A General Method To Solve Complex Problems By Combining Multiple Answer Set Programs

Marcello Balduccini

Intelligent Systems, OCTO
Eastman Kodak Company
Rochester, NY 14650-2102 USA
marcello.balduccini@gmail.com

Abstract This paper introduces the notion of Answer Set Programming (ASP) Machine, which is loosely inspired to Turing's Oracle Machines. The aim of this research is to use ASP "oracles" and ASP-represented transition systems to allow programmers to use exclusively ASP to solve problems that are beyond the expressive power of the basic language, rather than having to resort to auxiliary, often procedural programs. The advantage of our approach is a more uniform level of abstraction throughout the programs used to solve the problem, greater elaboration tolerance, as well as simplified proofs of the properties of programs, such as soundness and completeness.

1 Introduction

Answer Set Programming (ASP) [1,2] is a powerful programming paradigm that features sophisticated knowledge representation and reasoning capabilities.

In recent years, ASP has been used for a number of successful applications (e.g. [3,4,5,6]). One hurdle that programmers often face when using ASP, is that normal programs[1] can only solve NP-complete problems [7]. Whenever a problem of higher computational complexity is to be solved, either a suitable extension of ASP has to be selected (e.g. disjunctive programs, CR-Prolog [8], weak constraints [9]), or ad-hoc solutions have to be devised, such as writing an auxiliary, often procedural, program. In this second approach, the problem is reduced to computing the answer sets of a suitable sequence of ASP programs. For example, to find shortest plans, one can use the `#minimize` statement of LPARSE/SMODELS [10], or (in particular to have more control on the search strategy) write a normal program that solves the decision problem, and an auxiliary program that solves the optimization problem by selecting at each step a different plan length limit, and by checking if the normal program together with a specification of the given limit is consistent.

Unfortunately, it is often not easy to find suitable extensions of ASP that can be used in a natural way to solve the problem at hand. In that case, programmers are forced to write auxiliary programs. As we mentioned above, in other cases, one resorts to the use of auxiliary programs to have more control on the search strategy and increase efficiency.

[1] That is, logic programs with negation as failure but no disjunction.

Such auxiliary programs are typically relatively conceptually simple, but they still substantially complicate the task of proving properties of the overall program, such as soundness and completeness. Moreover, from a practical point of view, these auxiliary programs, being often procedural, require a substantial shift of perspective, and their writing involves the rather error-prone and cumbersome task of writing specifications at a substantially different level of abstraction compared to the ASP programs at the core of the application.

An additional motivation to the exclusive use of ASP stems from the recent observation [11,12] that describing an algorithm using a transition system instead of pseudocode makes it easier to prove its properties, compare it with other algorithms, and design new algorithms.

In this paper we explore an extension of the ASP paradigm that exploits these observations. Our approach to solving problems of high complexity is indeed based on reducing the task to computing the answer sets of a suitable sequence of ASP programs, but we use ASP to form such sequence. Although we do not suggest that our approach be used indiscriminately for general-purpose programming, we believe that it can be useful in simplifying the task of writing many ASP-based applications, as well as for testing search strategies before implementing with more efficient paradigms.

The paper is structured as follows. In Section 3 we describe our framework and show its application to a well-known NP-hard problem. In Section 4 we show how to the computations involved in our framework can be automated. In Section 5 we discuss related work. Finally, in Section 6 we draw conclusions and discuss future work.

2 Background

The syntax and semantics of answer set programming [1,2] are defined as follows. Let Σ be a signature containing constant, function and predicate symbols. Terms and atoms are formed as usual. A literal is either an atom a or its strong (also called classical or epistemic) negation $\neg a$. The sets of atoms and literals formed from Σ are denoted by $atoms(\Sigma)$ and $literals(\Sigma)$ respectively.

A *rule* is a statement of the form:[2]

$$[r]\ h \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n \tag{1}$$

where r is an optional name of the rule (a label useful when talking about the rule), h and l_i's are literals and *not* is the so-called *default negation*. The intuitive meaning of the rule is that a reasoner who believes $\{l_1, \ldots, l_m\}$ and has no reason to believe $\{l_{m+1}, \ldots, l_n\}$, has to believe h. We call h the *head* of the rule, and $\{l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n\}$ the *body* of the rule. Given a rule r, we denote its head and body by $head(r)$ and $body(r)$ respectively.

[2] For simplicity we focus on non-disjunctive programs. Our results extend to disjunctive (and other) programs in a natural way.

Often, rules of the form $h \leftarrow \text{not } h, l_1, \ldots, \text{not } l_n$ are abbreviated into $\leftarrow l_1, \ldots, \text{not } l_n$, and called *constraints*. The intuitive meaning of a constraint is that its body must not be satisfied.

An ASP *program* (or program for short) is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a set of rules over Σ. Often we denote programs by just the second element of the pair, and let the signature be defined implicitly. In that case, the signature of Π is denoted by $\Sigma(\Pi)$.

A set A of literals is *consistent* if no two complementary literals, a and $\neg a$, belong to A. A literal l is *satisfied* by a consistent set of literals A if $l \in A$. In this case, we write $A \models l$. If l is not satisfied by A, we write $A \not\models l$. A set $\{l_1, \ldots, l_k\}$ of literals is satisfied by a set A of literals ($A \models \{l_1, \ldots, l_k\}$) if each l_i is satisfied by A.

Programs not containing default negation are called *definite*. A consistent set of literals A is *closed* under a definite program Π if, for every rule of the form (1) such that the body of the rule is satisfied by A, the head belongs to A.

Definition 1. *A consistent set of literals A is an* answer set *of definite program Π if A is closed under all the rules of Π and A is set-theoretically minimal among the sets closed under all the rules of Π.*

The *reduct* of a program Π with respect to a set of literals A, denoted by Π^A, is the program obtained from Π by deleting:

- Every rule, r, such that $l \in A$ for some expression of the form not l from the body for r;
- All expressions of the form not l from the bodies of the remaining rules.

We are now ready to define the notion of answer set of a program.

Definition 2. *A consistent set of literals A is an* answer set *of program Π if it is an answer set of the reduct Π^A.*

To simplify the programming task, variables are often allowed to occur in ASP programs. A rule containing variables (called a *non-ground* rule) is then viewed as a shorthand for the set of its *ground instances*, obtained by replacing the variables in it by all the possible ground terms. Similarly, a non-ground program is viewed as a shorthand for the program consisting of the ground instances of its rules.

Later, we will also need the following notation. Given a program Π and a literal l, we say that *Π entails l*, and write $\Pi \models l$, if, for every answer set A of Π, $A \models l$. We say that *Π does not entail l*, and write $\Pi \not\models l$, if there exists one answer set A of Π such that $A \not\models l$.

3 ASP Machines and Execution Traces

Our approach is loosely inspired to the notion of Oracle Turing Machines [13], and consists in reducing the computation needed to solve a problem to a sequence of calls to

an "oracle" Ω, such that both the oracle and the program that generates the sequence of calls can be written in ASP. At this stage of the investigation, we focus on NP-complete oracles, as they can be implemented directly using ASP normal programs. Hence, from now on we identify an oracle with the normal program that implements it.

We denote the signature of an oracle Ω by Σ_Ω, and the set of literals, formed from Σ_Ω according to the usual conventions, by lit_Ω. An *input* to Ω is a consistent subset of lit_Ω.[3]

Given an input I, invoking oracle Ω is reduced to computing the answer sets of $I \cup \Omega$. To simplify dealing with situations in which $I \cup \Omega$ is inconsistent, we work under the convention that, for every input I, every answer set of $I \cup \Omega$ is non-empty.[4] Thus, an *output* of Ω (for input I) is an answer set of $I \cup \Omega$, or $\emptyset$ if $I \cup \Omega$ is inconsistent. Notice that, generally speaking, $I \cup \Omega$ may have multiple answer sets. We denote the *set of outputs* of Ω for I by $\Omega(I)$ (if $I \cup \Omega$ is inconsistent, it follows that $\Omega(I) = \{\emptyset\}$).

The computation that leads to calling the oracle is modeled as a sequence of transitions. Because of the similarity with the domain of reasoning about actions and change (see e.g. [14]), we call *fluents* the properties of interest of the states of the computation, whose truth value typically varies with state transitions. A *fluent literal* is either a fluent f or its negation $\neg f$. A state of the computation is a consistent set of fluent literals.[5]

A *configuration* is a pair $\langle \sigma, \rho \rangle$, where σ is a state of the computation and ρ is a consistent subset of lit_Ω. Intuitively, σ is the current state of the computation and ρ is one output from the latest call to the oracle. The initial configuration is $\langle \sigma^i, \emptyset \rangle$, where σ^i is a pre-determined initial state. Our goal is to use ASP to model a transition function that takes as input a configuration $\langle \sigma, \rho \rangle$ and returns a pair $\langle \sigma', \pi' \rangle$, where σ' is the next state of the computation and π' is the input to the next call to the oracle. The next configuration will then be $\langle \sigma', \rho' \rangle$, for some $\rho' \in \Omega(\pi')$ (there may be multiple next configurations). The computation terminates when it reaches a *terminal configuration*, which we will define using ASP. A terminal configuration may be successful, or failed, meaning that the configuration does not lead to a solution.

The formalization is as follows. Notice that, because here we focus on using a normal program to model the transition function, we restrict our attention to NP-complete transition functions. Given oracle Ω, let Σ_τ be a signature such that:

- all constant and function symbols of Σ_Ω occur also in Σ_τ;
- all the fluent literals of interest can be formed from function and constant symbols of Σ_τ;[6]
- all relation symbols of Σ_Ω are function symbols in Σ_τ;[7]

[3] Although here we use a different approach, one might also view Ω as an lp-function [14].

[4] This can be trivially achieved by ensuring that the oracle contains at least one fact.

[5] Although it is possible to require states of the computation to be also *complete* sets of fluent literals, that does not appear to play a major role in the formulation of our approach.

[6] As often done in the literature, we assume that either terms can contain strong negation, or a suitable function symbol and axioms are introduced to allow writing negative fluent literals.

[7] That is, we reify the relations of Σ_Ω.

– unary relation symbols $state$, $next\_state$, $param$, $result$, and zero-ary relation symbols $terminal$ and $failed$ belong to Σ_τ, and are called *fixed relations*.

Literals of Σ_τ are denoted by lit_τ and are formed according to the usual conventions. Notice that the conditions above, together with the normal assumption that function and relation symbols in a signature are disjoint, imply that the relation symbols of Σ_Ω and of Σ_τ are disjoint. In the rest of the paper, we use the term Ω-literals to refer to both the literals formed from Σ_Ω and to the corresponding terms formed from Σ_τ. The fixed relations of Σ_τ are used to encode the transition function and to provide an interface with the oracle. Intuitively:

– $state(L)$ says that fluent literal L holds in the current state of the computation;
– $next\_state(L)$ says that fluent literal L will hold in the next state of the computation;
– $terminal$ says that the current configuration is terminal;
– $failed$ says that the current configuration does not lead to a solution;
– $param(I)$ says that Ω-literal I is part of the input for the next call to oracle Ω;
– $result(O)$ says that Ω-literal O is part of an output of the latest call to Ω.

Given a relation r and a set of literals A, $A|_r$ (called restriction of A to r) denotes the set of literals of A formed by relation r. If r is a unary relation, by $A \downarrow_r$ we denote the set of arguments of the atoms from $A|_r$. For example, $\{r(a), r(c)\} \downarrow_r = \{a, c\}$. We also denote by $A \uparrow_r$ the set of atoms formed by unary relation r, with the elements of A as arguments. For example, $\{a, c\} \uparrow_r = \{r(a), r(c)\}$.

A configuration $\langle \sigma, \rho \rangle$ is encoded in ASP by means of relations $state$ and $result$. For example, the configuration $\langle \{f_1, f_2\}, \{r_1, r_2\} \rangle$ is encoded by the set of atoms $\{state(f_1), state(f_2), result(r_1), result(r_2)\}$. We denote the ASP encoding of a configuration γ by $\alpha(\gamma)$. More precisely, $\alpha(\langle \sigma, \rho \rangle) = (\sigma \uparrow_{state}) \cup (\rho \uparrow_{result})$.

The transition function is encoded by a program τ, over signature Σ_τ

Definition 3. *Program τ over Σ_τ is a* transition program *if, for every configuration γ:*

– *if some answer set of $\alpha(\gamma) \cup \tau$ entails $terminal$, then $\alpha(\gamma) \cup \tau$ has a unique answer set;*
– *for every answer set A of $\alpha(\gamma) \cup \tau$, $A \models failed$ implies $A \models terminal$.*

For simplicity, in this paper we focus on deterministic transitions, and consequently, for every configuration γ, $\alpha(\gamma) \cup \tau$ has a unique answer set (but recall that the oracle may have multiple answer sets).

Definition 4. *An* ASP machine *is a tuple $\langle \tau, \sigma^i, \Omega \rangle$, where τ is a transition program, Ω is an oracle, and σ^i is the initial state of the computation.*

To simplify the notation, we adopt the convention that the initial state of the computation is the set $\{initial\}$, where $initial$ is a suitable fluent literal from Σ_τ. In that case, an ASP machine is denoted simply by the pair $\langle \tau, \Omega \rangle$.

Definition 5. *An* execution trace *for ASP machine* $\langle \tau, \sigma^i, \Omega \rangle$ *from configuration* $\langle \sigma_0, \rho_0 \rangle$ *is a (possibly infinite) sequence:*

$$\langle \sigma_0, \rho_0, \pi_1, \sigma_1, \rho_1, \pi_2, \sigma_2, \rho_2, \pi_3, \sigma_3, \rho_3, \ldots \rangle,$$

where σ_i*'s are states of the computation, and* π_i*'s,* ρ_i*'s are subsets of* lit_Ω*, such that:*

- $\alpha(\langle \sigma_i, \rho_i \rangle) \cup \tau \models terminal$ *iff* σ_i, ρ_i *are the last two elements of the sequence;*
- *for every* i *such that* $\alpha(\langle \sigma_i, \rho_i \rangle) \cup \tau \not\models terminal$*, there exists an answer set* A *of* $\alpha(\langle \sigma_i, \rho_i \rangle) \cup \tau$ *such that: (i)* $\sigma_{i+1} = A \downarrow_{next\_state}$*, (ii)* $\pi_{i+1} = A \downarrow_{param}$*, and (iii)* $\rho_{i+1} \in \Omega(\pi_{i+1})$.

It follows from the definition that a configuration $\gamma_0 = \langle \sigma_0, \rho_0 \rangle$ is an execution trace from γ_0 for ASP machine $\langle \tau, \sigma^i, \Omega \rangle$ if $\alpha(\langle \sigma_0, \rho_0 \rangle) \cup \tau \models terminal$. By execution trace for an ASP machine $\langle \tau, \sigma^i, \Omega \rangle$, without any reference to a configuration, we mean an execution trace of the ASP machine from its *initial configuration* $\langle \sigma^i, \emptyset \rangle$.

We distinguish between *finite execution traces* and *infinite execution traces*. In a finite execution trace $\langle \sigma_0, \rho_0, \ldots, \pi_n, \sigma_n, \rho_n \rangle$, the pair formed by its last two elements, $\langle \sigma_n, \rho_n \rangle$, is called the *terminal configuration*. We also distinguish between *successful (finite) execution traces* and *failed (finite) execution traces*. A finite execution trace s is *failed* if its terminal configuration $\langle \sigma_n, \tau_n \rangle$ is such that $\alpha(\langle \sigma_i, \rho_i \rangle) \cup \tau \models failed$. Otherwise, s is *successful*.

It is not difficult to check that every finite execution trace has $2 + 3k$ elements for some $k \geq 0$. We call k the *number of transitions* in the execution trace.

Definition 6. *A set* $\sigma \cup \rho$ *is a* hyper answer set *of an ASP machine* $\langle \tau, \sigma^i, \Omega \rangle$ *if* $\langle \sigma, \rho \rangle$ *is a terminal configuration for some successful execution trace for* $\langle \tau, \sigma^i, \Omega \rangle$.

To better illustrate the framework developed so far, let us demonstrate how it can be applied to the task of solving the Traveling Salesman Problem (TSP). Given a directed graph G with edges labeled by weights (representing the cost of traveling from one vertex to the other) and an initial vertex v_0, the goal, in the TSP, is to find a Hamiltonian cycle C from v_0 such that the sum of the weights of the edges traversed by C (also called *cost*) is minimal. The TSP is known to be NP-hard, while the decision problem version of TSP is NP-complete. Although the TSP can be solved quite naturally with some extensions of ASP, such as the `#minimize` statement of LPARSE/SMODELS [10] or weak constraints from DLV [9], we chose this problem because it is well-known and has a simple structure. Furthermore, to demonstrate the programmer's control on the search strategy, we implement the transition program so that it performs binary search.

We use an oracle Ω_H to solve the decision problem version of TSP. That is, Ω_H finds a Hamiltonian cycle C from v_0 *such that the cost of* C *is less than or equal to some given limit* l.

A possible ASP program for Ω_H is:[8]

% Path P visits every vertex V at most once.
$\leftarrow vertex(V2), vertex(V1), vertex(V),$
$in(V1, V), in(V2, V), V1 \neq V2.$
$\leftarrow vertex(V2), vertex(V1), vertex(V),$
$in(V, V1), in(V, V2), V1 \neq V2.$

% Path P must visit every vertex of the graph.
$reached(V2) \leftarrow vertex(V1), vertex(V2), init(V1), in(V1, V2).$
$reached(V2) \leftarrow vertex(V1), vertex(V2), reached(V1), in(V1, V2).$

$\leftarrow vertex(V), \text{not } reached(V).$

$\{in(V1, V2) : edge(V1, V2)\}.$

% Edge cost – cost of a selected edge.
$[r_1]\ ecost(V1, V2, L) \leftarrow$
$vertex(V1), vertex(V2), in(V1, V2), length(V1, V2, L).$

$[r_2]\ cost(T) \leftarrow$
$T[ecost(V1, V2, C) : vertex(V1) : vertex(V2) : ldom(C) = C]T,$
$ldom(T).$

% Path P must have a cost no greater than the current limit.
$\leftarrow ldom(T), ldom(L), cost(T), limit(L), T > L.$

The input to the oracle is provided as follows: the list of vertexes is specified by relation $vertex$; the initial vertex is specified by relation $init$; an edge from v_1 to v_2 is specified by an atom $edge(v_1, v_2)$; the weight of the edge from v_1 to v_2 is specified by an atom $length(v_1, v_2, l)$;[9] the limit to the cost of the Hamiltonian cycle is specified as $limit(l)$; relation $ldom$ specifies the domain for the weights and path cost.[10]

The first 6 rules use well-know techniques to find Hamiltonian cycles. Rule r_1 determines the cost of each selected edge. Rule r_2 uses a special LPARSE/SMODELS [10] construct to compactly specify that $cost(T)$ must hold if T is the cost of the selected Hamiltonian cycle. Finally, the last constraint discards any cycles whose cost is greater than the given limit.

[8] The program shown here is an extension of the one described at `http://www.cs.ttu.edu/~mgelfond/FALL02/asp.pdf`. To increase readability, the program is written using constructs available in LPARSE/SMODELS [10]. Converting to the language described in Section 2 is not difficult.

[9] More compact representations are also possible, which avoid using two separate relations $edge$ and $length$, but we prefer this encoding as it allows us to build incrementally on top of the existing ASP solutions to the problem of finding Hamiltonian cycles in non-weighted graphs.

[10] The use of $ldom$ is required only for proper grounding by LPARSE. We include it here to make the program directly executable with LPARSE+SMODELS [10].

Consider the problem instance P_{TSP} encoded by:

$ldom(0..50).$

$vertex(s0; s1; s2; s3).$
$init(s0).$

$length(s0, s1, 6).\ \ length(s1, s2, 5).\ \ length(s2, s3, 4).$
$length(s3, s0, 3).\ \ length(s0, s2, 2).\ \ length(s1, s3, 1).$

% Edges oriented in the opposite direction, same weights.
$length(s1, s0, 6).\ \ length(s2, s1, 5).\ \ length(s3, s2, 4).$
$length(s0, s3, 3).\ \ length(s2, s0, 2).\ \ length(s3, s1, 1).$

$edge(V1, V2) \leftarrow length(V1, V2, L).$

It is not difficult to check that $\Omega_H \cup P_{TSP} \cup \{limit(25)\}$ has an answer set containing the atoms $\{in(s0, s1), in(s1, s3), in(s3, s2), in(s2, s0), cost(13)\}$, encoding the solution to the decision problem corresponding to path $\langle s0, s1, s3, s2, s0\rangle$, of cost 13.

A transition function that performs binary search is encoded by program, τ_H, consisting of the rules described next. The general idea is to maintain, as part of the state of the computation, an encoding of the current search interval and of its midpoint, used as the limit value for the next call to the oracle. Fluents of interest are, thus, $latest\_min(V)$, $latest\_max(V)$ and $latest\_limit(V)$. The first set of rules from τ_H detects terminal configurations, checks if a Hamiltonian cycle was found by the latest call to the oracle, and detects failed execution traces.

$terminal \leftarrow$
$\quad ldom(T),$
$\quad state(latest\_min(T)), state(latest\_max(T)).$

$hamcycle\_found \leftarrow$
$\quad ldom(T), result(cost(T)).$

$failed \leftarrow$
$\quad terminal,$ not $\ hamcycle\_found.$

The next set of rules uses information about the current state to determine the next search interval and its midpoint, and encode them using auxiliary relations, as follows:[11]

$new\_interval(dom\_min, dom\_max) \leftarrow state(first\_run).$

$new\_interval(X, Y) \leftarrow$
$\quad ldom(X), ldom(Y),$
$\quad hamcycle\_found, state(latest\_limit(Y)), state(latest\_min(X)).$

[11] The search could be made more efficient by taking into account, in the determination of the next search interval, the cost of the Hamiltonian cycle just found by the oracle. For illustrative purposes, however, we use the simpler technique described here.

$$new\_interval(X+1,Y) \leftarrow$$
$$\quad ldom(X), ldom(Y),$$
$$\quad \text{not } hamcycle\_found, state(latest\_limit(X)), state(latest\_max(Y)).$$

$$selected\_limit(T) \leftarrow$$
$$\quad ldom(X), ldom(Y), ldom(T), new\_interval(X,Y), T = ((X+Y)/2).$$

In the rules above, dom_min and dom_max are predefined constants that define the range of interest for the search. Also notice how $hamcycle\_found$ is used, above, to select either the left or right subinterval of the current search interval. The next set of rules determines the next state of the computation from the auxiliary relations just defined:

$$next\_state(latest\_min(X)) \leftarrow ldom(X), ldom(Y), new\_interval(X,Y).$$

$$next\_state(latest\_max(Y)) \leftarrow ldom(X), ldom(Y), new\_interval(X,Y).$$

$$next\_state(latest\_limit(T)) \leftarrow ldom(T), selected\_limit(T).$$

The final set of rules defines the parameters to be passed to the oracle, and in particular the value of the limit for the decision problem:

$$param(limit(V)) \leftarrow ldom(V), selected\_limit(V).$$
$$param(ldom(X)) \leftarrow ldom(X).$$
$$param(vertex(Vert)) \leftarrow vertex(Vert).$$
$$param(init(Vert)) \leftarrow init(Vert).$$
$$param(edge(V1,V2)) \leftarrow edge(V1,V2).$$
$$param(length(V1,V2,L)) \leftarrow length(V1,V2,L).$$

Let us now focus on constructing an execution trace $\langle \sigma_0, \rho_0, \pi_1, \sigma_1, \rho_1, \ldots \rangle$ for the ASP machine $\langle \tau_H \cup P_{TSP}, \Omega_H \rangle$ from the initial configuration $\gamma_0 = \langle \{initial\}, \emptyset \rangle$. To save space, we will construct the relevant portions of the answer sets of the various programs by using the informal meaning of the ASP rules, rather than mathematical proofs. Let dom_min, dom_max be respectively 0 and 50. It is not difficult to show that $\alpha(\gamma_0) \cup \tau_H$ has a unique answer set A_0 containing $new\_interval(0,50)$, $selected\_limit(25)$, $param(limit(25))$, and the corresponding definition of relation $next\_state$. According to our definition of execution trace, $\pi_1 = A_0 \downarrow_{param} = \{limit(25), vertex(s0), \ldots\} = P_{TSP} \cup \{limit(25)\}$ and $\sigma_1 = \{latest\_min(0), latest\_min(50), latest\_limit(25)\}$. Next, let us consider ρ_1. By definition, $\rho_1 \in \Omega_H(\pi_1)$. Notice that $\Omega_H \cup \pi_1$ has possibly multiple answer sets. One such answer set encodes the solution to the decision problem with $cost(13)$ shown earlier. Let us select that solution for the execution trace considered in this example. Hence, $\rho_1 = \{in(s0,s1), in(s1,s3), in(s3,s2), in(s2,s0), \ldots\}$. Let γ_1 denote $\langle \sigma_1, \rho_1 \rangle$. The next step consists in determining σ_2 and π_2 from the answer set, A_1, of $\alpha(\gamma_1) \cup \tau_H$. It can be shown that A_1 contains atoms $new\_interval(0,25)$ and $selected\_limit(12)$. Hence, $\sigma_2 = \{latest\_min(0), latest\_max(25), latest\_limit(12)\}$ and $\pi_2 = P_{TSP} \cup \{limit(12)\}$. Set $\Omega_H(\pi_2)$ contains two answer sets, both encoding solutions with cost 11. Let us pick arbitrarily one such answer set as ρ_2. At this point, σ_3 and π_3 can

be found as above: σ_3 is $\{latest\_min(0), latest\_max(12), latest\_limit(6)\}$, while π_3 is $P_{TSP} \cup \{limit(6)\}$. Because no Hamiltonian cycle exists of cost 6 or less for the given instance, $\Omega_H(\pi_3) = \{\emptyset\}$. Hence, $\rho_3 = \emptyset$. Let us now consider the answer set, A_3, of $\alpha(\langle\sigma_3, \rho_3\rangle)$. Obviously, $hamcycle\_found \notin A_3$. That causes the right search subinterval to be selected, that is $A_3 \supseteq \{new\_interval(7, 12), selected\_limit(9)\}$. Thus, $\pi_4 = P_{TSP} \cup \{limit(9)\}$. As the shortest Hamiltonian cycle for the problem instance considered here has cost 11, once again $\Omega_H(\pi_4) = \{\emptyset\}$. The search proceeds along these lines until subinterval $[11, 11]$ is selected. In other words, it is not difficult to show that there exists some index k such that $\sigma_k = \{latest\_min(11), latest\_max(11), latest\_limit(11)\}$ and $\pi_k = P_{TSP} \cup \{limit(11)\}$. Set $\Omega_H(\pi_k)$ contains two answer sets, both encoding solutions with cost 11. Let us select:

$$\rho_k = \{in(s0, s2), in(s2, s1), in(s1, s3), in(s3, s0), cost(11), \ldots\}.$$

Let us now consider $A_k = \alpha(\langle\sigma_k, \rho_k\rangle)$. Clearly $terminal \in A_k$. Moreover, $hamcycle\_found \in A_k$, which implies that $failed \notin A_k$. Therefore, $\langle\sigma_0, \rho_0, \pi_1, \ldots, \sigma_k, \rho_k\rangle$ is a successful execution trace, and $\sigma_k \cup \rho_k$ is a hyper answer set of the ASP machine. The hyper answer set encodes the solution, of cost 11, corresponding to the path $\langle s0, s2, s1, s3, s0\rangle$.

4 Computing Execution Traces

An algorithm that computes a successful finite execution trace for an ASP machine $\langle\tau, \Omega\rangle$ from configuration $\langle\sigma_0, \rho_0\rangle$ is shown below (Algorithm 1).

Algorithm 1: FindSuccessfulTrace

Input: ASP machine $\langle\tau, \Omega\rangle$
A configuration $\langle\sigma_0, \rho_0\rangle$
Output: A successful finite execution trace $\langle\sigma_0, \rho_0, \pi_1, \sigma_1, \rho_1, \ldots, \sigma_n, \rho_n\rangle$ or $\perp$ if none was found.

```
1  A := the answer set of τ ∪ α(⟨σ0, ρ0⟩)
2  if {terminal, failed} ⊆ A then return ⊥
3  if terminal ∈ A then return ⟨σ0, ρ0⟩
4  π := (A ↓param)
5  σ := (A ↓next_state)
6  O := Ω(π)
7  while O ≠ ∅ do
8      ρ := an arbitrary element of O
9      O := O \ {ρ}
10     s := FindSuccessfulTrace(⟨τ, Ω⟩, ⟨σ, ρ⟩)
11     if s ≠ ⊥ then return ⟨σ0, ρ0, π⟩ ∘ s
12 end
13 return ⊥
```

Given a configuration $\gamma_0 = \langle \sigma_0, \rho_0 \rangle$, algorithm FindSuccessfulTrace starts by checking whether γ_0 is failed. That is accomplished, as per the definition of execution trace, by checking whether the answer set A of $\tau \cup \alpha(\langle \sigma_0, \rho_0 \rangle)$ contains $\{terminal, failed\}$. If γ_0 is failed, the algorithm returns $\perp$. Next, FindSuccessfulTrace checks if γ_0 is terminal, and if so returns it. Otherwise, the algorithm extracts from A the next state of the computation, σ, and the parameters for the call to the oracle, π. The oracle is then invoked, and its outputs are stored in O. Notice that, by definition, O is guaranteed to be non-empty. Next, the algorithm attempts to construct a successful trace using each output of the oracle. To do that, an arbitrary element ρ of O is selected and removed from O. The algorithm is called recursively, to attempt to find a successful execution trace s from the new configuration $\langle \sigma, \rho \rangle$. If the attempt succeeds, the algorithm prepends to s the initial configuration as well as the parameters used in the call to the oracle and returns the resulting execution trace. Otherwise, the algorithm selects another output from the call to the oracle, and iterates. If all the attempts fail at constructing a successful execution trace from the configuration obtained from σ and an output of the oracle, the algorithm returns $\perp$. Let us now discuss soundness and completeness of the algorithm.

Lemma 1. *For every ASP machine $\langle \tau, \Omega \rangle$ and every configuration $\langle \sigma_0, \rho_0 \rangle$, if $FindSuccessfulTrace(\langle \tau, \Omega \rangle, \langle \sigma_0, \rho_0 \rangle)$ returns a sequence $s \neq \perp$, then the first two elements of s are σ_0 and ρ_0; that is,*

$$s = \langle \sigma_0, \rho_0, \ldots, \rangle.$$

It is not difficult to show that every sequence (that is, every result except for $\perp$) returned by $FindSuccessfulTrace$ has $2 + 3k$ elements, for some $k \geq 0$. Therefore, let us extend the notion of number of transitions to the sequences returned by algorithm $FindSuccessfulTrace$. We denote the number of transitions in such a sequence s by $|s|$.

Theorem 1. *For every ASP machine $\langle \tau, \Omega \rangle$ and every configuration $\langle \sigma_0, \rho_0 \rangle$, if $FindSuccessfulTrace(\langle \tau, \Omega \rangle, \langle \sigma_0, \rho_0 \rangle)$ returns a sequence $s \neq \perp$, then s is a successful execution sequence for $\langle \tau, \Omega \rangle$ from $\langle \sigma_0, \rho_0 \rangle$.*

Proof. *We proceed by induction on the number of transitions in the sequence returned by algorithm $FindSuccessfulTrace$.*

Base case: $|s| = 0$.
If $|s| = 0$, then by definition s contains two elements, that is, by Lemma 1, $s = \langle \sigma_0, \rho_0 \rangle$. Hence, s was returned at step 3 of the algorithm, which implies that $\tau \cup \alpha(\langle \sigma_0, \rho_0 \rangle)$ entails $terminal$, but it does not entail failed. By Definition 5, s is a successful execution trace.

Inductive step.
Let us assume that, if $FindSuccessfulTrace(\langle \tau, \Omega \rangle, \langle \sigma, \rho \rangle)$ returns a sequence s with $n - 1$ transitions, then s is a successful execution trace, and let us prove that, if $FindSuccessfulTrace(\langle \tau, \Omega \rangle, \langle \sigma, \rho \rangle)$ returns a sequence s' with n transitions, then s' is a successful execution trace.

Because $|s'| \geq 1$, s' contains at least $2 + 3 \cdot 1 = 5$ elements. Hence, the final result of algorithm $FindSuccessfulTrace$ must have been returned by step 11. Let σ_1, ρ_1,

π_1 denote the values of variables σ, ρ, and π at the moment of the execution of step 11, and notice that those variables had the same values at step 10. Consequently, there exists s such $s = FindSuccessfulTrace(\langle\tau, \Omega\rangle, \langle\sigma_1, \rho_1\rangle)$ and $s' = \langle\sigma_0, \rho_0, \pi_1\rangle \circ s$. By Lemma 1, the first two elements of s are σ_1 and ρ_1; that is, $s = \langle\sigma_1, \rho_1, \ldots\rangle$.

Because $|s'| = n$, $|s| = n-1$. By inductive hypothesis, s is a successful execution trace from $\langle\sigma_1, \rho_1\rangle$.

To prove the thesis, we need to show that the conditions of Definition 5 are satisfied. Because s has already been shown to be a successful execution trace, the conditions simplify to:

1. *$\alpha(\langle\sigma_0, \rho_0\rangle) \cup \tau$ entails neither $terminal$ nor $failed$;*
2. *the answer set, A, of $\alpha(\langle\sigma_0, \rho_0\rangle) \cup \tau$ is such that:*
   (a) $\sigma_1 = A \downarrow_{next\_state}$;
   (b) $\pi_1 = A \downarrow_{param}$;
   (c) $\rho_1 \in \Omega(\pi_1)$.

The fact that $\alpha(\langle\sigma_0, \rho_0\rangle) \cup \tau$ entails neither $terminal$ nor $failed$ follows from the observation that, if that were not the case, steps 2 and 3 of the algorithm would have returned either $\perp$ or a sequence with 0 transitions.[12]

The fact that the other conditions hold is demonstrated as follows. Notice that the value of variable π does not change between step 4 and step 11. We have already established that the value of π at step 11 is π_1. Hence, step 4 guarantees that $\pi_1 = A \downarrow_{param}$. With a similar reasoning, we conclude that $\sigma_1 = A \downarrow_{next\_state}$ and $\rho_1 \in \Omega(\pi_1)$.

Therefore, s' is an execution trace by Definition 5. The fact that it is finite follows from the fact that s is finite. Finally, s' is successful because, if step 2 detects a failed execution sequence, the algorithm returns $\perp$ instead of a sequence.

Obviously, the algorithm is not complete, as it finds only one successful execution trace. What is more important, however, is that not even termination is guaranteed: in fact, if an infinite execution trace exists, nothing prevents the algorithm from attempting to construct it. To make guarantees about termination, we need to consider a more restricted class of ASP machines.

Definition 7. *An ASP machine $\mu = \langle\tau, \Omega\rangle$ is* finite *if every execution trace of μ is finite.*

Finite ASP machines allow one to guarantee not only termination, but also a weak notion of completeness.

Theorem 2. *For every finite ASP machine $\mu = \langle\tau, \Omega\rangle$:*

1. *$FindSuccessfulTrace(\mu, \langle\sigma^i, \emptyset\rangle)$ terminates;*
2. *If a successful execution trace exists for μ, $FindSuccessfulTrace(\mu, \langle\sigma^i, \emptyset\rangle)$ returns a successful execution trace.*

Proof. *Both statements can be proven by induction similarly to Theorem 1.*

[12] Regarding step 2, the fact that τ is a transition program guarantees that, if $failed$ is entailed by $\alpha(\langle\sigma_0, \rho_0\rangle) \cup \tau$, then $terminal$ is also entailed.

It is possible to extend $FindSuccessfulTrace$ into an algorithm that returns a set of successful finite execution traces, as shown below (Algorithm 2). Not only the extended algorithm is sound, but, for finite ASP machines, it can also be shown to terminate and be complete.

Algorithm 2: FindAllSuccessfulTraces

Input: ASP machine $\langle \tau, \Omega \rangle$
A configuration $\langle \sigma_0, \rho_0 \rangle$
Output: A (possibly empty) set of successful finite execution traces.

```
1  A := the answer set of τ ∪ α(⟨σ0, ρ0⟩)
2  if {terminal, failed} ⊆ A then return ∅
3  if terminal ∈ A then return {⟨σ0, ρ0⟩}
4  π := (A ↓param)
5  σ := (A ↓next_state)
6  H := ∅
7  O := Ω(π)
8  while O ≠ ∅ do
9      ρ := an arbitrary element of O
10     O := O \ {ρ}
11     S := FindAllSuccessfulTraces(⟨τ, Ω⟩, ⟨σ, ρ⟩)
12     if S ≠ ∅ then H := H ∪ {⟨σ0, ρ0, π⟩ ∘ s | s ∈ S}
13 end
14 return H
```

5 Related Work

In [11] and [12], it is argued that describing an algorithm using a transition system instead of pseudocode makes it easier to prove its properties, compare it with other algorithms, and design new algorithms.

The fact that ASP provides a convenient framework to represent state transitions was highlighted by the research on using ASP for reasoning about actions and change, see e.g. [15,14,4].

Several extensions of the language of ASP allow one to deal with problems of complexity higher than NP-complete, e.g. [1,9,8,16].

The GnT system [17] finds answer sets of disjunctive logic programs by computing the answer sets of two normal programs automatically derived from the original program. Our approach is more general than [17] in that the oracle and the transition program are independently programmed, rather than obtained from automatic translation, and thus our technique can be applied to solve a variety of problems besides just the computation of the answer sets of disjunctive programs.

In [7], an ASP program is used to simulate a non-deterministic Turing machine. Hence, the encoding presented there is at a considerably lower level of abstraction than the one we discussed. Furthermore, [7] is focused upon a single ASP program, while in the present work a considerable effort was devoted to developing a suitable framework allowing the interaction between two ASP programs.

6 Conclusions and Future Work

In this paper we have explored the use of answer set programming to solve problems that cannot be solved with a normal program. Although extensions of ASP exist that allow solving some of these problems, for practical applications often programmers are forced to resort to writing auxiliary (often procedural) programs, which reduce the task of solving the problem to that of computing the answer sets of a suitable sequence of ASP programs. The rather different level of abstraction used in the auxiliary programs, compared to the ASP programs at the core of the application, causes various difficulties, and makes it hard to prove properties of the overall program. Our approach is based on the consideration that algorithms can be represented by transition systems, and that ASP has been proven to be a useful tool for representing state transitions. By using ASP to encode the auxiliary algorithms needed to solve problems of complexity beyond NP-complete, we remove the problems introduced by the use of different levels of abstraction, and simplify proving the properties of the overall program.

This paper has been focused upon solving problems by calling an NP-complete oracle. However, our definitions extend to more powerful oracles, ultimately allowing to use an ASP machine as an oracle to another ASP machine. Another interesting extension consists in allowing the use of multiple oracles, with the transition program determining which oracles to execute at each transition.

Finally, although here we have mostly focused on finite execution sequences, we believe that infinite execution sequences deserve attention. In fact, infinite execution sequences appear to be useful in the specification of closed-loop algorithms, such as agent control loops. In this area, ASP machines might provide an interesting middle-ground between the fully-logical specifications (e.g. [18]) and the mixed procedural/logical specifications (e.g. [19]) of control loops.

References

1. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing (1991) 365–385
2. Marek, V.W., Truszczynski, M. The Logic Programming Paradigm: a 25-Year Perspective. In: Stable models and an alternative logic programming paradigm. Springer Verlag, Berlin (1999) 375–398
3. Soininen, T., Niemela, I.: Developing a declarative rule language for applications in product configuration. In: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages. (May 1999)
4. Balduccini, M., Gelfond, M., Nogueira, M.: Answer Set Based Design of Knowledge Systems. Annals of Mathematics and Artificial Intelligence (2006)

5. Baral, C., Chancellor, K., Tran, N., Joy, A., Berens, M.: A Knowledge Based Approach for Representing and Reasoning About Cell Signalling Networks. In: Proceedings of the European Conference on Computational Biology, Supplement on Bioinformatics. (2004) 15–22
6. Son, T.C., Sakama, C.: Negotiation Using Logic Programming with Consistency Restoring Rules. In: IJCAI'09. (2009)
7. Marek, V.W., Remmel, J.B.: On the expressibility of stable logic programming. Journal of Theory and Practice of Logic Programming (TPLP) **3**(4–5) (2003) 551–567
8. Balduccini, M., Gelfond, M.: Logic Programs with Consistency-Restoring Rules. In Doherty, P., McCarthy, J., Williams, M.A., eds.: International Symposium on Logical Formalization of Commonsense Reasoning. AAAI 2003 Spring Symposium Series (Mar 2003) 9–18
9. Buccafurri, F., Leone, N., Rullo, P.: Strong and Weak Constraints in Disjunctive Datalog. In: Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97). Volume 1265 of Lecture Notes in Artificial Intelligence (LNCS). (1997) 2–17
10. Niemela, I., Simons, P., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1–2) (Jun 2002) 181–234
11. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Module Theories: From an Abstract Davis-Putnam-Longemann-Loveland Procedure to DPLL(T). Journal of Artificial Intelligence Research **53**(6) (2006) 937–977
12. Lierler, Y.: Abstract Answer Set Solvers. In: Proceedings of the 24th International Conference on Logic Programming (ICLP08). (Dec 2008) 377–391
13. Turing, A.M.: Systems of logic based on ordinals. Proceedings of the London Mathematical Society. Second Series **45** (1939) 161–228
14. Gelfond, M.: Representing Knowledge in A-Prolog. In Kakas, A.C., Sadri, F., eds.: Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II. Volume 2408., Springer Verlag, Berlin (2002) 413–451
15. Lifschitz, V., Turner, H.: Representing transition systems by logic programs. In: Proceedings of the 5th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR-99). Number 1730 in Lecture Notes in Artificial Intelligence (LNCS), Springer Verlag, Berlin (1999) 92–106
16. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 03), Morgan Kaufmann (Aug 2003)
17. Janhunen, T., Niemela, I., Simons, P., You, J.H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. In: Principles of Knowledge Representation and Reasoning: Proceedings of the 7th International Conference (KR00), Morgan Kaufmann (2000) 411–419
18. Levesque, H.J., Reiter, R., Lin, F., Scherl, R.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming **31** (1997)
19. Balduccini, M., Gelfond, M.: The AAA Architecture: An Overview. In: AAAI Spring Symposium 2008 on Architectures for Intelligent Theory-Based Agents (AITA08). (Mar 2008)