Conflict-driven ASP Solving with External Sources

Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl∗
Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
(*e-mail:* {eiter,fink,tkren,redl}@kr.tuwien.ac.at)

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

Abstract

Answer Set Programming (ASP) is a well-known problem solving approach based on nonmonotonic logic programs and efficient solvers. To enable access to external information, HEX-programs extend programs with *external atoms*, which allow for a bidirectional communication between the logic program and external sources of computation (e.g., description logic reasoners and Web resources). Current solvers evaluate HEX-programs by a translation to ASP itself, in which values of external atoms are guessed and verified after the ordinary answer set computation. This elegant approach does not scale with the number of external accesses in general, in particular in presence of nondeterminism (which is instrumental for ASP). In this paper, we present a novel, native algorithm for evaluating HEX-programs which uses learning techniques. In particular, we extend conflict-driven ASP solving techniques, which prevent the solver from running into the same conflict again, from ordinary to HEX-programs. We show how to gain additional knowledge from external source evaluations and how to use it in a conflict-driven algorithm. We first target the uninformed case, i.e., when we have no extra information on external sources, and then extend our approach to the case where additional meta-information is available. Experiments show that learning from external sources can significantly decrease both the runtime and the number of considered candidate compatible sets.

KEYWORDS: Answer Set Programming, Nonmonotonic Reasoning, Conflict-Driven Clause Learning

1 Introduction

Answer Set Programming (ASP) is a declarative programming approach (Niemelä 1999; Marek and Truszczyński 1999; Lifschitz 2002), in which solutions to a problem correspond to answer sets (Gelfond and Lifschitz 1991) of a logic program, which are computed using an ASP solver. While this approach has turned out, thanks to expressive and efficient systems like SMODELS (Simons et al. 2002), DLV (Leone et al. 2006), ASSAT (Lin and Zhao 2004), cmodels (Giunchiglia et al. 2006), and CLASP (Gebser et al. 2012; Gebser et al. 2011), to be fruitful for a range of applications, cf. (Brewka et al. 2011), current trends in distributed systems and the World Wide Web, for instance, revealed the need for access to external sources in a program, ranging from light-weight data access (e.g., XML, RDF, or data bases) to knowledge-intensive formalisms (e.g., description logics).

To cater for this need, HEX-programs (Eiter et al. 2005) extend ASP with so called external atoms, through which the user can couple any external data source with a logic

∗ This research has been supported by the Austrian Science Fund (FWF) project P20840, P20841, and P24090, and by the Vienna Science and Technology Fund (WWTF) project ICT 08-020.

program. Roughly, such atoms pass information from the program, given by predicates and constants, to an external source which returns output values of an (abstract) function that it computes. This extension is convenient and has been exploited for applications in different areas, cf. (Eiter et al. 2011), and it is also very expressive since recursive data exchange between the logic program and external sources is possible. Advanced reasoning applications like default reasoning over description logic ontologies (Eiter et al. 2008; Dao-Tran et al. 2009) or reasoning over Nonmonotonic Multi-Context Systems (Brewka and Eiter 2007; Eiter et al. 2010) take advantage of it.

Current algorithms for evaluating HEX-programs use a translation approach and rewrite them to ordinary ASP programs. The idea is to guess the truth values of external atoms (i.e., whether a particular fact is in the "output" of the external source access) in a modified program; after computing answer sets, a compatibility test checks whether the guesses coincide with the actual source behavior. While elegant, this approach is a bottleneck in advanced applications including those mentioned above. It does not scale, as blind guessing leads to an explosion of candidate answer sets, many of which might fail the compatibility test. Furthermore, a blackbox view of external sources disables any pruning of the search space in the ASP translation, and even if properties would be known, it is sheer impossible to make use of them in ordinary ASP evaluation on-the-fly using standard solvers.

To overcome this bottleneck, a new evaluation method is needed. In this paper, we thus present a novel algorithm for evaluating HEX-programs, described in Section 3, which avoids the simple ASP translation approach. It has three key features.

• First, it natively builds model candidates from first principles and accesses external sources already during the model search, which allows to prune candidates early.

• Second, it considers external sources no longer as black boxes, but exploits meta-knowledge about their internals.

• And third, it takes up modern SAT and ASP solving techniques based on *clause learning* (Biere et al. 2009), which led to very efficient *conflict-driven* algorithms for answer-set computation (Gebser et al. 2012; Drescher et al. 2008), and extends them to external sources, which is a major contribution of this work. To this end, we introduce *external behavior learning (EBL)*, which generates conflict clauses (nogoods) after external source evaluation (Section 3). We do this in Section 4, first in the uninformed case (Section 4.1), where no meta-information about the external source is available, except that a certain input generates a certain output. We then exploit meta-information[1] about external sources (properties such as monotonicity and functionality) to learn even more effective nogoods which restrict the search space further (Section 4.2).

We have implemented the new algorithm and incorporated it into the DLVHEX prototype system.[2] It is designed in an extensible fashion, such that the provider of external sources can specify refined learning functions which exploit specific knowledge about the source. Our theoretical work is confirmed by experiments that we conducted with our prototype on synthetic benchmarks and programs motivated by real-world applications (Section 5). In several cases, significant performance improvements compared to the previous algorithm are obtained, which shows the suitability and potential of the new approach.

[1] Not to be confused with semantically annotated data, which is not considered here.
[2] http://www.kr.tuwien.ac.at/research/systems/dlvhex/

2 Preliminaries

In this section, we introduce syntax and semantics of HEX-programs and, following (Drescher et al. 2008), conflict-driven SAT and answer set solving. We start with basic definitions.

A (signed) literal is a positive or a negated ground atom $\mathbf{T}a$ or $\mathbf{F}a$, where ground atom a is of form $p(c_1, \dots, c_\ell)$, with predicate p and function-symbol free ground terms $c_1, \dots, c_\ell$, abbreviated as $p(\mathbf{c})$. For a literal $\sigma = \mathbf{T}a$ or $\sigma = \mathbf{F}a$, let $\overline{\sigma}$ denote its negation, i.e. $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$. An assignment $\mathbf{A}$ over a (finite) set of atoms $\mathcal{A}$ is a consistent set of signed literals $\mathbf{T}a$ or $\mathbf{F}a$, where $\mathbf{T}a$ expresses that $a \in \mathcal{A}$ is true and $\mathbf{F}a$ that it is false.

We write $\mathbf{A}^{\mathbf{T}}$ to refer to the set of elements $\mathbf{A}^{\mathbf{T}} = \{a \mid \mathbf{T}a \in \mathbf{A}\}$ and $\mathbf{A}^{\mathbf{F}}$ to refer to $\mathbf{A}^{\mathbf{F}} = \{a \mid \mathbf{F}a \in \mathbf{A}\}$. The extension of a predicate symbol q wrt. an assignment $\mathbf{A}$ is defined as $ext(q, \mathbf{A}) = \{\mathbf{c} \mid \mathbf{T}q(\mathbf{c}) \in \mathbf{A}\}$. Let further $\mathbf{A}|_q$ be the set of all signed literals over atoms of form $q(\mathbf{c})$ in $\mathbf{A}$. For a list $\mathbf{q} = q_1, \dots, q_k$ of predicates, we let $\mathbf{A}|_{\mathbf{q}} = \mathbf{A}|_{q_1} \cup \cdots \cup \mathbf{A}|_{q_k}$.

A *nogood* $\{L_1, \dots, L_n\}$ is a set of (signed) literals $L_i, 1 \leq i \leq n$. An assignment $\mathbf{A}$ is a *solution* to a nogood δ resp. a set of nogoods Δ, iff $\delta \not\subseteq \mathbf{A}$ resp. $\delta \not\subseteq \mathbf{A}$ for all $\delta \in \Delta$.

2.1 HEX*-Programs*

We briefly recall HEX-programs, which have been introduced in Eiter et al. (2005) as a generalization of (disjunctive) extended logic programs under the answer set semantics (Gelfond and Lifschitz 1991); for more details and background, we refer to Eiter et al. (2005).

Syntax. HEX-programs extend ordinary ASP programs by *external atoms*, which enable a bidirectional interaction between a program and external sources of computation. External atoms have a list of input parameters (constants or predicate names) and a list of output parameters. Informally, to evaluate an external atom, the reasoner passes the constants and extensions of the predicates in the input tuple to the external source associated with the external atom, which is plugged into the reasoner. The external source computes an output tuple, which is matched with the output list. More formally, a *ground external atom* is of the form $\&g[\mathbf{p}](\mathbf{c})$, where $\mathbf{p} = p_1, \dots, p_k$ are constant input parameters (predicate names or object constants), and $\mathbf{c} = c_1, \dots, c_l$ are constant output terms.

Ground HEX-programs are then defined similar to ground ordinary ASP programs.

Definition 1 (Ground HEX-programs) *A ground* HEX*-program consists of rules of form*

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \dots, b_m, \text{not } b_{m+1}, \dots, \text{not } b_n \ ,$$

where each a_i for $1 \leq i \leq k$ is a ground atom $p(c_1, \dots, c_\ell)$ with constants c_j, $1 \leq j \leq \ell$, and each b_i for $1 \leq i \leq n$ is either a classical ground atom or a ground external atom.[3]

The *head* of a rule r is $H(r) = \{a_1, \dots, a_k\}$ and the *body* is $B(r) = \{b_1, \dots, b_m, \text{not } b_{m+1}, \dots, \text{not } b_n\}$. We call b or $\text{not}\, b$ in a rule body a *default literal*; $B^+(r) = \{b_1, \dots, b_m\}$ is the *positive body*, $B^-(r) = \{b_{m+1}, \dots, b_n\}$ is the *negative body*.

In Sections 4 and 5 we will also make use of non-ground programs. However, we restrict our theoretical investigation to ground programs as suitable safety conditions allow for application of grounding procedure (Eiter et al. 2006).

[3] For simplicity, we do not formally introduce strong negation but see classical literals of form $\neg a$ as new atoms together with a constraint which disallows that a and $\neg a$ are simultaneously true.

Semantics and Evaluation. The semantics of a ground external atom $\&g[\mathbf{p}](\mathbf{c})$ wrt. an assignment $\mathbf{A}$ is given by the value of a $1{+}k{+}l$-ary Boolean *oracle function* $f_{\&g}$ that is defined for all possible values of $\mathbf{A}$, $\mathbf{p}$ and $\mathbf{c}$. Thus, $\&g[\mathbf{p}](\mathbf{c})$ is true relative to $\mathbf{A}$ if and only if it holds that $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = 1$. Satisfaction of ordinary rules and ASP programs (Gelfond and Lifschitz 1991) is then extended to HEX-rules and programs in the obvious way, and the notion of extension $ext(\cdot, \mathbf{A})$ for external predicates $\&g$ with input lists $\mathbf{p}$ is naturally defined by $ext(\&g[\mathbf{p}], \mathbf{A}) = \{\mathbf{c} \mid f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = 1\}$.

The answer sets of a HEX-program Π are determined by the DLVHEX solver using a transformation to ordinary ASP programs as follows. Each external atom $\&g[\mathbf{p}](\mathbf{c})$ in Π is replaced by an ordinary ground *replacement atom* $e_{\&g[\mathbf{p}]}(\mathbf{c})$ and a rule $e_{\&g[\mathbf{p}]}(\mathbf{c}) \vee ne_{\&g[\mathbf{p}]}(\mathbf{c}) \leftarrow$ is added to the program. The answer sets of the resulting *guessing program* $\hat{\Pi}$ are determined by an ordinary ASP solver and projected to non-replacement atoms. However, the resulting assignments are not necessarily models of Π, as the value of $\&g[\mathbf{p}]$ under $f_{\&g}$ can be different from the one of $e_{\&g[\mathbf{p}]}(\mathbf{c})$. Each answer set of $\hat{\Pi}$ is thus a *candidate compatible set* (or *model candidate*) which must be checked against the external sources. If no discrepancy is found, the model candidate is a *compatible set* of Π. More precisely,

Definition 2 (Compatible Set) *A* compatible set *of a program* Π *is an assignment* $\mathbf{A}$

(i) *which is an answer set (Gelfond and Lifschitz 1991) of the* guessing program $\hat{\Pi}$*, and*

(ii) $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = 1$ *iff* $\mathbf{T}e_{\&g[\mathbf{p}]}(\mathbf{c}) \in \mathbf{A}$ *for all external atoms* $\&g[\mathbf{p}](\mathbf{c})$ *in* Π*, i.e. the guessed values coincide with the actual output under the input from* $\mathbf{A}$.

The compatible sets of Π computed by DLVHEX include (modulo $A(\Pi)$) all answer sets of Π as defined in Eiter et al. (2005) using the FLP reduct (Faber et al. 2011), which we refer to as FLP-answer sets; with an additional test on candidate answer sets A (which is easily formulated as compatible set existence for a variant of Π), the FLP-answer sets can be obtained. By default, DLVHEX computes compatible sets with smallest true part on the original atoms; this leads to answer sets as follows.

Definition 3 (Answer Set) *An (*DLVHEX*) answer set of* Π *is any set* $S \subseteq \{\mathbf{T}a \mid a \in A(\Pi)\}$ *such that (i)* $S = \{\mathbf{T}a \mid a \in A(\Pi)\} \cap \mathbf{A}$ *for some compatible set* $\mathbf{A}$ *of* Π *and (ii)* $\{\mathbf{T}a \mid a \in A(\Pi)\} \cap \mathbf{A} \not\subset S$ *for every compatible set* $\mathbf{A}$ *of* Π.

The answer sets in Definition 3 include all FLP-answer sets, and in fact often coincide with them (as in all examples we consider). Computing the (minimal) compatible sets is thus a key problem for HEX-programs on which we focus here.

2.2 Conflict-driven Clause Learning and Nonchronological Backtracking

Recall that DPLL-style SAT solvers rely on an alternation of drawing deterministic consequences and guessing the truth value of an atom towards a complete interpretation. Deterministic consequences are drawn by the basic operation of *unit propagation*, i.e., whenever all but one signed literals of a nogood are satisfied, the last one must be false. The solver stores an integer *decision level* dl, written $@dl$ as postfix to the signed literal. An atom which is set by unit propagation gets the highest decision level of all already assigned atoms, whereas guessing increments the current decision level.

Most modern SAT solver are *conflict-driven*, i.e., they learn additional nogoods when current assignment violates a nogood. This prevents the solver from running into the same conflict again. The learned nogood is determined by initially setting the conflict nogood to the violated one. As long as it contains multiple literals from the same decision level, it is resolved with the *reason* of one of these literals, i.e., the nogood which implied it.

Example 1 Consider the nogoods

$$\{\mathbf{T}a, \mathbf{T}b\}, \{\mathbf{T}a, \mathbf{T}c\}, \{\mathbf{F}a, \mathbf{T}x, \mathbf{T}y\}, \{\mathbf{F}a, \mathbf{T}x, \mathbf{F}y\}, \{\mathbf{F}a, \mathbf{F}x, \mathbf{T}y\}, \{\mathbf{F}a, \mathbf{F}x, \mathbf{F}y\}$$

and suppose the assignment is $\mathbf{A} = \{\mathbf{F}a@1, \mathbf{T}b@2, \mathbf{T}c@3, \mathbf{T}x@4\}$. Then the third nogood is unit and implies $\mathbf{F}y@4$, which violates the fourth nogood $\{\mathbf{F}a, \mathbf{T}x, \mathbf{F}y\}$. As it contains multiple literals (x and y) which were set at decision level 4, it is resolved with the reason for setting y to false, which is the nogood $\{\mathbf{F}a, \mathbf{T}x, \mathbf{T}y\}$. This results in the nogood $\{\mathbf{F}a, \mathbf{T}x\}$, which contains the single literal x set at decision level 4, and thus is the learned nogood.

In standard clause notation, the nogood set corresponds to

$$(\neg a \vee \neg b) \wedge (\neg a \vee \neg c) \wedge (a \vee \neg x \vee \neg y) \wedge (a \vee \neg x \vee y) \wedge (a \vee x \vee \neg y) \wedge (a \vee x \vee y)$$

and the violated clause is $(a \vee \neg x \vee y)$. It is resolved with $(a \vee \neg x \vee \neg y)$ and results in the learned clause $(a \vee \neg x)$. □

State-of-the-art SAT and ASP solvers backtrack then to the second-highest decision level in the learned nogood. In Example 1, this is decision level 1. All assignments after decision level 1 are undone ($\mathbf{T}b@2$, $\mathbf{T}c@3$, $\mathbf{T}x@4$). Only variable $\mathbf{F}a@1$ remains assigned. This makes the new nogood $\{\mathbf{F}a, \mathbf{T}x\}$ unit and derives $\mathbf{F}x$ at decision level 1.

2.3 Conflict-driven ASP Solving

In this subsection we summarize conflict-driven (disjunctive) answer-set solving (Gebser et al. 2012; Drescher et al. 2008). It corresponds to Algorithm HEX-CDNL without Part (c), (cf. Section 3, where we also discuss Part (c)). Subsequently, we provide a summary of the base algorithm; for details we refer to Gebser et al. (2012) and Drescher et al. (2008).

To employ conflict-driven techniques from SAT solving in ASP, programs are represented as sets of nogoods. For a program Π, let $A(\Pi)$ be the set of all atoms occurring in Π, and let $BA(\Pi) = \{B(r) \mid r \in \Pi\}$ be the set of all rule bodies of Π, viewed as fresh atoms.

We first define the set $\gamma(C) = \{\{\mathbf{F}C\} \cup \{\mathbf{t}\ell \mid \ell \in C\}\} \cup \{\{\mathbf{T}C, \mathbf{f}\ell\} \mid \ell \in C\}$ of nogoods to encode that a set C of default literals must be assigned $\mathbf{T}$ or $\mathbf{F}$ in terms of the conjunction of its elements, where $\mathbf{t}\,\mathrm{not}\,a = \mathbf{F}a$, $\mathbf{t}a = \mathbf{T}a$, $\mathbf{f}\,\mathrm{not}\,a = \mathbf{T}a$, and $\mathbf{f}a = \mathbf{F}a$. That is, the conjunction is true iff each literal is true. Clark's completion Δ_Π of a program Π over atoms $A(\Pi) \cup BA(\Pi)$ is the set of nogoods

$$\Delta_\Pi = \bigcup\nolimits_{r \in \Pi} (\gamma(B(r)) \cup \{\{\mathbf{T}B(r)\} \cup \{\mathbf{F}a \mid a \in H(r)\}\}) \ .$$

The body of a rule is true iff each literal is true, and if the body is true, a head literal must also be true. Unless a program is tight (Fages 1994), Clark's completion does not fully capture the semantics of a program; unfounded sets may occur, i.e., sets of atoms which only cyclically support each other, called a *loop*. Avoidance of unfounded sets requires additional *loop nogoods*, but as there are exponentially many, they are only introduced on-the-fly.

Disjunctive programs require additional concepts. Neglecting details, it is common to use additional nogoods $\Theta_{sh(\Pi)}$ derived from the *shifted program* $sh(\Pi)$, which encode the loop formulas of singleton loops; a comprehensive study is available in Drescher et al. (2008).

With these concepts we are ready to describe the basic algorithm for answer set computation shown in HEX-CDNL. The algorithm keeps a set $\Delta_{\Pi} \cup \Theta_{sh(\Pi)}$ of "static" nogoods (from Clark's completion and from singular loops), and a set ∇ of "dynamic" nogoods which are learned from conflicts and unfounded sets during execution. While constructing the assignment $\mathbf{A}$, the algorithm stores for each atom $a \in A(\Pi)$ a *decision level* dl. The decision level is initially 0 and incremented for each choice. Deterministic consequences of a set of assigned values have the same decision level as the highest decision level in this set.

The main loop iteratively derives deterministic consequences using Propagation trying to complete the assignment. This includes both unit propagation and unfounded set propagation. Unit propagation derives $\overline{d}$ if $\delta \setminus \{d\} \subseteq \mathbf{A}$ for some nogood δ, i.e. all but one literal of a nogood are satisfied, therefore the last one needs to be falsified. Unfounded set propagation detects atoms which only cyclically support each other and falsifies them.

Part (a) checks if there is a conflict, i.e. a violated nogood $\delta \subseteq \mathbf{A}$. If this is the case we need to backtrack. For this purpose we use Analysis to compute a learned nogood ϵ and a backtrack decision level k. The learned nogood is added to the set of dynamic nogoods, and assignments above decision level k are undone. Otherwise, Part (b) checks if the assignment is complete. In this case, a final unfounded set check is necessary due to disjunctive heads. If the candidate is founded, it is an answer set. Otherwise we select a violated loop nogood δ from the set $\lambda_{\hat{\Pi}}(U)$ of all loop nogoods for an unfounded set U (for the definition see Drescher et al. 2008), we do conflict analysis and backtrack. If no more deterministic consequences can be derived and the assignment is still incomplete, we need to guess in Part (d) and increment the decision level. The function Select implements a variable selection heuristic. In the simplest case it chooses an arbitrary yet unassigned variable, but state-of-the-art heuristics are more sophisticated. E.g., Goldberg and Novikov (2007) prefer variables which are involved in recent conflicts.

3 Algorithms for Conflict-driven HEX-Program Solving

We present now our new, genuine algorithms for HEX-program evaluation. They are based on Drescher et al. (2008), but integrate additional novel learning techniques to capture the semantics of external atoms. The term *learning* refers to the process of adding further nogoods to the nogood set as the search space is explored. They are classically derived from conflict situations to avoid similar conflicts during further search, as described above.

We add a second type of learning which captures the behavior of external sources, called *external behavior learning* (EBL). Whenever an external atom is evaluated, the algorithm might learn from the call. If we have no further information about the internals of a source, we may learn only very general input-output-relationships, if we have more information we can learn more effective nogoods. In general, we can associate a *learning-function* with each external source. For the sake of introducing the evaluation algorithms, however, in this section we abstractly consider a set of nogoods learned from the evaluation of some external predicate with input list $\&g[\mathbf{p}]$, if evaluated under an assignment $\mathbf{A}$, denoted by $\Lambda(\&g[\mathbf{p}], \mathbf{A})$. The next section will provide definitions of particular nogoods that can be learned for various types of external sources, i.e., to instantiate $\Lambda(\cdot, \cdot)$. The crucial requirement for learned nogoods is *correctness*, which intuitively holds if the nogood can be added without eliminating compatible sets.

Algorithm HEX-**Eval**

```
Input: A HEX-program $\Pi$
Output: All answer sets of $\Pi$
$\hat{\Pi} \leftarrow \Pi$ with ext. atoms $\&g[\mathbf{p}](\mathbf{c})$ replaced by $e_{\&g[\mathbf{p}]}(\mathbf{c})$
Add guessing rules for all replacement atoms to $\hat{\Pi}$
$\nabla \leftarrow \emptyset$      // set of dynamic nogoods
$\Gamma \leftarrow \emptyset$    // set of all compatible sets
while $\mathbf{C} \neq \bot$ do                                          (a)
    $\mathbf{C} \leftarrow \bot$
    $inconsistent \leftarrow false$
    while $\mathbf{C} = \bot$ and $inconsistent = false$ do              (b)
        $\mathbf{A} \leftarrow$HEX-CDNL($\Pi$,$\hat{\Pi}$,$\nabla$)                             (c)
        if $\mathbf{A} = \bot$ then $inconsistent \leftarrow true$
        else
            $compatible \leftarrow true$
            for all external atoms $\&g[\mathbf{p}]$ in $\Pi$ do                 (d)
                Evaluate $\&g[\mathbf{p}]$ under $\mathbf{A}$
                $\nabla \leftarrow \nabla \cup \Lambda(\&g[\mathbf{p}], \mathbf{A})$                     (e)
                Let $\mathbf{A}^{\&g[\mathbf{p}](\mathbf{c})} = 1 \Leftrightarrow \mathbf{T}e_{\&g[\mathbf{p}](\mathbf{c})} \in \mathbf{A}$
                if $\exists \mathbf{c}\colon f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) \neq \mathbf{A}^{\&g[\mathbf{p}](\mathbf{c})}$ then
                    Add $\mathbf{A}$ to $\nabla$
                    $compatible \leftarrow false$
            if $compatible$ then $\mathbf{C} \leftarrow \mathbf{A}$
    if $inconsistent = false$ then
        // $\mathbf{C}$ is a compatible set of $\Pi$
        $\nabla \leftarrow \nabla \cup \{\mathbf{C}\}$ and $\Gamma \leftarrow \Gamma \cup \{\mathbf{C}\}$
return $\subseteq$-minimal $\{\{\mathbf{T}a \in \mathbf{A} \mid a \in A(\Pi)\} \mid \mathbf{A} \in \Gamma\}$
```

Algorithm HEX-**CDNL**

```
Input: A program $\Pi$, its guessing program $\hat{\Pi}$, a set of correct
       nogoods $\nabla$ of $\Pi$
Output: An answer set of $\hat{\Pi}$ (candidate for a compatible set
        of $\Pi$) which is a solution to all nogoods $d \in \nabla$, or $\bot$
        if none exists
$\mathbf{A} \leftarrow \emptyset$     // over $A(\hat{\Pi}) \cup BA(\hat{\Pi}) \cup BA(sh(\hat{\Pi}))$
$dl \leftarrow 0$           // decision level
while true do
    $(\mathbf{A}, \nabla) \leftarrow$ Propagation($\hat{\Pi}, \nabla, \mathbf{A}$)
    if $\delta \subseteq \mathbf{A}$ for some $\delta \in \Delta_{\hat{\Pi}} \cup \Theta_{sh(\hat{\Pi})} \cup \nabla$ then          (a)
        if $dl = 0$ then return $\bot$
        $(\epsilon, k) \leftarrow$ Analysis($\delta, \hat{\Pi}, \nabla, \mathbf{A}$)
        $\nabla \leftarrow \nabla \cup \{\epsilon\}$ and $dl \leftarrow k$
        $\mathbf{A} \leftarrow \mathbf{A} \setminus \{\sigma \in \mathbf{A} \mid k < dl(\sigma)\}$
    else if $\mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}} = A(\hat{\Pi}) \cup BA(\hat{\Pi}) \cup BA(sh(\hat{\Pi}))$ then  (b)
        $U \leftarrow$ UnfoundedSet($\hat{\Pi}, \mathbf{A}$)
        if $U \neq \emptyset$ then
            let $\delta \in \lambda_{\hat{\Pi}}(U)$ such that $\delta \subseteq \mathbf{A}$
            if $\{\sigma \in \delta \mid 0 < dl(\sigma)\} = \emptyset$ then return $\bot$
            $(\epsilon, k) \leftarrow$ Analysis($\delta, \hat{\Pi}, \nabla, \mathbf{A}$)
            $\nabla \leftarrow \nabla \cup \{\epsilon\}$ and $dl \leftarrow k$
            $\mathbf{A} \leftarrow \mathbf{A} \setminus \{\sigma \in \mathbf{A} \mid k < dl(\sigma)\}$
        else return $\mathbf{A}^{\mathbf{T}} \cap A(\hat{\Pi})$
    else if Heuristic decides to evaluate $\&g[\mathbf{p}]$ then                (c)
        Evaluate $\&g[\mathbf{p}]$ under $\mathbf{A}$ and set
        $\nabla \leftarrow \nabla \cup \Lambda(\&g[\mathbf{p}], \mathbf{A})$
    else                                                                      (d)
        $\sigma \leftarrow$ Select($\hat{\Pi}, \nabla, \mathbf{A}$) and $dl \leftarrow dl + 1$
        $\mathbf{A} \leftarrow \mathbf{A} \circ (\sigma)$
```

Definition 4 (Correct Nogoods) *A nogood δ is* correct wrt. a program Π*, if all compatible sets of* Π *are solutions to δ.*

In our subsequent exposition we assume that the program Π is clear from the context. The overall approach consists of two parts. First, HEX-CDNL computes model candidates; it is essentially an ordinary ASP solver, but includes calls to external sources in order to learn additional nogoods. The external calls in this algorithm are not required for correctness of the algorithm, but may influence performance dramatically as discussed in Section 5. Second, Algorithm HEX-Eval uses Algorithm HEX-CDNL to produce model candidates and checks each of them against the external sources (followed by a minimality check). Here, the external calls are crucial for correctness of the algorithm.

For computing a model candidate, HEX-CDNL basically employs the conflict-driven approach presented in Drescher et al. (2008) as summarized in Section 2, where the main difference is the addition of Part (c). Our extension is driven by the following idea: whenever (unit and unfounded set) propagation does not derive any further atoms and the assignment is still incomplete, the algorithm possibly evaluates external atoms (driven by a heuristic) instead of simply guessing truth values. This might lead to the addition of new nogoods, which can in turn cause the propagation procedure to derive further atoms. Guessing of truth values only becomes necessary if no deterministic conclusions can be drawn and the evaluation of external atoms does not yield further nogoods; guessing also occurs if the heuristic does not decide to evaluate.

For a more formal treatment, let $\mathcal{E}$ be the set of all external predicates with input list that occur in Π, and let $\mathcal{D}$ be the set of all signed literals over atoms in $A(\Pi) \cup A(\hat{\Pi}) \cup BA(\hat{\Pi})$.

Then, a *learning function* for Π is a mapping $\Lambda : \mathcal{E} \times 2^{\mathcal{D}} \mapsto 2^{2^{\mathcal{D}}}$. We extend our notion of correct nogoods to correct learning functions $\Lambda(\cdot, \cdot)$, as follows:

Definition 5 *A learning function* Λ *is* correct *for a program* Π, *iff all* $d \in \Lambda(\&g[\mathbf{p}], \mathbf{A})$ *are correct for* Π, *for all* $\&g[\mathbf{p}]$ *in* $\mathcal{E}$ *and* $\mathbf{A} \in 2^{\mathcal{D}}$.

Restricting to learning functions that are correct for Π, the following results hold.

Proposition 1 *If for input* Π, $\hat{\Pi}$ *and* ∇, HEX-*CDNL returns (i) an interpretation* $\mathbf{A}$, *then* $\mathbf{A}$ *is an answer set of* $\hat{\Pi}$ *and a solution to* ∇*; (ii)* $\bot$, *then* Π *has no compatible set that is a solution to* ∇.

Proof (Sketch). (i) The proof mainly follows (Drescher et al. 2008). In our algorithm we have potentially more nogoods, which can never produce further answer sets but only eliminate them. Hence, each produced interpretation $\mathbf{A}$ is an answer set of $\hat{\Pi}$. (ii) By completeness of Drescher et al. (2008) we only need to justify that adding $\Lambda(\&g[\mathbf{p}], \mathbf{A})$ after evaluation of $\&g[\mathbf{p}]$ does not eliminate compatible sets of Π. For this purpose we need to show that when one of the added nogoods fires, the interpretation is incompatible with the external sources anyway. But this follows from the correctness of $\Lambda(\cdot, \cdot)$ and (for derived nogoods) from the completeness of Drescher et al. (2008). □

The basic idea of HEX-Eval is to compute all compatible sets of Π by the loop at (a) and checking subset-minimality afterwards. For computing compatible sets, the loop at (b) uses HEX-CDNL to compute answer sets of $\hat{\Pi}$ in (c), i.e., candidate compatible sets of Π, and subsequently checks compatibility for each external atom in (d). Here the external calls are crucial for correctness. However, different from the translation approach, the external source evaluation serves not only for compatibility checking, but also for generating additional dynamic nogoods $\Lambda(\&g[\mathbf{p}], \mathbf{A})$ in Part (e). We have the following result.

Proposition 2 HEX-*Eval computes all answer sets of* Π.

Proof (Sketch). We first show that the loop at (b) yields after termination a compatible set $\mathbf{C}$ of Π that is a solution of ∇ at the stage of entering the loop iff such a compatible set does exist, and yields $\mathbf{C} = \bot$ iff no such compatible set exists.

Suppose that $\mathbf{C} \neq \bot$ after the loop. Then $\mathbf{C}$ was assigned $\mathbf{A} \neq \bot$, which was returned by HEX-CDNL(Π, $\hat{\Pi}$, ∇). From Proposition 1 (ii) it follows that $\mathbf{C}$ is an answer set of $\hat{\Pi}$ and a solution to ∇. Thus (i) of Definition 2 holds. As $compatible = true$, the for loop guarantees the compatibility with the external sources in (ii) of Definition 2: if some source output on input from $\mathbf{C}$ is not compatible with the guess, $\mathbf{C}$ is rejected (and added as nogood). Otherwise $\mathbf{C}$ coincides with the behavior of the external sources, i.e., it satisfies (ii) of Definition 2. Thus, $\mathbf{C}$ is a compatible set of Π wrt. ∇ at call time. As only correct nogoods are added to ∇, it is also a compatible set of Π wrt. the initial set ∇.

Otherwise, after the loop $\mathbf{C} = \bot$. Then $inconsistent = true$, which means that the call HEX-CDNL(Π, $\hat{\Pi}$, ∇) returned $\bot$. By Proposition 1 (ii) there is no answer set of $\hat{\Pi}$ which is a solution to ∇. As only correct nogoods were added to ∇, there exists also no answer set of $\hat{\Pi}$ which is a solution to the original set ∇. Thus the loop at (b) operates as desired.

The loop at (a) then enumerates one by one all compatible sets and terminates: the update of ∇ with $\mathbf{C}$ prevents recomputing $\mathbf{C}$, and thus the number of compatible sets decreases. As by Definition 3 the answer sets of Π are the compatible sets with subset-minimal true part of original literals, the overall algorithm correctly outputs all answer sets of Π. □

Example 2 Let *&empty* be an external atom with one (nonmonotonic) predicate input p, such that its output is c_0 if the extension of p is empty and c_1 otherwise. Consider the program Π_e consisting of the rules

$$p(c_0).\ dom(c_0).\ dom(c_1).\ dom(c_2).\quad p(X) \leftarrow dom(X), \&empty[p](X)$$

Algorithm HEX-Eval transforms Π_e into the guessing program $\hat{\Pi}_e$:

$$p(c_0).\ dom(c_0).\ dom(c_1).\ dom(c_2).\quad p(X) \leftarrow dom(X), e_{\&empty[p]}(X).$$
$$e_{\&empty[p]}(X) \vee ne_{\&empty[p]}(X) \leftarrow dom(X).$$

The traditional evaluation strategy without learning will then produce 2^3 model candidates in HEX-CDNL, which are subsequently checked in HEX-Eval. For instance, the guess $\{\mathbf{T}ne_{\&empty[p]}(c_0), \mathbf{T}e_{\&empty[p]}(c_1), \mathbf{T}ne_{\&empty[p]}(c_2)\}$ leads to the model candidate $\{\mathbf{T}ne_{\&empty[p]}(c_0), \mathbf{T}e_{\&empty[p]}(c_1), \mathbf{T}ne_{\&empty[p]}(c_2), \mathbf{T}p(c_1)\}$ (neglecting false atoms and facts). This is also the only model candiate which passes the compatibility check: $p(c_0)$ is always true, and therefore $e_{\&empty[p]}(c_1)$ must also be true due to definition of the external atom. This allows for deriving $p(c_1)$ by the first rule of the program. All other atoms are false due to minimality of answer sets. □

The effects of the additionally learned nogoods will be discussed in Section 4 after having formally specified concrete $\Lambda(\&g[\mathbf{p}], \mathbf{A})$ for various types of external sources.

4 Nogoods for External Behavior Learning

We now discuss nogoods generated for external behavior learning (EBL) in detail. EBL is triggered by external source evaluations instead of conflicts. The basic idea is to integrate knowledge about the external source behavior into the program to guide the search. The program evaluation then starts with an empty set of learned nogoods and the preprocessor generates a guessing rule for each ground external atom, as discussed in Section 2. Further nogoods are added during the evaluation as more information about external sources becomes available. This is in contrast to traditional evaluation, where external atoms are assigned arbitrary truth values which are checked only after the assignment was completed.

We will first show how to construct useful learned nogoods after evaluating external atoms, if we have no further information about the internals of external sources, called *uninformed learning*. In this case we can only learn simple input/output relationships. Subsequently we consider *informed learning*, where additional information about properties of external sources is available. This allows for using more elaborated learning strategies.

4.1 Uninformed Learning

We first assume that we do not have information about the internals and consider external sources as black boxes. Hence, we can just apply very general rules for learning: whenever an external predicate with input list $\&g[\mathbf{p}]$ is evaluated under an assignment $\mathbf{A}$, we learn that the input $\mathbf{A}|_{\mathbf{p}}$ for $\mathbf{p} = p_1, \ldots, p_n$ to the external atom $\&g$ produces the output $ext(\&g[\mathbf{p}], \mathbf{A})$. This can be formalized as the following set of nogoods.

Definition 6 *The learning function for a general external predicate with input list* $\&g[\mathbf{p}]$ *in program* Π *under assignment* $\mathbf{A}$ *is defined as*

$$\Lambda_g(\&g[\mathbf{p}], \mathbf{A}) = \left\{ \mathbf{A}|_{\mathbf{p}} \cup \{\mathbf{F}e_{\&g[\mathbf{p}]}(\mathbf{c})\} \mid \mathbf{c} \in ext(\&g[\mathbf{p}], \mathbf{A}) \right\} .$$

Table 1: Learned Nogoods of Example 3

Guess	Learned Nogood
$\left\{\begin{array}{l}\mathbf{T}e_{\&empty[p]}(c_0), \mathbf{T}ne_{\&empty[p]}(c_1),\\ \mathbf{T}ne_{\&empty[p]}(c_2)\end{array}\right\}$	$\{\mathbf{T}p(c_0), \mathbf{F}p(c_1), \mathbf{F}p(c_2), \mathbf{F}e_{\&empty[p]}(c_1)\}$
$\left\{\begin{array}{l}\mathbf{T}e_{\&empty[p]}(c_0), \mathbf{T}ne_{\&empty[p]}(c_1),\\ \mathbf{T}e_{\&empty[p]}(c_2), p(c_2)\end{array}\right\}$	$\{\mathbf{T}p(c_0), \mathbf{F}p(c_1), \mathbf{T}p(c_2), \mathbf{F}e_{\&empty[p]}(c_1)\}$
$\left\{\begin{array}{l}\mathbf{T}e_{\&empty[p]}(c_0), \mathbf{T}e_{\&empty[p]}(c_1),\\ \mathbf{T}ne_{\&empty[p]}(c_2), p(c_1)\end{array}\right\}$	$\{\mathbf{T}p(c_0), \mathbf{T}p(c_1), \mathbf{F}p(c_2), \mathbf{F}e_{\&empty[p]}(c_1)\}$
$\left\{\begin{array}{l}\mathbf{T}e_{\&empty[p]}(c_0), \mathbf{T}e_{\&empty[p]}(c_1),\\ \mathbf{T}e_{\&empty[p]}(c_2), p(c_1), p(c_2)\end{array}\right\}$	$\{\mathbf{T}p(c_0), \mathbf{T}p(c_1), \mathbf{T}p(c_2), \mathbf{F}e_{\&empty[p]}(c_1)\}$

In the simplest case, an external atom has no input and the learned nogoods are unary, i.e., of the form $\{\mathbf{F}e_{\&g[]}(\mathbf{c})\}$. Thus, it is learned that certain tuples are in the output of the external source, i.e. they must not be false. For external sources with input predicates, the added rules encode the relationship between the output tuples and the provided input.

Example 3 (ctd.) Recall Π_e from Example 2. Without learning, the algorithms produce 2^3 model candidates and check them subsequently. It turns out that EBL allows for falsification of some of the guesses without actually evaluating the external atoms. Suppose the reasoner first tries the guesses containing literal $\mathbf{T}e_{\&empty[p]}(c_0)$. While they are checked against the external sources, the described learning function allows for adding the externally learned nogoods shown in Table 1. Observe that the combination $\mathbf{T}p(c_0), \mathbf{F}p(c_1), \mathbf{F}p(c_2)$ will be reconstructed also for different choices of the guessing variables. As $p(c_0)$ is a fact, it is true independent of the choice between $e_{\&empty[p]}(c_0)$ and $ne_{\&empty[p]}(c_0)$. E.g., the guess $\mathbf{F}e_{\&empty[p]}(c_0)$, $\mathbf{F}e_{\&empty[p]}(c_1)$, $\mathbf{F}e_{\&empty[p]}(c_2)$ leads to the same extension of p. This allows for reusing the nogood, which is immediately invalidated without evaluating the external atoms. Different guesses with the same input to an external source allow for reusing learned nogoods, at the latest when the candidate is complete, but before the external source is called for validation. However, very often learning allows for discarding guesses even earlier. For instance, we can derive $\{\mathbf{T}p(c_0), \mathbf{F}e_{\&empty[p]}(c_1)\}$ from the nogoods above in 3 resolution steps. Such derived nogoods will be learned after running into a couple of conflicts. We can derive $\mathbf{T}e_{\&empty[p]}(c_1)$ from $p(c_0)$ even before the truth value of $\mathbf{F}e_{\&empty[p]}(c_1)$ is set, i.e., external learning guides the search while the traditional evaluation algorithm considers the behavior of external sources only during postprocessing. □

For the next result, let Π be a program which contains an external atom of form $\&g[\mathbf{p}](\cdot)$.

Lemma 1 *For all assignments* $\mathbf{A}$*, the nogoods* $\Lambda_g(\&g[\mathbf{p}], \mathbf{A})$ *(Def. 6) are correct wrt.* Π.

Proof (Sketch). The added nogood for an output tuple $\mathbf{c} \in \mathit{ext}(\&g[\mathbf{p}], \mathbf{A})$ contains $\mathbf{A}|_{\mathbf{p}}$ and the negated replacement atom $\mathbf{F}e_{\&g[\mathbf{p}]}(\mathbf{c})$. If the nogood fires, then the guess was wrong as the replacement atom is guessed false but the tuple $(\mathbf{c})$ is in the output. Hence, the interpretation is not compatible and cannot be an answer set anyway. □

4.2 Informed Learning

The learned nogoods of the above form can become quite large as they include the whole input to the external source. However, known properties of external sources can be exploited in order to learn smaller and more general nogoods. For example, if one of the input parameters of an external source is monotonic, it is not necessary to include information about false atoms in its extension, as the output will not shrink given larger input.

Properties for informed learning can be stated on the level of either *predicates* or individual *external atoms*. The former means that all usages of the predicate have the property. To understand this, consider predicate $\&union$ which takes two predicate inputs p and q and computes the set of all elements which are in at least one of the extensions of p or q. It will be *always* monotonic in both parameters, independently of its usage in a program. While an external source may lack a property in general, it may hold for particular usages.

Example 4 Consider an external atom $\&db[r_1, \dots, r_n, \mathit{query}](\mathbf{X})$ as an interface to an SQL query processor, which evaluates a given query (given as string) over tables (relations) provided by predicates $r_1, \dots, r_n$. In general, the atom will be nonmonotonic, but for special queries (e.g., simple selection of all tuples), it will be monotonic. □

Next, we discuss two particular cases of informed learning which customize the default learning function for generic external sources by exploiting properties of external sources, and finally present examples where the learning of user-defined nogoods might be useful.

Monotonic Atoms. A parameter p_i of an external atom $\&g$ is called *monotonic*, if $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = 1$ implies $f_{\&g}(\mathbf{A}', \mathbf{p}, \mathbf{c}) = 1$ for all $\mathbf{A}'$ with $\mathbf{A}'|_{p_i} \supseteq \mathbf{A}|_{p_i}$ and $\mathbf{A}'|_{p'} = \mathbf{A}|_{p'}$ for all other $p' \neq p_i$. The learned nogoods $\Lambda(\&g[\mathbf{p}], \mathbf{A})$ after evaluating $\&g[\mathbf{p}]$ are not required to include $\mathbf{F}p_i(t_1, \dots, t_\ell)$ for monotonic $p_i \in \mathbf{p}$. That is, for an external predicate with input list $\&g[\mathbf{p}]$ with monotonic input parameters $\mathbf{p_m} \subseteq \mathbf{p}$ and nonmonotonic parameters $\mathbf{p_n} = \mathbf{p} \setminus \mathbf{p_m}$, the set of learned nogoods can be restricted as follows.

Definition 7 *The learning function for an external predicate &g with input list* $\mathbf{p}$ *in program* Π *under assignment* $\mathbf{A}$*, such that &g is monotonic in* $\mathbf{p_m} \subseteq \mathbf{p}$*, is defined as*

$$\Lambda_m(\&g[\mathbf{p}], \mathbf{A}) = \Big\{ \{\mathbf{T}a \in \mathbf{A}|_{\mathbf{p_m}}\} \cup \mathbf{A}|_{\mathbf{p_n}} \cup \{\mathbf{F}e_{\&g[\mathbf{p}]}(\mathbf{c})\} \mid \mathbf{c} \in \mathit{ext}(\&g[\mathbf{p}], \mathbf{A}) \Big\} \ .$$

Example 5 Consider the external atom $\&\mathit{diff}[p, q](X)$ which computes the set of all elements X that are in the extension of p, but not in the extension of q. Suppose it is evaluated under $\mathbf{A}$, s.t. $\mathit{ext}(p, \mathbf{A}) = \{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}p(c)\}$ and $\mathit{ext}(q, \mathbf{A}) = \{\mathbf{F}q(a), \mathbf{T}q(b), \mathbf{F}q(c)\}$. Then the output of the atom is $\mathit{ext}(\&\mathit{diff}[p, q], \mathbf{A}) = \{a\}$ and the (only) naively learned nogood is $\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}p(c), \mathbf{F}q(a), \mathbf{T}q(b), \mathbf{F}q(c), \mathbf{F}e_{\&\mathit{diff}[p,q]}(a)\}$. However, due to monotonicity of $\&\mathit{diff}[p, q]$ in p, it is not necessary to include $\mathbf{F}p(c)$ in the nogood; the output of the external source will not shrink even if $p(c)$ becomes true. Therefore the (more general) nogood $\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b), \mathbf{F}q(c), \mathbf{F}e_{\&\mathit{diff}[p,q]}(a)\}$ suffices to correctly describe the input-output behavior. □

Functional Atoms. When evaluating $\&g[\mathbf{p}]$ with some functional $\&g$ under assignment $\mathbf{A}$, only one output tuple can be contained in $\mathit{ext}(\&g[\mathbf{p}], \mathbf{A})$, formally: for all assignments $\mathbf{A}$ and all $\mathbf{c}$, if $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = 1$ then $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}') = 0$ for all $\mathbf{c}' \neq \mathbf{c}$. Therefore the following nogoods may be added right from the beginning.

Definition 8 *The learning function for a functional external predicate &g with input list* $\mathbf{p}$ *in program* Π *under assignment* $\mathbf{A}$ *is defined as*

$$\Lambda_f(\&g[\mathbf{p}], \mathbf{A}) = \big\{\{\mathbf{T}e_{\&g[\mathbf{p}]}(\mathbf{c}), \mathbf{T}e_{\&g[\mathbf{p}]}(\mathbf{c}')\} \mid \mathbf{c} \neq \mathbf{c}'\big\} \ .$$

However, our implementation of this learning rule does not generate all pairs of output tuples beforehand. Instead, it memorizes all generated output tuples $\mathbf{c^i}$, $1 \le i \le k$ during evaluation of external sources. Whenever a new output tuple $\mathbf{c}'$ is added, it also adds all nogoods which force previously derived output tuples $\mathbf{c^i}$ to be false.

Example 6 Consider the rules

$$\begin{aligned} out(X) &\leftarrow \&concat[A, x](X), strings(A), dom(X) \\ strings(X) &\leftarrow dom(X), \text{not } out(X) \end{aligned}$$

where $\&concat[a, b](c)$ is true iff string c is the concatenation of strings a and b, and observe that the external atom is involved in a cycle through negation. As the extension of the domain dom can be large, many ground instances of the external atom are generated. The old evaluation algorithm guesses their truth values completely uninformed. E.g., $e_{\&concat}(x, x, xx)$ (the replacement atom of $\&concat[A, x](X)$ with $A = x$ and $X = xx$, where $dom(x)$ and $dom(xx)$ are supposed to be facts) is in each guess set randomly to true or to false, independent of previous guesses. In contrast, with learning over external sources, the algorithm learns after the first evaluation that $e_{\&concat}(x, x, xx)$ must be true. Knowing that *&concat* is functional, all atoms $e_{\&concat}(x, x, O)$ with $O \neq xx$ must also be false. □

For the next result, let Π be a program which contains an external atom of form $\&g[\mathbf{p}](\cdot)$.

Lemma 2 *For all assignments* $\mathbf{A}$*, (i) the nogoods* $\Lambda_m(\&g[\mathbf{p}], \mathbf{A})$ *(Def. 7) are correct wrt.* Π*, and (ii) if &g is functional, the nogoods* $\Lambda_f(\&g[\mathbf{p}], \mathbf{A})$ *(Def. 8) are correct wrt.* Π.

Proof (Sketch). For monotonic external sources we must show that negative input literals over monotonic parameters can be removed from the learned nogoods without affecting correctness. For uninformed learning, we argued that for output tuple $\mathbf{c} \in ext(\&g[\mathbf{p}], \mathbf{A})$, the replacement atom $e_{\&g[\mathbf{p}]}(\mathbf{c})$ must not be be guessed false if the input to $\&g[\mathbf{p}](\mathbf{c})$ is $\mathbf{A}|_{\mathbf{p}}$ under assignment $\mathbf{A}$. However, as the output of $\&g$ grows monotonically with the extension of a monotonic parameter $p \in \mathbf{p_m}$, the same applies for any $\mathbf{A}'$ which is "larger" in p, i.e., $\{\mathbf{T}a \in \mathbf{A}'|_p\} \supseteq \{\mathbf{T}a \in \mathbf{A}|_p\}$ and consequently $\{\mathbf{F}a \in \mathbf{A}'|_p\} \subseteq \{\mathbf{F}a \in \mathbf{A}|_p\}$. Hence, the negative literals are not relevant wrt. output tuple $\mathbf{c}$ and can be removed from the nogood.

For functional $\&g$, we must show that the nogoods $\big\{\{\mathbf{T}e_{\&g[\mathbf{p}]}(\mathbf{c}), \mathbf{T}e_{\&g[\mathbf{p}]}(\mathbf{c}')\} \mid \mathbf{c} \neq \mathbf{c}'\big\}$ are correct. Due to functionality, the external source cannot return more than one output tuple for the same input. Therefore no such guess can be an answer set as it is not compatible. Hence, the nogoods do not eliminate possible answer sets. □

User-defined Learning. In many cases the developer of an external atom has more information about the internal behavior. This allows for defining more effective nogoods. It is therefore beneficial to give the user the possibility to customize learning functions. Currently, user-defined functions need to directly specify the learned nogoods. The development of a user-friendly language for writing learning functions is subject to future work.

Example 7 Consider the program

$$\begin{aligned} r(X, Y) \vee nr(X, Y) &\leftarrow d(X), d(Y) \\ r(V, W) &\leftarrow \&tc[r](V, W), d(V), d(W) \end{aligned}$$

It guesses, for some set of nodes $d(X)$, all subgraphs of the complete graph. Suppose $\&tc[r]$ checks if the edge selection $r(X, Y)$ is $\underline{t}$ransitively $\underline{c}$losed; if this is the case, the output is empty, otherwise the set of missing transitive edges is returned. For instance, if the extension of r is $\{(a, b), (b, c)\}$, then the output of $\&tc$ will be $\{(a, c)\}$, as this edge is missing in order to make the graph transitively closed. The second rule eliminates all subgraphs which are not transitively closed. Note that $\&tc$ is nonmonotonic. The guessing program is

$$\begin{aligned} r(X, Y) \vee nr(X, Y) &\leftarrow d(X), d(Y) \\ r(V, W) &\leftarrow e_{\&tc[r]}(V, W), d(V), d(W) \\ e_{\&tc[r]}(V, W) \vee ne_{\&tc[r]}(V, W) &\leftarrow d(V), d(W) \end{aligned}$$

The naive implementation guesses for n nodes all $2^{\frac{n(n-1)}{2}}$ subgraphs and checks the transitive closure for each of them, which is costly. Consider the domain $D = \{a, b, c, d, e, f\}$. After checking one selection with $r(a, b), r(b, c), nr(a, c)$, we know that *no* selection containing these three literals will be transitively closed. This can be formalized as a user-defined learning function. Suppose we have just checked our first guess $r(a, b), r(b, c)$, and $nr(x, y)$ for all other $(x, y) \in D \times D$. Compared to the nogood learned by the general learning function, the nogood $\{\mathbf{T}r(a, b), \mathbf{T}r(b, c), \mathbf{F}r(a, c), \mathbf{F}e_{\&tc[r]}(a, c)\}$ is a more general description of the conflict reason, containing only relevant edges. It is immediately violated and future guesses containing $\{\mathbf{T}r(a, b), \mathbf{T}r(b, c), \mathbf{F}r(a, c)\}$ are avoided. $\square$

Example 8 (Linearity) A useful learning function for $\&\mathit{diff}[p, q](X)$ is the following: whenever an element is in p but not in q, it belongs to the output of the external atom. This user-defined function works elementwise and produces nogoods with three literals each. We call this property *linearity*. In contrast, the naive learning function from the Section 4.1 includes the complete extensions of p and q in the nogoods, which are less general. $\square$

For user-defined learning, correctness of the learning function must be asserted.

5 Implementation and Evaluation

We have integrated CLASP into our reasoner DLVHEX; previous versions of DLVHEX used just DLV. In order to learn nogoods from external sources we exploit CLASP's SMT interface, which was previously used for the special case of constraint answer set solving and implemented in the CLINGCON system (Gebser et al. 2009; Ostrowski and Schaub 2012). We compare three configurations: DLVHEX with DLV backend, DLVHEX with (conflict-driven) CLASP backend but without EBL, and DLVHEX with CLASP backend and EBL.

For our experiments we used variants of the above examples, the DLVHEX test suite, and default reasoning over ontologies. It appeared that learning has high potential to reduce the number of candidate models. Also the number of total variable assignments and backtracks during search decreased drastically in many cases. This suggests that candidate rejection often needs only parts of interpretations and is possible early in the evaluation. All benchmarks were carried out on a machine with two 12-core AMD Opteron 6176 SE CPUs and 128 GB RAM, running Linux and using CLASP 2.0.5 and DLV Dec 21 2011 as solver backends. For each benchmark instance, the average of three runs was calculated, having a timeout of 300 seconds, and a memout of 2 GB for each run. We report runtime in seconds; gains and speedups are given as a factor.

Set Partitioning. The following program partitions a set S into two subsets $S_1, S_2 \subseteq S$ such that $|S_1| \leq 2$. The partitioning criterion is expressed by two rules for $S_1 = S \setminus S_2$ and $S_2 = S \setminus S_1$. The implementation is by the use of external atom *&diff* (cf. Example 5):

$$\begin{aligned}
&dom(c_1).\ \cdots\ dom(c_n).\\
nsel(X) &\leftarrow dom(X), \&diff[dom, sel](X).\\
sel(X) &\leftarrow dom(X), \&diff[dom, nsel](X).\\
&\leftarrow sel(X), sel(Y), sel(Z), X \neq Y, X \neq Z, Y \neq Z.
\end{aligned}$$

The results in Table 2a compare the run of the reasoner with different configurations for computing (i) all models resp. (ii) the first model. In both cases, using the conflict-driven CLASP reasoner instead of DLV as backend already improves efficiency. Adding EBL leads to a further improvement: in case (ii), the formerly exponentially growing runtime becomes almost constant. When computing all answer sets, the runtime is still exponential as exponentially many subset choices must be considered (due to the encoding); however, also in this case many of them can be pruned early by learning, which makes the runtime appear linear for the shown range of instance sizes. Moreover, our experiments show that the delay between the models decreases over time when EBL is used (not shown in the table), while it is constant without EBL due to the generation of additional nogoods.

Default Reasoning over Description Logic Ontologies. We consider now a more realistic scenario using the DL-plugin (Eiter et al. 2008) for DLVHEX, which integrates description logics (DL) knowledge bases and nonmonotonic logic programs. The DL-Plugin allows to access an ontology using the description logic reasoner RacerPro 1.9.0 (http://www.racer-systems.com/). For our first experiment, consider the program (shown left) and the terminological part of a DL knowledge base on the right:

$$\begin{aligned}
birds(X) &\leftarrow DL[Bird](X). & Flier &\sqsubseteq \neg NonFlier\\
flies(X) &\leftarrow birds(X), \text{not } neg\_flies(X). & Penguin &\sqsubseteq Bird\\
neg\_flies(X) &\leftarrow birds(X), DL[Flier \uplus flies; \neg Flier](X). & Penguin &\sqsubseteq NonFlier
\end{aligned}$$

This encoding realizes the classic Tweety bird example using DL-atoms (which is an alternative syntax for external atoms in this example and allows to express queries over description logics in a more accessible way). The ontology states that *Flier* is disjoint with *NonFlier*, and that penguins are birds and do not fly; the rules express that birds fly by default, i.e., unless the contrary is derived. The program amounts to the Ω-transformation of default logic over ontologies to dl-programs (Dao-Tran et al. 2009), where the last rule ensures consistency of the guess with the DL ontology. If the assertional part of the DL knowledge base contains $Penguin(tweety)$, then $flies(tweety)$ is inconsistent with the given DL-program ($neg\_flies(tweety)$ is derived by monotonicity of DL atoms and $flies(tweety)$ loses its support). Note that defaults cannot be encoded in standard (monotonic) description logics, which is achieved here by the cyclic interaction of DL-rules and the DL knowledge base.

As all individuals appear in the extension of the predicate *flier*, all of them are considered simultaneously. This requires a guess on the ability to fly for each individual and a subsequent check, leading to a combinatorial explosion. Intuitively, however, the property can be determined for each individual independently. Hence, a query may be split into independent subqueries, which is achieved by our learning function for *linear sources* in Example 8. The learned nogoods are smaller and more candidate models are eliminated. Table 2b shows the

Table 2: Benchmark Results (runtime in seconds, timeout 300s)

(a) Set Partitioning

# elements	all models			first model		
	DLV	CLASP w/o EBL	CLASP w EBL	DLV	CLASP w/o EBL	CLASP w EBL
1	0.07	0.08	0.07	0.08	0.07	0.07
5	0.20	0.16	0.10	0.08	0.08	0.07
10	12.98	9.56	0.17	0.56	0.28	0.07
11	38.51	21.73	0.19	0.93	0.63	0.08
12	89.46	49.51	0.19	1.69	1.13	0.08
13	218.49	111.37	0.20	3.53	2.31	0.10
14	—	262.67	0.28	8.76	3.69	0.10
⋮	—	—	⋮	⋮	⋮	⋮
18	—	—	0.45	128.79	62.58	0.12
19	—	—	0.42	—	95.39	0.10
20	—	—	0.54	—	91.16	0.11

(b) Bird-Penguin

# individuals	DLV	CLASP w/o EBL	CLASP w EBL
1	0.50	0.15	0.14
5	1.90	1.98	0.59
6	4.02	4.28	0.25
7	8.32	7.95	0.60
8	16.11	16.39	0.29
9	33.29	34.35	0.35
10	83.75	94.62	0.42
11	229.20	230.75	4.45
12	—	—	1.10
⋮	—	—	⋮
20	—	—	2.70

(c) Wine Ontology

Instance	concept completion		gain	
	CLASP w/o EBL	CLASP w EBL	max	avg
wine_0	25	31	33.02	6.93
wine_1	16	25	16.05	5.78
wine_2	14	22	11.82	4.27
wine_3	4	17	10.09	4.02
wine_4	4	17	6.83	2.87
wine_5	4	16	5.22	2.34
wine_6	4	13	2.83	1.52
wine_7	4	12	1.81	1.14
wine_8	4	4	1.88	1.08

(d) MCS

# contexts	DLV	CLASP w/o EBL	CLASP w EBL
3	0.07	0.05	0.04
4	1.04	0.68	0.14
5	0.23	0.15	0.05
6	2.63	1.44	0.12
7	8.71	4.39	0.17

runtime for different numbers of individuals and evaluation with and without EBL. The runs with EBL exhibit a significant speedup, as they exclude many model candidates, whereas the performance of the DLV and the CLASP backend without EBL is almost identical (unlike in the first example); here, most of the time is spent calling the description logic reasoner and not for the evaluation of the logic program.

The findings carry over to large ontologies (DL knowledge bases) used in real-world applications. We did similar experiments with a scaled version of the wine ontology (http://kaon2.semanticweb.org/download/test_ontologies.zip). The instances differ in the size of the ABox (ranging from 247 individuals in wine_0 to 20007 in wine_8) and in several other parameters (e.g., on the number of concept inclusions and concept equivalences; Motik and Sattler (2006) describe the particular instances wine_i). We implemented a number of default rules using an analogous encoding as above: e.g., wines not derivable to be dry are not dry, wines which are not sweet are assumed to be dry, wines are white by default unless they are known to be red. Here, we discuss the results of the latter scenario. The experiments classified the wines in the 34 main concepts of the ontology (the immediate subconcepts of the concept $Wine$, e.g., $DessertWine$ and $ItalianWine$), which have varying numbers of known concept memberships (e.g., ranging from 0 to 43, and 8 on average, in wine_0) and percentiles of red wines among them (from 0% to 100%, and 47% on average). The results are summarized in Table 2c. There, entries for concept completion state the number of classified concepts. Again, there is almost no difference between the DLV and the CLASP backend without EBL, but EBL leads to a significant improvement for most concepts and ontology sizes. E.g., there is a gain for 16 out of the 34 concepts of the wine_0 runs, as EBL can exploit linearity. Furthermore, we observed that 6 additional instances can be solved within the 300 seconds time limit. If a concept could be classified both with and without EBL, we could observe a gain of up to 33.02 (on average 6.93). As expected, larger categories profit more from EBL as we can reuse learned nogoods in these instances.

Besides Ω, Dao-Tran et al. (2009) describe other transformations of default rules over description logics. Experiments with this transformations revealed that the structure of the resulting HEX-programs prohibits an effective reuse of learned nogoods. Hence, the overall picture does not show a significant gain with EBL for these encodings, we could however still observe a small improvement for some runs.

Multi-Context Systems (MCS). MCS (Brewka and Eiter 2007) is a formalism for interlinking multiple knowledge-based systems (the contexts). Eiter et al. (2010) define *inconsistency explanations (IE)* for MCS, and present a system for finding such explanations on top of DLVHEX. In our benchmarks we computed explanations for inconsistent multi-context systems with 3 up to 7 contexts. For each number we computed the average runtime over several instances with different topologies (tree, zigzag, diamond), which were randomly created with an available benchmark generator, and report the results in Table 2d.

Unlike in the previous benchmark we could already observe a speedup of up to 1.98 when using CLASP instead of the DLV backend. This is because of two reasons: first, CLASP is more efficient than DLV for the given problem, and second, CLASP was tightly integrated into DLVHEX, whereas using DLV requires interprocess communication. However, the most important aspect is again EBL, which leads to a further significant speedup with a factor of up to 25.82 compared to CLASP without EBL.

Logic Puzzles. Another experiment concerns logic puzzles. We encoded *Sudoku* as a HEX-program, such that the logic program makes a guess of assignments to the fields and an external atom is used for verifying the answer. In case of a negative verification result, the external atom indicates by user-defined learning rules the reason of the inconsistency, encoded a pair of assignments to fields which contradict one of the uniqueness rules.

As expected, all instances times out without EBL, because the logic program has no information about the rules of the puzzle and blindly guesses all assignments, which are subsequently checked by the external atom. But with EBL, the Sudoku instances could be solved in several seconds.

More details on the experiments and links to benchmarks and benchmark generators can be found at http:/www.kr.tuwien.ac.at/research/systems/dlvhex/experiments.html.

6 Discussion and Conclusion

The basic idea of our algorithm is related to constraint ASP solving presented in Gebser et al. (2009), and Ostrowski and Schaub (2012), which is realized in the CLINGCON system. External atom evaluation in our algorithm can superficially be regarded as constraint propagation. However, while both,Gebser et al. (2009) and Ostrowski and Schaub (2012), consider a particular application, we deal with a more abstract interface to external sources. An important difference between CLINGCON and EBL is that the constraint solver is seen as a black box, whereas we exploit known properties of external sources. Moreover, we support *user-defined learning*, i.e., customization of the default construction of conflict clauses to incorporate knowledge about the sources, as discussed in Section 4. Another difference is the construction of conflict clauses. ASP with CP has special constraint atoms, which may be contradictory, e.g., $\mathbf{T}(X > 10)$ and $\mathbf{T}(X = 5)$. The learned clauses are sets of constraint literals, which are kept as small as possible. In our algorithm we have usually *no* conflicts between ground external atoms as output atoms are mostly independent of each other (excepting e.g. functional sources). Instead, we have a strong relationship between the input and the output. This is reflected by conflict clauses which usually consist of (relevant) input atoms and the negation of one output atom. As in constraint ASP solving, the key for efficiency is keeping conflict clauses small.

We have extended conflict-driven ASP solving techniques from ordinary ASP to HEX-programs, which allow for using external atoms to access external sources. Our approach uses two types of learning. The classical type is conflict-driven clause learning, which derives conflict nogoods from conflict situations while the search tree is traversed. Adding such nogoods prevents the algorithm from running into similar conflicts again.

Our main contribution is a second type of learning which we call *external behavior learning* (EBL). Whenever external atoms are evaluated, further nogoods may be added which capture parts of the external source behavior. In the simplest case these nogoods encode that a certain input to the source leads to a certain output. This default learning function can be customized to learn shorter or more general nogoods. Customization is either done explicitly by the user, or learning functions are derived automatically from known properties of external atoms, which can be stated either on the level of external predicates or on the level of atoms. Currently we exploit monotonicity and functionality.

Future work includes the identification of further properties which allow for automatic derivation of learning functions. We further plan the development of a user-friendly language for writing user-defined learning functions. Currently, they require to specify the learned nogoods by hand. It may be more convenient to write rules that a certain input to an external source leads to a certain output, in (a restricted variant of) ASP or a more convenient language. The challenge is that evaluation of learning rules introduces additional overhead, hence there is another tradeoff between costs and benefit of EBL. Finally, also the development of heuristics for lazy evaluation of external sources is subject to future work.

References

BIERE, A., HEULE, M. J. H., VAN MAAREN, H., AND WALSH, T., Eds. 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press.

BREWKA, G. AND EITER, T. 2007. Equilibria in Heterogeneous Nonmonotonic Multi-Context Systems. In *AAAI'07*. AAAI Press, 385–390.

BREWKA, G., EITER, T., AND TRUSZCZYŃSKI, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

DAO-TRAN, M., EITER, T., AND KRENNWALLNER, T. 2009. Realizing default logic over description logic knowledge bases. In *ECSQARU'09*. Springer, 602–613.

DRESCHER, C., GEBSER, M., GROTE, T., KAUFMANN, B., KÖNIG, A., OSTROWSKI, M., AND SCHAUB, T. 2008. Conflict-driven disjunctive answer set solving. In *KR'08*. AAAI Press, 422–432.

EITER, T., FINK, M., IANNI, G., KRENNWALLNER, T., AND SCHÜLLER, P. 2011. Pushing efficient evaluation of HEX programs by modular decomposition. In *LPNMR'11*. Springer, 93–106.

EITER, T., FINK, M., SCHÜLLER, P., AND WEINZIERL, A. 2010. Finding explanations of inconsistency in multi-context systems. In *KR'10*. AAAI Press, 329–339.

EITER, T., IANNI, G., KRENNWALLNER, T., AND SCHINDLAUER, R. 2008. Exploiting conjunctive queries in description logic programs. *Ann. Math. Artif. Intell. 53,* 1–4, 115–152.

EITER, T., IANNI, G., LUKASIEWICZ, T., SCHINDLAUER, R., AND TOMPITS, H. 2008. Combining answer set programming with description logics for the semantic web. *Artif. Intell. 172,* 12-13, 1495–1539.

EITER, T., IANNI, G., SCHINDLAUER, R., AND TOMPITS, H. 2005. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In *IJCAI'05*. Professional Book Center, 90–96.

EITER, T., IANNI, G., SCHINDLAUER, R., AND TOMPITS, H. 2006. Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning. In *ESWC'06*. Springer, 273–287.

FABER, W., LEONE, N., AND PFEIFER, G. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell. 175,* 1, 278–298.

FAGES, F. 1994. Consistency of clark's completion and existence of stable models. *J. Meth. Logic Comp. Sci. 1*, 51–60.

GEBSER, M., KAUFMANN, B., KAMINSKI, R., OSTROWSKI, M., SCHAUB, T., AND SCHNEIDER, M. T. 2011. Potassco: The Potsdam answer set solving collection. *AI Commun. 24,* 2, 107–124.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artif. Intell. 187-188*, 52–89.

GEBSER, M., OSTROWSKI, M., AND SCHAUB, T. 2009. Constraint answer set solving. In *ICLP'09*. Springer, 235–249.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generat. Comput. 9,* 3–4, 365–386.

GIUNCHIGLIA, E., LIERLER, Y., AND MARATEA, M. 2006. Answer set programming based on propositional satisfiability. *J. Autom. Reason. 36,* 4, 345–377.

GOLDBERG, E. AND NOVIKOV, Y. 2007. BerkMin: A fast and robust SAT-solver. *Discrete Appl. Math. 155,* 12, 1549–1561.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S., AND SCARCELLO, F. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Logic 7,* 3, 499–562.

LIFSCHITZ, V. 2002. Answer Set Programming and Plan Generation. *Artif. Intell. 138*, 39–54.

LIN, F. AND ZHAO, Y. 2004. ASSAT: computing answer sets of a logic program by SAT solvers. *Artif. Intell. 157,* 1–2, 115–137.

MAREK, V. W. AND TRUSZCZYŃSKI, M. 1999. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm*. Springer, 375–398.

MOTIK, B. AND SATTLER, U. 2006. A comparison of reasoning techniques for querying large description logic ABoxes. In *LPAR'06*. Springer, 227–241.

NIEMELÄ, I. 1999. Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. *Ann. Math. Artif. Intell. 25,* 3–4, 241–273.

OSTROWSKI, M. AND SCHAUB, T. 2012. ASP modulo CSP: The clingcon system. *Theor. Pract. Log. Prog., Special Issue 28th Intl. Conf. Logic Programming*. To appear.

SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artif. Intell. 138,* 1-2, 181–234.