* a certain complexity class iff the formalism is in the class and every problem in that class can be expressed in the formalism. The latter part is normally proved by taking an arbitrary expression in a normal (or general) form[6] for the particular complexity class and by showing that it can be expressed in the formalism.

By using the results from [14, 12], the following normal form for the complexity class Σ_k^P, with $k \geq 2$, can be obtained. First, we have to consider a signature $\sigma = (O, F, P)$, with O finite and $F = \emptyset$, i.e. we do not allow function symbols. A finite database over σ is any finite subset of the Herbrand Base over σ. Secondly, we have three predicates that do not occur in P, i.e. *succ, first* and *last*. Enumeration literals are literals over the signature $(O, \emptyset, \{succ, first, last\})$ that satisfy the conditions:

- *succ* describes an enumeration of the elements in O; and
- *first* and *last* contain the first and last element in the enumeration respectively.

Intuitively, *succ* is a binary predicate such that $succ(x, y)$ means that y is the successor of x. Further, *first* and *last* are unary predicates.

A collection S of finite databases over the signature $\sigma = (O, \emptyset, P)$ is in Σ_k^P iff there is a second order formula of the form

$$\phi = Q_1 U^1_{1,\ldots,m_1} Q_2 U^2_{1,\ldots,m_2} \ldots Q_k U^k_{1,\ldots,m_k} \exists \overline{x} \cdot \theta_1(\overline{x}) \vee \cdots \vee \theta_l(\overline{x}) \ ,$$

where $Q_i = \exists$ if i is odd, $Q_i = \forall$ if i is even, $U^i_{1,\ldots,m_i}$ $(1 \leq i \leq k)$ are finite sets of predicate symbols and $\theta_i(\overline{x})$ $(1 \leq i \leq l)$ are conjunctions of enumeration literals or literals involving predicates in $P \cup \{ U^1_{1,\ldots,m_1}, U^2_{1,\ldots,m_2}, \ldots, U^k_{1,\ldots,m_k} \}$ such that for any finite database w over σ, $w \in S$ iff w satisfies ϕ.

Again, we first consider the case in which the number of programs in the sequence is fixed by a number $n \in \mathbb{N}$.

Theorem 3. *The global answer set semantics for cooperating program systems with a fixed number n of programs captures Σ_n^P.*

Proof Sketch. Membership Σ_n^P: The result follows directly from the membership part of the proof of Theorem 1.

[5] A good example of this fact is the query class *fixpoint*, which is $PTIME$-complete but cannot express the simple query $even(R)$ to check if $|R|$ is even. See e.g. [7, 2] for a more detailed explanation on the difference between completeness and expressiveness (or capturing).

[6] A normal (or general) form of a complexity class is a form in which every problem in the class can be expressed. Note that not every complexity class necessarily has a general form.

Capture Σ_n^P: This proof is a generalization of the technique used in the hardness proof of Theorem 1. Further, the construction of the programs, especially the first program, is based on the proof of Theorem 6.3.2. in [2], where it is shown that disjunctive logic programming under the brave semantics captures Σ_2^P.

However, we first have to consider the case where $n = 1$ separately, as the general form discussed above only holds for $n \geq 2$. It is easy to see that the global answer set semantics for cooperating systems of a single program coincides with the classical answer set semantics, for which capturing of $\Sigma_1^P = \mathcal{NP}$ is already proven in the literature (e.g. in [2]).

To prove that any problem of Σ_n^P, with $n \geq 2$, can be expressed in a cooperating program system of n programs under the global answer set semantics, we have to show a construction of such a system $\langle P_i \rangle_{i=1,\ldots,n}$ such that a finite database w satisfies the formula

$$\phi = \exists U^1_{1,\ldots,m_1} \forall U^2_{1,\ldots,m_2} \ldots Q_n U^n_{1,\ldots,m_n} \exists \overline{x} \cdot \theta_1(\overline{x}) \vee \cdots \vee \theta_l(\overline{x}) \ ,$$

with everything defined as in the general form for Σ_n^P described before, iff $\langle P_i \rangle_{i=1,\ldots,n}$ has a global answer set containing *sat*.

The first program[7] P_1 in the sequence contains, beside the facts that introduce the database w (as w'), the following rules:

- For the enumeration of the predicates $U^1_{1,\ldots,m_1}, U^2_{1,\ldots,m_2}, \ldots, U^n_{1,\ldots,m_n}$, we have the rules:

  $$U_k^{i\prime}(\overline{w_k^i}) \leftarrow \mathit{not}\ \neg U_k^{i\prime}(\overline{w_k^i}) \qquad\qquad \neg U_k^{i\prime}(\overline{w_k^i}) \leftarrow \mathit{not}\ U_k^{i\prime}(\overline{w_k^i})$$

  for $1 \leq i \leq n$ and $1 \leq k \leq m_i$.
- To introduce the linear ordering, we need a set of rules similar to the ones used in Section 2.1.13. of [2] (see the technical report [25] for a detailed description). This set of rules has the property that when a linear ordering is established, the literal $linear'$ is derived.
- To check satisfiability, we use the rules

  $$sat' \leftarrow \theta_i{}'(\overline{x}), linear'$$

  for $1 \leq i \leq l$.

The other programs of the sequence are defined, similar to the hardness proof of Theorem 1, by using two skeletons $P^i_\forall$ and $P^i_\exists$. First, both skeletons have the following set of rules P^i in common

- **keep** $\{\, U_k^j(\overline{w_k^j}) \ , \ \neg U_k^j(\overline{w_k^j}) \mid (1 \leq j < i) \wedge (1 \leq k \leq m_i) \}$,
- **keep** $\{$*facts of the linear ordering*$\}$,
- **keep** w ,
- $\{\, U_k^{j\prime}(\overline{w_k^j}) \leftarrow \mathit{not}\ \neg U_k^{j\prime}(\overline{w_k^j}) \ ; \ \neg U_k^{j\prime}(\overline{w_k^j}) \leftarrow \mathit{not}\ U_k^{j\prime}(\overline{w_k^j}) \mid (i \leq j \leq n) \wedge (1 \leq k \leq m_i) \}$, and

[7] For clarity, we will use non-grounded rules, but we assume that the reader is familiar with obtaining the grounded versions of non-grounded rules.

– $\{sat' \leftarrow \theta_i{}'(\overline{x}), linear' \mid 1 \leq i \leq l\}$.

Now, we define the programs $P^i_\forall$ and $P^i_\exists$ as $P^i_\forall = P^i \cup \{ \leftarrow sat' ; \ \leftarrow not\ sat\}$ and $P^i_\exists = P^i \cup \{ \leftarrow not\ sat' ; \ \leftarrow sat\}$ respectively.

Besides P_1, the remaining programs in the sequence are defined by:

– if n is even, then $P_i = P^{n+2-i}_\forall$ when i even and $P_i = P^{n+2-i}_\exists$ when $i > 1$ odd;
– if n is odd, then $P_i = P^{n+2-i}_\exists$ when i even and $P_i = P^{n+2-i}_\forall$ when $i > 1$ odd.

It is not difficult to see (similar to the hardness proof of Theorem 1) that the above constructed program will only generate, for a given input database w, global answer sets that contain *sat* iff ϕ is satisfied. □

When we drop the fixed length of the sequence, the above result can be easily generalized to arbitrary cooperating program systems.

Corollary 1. *The global answer set semantics for cooperating program systems captures*[8] $\mathcal{PH}$*, i.e. the polynomial hierarchy.*

The above result yields that the presented framework is able to encode each problem in the polynomial hierarchy in a modular way, making the framework useful for complex knowledge reasoning tasks, e.g. involving multiple optimization steps.

4 Relationships to Other Approaches

In [5], answer set optimization (ASO) programs are presented. Such ASO programs consist of a generator program and a sequence of optimizing programs. To perform the optimization, the latter programs use rules similar to ordered disjunction [3], i.e. rules of the form $c_1 < \cdots < c_n \leftarrow \beta$ which intuitively read: when β is true, making c_1 true is the most preferred option and only when c_1 cannot be made true, the next best option is to make c_2 true, ... Solutions of the generator program that are optimal w.r.t. the first optimizing program and, among those, are optimal w.r.t. the second optimizing program, and so on, are called preferred solutions for the ASO program.

The framework of ASO programming looks very similar to our approach, i.e. just consider the generator program as program P_1 and the optimizing programs as programs $P_2, \ldots, P_n$. However, ASO programs are far more limited w.r.t. their expressiveness, due to the syntactical and semantical restrictions of the optimizing programs in comparison to our approach where arbitrary programs can be used to do the optimization. It turns out that the expressiveness of an ASO program does not depend on the length of the sequence of optimizing programs: it is always Σ^P_2-complete. Hence ASO programs can be captured by the presented cooperating program systems in this paper using a pair of programs. The construction of these two programs simulating ASO programs is subject to further research.

Weak constraints were introduced in [6] as a relaxation of the concept of a constraint. Intuitively, a weak constraint is allowed to be violated, but only as a last resort, meaning that one tries to minimize the number of violated constraints. Additionally, weak constraints are allowed to be hierarchically layered by means of a sequence of sets of weak

[8] Note that while the semantics captures $\mathcal{PH}$, it can never be complete for it as the hierarchy would than collapse.

constraints. Intuitively, one first chooses the answer sets that minimize the number of violated constraints in the first set of weak constraints in the sequence, and then, among those, one chooses the answer sets that minimize the number of violated constraints in the second set, etc.

Again, this approach can be seen as a kind of cooperating programming system. The complexity of such a system, independent of the number of sets of weak constraints, is at most Δ_3^P-complete. Thus, using the presented cooperating programming system from Section 2, a sequence of three programs suffice to capture the most expressive form of that formalism.

The framework developed in this paper can be seen as a more general version of the idea presented in [15], where a guess and a check program are combined into a single disjunctive program such that the answer sets of this program coincide with the solutions that can be obtained from the guess program and successfully checked by the check program. The approach in this paper allows to combine a guess program and multiple check programs, which each have to be applied in turn, into a single cooperating system such that the global answer sets correspond to solutions that can be guessed by the guess program and subsequently verified by the check programs.

In [26, 19], hierarchies of preferences on a single program are presented. The preferences are expressible on both the literals and the rules in that program. It is shown that for a sequence of n preference relations the complexity of the system is Σ_{n+1}^P-complete. The semantics proposed in Section 2 is a generalization of that approach: instead of using one global program with a sequence of preferences expressed on that program, we use a sequence of, in general, different programs, thus allowing a separate optimizing strategy for each individual program. To capture a hierarchy of n preference relations, we need $n + 1$ cooperating programs: the first one will correspond with the global program, while the rest will correspond to the n preference relations. The system described in Example 3 can be seen as a translation of such a preference hierarchy. Intuitively, the program P_2 describes the preference relation $experienced < inexperienced$, while P_3 implements the relation $young < old$; $inexperienced < experienced$. Finally, P_4 corresponds to the single preference $female < male$. This also suggests that the present framework may be useful to encode, in a unified way, sequential communication between programs supporting different higher level language constructs such as preference orders.

Updates of logic programs [1, 10] can be seen as a form of sequential communication. However, the approaches presented in the literature are limited to solving problems located in the first or second level of the polynomial hierarchy.

[21] presents composition of logic programs as a way to solve decision making in agent systems. Intuitively, for two programs P_1 and P_2 the system tries to compute a program P such that each answer set S of P is of the form $S = S_1 \cup S_2$ (or $S = S_1 \cap S_2$), where S_1 and S_2 are answer sets of P_1 and P_2 respectively. Clearly, this approach can be extended to sequences of programs, but it is different from the one presented in this paper in the sense that we apply each program in the sequence in turn, while all programs in the former approach are applied at once in the composed program P'. This explains why the complexity of the former semantics remains the same as that of the underlying answer set semantics.

Finally, the concept of cooperation for decision making is also used in other areas than answer set programming, e.g. in the context of concurrent or distributed theorem-proving [8, 17]. In [17], the idea is to split up and distribute a set of axioms and a theory among a number of agents that each derive new knowledge to prove the theory (using only the part of the knowledge they received) and who communicate their newly derived knowledge to the other agents in the system. [8] handles the same problem in a different way, i.e. each agent has its own strategy to prove the theory and after an amount of time the results of the cooperating agents are evaluated. If an agent scored badly during this evaluation, the system can decide to replace it with a new agent having another proof strategy. In the end, one wants to obtain a team of agents that performs best to solve the given problem.

5 Conclusions and Directions for Further Research

We presented a framework suitable for solving hierarchical decision problems using logic programs that cooperate via a sequential communication channel. The resulting semantics turns out to be rather expressive, as it essentially covers the polynomial hierarchy, thus enabling further complex applications. E.g., the framework could be used to develop implementations for diagnostic systems at the third level of the polynomial hierarchy [11, 13, 27].

Future work comprises the development of a dedicated implementation of the approach, using existing answer set solvers, e.g. DLV [16] or SMODELS [22], possibly in a distributed environment. Such an implementation will use a control structure that communicates candidate solutions between consecutive programs. When a program P_i receives a solution S from P_{i-1}, it attempts to compute an improvement S'. If no such S' exists, S is acceptable to P_i and is communicated to P_{i+1}. Otherwise, S' is send back to P_{i-1}, who verifies its acceptability (for P_{i-1}). If it is, P_i starts over to check if it can (or cannot) improve upon S'. On the other hand, when S' is not acceptable to P_{i-1}, P_i generates another improvement and starts over again. For efficiency, each program can hold some kind of success- and failure list containing solutions that have already been tested for acceptability and were either accepted or rejected.

In the context of an implementation, it is also interesting to investigate which conditions a program has to fulfill in order for it not to lift the complexity up with one level in the polynomial hierarchy, yielding possible optimizations of the computation and communication process.

Finally, we plan to look into a broader class of communication structures, e.g. a tree or, more generally, a (strict) partial ordering of programs, or even cyclic structures.

References

[1] J. J. Alferes, L. J. A., L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic logic programming. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning*, pages 98–111. Morgan Kaufmann, 1998.

[2] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.

[3] G. Brewka. Logic programming with ordered disjunction. In *Proceedings of the 18th National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence*, pages 100–105. AAAI Press, 2002.

[4] G. Brewka, I. Niemela, and T. Syrjanen. Implementing ordered disjunction using answer set solvers for normal programs. In *European Conference, JELIA 2002*, volume 2424 of *LNAI*, pages 444–455, Cosenza, Italy, September 2002. Springer.

[5] G. Brewka, I. Niemelä, and M. Truszczynski. Answer set optimization. In G. Gottlob and T. Walsh, editors, *IJCAI*, pages 867–872. Morgan Kaufmann, 2003.

[6] F. Buccafurri, N. Leone, and P. Rullo. Strong and weak constraints in disjunctive datalog. In *Proceedings of the 4th International Conference on Logic Programming (LPNMR '97)*, pages 2–17, 1997.

[7] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[8] J. Denzinger and M. Kronenburg. Planning for distributed theorem proving: The teamwork approach. In *Advances in Artificial Intelligence, 20th Annual German Conference on Artificial Intelligence (KI-96)*, volume 1137 of *LNCS*, pages 43–56. Springer, 1996.

[9] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The diagnosis frontend of the dlv system. *AI Communications*, 12(1-2):99–111, 1999.

[10] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Considerations on updates of logic programs. In *European Workshop, JELIA 2000*, volume 1919 of *Lecture Notes in Artificial Intelligence*, pages 2–20, Malaga, Spain, September–October 2000. Springer Verlag.

[11] T. Eiter and G. Gottlob. The complexity of logic-based abduction. *Journal of the Association for Computing Machinery*, 42(1):3–42, 1995.

[12] T. Eiter, G. Gottlob, and Y. Gurevich. Normal forms for second-order logic over finite structures, and classification of np optimization problems. *Annals of Pure and Applied Logic*, 78(1-3):111–125, 1996.

[13] T. Eiter, G. Gottlob, and N. Leone. Abduction from logic programs: Semantics and complexity. *Theoretical Computer Science*, 189(1-2):129–177, 1997.

[14] T. Eiter, G. Gottlob, and H. Mannila. Adding disjunction to datalog. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 267–278. ACM Press, 1994.

[15] T. Eiter and A. Polleres. Towards automated integration of guess and check programs in answer set programming. In V. Lifschitz and I. Niemelä, editors, *LPNMR*, volume 2923 of *LNCS*, pages 100–113, Fort Lauderdale, FL, USA, January 2004. Springer.

[16] W. Faber and G. Pfeifer. dlv homepage. http://www.dbai.tuwien.ac.at/proj/dlv/.

[17] M. Fisher. An open approach to concurrent theorem-proving. *Parallel Processing for Artificial Intelligence*, 3, 1997.

[18] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, pages 1070–1080, Seattle, Washington, August 1988. The MIT Press.

[19] S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Hierarchical decision making by autonomous agents. In *Logics in Artificial Intelligence, 9th European Conference, Proceedings, JELIA 2004*, volume 3229 of *LNCS*, pages 44–56. Springer, 2004.

[20] V. Lifschitz. Answer set programming and plan generation. *Journal of Artificial Intelligence*, 138(1-2):39–54, 2002.

[21] C. Sakama and K. Inoue. Coordination between logical agents. In *Proceedings of the 5th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA-V)*, volume 3487 of *Lecture Notes in Artificial Intelligence*, pages 161–177. Springer, 2005.

[22] P. Simons. smodels homepage. http://www.tcs.hut.fi/Software/smodels/.

[23] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL '99)*, LNCS, San Antonio, Texas, 1999. Springer.

[24] L. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *Proceedings of the 5th ACM Symposium on Theory of Computing (STOC '73)*, pages 1–9, 1973.

[25] D. Van Nieuwenborgh, S. Heymans, and D. Vermeir. Cooperating answer set programming. Technical report. Vrije Universiteit Brussel, Dept. of Computer Science, 2006, http://tinf2.vub.ac.be/~dvnieuwe/iclp2006technical.ps.

[26] D. Van Nieuwenborgh, S. Heymans, and D. Vermeir. On programs with linearly ordered multiple preferences. In B. Demoen and V. Lifschitz, editors, *Proceedings of 20th International Conference on Logic Programming (ICLP 2004)*, number 3132 in LNCS, pages 180–194. Springer, 2004.

[27] D. Van Nieuwenborgh and D. Vermeir. Ordered programs as abductive systems. In *Proceedings of the APPIA-GULP-PRODE Conference on Declarative Programming (AGP2003)*, pages 374–385, Regio di Calabria, Italy, 2003.