Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 34–43
http://www.floc-conference.org/ICLP-home.html

COMMUNICATING ANSWER SET PROGRAMS

KIM BAUTERS [1] AND JEROEN JANSSEN [2] AND STEVEN SCHOCKAERT [1] AND DIRK VERMEIR [2] AND MARTINE DE COCK [3,1]

[1] Department of Applied Mathematics and Computer Science, Universiteit Gent
Krijgslaan 281, 9000 Gent, Belgium
E-mail address: {kim.bauters,steven.schockaert}@ugent.be

[2] Department of Computer Science, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
E-mail address: {jeroen.janssen,dvermeir}@vub.ac.be

[3] Institute of Technology, University of Washington
1900 Commerce Street, WA-98402 Tacoma, USA
E-mail address: mdecock@u.washington.edu

Abstract. Answer set programming is a form of declarative programming that has proven very successful in succinctly formulating and solving complex problems. Although mechanisms for representing and reasoning with the combined answer set programs of multiple agents have already been proposed, the actual gain in expressivity when adding communication has not been thoroughly studied. We show that allowing simple programs to talk to each other results in the same expressivity as adding negation-as-failure. Furthermore, we show that the ability to focus on one program in a network of simple programs results in the same expressivity as adding disjunction in the head of the rules.

1. Introduction

The idea of answer set programming (ASP) is to represent the requirements of a computational problem by a logic program P such that particular minimal models of P, called answer sets and usually defined using some form of the stable model semantics [Gel88], correspond to the solutions of the original problem [Lif02]. The research on multi-context systems has, among other things, been concerned with studying how a group of simple agents can cooperate to find the solutions of global problems [Roe05, Bre07]. We start with an introductory example to illustrate how the ideas of multi-context systems can be used to solve problems in the ASP setting.

Kim Bauters and Jeroen Janssen are funded by a joint Research Foundation-Flanders (FWO) project. Steven Schockaert is a postdoctoral fellow of the Research Foundation-Flanders (FWO).

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.34

Example 1.1. A hotspot network consists of two hotspots H_1 and H_2. The hotspots are wired to each other to share an internet connection and provide wireless access to users in the area. A user U tries to connect to the closest detectable hotspot *e.g.* H_1. Now assume that H_1 is no longer accessible. H_1 cannot find this out by itself, nor can it rely on users telling this since they cannot connect. The rules below illustrate how we can model this knowledge using the communicating programs we describe in Section 2.2. For compactness, we abbreviate *accessible* as a, *access* as c, *problem* as p and *optimal* as o. Consider the program $\mathcal{P}_{intro}$ with the rules:

$$H_1 : \neg a \leftarrow H_2 : p \quad \text{[r1]} \qquad U : o \leftarrow H_1 : a \quad \text{[r5]}$$
$$H_2 : a \leftarrow \quad \text{[r2]} \qquad U : \neg o \leftarrow H_2 : a, not\ H_1 : a \quad \text{[r6]}$$
$$H_2 : \neg a \leftarrow H_1 : p \quad \text{[r3]} \qquad U : c \leftarrow H_1 : a \quad \text{[r7]}$$
$$H_2 : p \leftarrow U : \neg o \quad \text{[r4]} \qquad U : c \leftarrow H_2 : a \quad \text{[r8]}$$

Let $H_1 = \{\mathsf{r1}\}$, $H_2 = \{\mathsf{r2}, \mathsf{r3}, \mathsf{r4}\}$ and $U = \{\mathsf{r5}, \mathsf{r6}, \mathsf{r7}, \mathsf{r8}\}$. Note how the rules of the first hotspot H_1 differ from those of the second hotspot H_2, *i.e.* they are in different states. Indeed, the first hotspot H_1 cannot rely on the user to tell that there is a problem and it is not accessible. The second hotspot does not have these restrictions. It is easy to see that user U can deduce that she has access (r2, r8), though this access is not optimal (r6). The second hotspot detects this (r4) and concludes that there is a problem, allowing H_1 to derive that it is not accessible (r1).

In this paper we systematically study the effect of adding such kind of communication to ASP in terms of expressiveness. The communication between ASP programs that we propose is similar in spirit to the work in [Roe05, Bre07, Buc08]. Studying the expressiveness with a focus on simple ASP programs, however, is in contrast to approaches such as [De05, Van07] that start from expressive ASP variants, which obscures the analysis of the effect of communication on the expressiveness. A first contribution of this paper is that communicating simple programs can solve problems at the first level of the polynomial hierarchy and that communicating normal ASP programs do not offer any additional expressiveness. The second contribution is the introduction of a new, intuitive form of communication that allows for communicating simple ASP programs to solve problems at the second level of the polynomial hierarchy. The hardness results that we present in this paper in a sense complement the membership results from [Bre07]. However, our definitions of communicating ASP and minimality differ slightly, complicating a direct comparison of the results.

The remainder of this paper is organized as follows. Section 2 recalls the basic concepts and results from ASP which we use in this paper, and explores the syntax and semantics of communicating programs. In Section 3 we show that these communicating simple programs are capable of simulating normal programs that have negation-as-failure. In Section 4 we introduce focused communicating programs and show how networks of simple agents can simulate disjunctive ASP programs. Related work is discussed in Section 5 and Section 6 provides some final remarks.

2. Preliminaries

2.1. Answer set programming

We first recall the basic concepts and results from ASP that are used in this paper. To define ASP programs, we start from a countable set of atoms and we define a *literal* l as an atom a or its classical negation $\neg a$. If L is a set of literals, we use $\neg L$ to denote the set $\{\neg l \mid l \in L\}$ where, by definition, $\neg\neg a = a$. A set of literals L is *consistent* if $L \cap \neg L = \emptyset$. An *extended literal* is either a literal or a literal preceded by *not* which we call the negation-as-failure operator. For a set of literals L, we use $not(L)$ to denote the set $\{not\ l \mid l \in L\}$.

A *disjunctive rule* is an expression of the form $\gamma \leftarrow (\alpha \cup not(\beta))$ where γ is a set of literals (interpreted as a disjunction, denoted as $l_1; \ldots; l_n$) called the head of the rule and $(\alpha \cup not(\beta))$ (interpreted as a conjunction) is the body of the rule with α and β sets of literals. A *positive disjunctive rule* is a disjunctive rule without negation-as-failure in the body, *i.e.* with $\beta = \emptyset$. A *disjunctive program* P is a finite set of disjunctive rules. The *Herbrand base* $\mathcal{B}_P$ of P is the set of atoms appearing in program P. A (partial) *interpretation* I of P is any consistent set of literals $I \subseteq (\mathcal{B}_P \cup \neg\mathcal{B}_P)$. I is *total* iff $I \cup \neg I = \mathcal{B}_P \cup \neg\mathcal{B}_P$.

A *normal rule* is a disjunctive rule with at most one literal l in the head. A *normal program* P is a finite set of normal rules. A *simple rule* is a normal rule without negation-as-failure in the body. A *simple program* P is a finite set of simple rules. The *immediate consequence operator* T_P of a simple program P *w.r.t.* an interpretation I is defined as

$$T_P(I) = I \cup \{l \mid ((l \leftarrow \alpha) \in P) \wedge (\alpha \subseteq I)\}. \tag{2.1}$$

We use $P^\star$ to denote the fixpoint which is obtained by repeatedly applying T_P starting from the empty interpretation, *i.e.* the least fixpoint of T_P *w.r.t.* set inclusion. An interpretation I is an *answer set* of a simple program P iff $I = P^\star$.

The *reduct* P^I of a disjunctive program P *w.r.t.* the interpretation I is defined as $P^I = \{\gamma \leftarrow \alpha \mid (\gamma \leftarrow \alpha \cup not(\beta)) \in P, \beta \cap I = \emptyset\}$. I is an answer set of the disjunctive program P when I is the minimal model *w.r.t.* set inclusion of P^I. In the specific case of normal programs, answer sets can also be characterized in terms of fixpoints. Specifically, it is easy to see that the reduct P^I is a simple program. I is an answer set of the normal program P iff $\left(P^I\right)^\star = I$, *i.e.* if I is the answer set of the reduct P^I.

2.2. Communicating programs

The underlying intuition of communication between ASP programs is that of a function call or, in terms of agents, asking questions to other agents. This communication is based on a new kind of literal '$Q{:}l$', as in [Roe05, Bre07]. If the literal l is not in the answer set of Q then $Q{:}l$ is false; otherwise $Q{:}l$ is true. The semantics are closely related to the minimal semantics in [Bre07] and especially the semantics in [Buc08].

Let $\mathcal{P}$ be a finite set of program names. A *$\mathcal{P}$-situated literal* is an expression of the form $Q{:}l$ with $Q \in \mathcal{P}$ and l a literal. For $R \in \mathcal{P}$, a $\mathcal{P}$-situated literal $Q{:}l$ is called *R-local* if $Q = R$. For a set of literals L, we use $Q{:}L$ as a shorthand for $\{Q{:}l \mid l \in L\}$. For a set of $\mathcal{P}$-situated literals X and $Q \in \mathcal{P}$, we use $X_{\downarrow Q}$ to denote $\{l \mid Q{:}l \in X\}$, i.e. the projection of X on Q. A set of $\mathcal{P}$-situated literals X is *consistent* iff $X_{\downarrow Q}$ is consistent for all $Q \in \mathcal{P}$. By $\neg X$ we denote the set $\{Q{:}\neg l \mid Q{:}l \in X\}$ where we define $Q{:}\neg\neg l = Q{:}l$. An *extended $\mathcal{P}$-situated literal* is either a $\mathcal{P}$-situated literal or a $\mathcal{P}$-situated literal preceded by *not*. For

a set of $\mathcal{P}$-situated literals X, we use $not(X)$ to denote the set $\{not\ Q{:}l \mid Q{:}l \in X\}$. For a set of extended $\mathcal{P}$-situated literals X we denote by X_{pos} the set of $\mathcal{P}$-situated literals in X, *i.e.* those extended $\mathcal{P}$-situated literals in X that are not preceded by negation-as-failure, while $X_{\text{neg}} = \{Q{:}l \mid not\ Q{:}l \in X\}$.

A *$\mathcal{P}$-situated normal rule* is an expression of the form $Q : l \leftarrow (\alpha \cup not(\beta))$ where $Q : l$ is a single $\mathcal{P}$-situated literal, called the head of the rule, and $(\alpha \cup not(\beta))$ is called the body of the rule with α and β sets of $\mathcal{P}$-situated literals. A $\mathcal{P}$-situated normal rule $Q{:}l \leftarrow (\alpha \cup not\ (\beta))$ is called R-local whenever $Q = R$. A *$\mathcal{P}$-component normal program* Q is a finite set of Q-local $\mathcal{P}$-situated normal rules. Henceforth we shall use $\mathcal{P}$ to both denote the set of program names and to denote the set of actual $\mathcal{P}$-component normal programs. A *communicating normal program* $\mathcal{P}$ is then a finite set of $\mathcal{P}$-component normal programs.

A *$\mathcal{P}$-situated simple rule* is an expression of the form $Q{:}l \leftarrow \alpha$, *i.e.* a $\mathcal{P}$-situated normal rule without negation-as-failure in the body. A *$\mathcal{P}$-component simple program* Q is a finite set of Q-local $\mathcal{P}$-situated simple rules. A *communicating simple program* $\mathcal{P}$ is then a finite set of $\mathcal{P}$-component simple programs.

In the remainder of this paper we drop the $\mathcal{P}$-prefix whenever the set $\mathcal{P}$ is clear from the context. Whenever the name of the component normal program Q is clear, we write l instead of $Q : l$ for Q-local situated literals. For notational convenience, we write communicating program for communicating normal program. Finally note that a communicating normal (simple) program with only one component program trivially corresponds to a normal (simple) program.

Similar as for a normal program, we can define the *Herbrand base* for a component program Q as the set of atoms occurring in Q, which we denote as $\mathcal{B}_Q$. The Herbrand base of a communicating program $\mathcal{P}$ is defined as $\mathcal{B}_{\mathcal{P}} = \left\{Q{:}a \mid Q \in \mathcal{P} \text{ and } a \in \bigcup_{R \in \mathcal{P}} \mathcal{B}_R\right\}$. We say that a (partial) interpretation I of a communicating program $\mathcal{P}$ is any consistent subset $I \subseteq (\mathcal{B}_{\mathcal{P}} \cup \neg\mathcal{B}_{\mathcal{P}})$. Given an interpretation I of a communicating program $\mathcal{P}$, the reduct Q^I for $Q \in \mathcal{P}$ is the component simple program obtained by deleting

- each rule with an extended situated literal $not\ R{:}l$ in the body such that $R{:}l \in I$;
- each remaining extended situated literal of the form $not\ R{:}l$;
- each rule with a situated literal $R{:}l$ in the body that is not Q-local with $R{:}l \notin I$;
- each situated literal $R{:}l$ that is not Q-local and such that $R{:}l \in I$.

The underlying intuition of the reduct is clear. Analogous to the definition of a reduct of a normal programs [Gel88], the reduct of a communicating program defines a way to reduce this program relative to some guess I. The reduct of a communicating program is a communicating simple program that only contains component simple programs Q with Q-local situated literals. That is, each component simple program Q corresponds to a classical simple program. We tackle the problem of self-references in [Buc08] by treating Q-local situated literals in a different way. Since the communication is based on belief and internal reasoning is based on knowledge, this allows for "mutual influence" as in [Bre07, Buc08] where the belief of an agent can be supported by the agent itself, via belief in other agents. Also note that the belief between agents is the belief as identified in [Lif99], *i.e.* $Q : l$ is true whenever "$\neg not\ Q{:}l$" is true under the syntax and semantics introduced in [Lif99] for nested logic programs and when treating $Q{:}l$ as a fresh atom.

Definition 2.1. We say that an interpretation I of a communicating program $\mathcal{P}$ is an *answer set* of $\mathcal{P}$ if and only if we have that $\forall Q \in \mathcal{P} \cdot (Q{:}I_{\downarrow Q}) = \left(Q^I\right)^\star$.

Example 2.2. Consider the communicating program $\mathcal{P}_{intro}$ from Example 1.1. It is easy to see that $M = \{H_1 : \neg a, H_2 : a, H_2 : p, U : \neg o, U : c\}$ is the unique answer set of $\mathcal{P}_{intro}$. Indeed, we obtain the reducts $(H_1)^M = \{\neg a \leftarrow\}$, $(H_2)^M = \{a \leftarrow, p \leftarrow\}$ and $(U)^M = \{\neg o \leftarrow, c \leftarrow\}$ which have the answer sets $\{\neg a\}, \{a, p\}$ and $\{\neg o, c\}$, respectively.

3. Simulating Negation-as-Failure with Communication

The addition of communication to ASP programs provides added expressiveness and an increase in computational complexity, which we illustrate in this section. We show that a communicating simple program can simulate normal programs, where simple programs are P-complete and normal programs are NP-complete [Bar03]. Furthermore, we illustrate that, surprisingly, there is no difference in terms of computational complexity between communicating simple programs and communicating normal programs.

We start by giving an example of the transformation that allows to simulate (communicating) normal programs using communicating simple programs. Afterwards, we give a formal definition of this transformation.

Example 3.1. Consider the communicating normal program $\mathcal{E}$ with the rules

$$Q_1 : a \leftarrow not\ Q_2 : b$$
$$Q_2 : b \leftarrow not\ Q_1 : a.$$

When $Q_1 = Q_2$ this example corresponds to a normal program. The transformation we propose below results in the communicating simple program $\mathcal{P} = \{Q_1', Q_2', N_1, N_2\}$:

$$\begin{aligned} Q_1' : a &\leftarrow N_2 : \neg(b)^\dagger & \qquad N_1 : (a)^\dagger &\leftarrow Q_1' : a \\ Q_2' : b &\leftarrow N_1 : \neg(a)^\dagger & \qquad N_2 : (b)^\dagger &\leftarrow Q_2' : b \\ Q_1' : \neg(a)^\dagger &\leftarrow N_1 : \neg(a)^\dagger & \qquad N_1 : \neg(a)^\dagger &\leftarrow Q_1' : \neg(a)^\dagger \\ Q_2' : \neg(b)^\dagger &\leftarrow N_2 : \neg(b)^\dagger & \qquad N_2 : \neg(b)^\dagger &\leftarrow Q_2' : \neg(b)^\dagger. \end{aligned}$$

The transformation creates two types of *'worlds'*, Q_i' and N_i with $1 \leq i \leq 2$, which are all component programs. Q_i' is similar to Q_i, although occurrences of extended situated literals of the form $not\ Q_i : l$ are replaced by $N_i : \neg(l)^\dagger$, with $(l)^\dagger$ a fresh literal. The non-monotonicity associated with negation-as-failure is simulated by introducing the rules $\neg(l)^\dagger \leftarrow N_i : \neg(l)^\dagger$ and $\neg(l)^\dagger \leftarrow Q_i' : \neg(l)^\dagger$ in Q_i' and N_i, respectively. Finally, we add rules of the form $(l)^\dagger \leftarrow Q_i' : l$ to N_i, creating an inconsistency when N_i believes $\neg(l)^\dagger$ when Q_i' believes l.

The resulting communicating simple program $\mathcal{P}$ is an equivalent program in that its answer sets correspond to those of the original communicating program, yet without using negation-as-failure. Indeed, the answer sets of $\mathcal{E}$ are $\{Q_1 : a\}$ and $\{Q_2 : b\}$ and the answer sets of $\mathcal{P}$ are $\left\{Q_1' : a, Q_2' : \neg(b)^\dagger, N_2 : \neg(b)^\dagger, N_1 : (a)^\dagger\right\}$ and $\left\{Q_2' : b, Q_1' : \neg(a)^\dagger, N_1 : \neg(a)^\dagger, N_2 : (b)^\dagger\right\}$. Note furthermore how this is a polynomial transformation with at most $3 \cdot |\mathcal{E}_{\text{neg}}|$ additional rules with $\mathcal{E}_{\text{neg}}$ as defined in Definition 3.2.

Definition 3.2. Let $\mathcal{E} = \{Q_1, \ldots, Q_n\}$ be a communicating program. The communicating simple program $\mathcal{P} = \{Q_1', \ldots, Q_n', N_1, \ldots, N_n\}$ with $1 \leq i, j \leq n$ that simulates $\mathcal{E}$ is defined

by

$$
\begin{aligned}
Q'_i &= \left\{ l \leftarrow \alpha'_{\text{pos}} \cup \left\{ N_j{:}\neg(k)^\dagger \mid Q_j{:}k \in \alpha_{\text{neg}} \right\} \mid (l \leftarrow \alpha) \in Q_i \right\} && (3.1)\\
&\cup \left\{ \neg(b)^\dagger \leftarrow N_i{:}\neg(b)^\dagger \mid Q_i{:}b \in \mathcal{E}_{\text{neg}} \right\} && (3.2)\\
N_i &= \left\{ \neg(b)^\dagger \leftarrow Q'_i{:}\neg(b)^\dagger \mid Q_i{:}b \in \mathcal{E}_{\text{neg}} \right\} && (3.3)\\
&\cup \left\{ (b)^\dagger \leftarrow Q'_i{:}b \mid Q_i{:}b \in \mathcal{E}_{\text{neg}} \right\} && (3.4)
\end{aligned}
$$

with $\alpha'_{pos} = \left\{ Q'_j{:}l \mid Q_j{:}l \in \alpha_{pos} \right\}$ and $\mathcal{E}_{\text{neg}} = \bigcup_{i=1}^{n} \left(\bigcup_{(a \leftarrow \alpha) \in Q_i} \alpha_{\text{neg}} \right)$.

Recall that both $\neg(b)^\dagger$ and $(b)^\dagger$ are fresh literals that intuitively correspond to $\neg b$ and b. We use Q'_i+ to denote the rules in Q'_i defined by (3.1) and Q'_i- to denote the rules in Q'_i defined by (3.2).

Intuitively, the transformation employs the non-monotonic property of the belief underlying the situated literals to simulate negation-as-failure. This is obtained from the interplay between the rules (3.2) and (3.3). As such, we can use the new literal '$\neg(b)^\dagger$' instead of the original extended (situated) literal '*not* b', allowing us to rewrite the rules as we do in (3.1). In order to ensure that the simulation works, even when the program we want to simulate contains true negation, we need to specify some additional bookkeeping (3.4).

As becomes clear from Proposition 3.3 and Proposition 3.4, the above transformation preserves the semantics of the original program. Since we can easily rewrite any normal program as a communicating normal program, the importance of this is thus twofold. On one hand, we reveal that communicating normal programs do not have any additional expressive power over communicating simple programs. On the other hand, it follows that the expressiveness of communicating simple programs allows us to solve NP-complete problems, since finding the answer set of normal programs is an NP-complete problem [Bar03].

Proposition 3.3. *Let $\mathcal{P} = \{Q_1, \ldots, Q_n\}$ and let $\mathcal{P}' = \{Q'_1, \ldots, Q'_n, N_1, \ldots, N_n\}$ with $\mathcal{P}$ a communicating program and $\mathcal{P}'$ the communicating simple program that simulates $\mathcal{P}$ as defined in Definition 3.2. If M is an answer set of $\mathcal{P}$, then M' is an answer set of $\mathcal{P}'$ with M' defined as:*

$$
\begin{aligned}
M' = &\left\{ Q'_i{:}a \mid a \in M_{\downarrow Q_i}, Q_i \in \mathcal{P} \right\}\\
&\cup \left\{ Q'_i{:}\neg(b)^\dagger \mid b \notin M_{\downarrow Q_i}, Q_i \in \mathcal{P} \right\}\\
&\cup \left\{ N_i{:}\neg(b)^\dagger \mid b \notin M_{\downarrow Q_i}, Q_i \in \mathcal{P} \right\}\\
&\cup \left\{ N_i{:}(a)^\dagger \mid a \in M_{\downarrow Q_i}, Q_i \in \mathcal{P} \right\}.
\end{aligned} \tag{3.5}
$$

Proposition 3.4. *Let $\mathcal{P} = \{Q_1, \ldots, Q_n\}$ and let $\mathcal{P}' = \{Q'_1, \ldots, Q'_n, N_1, \ldots, N_n\}$ with $\mathcal{P}$ a communicating program and $\mathcal{P}'$ the communicating simple program that simulates $\mathcal{P}$. Assume that M' is an answer set of $\mathcal{P}'$ and that $(M')_{\downarrow N_i}$ is total* w.r.t. *$\mathcal{B}_{N_i}$ for all $i \in \{1, \ldots, n\}$. Then the interpretation M defined as*

$$
M = \left\{ Q_i{:}b \mid b \in \left(\left(Q'_i+ \right)^{M'} \right)^\star \right\} \tag{3.6}
$$

is an answer set of $\mathcal{P}$. ∎

Note that the requirement for M' to be a total answer set of $\mathcal{P}$ in N_i is necessary in this last proposition, as demonstrated by the following example.

Example 3.5. Consider the normal program $R = \{a \leftarrow not\ a\}$ which has no answer sets. The corresponding communicating simple program $\mathcal{P} = \{Q, N\}$ has the following rules:

$$Q : a \leftarrow N : \neg(a)^\dagger \qquad N : \neg(a)^\dagger \leftarrow Q : \neg(a)^\dagger$$
$$Q : \neg(a)^\dagger \leftarrow N : \neg(a)^\dagger \qquad N : (a)^\dagger \leftarrow Q : a.$$

It is easy to see that $I = \emptyset$ is an answer set of $\mathcal{P}$ since we have $Q^I = N^I = \emptyset$.

4. Focused Communicating Programs

In this section, we extend the semantics of communicating programs in such a way that it is possible to focus on a single component program. That is, we indicate that we are not interested in the answer sets of the entire network of component programs, but only in answer sets of a single component program. The underlying intuition is that of auxiliary functions or, in terms of agents, a team governed by a leader who forwards (and possibly amends) the conclusions. We are thus varying the communication mechanism, without altering the expressiveness of the agents in the network.

Definition 4.1. Let $\mathcal{P}$ be a communicating program and $Q \in \mathcal{P}$ a component program. A *Q-focused answer set* of $\mathcal{P}$ is any subset-minimal element of

$$\{M_{\downarrow Q} \mid M \text{ an answer set of } \mathcal{P}\}.$$

If we are only interested in Q-focused answer sets, then $\mathcal{P}$ is called a Q-focused communicating program, denoted as $\mathcal{P}_{\downarrow Q}$. As before, we drop the Q-prefix when the component program Q is clear from the context.

Example 4.2. Consider the communicating program $\mathcal{P}_{focus} = \{Q, R\}$ with the rules

$$Q = \{a \leftarrow, b \leftarrow, c \leftarrow not\ R : c\}$$
$$R = \{a \leftarrow not\ c, c \leftarrow not\ a, d \leftarrow c\}.$$

The communicating program $\mathcal{P}_{focus}$ has two answer sets, namely $M_1 = Q : \{a, b, c\} \cup \{R : a\}$ and $M_2 = Q : \{a, b\} \cup R : \{c, d\}$. The only Q-focused answer set of $\mathcal{P}_{focus}$ is $\{a, b\}$ since $M_{1\downarrow Q} = \{a, b, c\}$ and $M_{2\downarrow Q} = \{a, b\}$.

This simple extension is all that is needed to take another step in the complexity hierarchy. That is, the complexity of finding the answer sets of a focused communicating program is Σ_2^{P}-hard.[1] Before we state this result, we first explain that any positive disjunctive program can be simulated using focused communicating programs. The underlying intuition is straightforward. We delegate the disjunction in the head to a new component program where we simulate the corresponding choice using negation-as-failure. The results of these component programs are then grouped in an aggregate component program on which we focus to ensure that we only retain the minimal models that correspond with the answer sets of the original positive disjunctive program. We start with an example to illustrate the simulation.

[1]Recall that Σ_2^{P} is the class of problems that can be solved in polynomial time on a non-deterministic machine with an NP oracle, *i.e.* $\Sigma_2^{\mathsf{P}} = \mathsf{NP}^{\mathsf{NP}}$.

Example 4.3. Consider the positive disjunctive program $D = \{a; b \leftarrow, a \leftarrow b, b \leftarrow a\}$. The corresponding focused program $(\mathcal{P}_{simulate})_{\downarrow Q} = \{Q, R_1\}$ has the following rules:

$$\begin{array}{ll} R_1{:}a \leftarrow not\ R_1{:}b & Q{:}a \leftarrow R_1{:}a \\ R_1{:}b \leftarrow not\ R_1{:}a & Q{:}b \leftarrow R_1{:}b \\ & Q{:}a \leftarrow Q{:}b \\ & Q{:}b \leftarrow Q{:}a \end{array}$$

The answer sets of $\mathcal{P}_{simulate}$ are $\{R_1{:}a\} \cup Q{:}\{a, b\}$ and $\{R_1{:}b\} \cup Q{:}\{a, b\}$. The unique answer set of $(\mathcal{P}_{simulate})_{\downarrow Q}$ is therefore $\{a, b\}$, which is also the unique answer set of D.

Definition 4.4. Let $D = \{r_1, \ldots, r_n, r_{n+1}, \ldots, r_s\}$ be a positive disjunctive program where $r_i = \gamma_i \leftarrow \alpha_i$ such that $|\gamma_i| > 1$ for $i \in \{1, \ldots, n\}$ and $|\gamma_i| \in \{0, 1\}$ for $i \in \{n+1, \ldots, s\}$. The focused program that simulates D, $\mathcal{P}_{\downarrow Q} = \{Q, R_1, \ldots, R_n\}$, is defined by

$$\begin{aligned} Q = &\{r_i \mid i \in \{n+1, \ldots, s\}\} \\ &\cup \{l \leftarrow \{R_i{:}l\} \cup \alpha_i \mid i \in \{1, \ldots, n\}, l \in \gamma_i\} \end{aligned} \tag{4.1}$$

where for $i \in \{1, \ldots, n\}$ we have

$$R_i = \{l \leftarrow not(\gamma_i \setminus \{l\}) \mid l \in \gamma_i\}. \tag{4.2}$$

Proposition 4.5. *Let D be a positive disjunctive program and $\mathcal{P}_{\downarrow Q}$ the focused communicating program that simulates D. M is an answer set of D iff M is an answer set of $\mathcal{P}_{\downarrow Q}$.* ∎

We can thus use focused communicating programs to solve existential-universal quantifiable boolean formulae (*e.g.* by simulating the disjunctive ASP program proposed in [Bar03]). This can be used as the basis of a proof to show that finding the answer sets of focused communicating programs is in Σ_2^{P}.

Corollary 4.6. *Deciding whether a Q-focused communicating (simple) program $\mathcal{P}_{\downarrow Q}$ with two or more component programs has an answer set containing a specific literal l is Σ_2^{P}-hard. Membership in Σ_2^{P} can also be shown, thus this problem is Σ_2^{P}-complete.* ∎

5. Related Work

A large body of research has been devoted to combining logic programming with multi-agent or multi-context ideas for various reasons. Among others, the logic can be used to describe the (rational) behaviour of the agents in a multi-agent network, as in [Del99]. It can be used to combine different flavours of logic programming languages [Luo05, Eit08]. It can be used to externally solve tasks for which ASP is not suited, yet remaining in a declarative framework [Eit06]. It can also be used as a form of cooperation, where multiple agents or contexts collaborate to solve a difficult problem [De05, Van07].

The approach described in this paper falls into this last category and studies the expressiveness of the communication component in communicating ASP. In contrast to [De05, Van07] our approach is based on simple programs and on asking for information instead of pushing (partial) answer sets to the next ASP program in the network. Like in [De05], but in contrast with [Van07], we allow circular communication between programs and do not force a linear network of ASP programs that in turn refine the results of previous steps.

Table 1: Complexity of Communicating Answer Set Programming

	no communication	with communication	focused communication
simple program	P-hard	NP-hard	Σ_2^P-hard
normal program	NP-hard	NP-hard	Σ_2^P-hard

Complexity studies in this setting have been performed but with some notable differences. For example, [Bre07] generalises towards heterogenous non-monotonic multi-context systems in which different flavours of logic programming languages work together to solve a problem.

It is shown that the complexity of verifying whether some literal is contained in some (resp. all) solutions is in Σ_k^P (resp. Π_k^P), where the value of k depends on the underlying logic that is used.

In [DT09], recursive modular nonmonotonic logic programs (MLP) under the ASP semantics are considered. The main difference between MLP and our simple communication is that our communication is parameter-less, *i.e.* the truth of a situated literal is not dependent on parameters passed by the situated literal to the target component program.

The work in this paper is different from all of the above in that it studies the expressiveness of communicating answer set programs with simple rules while varying the mechanisms for parameter-less communication between the agents.

6. Conclusion

In this paper we have systematically studied the effect of adding communication to ASP in terms of expressiveness and computational complexity. One of the most interesting results is that communicating simple programs (without negation-as-failure) are expressive enough to simulate communicating normal programs (with negation-as-failure). To show this, we have provided an actual translation of a communication normal ASP program into an equivalent communicating ASP program with only simple rules. Since normal programs are a special case of communicating normal programs, and solving normal programs is known to be NP-complete, this entails that solving communicating simple programs is an NP-hard problem.

Additionally, we introduce focused communicating programs where we "focus" on the results of a single component program. The other component programs can still contribute to solving the problem at hand, but they no longer have a direct influence over the resulting answer set. Indeed, the component program on which we focus can override any and all conclusions. Such focused communicating programs can easily be obtained by varying the parameter-less communication mechanism found in the communicating programs introduced in the first part of this paper. Focused communicating programs can be used to simulate programs with disjunctive rules without negation-as-failure and are able to solve problems in Σ_2^P. Table 1 summarises our main results.

Acknowledgment

The authors wish to thank the anonymous reviewers for their references to related work, as well as for their comments and suggestions that helped to improve the quality of this

paper. Special thanks go out to Pascal Nicolas, Claire Lefèvre and Laurent Garcia for their fruitful discussions that led to new insights.

References

[Bar03] Chitta Baral. *Knowledge, Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[Bre07] Gerhard Brewka and Thomas Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proc. of AAAI07*, pp. 385–390. 2007.

[Buc08] Francesco Buccafurri, Gianluca Caminiti, and Rosario Laurendi. A logic language with stable model semantics for social reasoning. In *Proc. of ICLP08*, pp. 718–723. 2008.

[De05] Marina De Vos, Tom Crick, Julian Padget, Martin Brain, Owen Cliffe, and Jonathan Needham. LAIMA: A multi-agent platform using ordered choice logic programming. In *Declarative Agent Languages and Technologies III*, pp. 72–88. 2005.

[Del99] Pierangelo Dell'Acqua, Fariba Sadri, and Francesca Toni. Communicating agents. In *Proceedings of the International Workshop on Multi-Agent Systems in Logic Programming*. 1999.

[DT09] Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Modular nonmonotonic logic programming revisited. In *Proc. of ICLP*, pp. 145–159. 2009.

[Eit06] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. dlvhex: A tool for semantic-web reasoning under the answer-set semantics. In *Proceedings of International Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services*, pp. 33–39. 2006.

[Eit08] Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artifial Intelligence*, 172(12–13):1495–1539, 2008.

[Gel88] Michael Gelfond and Vladimir Lifzchitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pp. 1081–1086. 1988.

[Lif99] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Ann. Math. Artif. Intell.*, 25(3-4):369–389, 1999.

[Lif02] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.

[Luo05] Jiewen Luo, Zhongzhi Shi, Maoguang Wang, and He Huang. Multi-agent cooperation: A description logic view. In *Proc. of PRIMA05*, pp. 365–379. 2005.

[Roe05] Floris Roelofsen and Luciano Serafini. Minimal and absent information in contexts. In *Proc. of IJCAI05*, pp. 558–563. 2005.

[Van07] Davy Van Nieuwenborgh, Marina De Vos, Stijn Heymans, and Dirk Vermeir. Hierarchical decision making in multi-agent systems using answer set programming. In *Proc. of CLIMA07*. 2007.