The DLVHEX *System for Knowledge Representation: Recent Advances (System Description)* ∗

Christoph Redl

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
(*e-mail:* redl@kr.tuwien.ac.at)

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

Abstract

The DLVHEX system implements the HEX-semantics, which integrates answer set programming (ASP) with arbitrary external sources. Since its first release ten years ago, significant advancements were achieved. Most importantly, the exploitation of properties of external sources led to efficiency improvements and flexibility enhancements of the language, and technical improvements on the system side increased user's convenience. In this paper, we present the current status of the system and point out the most important recent enhancements over early versions. While existing literature focuses on theoretical aspects and specific components, a bird's eye view of the overall system is missing. In order to promote the system for real-world applications, we further present applications which were already successfully realized on top of DLVHEX.

KEYWORDS: Answer Set Programming, Nonmonotonic Reasoning, Knowledge representation

1 Introduction

Answer Set Programming (ASP) is a declarative programming approach which has been gaining popularity for many applications in artificial intelligence and beyond (Brewka et al. 2011). Features such as the use of variables as a shortcut for all ground instances, aggregates and optimization statements, distinguish ASP from SAT and simplify the process of problem solving in many cases. However, since not all data or computation sources can (easily and effectively) be encoded in an ASP program, extensions of the formalism towards the integration of other formalisms are needed.

To this end, HEX-programs extend ASP with arbitrary external sources (which are realized in C++ or Python) by the use of so-called *external atoms*. Intuitively, the logic program sends information, given by constants and/or predicate extensions, to the external source, which returns output values that are imported into the program. For instance, the external atom $\&synonym[metro](X)$ might be used to find the synonyms X of $metro$, e.g. $subway$ and $tube$. Notably, external atoms can be nonmonotonic, introduce new values which are not part of the program (*value invention*), and can be used in recursive rules. The generality of external sources is in contrast to previous and dedicated formalisms such as DL-programs (Eiter et al. 2004) or constraint ASP (Ostrowski and Schaub 2012), which integrate ASP only with a concrete other formalism. HEX-programs subsume these other formalisms and also well-known ASP extensions such as aggregates.

However, expressiveness of a formalism alone it not sufficient. Instead, also an efficient and convenient implementation is needed to attract users; recall that also the success of ASP

∗ This research has been supported by the Austrian Science Fund (FWF) project P27730.

depends considerably on expressive, efficient and easy-to-use systems like CLASP (Gebser et al. 2011)[1], DLV (Leone et al. 2006)[2], and SMODELS (Simons et al. 2002)[3]. The HEX-semantics was implemented in the DLVHEX system (Eiter et al. 2006a)[4] on top of GRINGO and CLASP (Gebser et al. 2011). The system celebrates its 10th anniversary this year and was released in version 2.5.0 earlier this year. While early versions were mainly intended to be used for experimental purposes, only in the last three years much effort was spent on turning DLVHEX into a system for KR tasks which can conveniently be used beyond experimental purposes. To this end, we have overcome former limitations of the system which prevented its application in practice, including former efficiency problems, restrictions of the language, and technical limitations on the system side.

In this paper, we report about this progress. While some (but not all) of the enhancements discussed in the following have already been presented in dedicated works, this was from an algorithmic perspective and with focus on specific subproblems which occur when evaluating a HEX-program. In contrast, this paper provides a bird's eye view of the system from the user's perspective, which is missing so far.

After briefly recalling HEX-programs in Section 2, we present the novelties compared to earlier versions of the system. We group the enhancements in two main sections:

- Section 3 presents enhancements based on the **exploitation of known properties of external sources** such as monotonicity or functionality. We first discuss the types of properties supported by the system and show how they are specified (Section 3.1). Afterwards, we give an overview about how they are used within the system. To this end, we present the two main features based on them, namely *scalability boosts* by advanced learning techniques (Section 3.2) and *language flexibility* due to reduced syntactic limitations (Section 3.3).
- Section 4 presents recent extensions towards **usability and system features**. This includes a novel convenient programming interface for providers of external sources (Section 4.1), the integration of support for popular ASP extensions and interoperability (Section 4.2), and a new dissemination strategy which respects previous user feedback.

Afterwards, we give an overview about existing applications based on HEX-programs in Section 5 and discuss how they can benefit from the system improvements. We conclude in Section 6.

2 HEX-Programs

HEX-programs (Eiter et al. 2005) are a generalization of (disjunctive) extended logic programs under the answer set semantics (Gelfond and Lifschitz 1991) with external atoms. Besides *ordinary atoms* of the form $p(\mathbf{t})$, where p is a predicate and $\mathbf{t} = t_1, \ldots, t_\ell$ is a list of terms (such as strings, integers, symbolic constants, nested function terms), rule bodies may also contain *external atoms* of the form $\&g[\mathbf{X}](\mathbf{Y})$, where $\&g$ is an external predicate, $\mathbf{X} = X_1, \ldots, X_l$ and each X_i is an *input parameter* (which can be either a constant or variable term, or a predicate), and $\mathbf{Y} = Y_1, \ldots, Y_k$ and each Y_i is an *output term*.

Syntax. A HEX-*program* (or *program*) consists of rules r of form

$$a_1 \vee \cdots \vee a_h \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n \ , \tag{1}$$

[1] http://potassco.sourceforge.net
[2] http://www.dlvsystem.com
[3] http://www.tcs.hut.fi/Software/smodels
[4] http://www.kr.tuwien.ac.at/research/systems/dlvhex

where each a_i is an (ordinary) atom and each b_j is either an ordinary atom or an external atom, and $h + n > 0$; for such a rule r let $B(r) = \{b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n\}$ denote its *body*.

Semantics. An *assignment* $\mathbf{A}$ is a consistent set of literals of form $\mathbf{T}a$ or $\mathbf{F}a$, where a is an atom which is said to be *true* in $\mathbf{A}$ if $\mathbf{T}a \in \mathbf{A}$, *false* if $\mathbf{F}a \in \mathbf{A}$, and *undefined* otherwise. We say that $\mathbf{A}$ is *complete* over a program Π if for all atoms a in Π we have either $\mathbf{T}a \in \mathbf{A}$ or $\mathbf{F}a \in \mathbf{A}$.

The semantics of a HEX-program Π is defined via its grounding $grnd(\Pi)$ over a Herbrand universe of constants $\mathcal{C}$ as usual, where $\mathcal{C}$ can contain constants which are not in the program and might even be infinite. The value of a ground external atom $\&g[\mathbf{p}](\mathbf{c})$ wrt. an assignment $\mathbf{A}$ is given by the value $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c})$ of a decidable $1{+}k{+}l$-ary three-valued *oracle function* $f_{\&g}$, where k and l are the lengths of $\mathbf{p}$ and $\mathbf{c}$, respectively[5]. The oracle function evaluates to true, false or unknown ($\mathbf{T}$, $\mathbf{F}$ or $\mathbf{U}$), where we assume that (i) it evaluates to true or false if $\mathbf{A}$ is complete over Π, and (ii) we have $f_{\&g}(\mathbf{A}', \mathbf{p}, \mathbf{c}) = f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c})$ whenever $\mathbf{A}' \supseteq \mathbf{A}$ and $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) \in \{\mathbf{T}, \mathbf{F}\}$, i.e., evaluations to true or false do not change when the assignment becomes more complete; we call this property *knowledge-monotonicity*. In practice, one often abstracts from the Boolean view and sees an external predicate with input list $\&g[\mathbf{p}]$ as *computation* of output values $\mathbf{c}$, i.e., determining all values $\mathbf{c}$ such that $f_{\&g}(\mathbf{A}, \mathbf{p}, \mathbf{c}) = \mathbf{T}$.

We note that the definition of the oracle function for assignments which are not complete is only for efficiency improvement, as explained in detail in Section 3.2. For user's convenience and for backwards compatibility, it is also possible to use a two-valued (Boolean) oracle function which is only defined over complete assignments. It is then implicitly assumed to evaluate to unknown for all assignments which are not complete.

For (a set of) ground literals, rules, programs, etc., say O, satisfaction wrt. a complete assignment $\mathbf{A}$ extends naturally from ASP to HEX-programs, by taking external atoms into account. Satisfaction of O under $\mathbf{A}$ is denoted by $\mathbf{A} \models O$. In this case we say that $\mathbf{A}$ is a *model* of O.

An *answer set* of a HEX-program Π is a model $\mathbf{A}$ of the *FLP-reduct*[6] $f\Pi^{\mathbf{A}}$ of Π wrt. $\mathbf{A}$, given by $f\Pi^{\mathbf{A}} = \{r \in grnd(\Pi) \mid \mathbf{A} \models B(r)\}$ (Faber et al. 2011), which is subset-minimal, i.e., there exists no model $\mathbf{A}'$ of $f\Pi^{\mathbf{A}}$ s.t. $\{\mathbf{T}a \in \mathbf{A}'\} \subsetneq \{\mathbf{T}a \in \mathbf{A}\}$.

Technically, external atoms are realized as *plugins* of the reasoner using a *programming interface*. To this end, the provider of an external source basically implements its oracle function.

Example 1 *Consider the program*

$$\Pi = \left\{ \begin{array}{l} r_1 \colon start(s). \\ r_2 \colon scc(X) \leftarrow start(X). \quad r_3 \colon scc(Y) \leftarrow scc(X), \&edge[X](Y). \end{array} \right\}$$

where r_1 selects a node s from an externally defined (finite) graph, and r_2 and r_3 recursively compute the strongly connected component of s. To this end, the external atom $\&edge[X](Y)$ is used, which is true if Y is directly reachable from X (and false otherwise).

The implementation of $\&edge[X](Y)$ may look as follows (API details follow in Section 4.1):

```
def edge(x):
  graph=((1,2),(1,3),(2,3))        # simplified implementation; real ones may read a DOT file
  for edge in graph:               # search for outgoing edges of node x
    if edge[0]==x.intValue():
      dlvhex.output((edge[1],))    # output edge target
```

[5] In previous works, oracle functions were two-valued; we come back to this extension (Eiter et al. 2016) in Section 3.

[6] The FLP-reduct is equivalent to the traditional reduct for ordinary logic programs (Gelfond and Lifschitz 1991), but more attractive for extensions such as aggregates or external atoms.

3 Exploiting External Source Properties

External sources were seen as black boxes in earlier versions of DLVHEX. It was assumed that the system does not have any information about them, except that there is an oracle function which decides satisfaction of an external atom under a complete assignment. As a consequence, the room for optimizations in the algorithms was limited because the value of an external atom under one assignment did not allow for drawing any conclusions about its behavior under other assignments.

However, in many practical applications the provider of an external source and/or the HEX-programmer have additional knowledge about the behavior of the source, for instance, that the source is monotonic, functional, has a limited domain, returns only elements which are smaller than the input (according to some ordering), etc. Knowing such properties allows for implementing more specialized algorithms which are tailored to the particular external sources used in a program. We therefore identified a set of *properties* that external sources might have, and allow the user to specify the ones which are fulfilled by a concrete external source.

Example 2 *Suppose* $\&tail[X](Y)$ *is true whenever* Y *is the string which results from string* X *if the first character is dropped. Then the output is always smaller than the input wrt. string length.*

The system exploits these properties automatically, mainly for two purposes: in the *learning algorithms for scalability enhancements* and in the *grounding component for more flexibility of the language* due to reduced syntactic limitations; we discuss these two aspects in more detail in Sections 3.2 and 3.3, respectively. In addition, there are several other system components which exploit the properties to further speed up the evaluation, such as skipping various checks if their result is definite due to known behavior of external sources, partitioning a reasoning task into smaller independent tasks, avoiding unnecessary evaluations of external atoms, and drawing deterministic conclusions rather than guessing.

However, as this paper presents the system from user's perspective, we focus on *which* properties can be specified, *how* the user can do that, and give a rough idea of how the system makes use of this information, but we refrain from discussing the involved algorithms in detail. This is in line with the goal of these properties: the user can benefit from the advantages when specifying them, but without the need to care about how the system is going to exploit this information. Instead, the user can generally expect that the more information is available to the system, the more efficient evaluation will be; if the added information does not yield a speedup, it does at least no harm.[7] Some of the properties, such as monotonicity, do even lead to a drop of complexity from Σ_2^P to NP for answer set existence checking over ground disjunction-free programs, provided that external sources are polynomial (Faber et al. 2011).

Furthermore, properties also serve as *assertions*: if the reasoner observes a behavior of external sources which contradicts the declared properties, appropriate error messages are printed.

3.1 Specifying Properties

The specification of properties is supported in two ways. The first option is to declare them as part of the external source implementation via the *external source interface*. The second option is to specify them as part of the HEX-program using so-called *property tags*.

[7] The only property related to potential performance decrease is provision of a *three-valued semantics* as additional calls of the external source are sometimes counterproductive (Eiter et al. 2016). However, even then the property itself does not harm since it is only exploited by certain (non-default) evaluation heuristics selected via command-line options.

Specification via the External Source Interface. Properties are mostly specified via the (C++ or Python) programming interface for external sources. To this end, the procedural code which implements external atoms calls specific *setter methods* provided by the programming interface to inform the system that the source has certain properties.

Example 3 *The implementation of a hash function &md5$[X](Y)$ which computes for a string X its MD5 hash value Y might call* `prop.setFunctionality(true)` *to let* DLVHEX *know that for each X there is exactly one Y. This allows the system, for instance, to conclude that &md5$[x](y_2)$ is false without evaluating the external source, if it has already found a value $y_1 \neq y_2$ such that &md5$[x](y_1)$ is true.*

If a property is declared in this way, the external source is meant to *always* provide a certain behavior, independent of its usage in a certain HEX-program, like in case of the computation of a hash value. Another example is *&diff*$[p, q](X)$, which computes all values X which are in the extension of p but not in that of q wrt. assignment $\mathbf{A}$ (formally, these are all values x such that $f_{\&diff}(\mathbf{A}, p, q, x) = \mathbf{T}$). This external atom it is always monotone/antimonotone in the first/second parameter, which can be specified by calling `prop.addMonotonicInputPredicate(0)` and `prop.addAntimonotonicInputPredicate(1)` (cf. Example 8).

Specification via Property Tags. However, it might also be the case that only a specific usage of an external source in a concrete program has a property. Then the implementer of the external source cannot declare it yet; instead, only the implementer of the HEX-program has sufficient knowledge and can declare the property as part of an external atom in the program.

Example 4 *Suppose &greaterThan$[p, 10]()$ checks if the sum of integer values c s.t. $p(c)$ is true is greater than* 10*. It is not monotone in general if negative integers are allowed, but it is monotone if a program uses only positive integers. While the provider of the external source cannot assert the property, the user of the external source in a concrete program, who knows the context, can.*

To this end, the HEX language and implementation were extended such that external atoms can be followed by *property tags* of form $\langle list\ of\ properties \rangle$, where the list of properties is comma-separated. Each property is then a whitespace-separated list of constants, consisting of a *property type* (first element in the list), and a number of *property parameters* (remaining elements in the list), whose number depends on the property type and may also have default values. For example, *&diff*$[p, q](X)\langle monotonic\ p, antimonotonic\ q \rangle$ specifies two properties which declare that the external atom is monotonic in p and antimonotonic in q wrt. their extension in the input assignment. Here, the first property $monotonic\ p$ uses the property type *monotonic* and the property parameter p, while the second property $antimonotonic\ q$ uses the property type *antimonotonic* and the property parameter q. Another example is *&greaterThan*$[p, 10]()\langle monotonic \rangle$, which declares that the external source is monotonic in all parameters (because it is monotonic in p and it is trivially monotonic in constant input parameters because they are independent of the input assignment); the property type is *monotonic* and no property parameters are explicitly specified, which indicates by default that the source is monotonic in all inputs. Properties declared by tags are understood to hold *in addition* to those declared via the external source interface (stating conflicting properties is not possible with the currently available ones).

Supported properties. The following list gives an overview about the currently available properties and how to specify them if the property tag language is used (but all of them can be specified both via the external source interface or in property tags). Each property is explained with an example in order to show the property type and the expected property parameters.

- **Functionality**: $\&add[X,Y](Z)\langle functional\rangle$
  The external atom adds integers X and Y and is true for their sum Z. The source provides exactly one output value for a given input. There are no property parameters.
- **Monotonicity in a parameter**: $\&diff[p,q](X)\langle monotonic\ p\rangle$
  The external atom computes the difference of the extensions of p and q. The source is monotonic in predicate parameter p (i.e., if the extension of p increases, the output does not shrink), as indicated by the property parameter.
- **Global monotonicity**: $\&union[p,q](X)\langle monotonic\rangle$
  The source computes the set union of the extensions of p and q. It is monotonic in all parameters (indicated by the default value of the missing property parameter).
- **Antimonotonicity in a parameter**: $\&diff[p,q](X)\langle antimonotonic\ q\rangle$
  The source is antimonotonic in predicate parameter q (i.e., if the extension of q shrinks, the output does not shrink).
- **Global antimonotonicity**: $\&complement[p](X)\langle antimonotonic\rangle$
  The source computes the complement of the extension of p wrt. a fixed domain. It is antimonotonic in all parameters.
- **Linearity on atoms**: $\&union[p,q](X)\langle atomlevellinear\rangle$
  We have domain independence on the level of atoms, i.e., the source can be separately evaluated for each input atom s.t. the final result is the union of the results of all evaluations. For instance, the evaluation under assignment $\mathbf{A} = \{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{T}q(c)\}$, which yields $\{a, b, c\}$, can be split up into three evaluations under $\mathbf{A}_1 = \{\mathbf{T}p(a)\}$, $\mathbf{A}_2 = \{\mathbf{T}p(b)\}$ and $\mathbf{A}_3 = \{\mathbf{T}q(c)\}$, which yield $\{a\}$, $\{b\}$ and $\{c\}$, respectively, and their union the result of the evaluation under $\mathbf{A}$. There are no property parameters.
- **Linearity on tuples**: $\&diff[p,q](X)\langle tuplelevellinear\rangle$
  We have domain independence on the level of tuples in the extensions of predicate input parameters, i.e., the source can be separately evaluated for each pair of atoms $p(\mathbf{c})$ and $q(\mathbf{c})$ for all vectors of terms $\mathbf{c}$ s.t. the final result is the union of the results of all evaluations. For instance, the evaluation under $\mathbf{A} = \{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b)\}$, which yields $\{a\}$, can be split up into two evaluations under $\mathbf{A}_1 = \{\mathbf{T}p(a), \mathbf{F}q(a)\}$ and $\mathbf{A}_2 = \{\mathbf{T}p(b), \mathbf{T}q(b)\}$, which yield $\{a\}$ and $\emptyset$, respectively, and their union in the result of the evaluation under $\mathbf{A}$. However, it would not be correct to split $\mathbf{A}_2$ further up into $\mathbf{A}_{2.1} = \{\mathbf{T}p(b)\}$ and $\mathbf{A}_{2.2} = \{\mathbf{T}q(b)\}$ as they would yield the results $\{b\}$ and $\emptyset$, which would put b into the final result, which differs from the evaluation under $\mathbf{A}$. There are no property parameters.
- **Finite domain**: $\&edges[graph.dot](X,Y)\langle finitedomain\ 0, finitedomain\ 1\rangle$
  Imports the edges of a predefined graph. Both output values can have only finitely many different values. To this end, we specify two properties with type *finitedomain* with property parameters that identify the output terms X and Y by index (0 and 1, respectively).
- **Finite domain wrt. the input**: $\&diff[p,q](X)\langle relativefinitedomain\ 0\ 0\rangle$
  Only constants which already appear in the 0-th input (indicated by the first property parameter 0; points in this case to the predicate p) may occur as first output term (indicated by the second property parameter 0). Informally, the difference between sets represented by predicates p and q can only contain elements which appear in the set represented by p.
- **Finite fiber**: $\&sqrt[X](Z)\langle finitefiber\rangle$
  The source computes the square root of X. Each element in the output is only produced by finitely many different inputs (in this case, in fact, only by a single input value). There are no property parameters.

- **Well-ordering wrt. string lengths**: $\&tail[X](Z)\langle wellorderingstrlen\ 0\ 0\rangle$
  The source drops the first character of string X and returns the result in Z. The 0-th output (indicated by the second property parameter 0) is no longer than the longest string in the 0-th input (indicated by the first property parameter 0).
- **General well-ordering**: $\&decrement[X](Z)\langle wellordering\ 0\ 0\rangle$
  The external atom decrements a given integer. There is an ordering of all constants such that the 0-th output (second parameter) is no greater than the 0-th input (first parameter) wrt. this ordering.
- **Three-valued semantics**: $\&g[\mathbf{X}](\mathbf{Y})\langle providespartialanswer\rangle$
  The external source can be evaluated under partial assignments, i.e., it can handle assignments which do not define all atoms, but may evaluate to *undefined* ($\mathbf{U}$) in this case (can be used with any external source if implemented).

Note that properties are only useful if they are exploited by at least one solving technique or algorithm implemented in the reasoner. It is therefore *not* intended that typical users introduce custom properties, but only tag external atoms with existing ones from the above list. However, for advanced users who contribute to or customize the reasoner itself, the framework supports easy extension of the parser and data structures. Exploiting such a new property in the algorithms might be more sophisticated depending on the particular property and the envisaged goal.

3.2 Scalability Boost

Traditionally, ground HEX-programs have been evaluated by replacing each external atom $\&e[\mathbf{p}](\mathbf{c})$ by an ordinary atom $e_{\&e[\mathbf{p}]}(\mathbf{c})$ and introducing a rule $e_{\&e[\mathbf{p}]}(\mathbf{c}) \vee ne_{\&e[\mathbf{p}]}(\mathbf{c}) \leftarrow$ to guess its truth value; the resulting program is evaluated by an ordinary ASP solver to produce model candidates. Each candidate $\mathbf{A}$ is subsequently checked by testing (i) if the external atom guesses are correct, i.e., if $\mathbf{A} \models e_{\&e[\mathbf{p}]}(\mathbf{c})$ iff $\mathbf{A} \models \&e[\mathbf{p}](\mathbf{c})$ for all external atoms $\&e[\mathbf{p}](\mathbf{c})$, and (ii) if assignment $\mathbf{A}$ is a *subset-minimal* model of $f\Pi^{\mathbf{A}}$. If both conditions are satisfied, an answer set has been found. However, this approach did not scale well because there are exponentially many independent guesses in the number of external atoms in the ground program.

Basic approach. To overcome the problem, novel evaluation algorithms based on *conflict-driven* techniques have been introduced (Eiter et al. 2012). As in ordinary ASP solving, the input program is translated to a set of *nogoods*, i.e., a set of literals which must not be true at the same time. Given this representation, techniques from SAT solving are applied to find an assignment which satisfies all nogoods (Gebser et al. 2012). Notably, as the encoding as a set of nogoods is of exponential size due to *loop nogoods* which avoid cyclic justifications of atoms, those parts are generated only on-the-fly. Moreover, additional nogoods are learned from conflict situations, i.e., violated nogoods which cause the solver to backtrack; this is called *conflict-driven nogood learning*.

The extension of this algorithm towards the integration of external sources into the learning component works as follows. Whenever an external atom $\&e[\mathbf{p}](\mathbf{c})$ is evaluated under an assignment $\mathbf{A}$ in the checking part (i), the actual truth value under the assignment becomes evident. Then, regardless of whether the guessed value was correct or not, one can add a nogood which represents that $e_{\&e[\mathbf{p}]}(\mathbf{c})$ must be true under $\mathbf{A}$ if $\mathbf{A} \models \&e[\mathbf{p}](\mathbf{c})$ or that $e_{\&e[\mathbf{p}]}(\mathbf{c})$ must be false under $\mathbf{A}$ if $\mathbf{A} \not\models \&e[\mathbf{p}](\mathbf{c})$. If the guess was incorrect, the newly learned nogood will trigger backtracking, if the guess was correct, the learned nogood will prevent future wrong guesses.

Example 5 *As above, suppose $\&diff[p, q](X)$ computes the set difference between the extensions*

of predicates p and q and that it is evaluated under $\mathbf{A} = \{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b)\}$ with Herbrand universe $\mathcal{C} = \{a, b\}$. Then it can be learned that $\mathbf{A} \models e_{\&e[p,q]}(a)$ by adding the nogood $\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b), \mathbf{F}e_{\&e[p,q]}(a)\}$, i.e., whenever $p(a), p(b), q(b)$ are true and $q(a)$ is false, then $\&e[p, q](a)$ must not be false. Conversely, one can learn that $\mathbf{A} \not\models \&e[p, q](b)$ by adding nogood $\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b), \mathbf{T}e_{\&e[p,q]}(b)\}$.

Experimental results show a significant, up to exponential speedup (Eiter et al. 2014). This is explained by the exclusion of up to exponentially many guesses by the learned nogoods.

Exploiting external source properties. The technique was refined by exploiting additional knowledge about external sources in order to keep the learned nogoods small. In the previous example, atoms $p(a)$ and $q(a)$ in the assignment are in fact irrelevant when deciding whether $\&e[p, q](b)$ is true because constants a and b are independent (similarly for $p(b)$ and $q(b)$ when deciding $\&e[p, q](a)$). If this information is available to the system, it can be exploited to shrink nogoods to the relevant part such that the search space is pruned more effectively.

One way to gain the required information is to make use of the properties introduced in Section 3.1. In particular, the independence of a and b in the previous example can be derived from the property 'linearity on tuples'. Then the nogood $\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b), \mathbf{F}e_{\&e[p,q]}(a)\}$ can be reduced to $\{\mathbf{T}p(a), \mathbf{F}q(a), \mathbf{F}e_{\&e[p,q]}(a)\}$ and the nogood $\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b), \mathbf{T}e_{\&e[p,q]}(b)\}$ to $\{\mathbf{T}p(b), \mathbf{T}q(b), \mathbf{T}e_{\&e[p,q]}(b)\}$. If monotonicity in p is known in addition, then nogood $\{\mathbf{T}p(b), \mathbf{T}q(b), \mathbf{T}e_{\&e[p,q]}(b)\}$ can be further simplified to $\{\mathbf{T}q(b), \mathbf{T}e_{\&e[p,q]}(b)\}$ by dropping $\mathbf{T}q(a)$ because $\&e[p, q](b)$ will remain false even if $q(a)$ becomes false.

Exploiting three-valued oracle functions. Alternatively or in addition to external source properties, also three-valued oracle functions (cf. Section 2) can be exploited for shrinking learned nogoods to the essential part (Eiter et al. 2016). If the truth value is already known and will not change when the assignment becomes more complete, then the set of yet unassigned atoms is irrelevant for the output of the external source. This is exploited for nogood minimization as follows. Whenever a nogood is learned, the system iteratively tries to remove one of the input atoms and evaluate again in order to check if the truth value is still defined. If this is the case, the according atom is not necessary and can be removed from the nogood.

For instance, a proper implementation of a three-valued oracle function in the previous example allows for reducing $\{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{F}q(a), \mathbf{T}q(b), \mathbf{T}e_{\&e[p,q]}(b)\}$ to $\{\mathbf{T}q(b), \mathbf{T}e_{\&e[p,q]}(b)\}$ because whenever $\mathbf{T}q(b)$ is in the assignment, it is already definite that $\&diff[p, q](b)$ is false.

Discussion and Extensions. Whether to exploit external source properties, three-valued oracle functions, or both, depends largely on the use case. Depending on the type of external source to be realized, the implementation of a three-valued oracle function might be more challenging than of a Boolean one (implementing an algorithm which decides over partial assignments is in general more difficult than if all information is known). However, it allows for exploiting application-specific knowledge in an optimal way (Eiter et al. 2016). In contrast, tagging external sources with properties from a list is easy and can still lead to good efficiency.

3.3 Language Flexibility

External atoms may introduce constants which do not appear in the program (*value invention*). Obviously, this can in general lead to programs which do not have a finite grounding that has the same answer sets as the original program (which are defined via the full, possibly infinite grounding $grnd(\Pi)$). Since this inhibits grounding in general, it is crucial to identify classes of

programs for which the existence of such a finite grounding is guaranteed; we call this property *finite groundability*. Traditionally, *strong safety* was used, which basically forbids value invention by recursive external atoms (i.e., external atoms whose input possibly depends on its own output wrt. the predicate dependency graph, for a formal definition cf. Eiter et al. (2006b)). If only non-recursive external atoms introduce new values, termination is guaranteed. However, it turns out that this is only a sufficient but not a necessary criterion, i.e., strong safety is overly restrictive.

Example 6 *The program* Π *from Example 1 is* not *strongly safe because* $\&edge[X](Y)$ *is recursive (output* Y *may be input to the same external atom by another application of* r_3*) but may introduce values for* Y *which do not appear in* Π*. However, if one knows that the graph is finite, one can conclude that the recursive introduction of new values will end at some point.*

In the example, the criterion may be circumvented by importing the full domain a priori and adding *domain predicates*, i.e., adding $node(Y)$ to the body of r_3 and another rule $node(X) \leftarrow \&node[](X)$ to import all nodes. Then $\&edge[X](Y)$ does no longer invent values because all possible values for Y are determined in a non-recursive fashion using $\&node[](X)$. However, this comes at the price of importing the whole graph although only a small set of nodes might be in the strongly connected component of s.

Therefore, new safety criteria have been introduced which allow for exploiting both syntactic and semantic conditions to derive finite groundability, where the latter are based on external source properties as introduced in Section 3.1. So-called *liberally safe* HEX-programs are guaranteed to have a finite grounding which can be computed using a novel algorithm (Eiter et al. 2016).

Example 7 *Let* $\&tail[X](Y)$ *drop the first character of string* X *and return it as* Y*. Then* Y *is no longer than* X *and – even if used recursively – it is guaranteed that it can generate only finitely many strings because there are only finitely many strings with a length up to the one of* X*.*

In addition to the declaration of predefined properties, the generic framework is also extensible such that custom knowledge about external sources can be exploited. To this end, providers may implement *safety plugins*, which are integrated into the safety check. The safety check itself is fast (at most quadratic in the size of the non-ground program).

The system combines the available information, given by syntactic conditions, specified semantic properties and safety plugins in order to check safety of the program. This does not only allow for writing programs with fewer syntactic restrictions, but the implementation of some applications may be possible in the first place. For instance, in *route planning applications*, importing the whole map material a priori is practically impossible due to the large amount of data, while a selective import using liberal safety makes the application possible (Eiter et al. 2016).

In case a program is not safe, the system prints hints such as the rule and the variable for which finiteness during instantiation could not be proven. This information is intended to guide the user when providing more information in order to make the program safe, e.g., by adding properties from Section 3.1 which constrain the values of this variable further. Alternatively, a command-line option allows to disable the safety check altogether, in which case there is no guarantee that the reasoner terminates (putting this burden on the user).

4 Usability and System Features

In this section we present recent work on the system side to improve the user's convenience. We start with general remarks on the DLVHEX software and its dissemination. DLVHEX was previously

only available in source format (released under GNU LGPL) and only for Linux platforms. This deployment method turned out to be inconvenient for ASP programmers who want to use the system as is without custom modifications, thus we now provide pre-built binaries for all major platforms (Linux-based, OS X and Windows) in addition. We further created an online demo of the system under http://www.kr.tuwien.ac.at/research/systems/dlvhex/demo.php which allows for evaluating HEX-programs directly in the browser (the user may specify both the logic program and custom Python-implemented external atoms in two input fields). The demo comes with a small set of examples to demonstrate the main features of the KR formalism. We further provide a manual to support new users of the system (Eiter et al. 2015).

Next, the following two subsections give an overview of the new Python programming interface and interoperability of the system.

4.1 Python Programming Interface

With earlier versions of the system, users who wanted to integrate custom external sources had to write plugins in C++. While this was natural as the reasoner itself is implemented in C++, it was cumbersome and introduced development overhead even for experienced developers. This is because multiple configuration, source and header files need to be created even when realizing only a small and simple plugin. Also the compilation and linking overhead during development and debugging was considered inconvenient.

As a user-friendly alternative, DLVHEX 2.5.0 introduces a plugin API for Python-implemented external sources. A plugin consists of a single file (unless the user explicitly wants to use multiple files), which imports a dedicated `dlvhex` package and specifies a single method for each external atom. Thanks to higher-level features of Python and modern packages, this usually results in much shorter and simpler code than with C++-implemented plugins. A central `register` method exports the available external atoms and (optionally) their properties from Section 3.1 to DLVHEX.

Example 8 *The following snippet implements &diff*$[p,q](X)$ *for computing the values* X *which are in the extension of* p *but not in that of* q*. It is monotonic in* p *and antimonotonic in* q*.*

```
import dlvhex

def diff(p,q):
  for x in dlvhex.getTrueInputAtoms():          # for all true input atoms
    if x.tuple()[0] == p:                       # is it of form p(c)?
      if dlvhex.isFalse(dlvhex.storeAtom(       # is the corresponding q(c) false?
                        (q, x.tuple()[1]))):
        dlvhex.output((x.tuple()[1], ));        # then c is in the output

def register():
  prop = dlvhex.ExtSourceProperties()           # inform dlvhex about
  prop.addMonotonicInputPredicate(0)            # monotonicity/antimonotonicity
  prop.addAntimonotonicInputPredicate(1)        # in the first/second parameter
  dlvhex.addAtom("diff", (dlvhex.PREDICATE, dlvhex.PREDICATE), 1, prop)
```

On the command-line, the call `dlvhex2 --python-plugin=plugin.py prog.hex` loads the external atoms defined in `plugin.py` and then evaluates HEX-program `prog.hex`.

In the system, the Python programming interface is realized as a wrapper of the generic C++ interface as shown in Figure 1, where arcs model both control and data flow. That is, the Python interface uses only the C++ interface but does not communicate with the core reasoning components otherwise. This turns the Python interface in fact into a special C++ plugin. The performance gap between C++ and Python plugins is normally negligible (the update of the Python data structures it in the worst case linear in the number of input atoms), unless the plugin

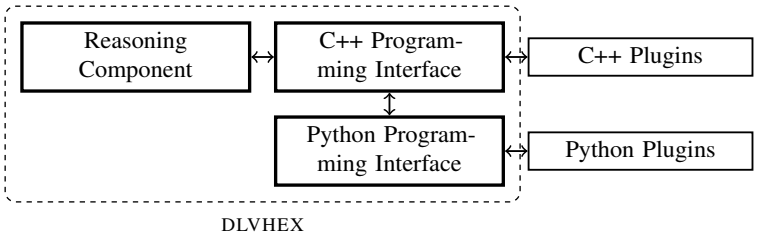

Fig. 1: Architecture of the Python Programming Interface

is itself computationally expensive. Wrappers for other languages can be added similarly and can also be implemented externally, i.e., they do not necessarily need to be part of the DLVHEX solver.

For a complete API description we refer to http://www.kr.tuwien.ac.at/research/systems/dlvhex/.

4.2 ASP-Core-2 Standard, Extensions and Interoperability

In the course of the organization of the fourth ASP competition, the input language of ASP systems was standardized in the *ASP-Core-2 input language format* (Calimeri et al. 2013). The DLVHEX system in its current version supports all features defined in the standard, including function symbols, choice rules, conditional literals, aggregates, and weak constraints. The supported language is therefore a strict superset of the standard.

The system further supports input and output in CSV format to improve interoperability with other systems such as Unix commands or spreadsheet applications. That is, facts may be read from the lines of a CSV file, where the different values are mapped to the arguments of a predicate. After the computation, the extension of a specified predicate may be written in CSV format to allow a seamless further processing by other applications. For instance, consider `salary.csv`:

```
joe,smith,2000
sue,johnson,2200
```

It can be read as facts $emp(1, joe, smith, 2000)$ and $emp(2, sue, johnson, 2200)$ (where the first element is the original line number if relevant) using the DLVHEX command-line option `--csvinput=emp,salary.csv`. Conversely, results can be output in CSV format.

5 Applications

We now discuss some applications which were realized on top of HEX-programs. In this paper, we focus on applications whose purpose was *not* to demonstrate HEX-programs or to evaluate the reasoner. Instead, the following applications are motivated by real needs and are interesting by themselves, while HEX-programs were merely a means for their realization. This witnesses that HEX-programs and DLVHEX can be fruitfully applied for real-world problems.

We discuss the effects of the described system improvements on the applications. However, since this paper gives an overview of the system and not all of the presented improvements are related to efficiency, not all of the following applications are suited as performance benchmarks. For an extensive empirical evaluation focused on efficiency we refer to Eiter et al. (2014) and Eiter et al. (2016).

Hybrid planning. The application comes from the robotics domain and consists of high-level planning and low-level feasibility checking (Erdem et al. 2016). High-level plans are sequences

of actions towards a goal, while low-level constraints (such as stability issues of robots or intersections of routes) exclude some of the sequences. Thus, not all such plans which are possible from high-level perspective are actually executable. The separation of the two levels is motivated by the observation that the full integration of low-level constraints into the model for high-level planning might blow up the encoding (while it might be feasible in other cases). An implementation of hybrid planning on top of HEX-programs was presented, where external atoms are used to perform low-level feasibility checking of high-level plans generated in the program.
Effects of improvements: The application uses hand-crafted *custom learning functions* which add custom nogoods during evaluation to improve efficiency, cf. Erdem et al. (2016). With the new DLVHEX version, three-valued oracle functions can be used instead, which allow for an easier realization of a similar learning behavior. Furthermore, the property *tuplelevellinear* can be exploited whenever feasibility checks can be split into multiple independent checks (e.g. of independent robots), and *relativefinitedomain* can be exploited for external atoms used for sensing objects (only objects which appear in the description of the world can be sensed).

Route planning. The combination of route planning with side constraints was realized on top of HEX (Eiter et al. 2016). An example is planning a tour through multiple locations, where the possibility to get refreshments should be included if the tour is longer than a limit.
Effects of improvements: Since the traditional criterion of strong safety disallows recursive value invention, previous system versions must import the whole map a priori. As this is infeasible for real-world data, the application can in fact only be realized on top of HEX by exploiting the improvements from Section 3.3. To this end, *finiteness* of the map used with liberal safety allows for importing only relevant parts of the map and solving the problem efficiently (Eiter et al. 2016).

Multi-context Systems. Multi-context systems are a framework for integrating heterogeneous knowledge-bases, called *contexts*, which are abstractly identified by sets of belief sets (Brewka and Eiter 2007). Their integration works via dedicated *bridge rules* which derive information in one context based on atoms in other contexts. The whole system may become inconsistent although the individual contexts are all consistent. A typical reasoning task is then *inconsistency analysis*, i.e., the computation of an inconsistency explanation (Eiter et al. 2010), which was realized on top of HEX-programs (Bögl et al. 2010). The main idea is to realize contexts as external sources. Then a HEX-program can access all contexts, compute candidate explanations, and check them against all contexts. Experimental results, which demonstrate effectiveness of the learning techniques from Section 3.2, can be found in the work by Eiter et al. (2014).
Effects of improvements: Already plain learning (general part of Section 3) is highly effective, cf. (Eiter et al. 2014). The external atoms are *functional*, which does however, since their output is 0-ary, not lead to additional performance improvements, but does at least not harm.

Complaint management. Citizens may raise complaints about issues such as noise or traffic jams as part of e-government. A system was realized on top of DLVHEX, which ranks complaints by their severity, such that priorities can be assigned (Zirtiloglu and Yolum 2008). While ontologies capture parts of the application, the authors combine them with HEX-programs due to the inherent support for nonmonotonic reasoning. This is motivated by the dynamic behavior of complaint management systems, which might need to adopt the ranking if new complaints are added.
Effects of improvements: The encoding makes use of recursive rules over external atoms, thus the evaluation involves nondeterministic guessing. However, as the external atoms use only constant input, they are independent of the assignment are thus trivially *monotonic* and *antimonotonic*. In this case, the techniques from Section 3.2 assign the correct truth value permanently after the first evaluation and thus this application is expected to benefit significantly from the improvements.

AI in computer games. *Angry-HEX* is an AI agent for the computer game *Angry Birds*[8] (Ianni et al. 2016) and was developed since 2012 for participation in the *AIBirds competition*[9]; it was a finalist in 2015. The goal is to shoot birds with a slingshot at pigs located in buildings of wood, stone and ice blocks in order to destroy them. While the game is a strategy and skills game when playing manually, an AI agent can precisely compute the trajectory and the angle and speed in order to hit the desired target. Thus, the main issue is the selection of the best target.

The strategy employed by Angry-HEX is to select the target which maximizes the estimated damage to pigs (primary goal) and to other objects (secondary goal). This is encoded as a HEX-program which guesses possible targets, estimates the damage for each, and uses weak constraints for optimization. However, the estimation of the damage requires physics simulation for deciding, for instance, which objects will fall if others are destroyed. As such a simulation cannot easily be done with rules alone, external atoms are used to interface with a physics simulator. Hence, the low-level simulation is done in external atoms while the high-level strategy is rule-based. The idea of this two-level approach is similar to the hybrid planning domain.
Effects of improvements: The application mainly benefits from the improvements in Section 4. It uses new language features from the ASP-Core-2 standard such as optimization statements. Moreover, until now a significant amount of development time was spent on low-level coding for interfacing physics libraries. The new Python interface is expected to speed up the development of the agent. Finally, the availability of binaries is more important than for other applications since the application needs to be run in an environment provided by the organizers of the competition.

6 Conclusion

The DLVHEX system implements HEX-programs and was first released ten years ago (Eiter et al. 2006a). Over time, it was significantly extended with new algorithms, features, programming interfaces, and user's resources. While it served mainly as an experimental framework in the beginnings, its advancement towards practical applicability started only in the last three years. We now reached a stable state, where all extensions envisaged for this major release are implemented.

In this paper, we gave a summary of version 2.5.0 and the most important recent enhancements. While literature on theoretical aspects and algorithms is preexisting, this paper focuses on the practical aspects which are relevant when realizing an application on top of HEX. After receiving positive feedback from individual users, we believe that informing the users succinctly about the enhancements will push the use not only of the new features but also of the system altogether.

The improvements concern *exploitation of known properties of external sources* for novel efficient evaluation algorithms and more flexibility of the language, and *recent system extensions* for improved user's convenience; the latter include a Python programming interface, additional material and a new dissemination strategy. Real applications, which emerged independently of the research on HEX, but were realized on top of DLVHEX, confirm the practicability of the approach.

References

BÖGL, M., EITER, T., FINK, M., AND SCHÜLLER, P. 2010. The MCS-IE system for explaining inconsistency in multi-context systems. In *In JELIA 2010*. 356–359.

[8] https://www.angrybirds.com
[9] https://aibirds.org

BREWKA, G. AND EITER, T. 2007. Equilibria in Heterogeneous Nonmonotonic Multi-Context Systems. In *AAAI*. AAAI Press, 385–390.

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

CALIMERI, F., FABER, W., GEBSER, M., IANNI, G., ROLAND KAMINSKI, T. K., LEONE, N., RICCA, F., AND SCHAUB, T. 2013. ASP-Core-2 Input Language Format.

EITER, T., FINK, M., KRENNWALLNER, T., AND REDL, C. 2012. Conflict-driven ASP solving with external sources. *Theory and Practice of Logic Programming: Special Issue ICLP*.

EITER, T., FINK, M., KRENNWALLNER, T., AND REDL, C. 2016. Domain expansion for asp-programs with external sources. *Artif. Intell. 233*, 84–121.

EITER, T., FINK, M., KRENNWALLNER, T., REDL, C., AND SCHÜLLER, P. 2014. Efficient HEX-program evaluation based on unfounded sets. *Journal of Artificial Intelligence Research 49*, 269–321.

EITER, T., FINK, M., SCHÜLLER, P., AND WEINZIERL, A. 2010. Finding explanations of inconsistency in Multi-Context Systems. In *KR*. AAAI Press, 329–339.

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 2005*, 90–96.

EITER, T., IANNI, G., SCHINDLAUER, R., AND TOMPITS, H. 2006a. dlvhex: A Prover for Semantic-Web Reasoning under the Answer-Set Semantics. In *the ICLP'06 Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services (ALPSWS2006)*. CEUR WS, 33–39.

EITER, T., IANNI, G., SCHINDLAUER, R., AND TOMPITS, H. 2006b. Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning. In *ESWC 2006*, 273–287.

EITER, T., KAMINSKI, T., REDL, C., AND WEINZIERL, A. 2016. Exploiting partial assignments for efficient evaluation of answer set programs with external source access. In *IJCAI 2016*, To appear.

EITER, T., LUKASIEWICZ, T., SCHINDLAUER, R., AND TOMPITS, H. 2004. Combining Answer Set Programming with Description Logics for the Semantic Web. In *KR 2004*, D. Dubois, C. Welty, and M.-A. Williams, Eds. AAAI Press, 141–151.

EITER, T., MEHULJIC, M., REDL, C., AND SCHÜLLER, P. 2015. User guide: dlvhex 2.x. Tech. Rep. INFSYS RR-1843-15-05, Vienna University of Technology, Institute for Information Systems. September.

ERDEM, E., PATOGLU, V., AND SCHÜLLER, P. 2016. A Systematic Analysis of Levels of Integration between High-Level Task Planning and Low-Level Feasibility Checks. *AI Communications, IOS Press*.

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

GEBSER, M., KAUFMANN, B., KAMINSKI, R., OSTROWSKI, M., SCHAUB, T., AND SCHNEIDER, M. 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.

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

IANNI, G., CALIMERI, F., GERMANO, S., HUMENBERGER, A., REDL, C., STEPANOVA, D., TUCCI, A., AND WIMMER, A. 2016. Angry-HEX: an artificial player for angry birds based on declarative knowledge bases. *IEEE Transactions on Computational Intelligence and AI in Games*.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S., AND SCARCELLO, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log. 7(3)*, 499–562.

OSTROWSKI, M. AND SCHAUB, T. 2012. ASP modulo CSP: the clingcon system. *CoRR abs/1210.2287*.

SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence 138*, 181–234.

ZIRTILOGLU, H. AND YOLUM, P. 2008. Ranking semantic information for e-government: complaints management. In *OBI 2008, Karlsruhe, Germany, October 27, 2008*, pages 5:1–5:7.