Answer Set Programming and Extensions

Unit 3 – Hybrid Extensions

Thomas Eiter

Institut für Informationsysteme, TU Wien

VTSA Summer School 2016, Liège, August 29-September 2, 2016

Austrian Science Fund (FWF) grants P24090, P26471, P27730
Unit Outline

1. Introduction

2. ASP + Concrete Theories

3. ASP + Abstract Theories: HEX

4. ASP + Abstract Theories: Clingo
Hybrid Extensions

- Language extensions before are within same semantic/computational framework

- **Need:**
  - interoperability with other logics, e.g. Description Logics
  - interfacing with programming languages, e.g. C++
  - access to general *external* sources of information, e.g. WordNet

- **Problems, issues:**
  - value invention: new ground terms might appear
  - impedance mismatch
  - black boxes: no (little) knowledge about external source
  - semantics: e.g. cyclic reference
Hybrid Extensions: Approaches

- **Embedded ASP**: akin to embedded SQL
  - XSB + stable models
  - DLV Wrapper: library to embed ASP in Java and control execution
  - Potassco: embeds ASP in Python and Lua progs w/ gringo and clasp

- **Bilateral interaction**
  - JASP [Febbraro et al., 2012]: ASP ↔ Java;
    ASP progs can access Java variables, answer sets are stored in Java

- **ASP + concrete theories $X$**: $X$ = ontologies (dl-programs), constraints (CASP), actions (acthex) ...
  - dlvhex/dlplugin, nlp-dl [E_ et al., 2005a],[E_ et al., 2008]
  - clingcon [Ostrowski and Schaub, 2012], inca [Drescher and Walsh, 2010]
    lc2casp [Cabalar et al., 2016]
  - mingo [Liu et al., 2012] dingo for ASP(SMT) [Janhunen et al., 2013]
  - EZSMT [Susman and Lierler, 2016], EZCSP [Balduccini, 2009]
  - acthex [Fink et al., 2013]

- **ASP + abstract “theory” $X$**: even non-logical
  - dlvhex
ASP+Ontologies: DL-programs

- An extension of answer set programs with queries to DL knowledge bases (DL KBs) [E_ et al., 2008]
- Queries can temporarily update the DL KB

*bidirectional flow of information*, with clean technical separation of DL engine and ASP solver ("loose coupling")

\[ \text{ASP Solver} \quad \text{?} \quad \text{DL Engine} \]

- Use dl-programs as "glue" for combining inferences on a DL KB.
- Experimental prototypes
  - NLP-DL (early) https://www.mat.unical.it/ianni/swlp/
  - dlvhex/dlplugin http://www.kr.tuwien.ac.at/research/systems/dlvhex/dlplugin.html
  - DReW http://www.kr.tuwien.ac.at/research/systems/drew/
**Example: Closed World Reasoning**

Handle incomplete information w/ disjunction: ECWA [Gelfond et al., 1986]

Let $\mathcal{K} = (O, P)$, where

- $O = \{\text{Artist} \equiv \text{Singer} \sqcup \text{Painter}, \text{Artist}(\text{Nash}), \text{Singer}(\text{Sting})\}$

- $P = \begin{cases}
\begin{aligned}
(r_1) & \quad \neg p(X) \leftarrow \text{DL[} ; \text{Artist}(X), \neg p(X). \\
(r_2) & \quad \neg s(X) \leftarrow \text{DL[} ; \text{Artist}(X), \neg s(X). \\
p(X) & \leftarrow \text{DL[} ; \text{Artist}(X), \\
& \quad \text{DL[} \text{Painter} \cup \neg p, \text{Singer} \cup \neg s; \text{Painter}(X). \\
& \quad s(X) \leftarrow \text{DL[} ; \text{Artist}(X), \\
& \quad \text{DL[} \text{Painter} \cup \neg p, \text{Singer} \cup \neg s; \text{Singer}(X). \\
\end{aligned}
\end{cases}$

- Under ECWA, *Sting* is not a painter, while *Nash* is one iff he is no singer
- CWA defaults $(r_1), (r_2)$: if $p$ ($s$) is not provable, conclude it’s false
- $\text{Painter} \cup \neg p, \text{Singer} \cup \neg s$ is temporary update of $O$ with $\neg p, \neg s$
- answer sets: $I_1 = \{s(\text{Sting}), \neg p(\text{Sting}), s(\text{Nash}), \neg p(\text{Nash})\}$, $I_2 = \{s(\text{Sting}), \neg p(\text{Sting}), p(\text{Nash}), \neg s(\text{Nash})\}$
- cyclic information flow is essential!
ASP + Constraint Programming

- Writing constraints straight in ASP faces grounding problems:

  \[ \text{ok}(X, Y) \leftarrow p(X, Y), X \leq 10000, Y \geq 50000, 2\times Y < X \]

  treat constraints as \textit{abstract predicates / objects}, avoid grounding

- Approaches:
  - ASP + difference constraints: \( X - Y \leq k \) [Janhunen et al., 2013]
  - ASP + linear inequalities: \( \sum_{i=1}^{k} a_i \times X_i \leq k \) [Mellarkod et al., 2008], [Gebser et al., 2009], [Liu et al., 2012]
  - ASP + linear inequalities + \( X \) : C-HEX [De Rosis et al., 2015]
  - Default constraints [Cabalar et al., 2016]
  - CP + ASP: stable model version of CP [Aziz et al., 2013]
  - solvers: acsolver, clingcon ezcsp, idp, mingo, dingo, lc2casp, ...

- Often focus finite domains

- Lierler [2014]: key features of constraint ASP languages & systems

- formal link Constraint ASP \iff SMT: [Lierler and Susman, 2016]
HEX Programs

- Motivated to meet needs of heterogeneous data access on the Web
- Generalize DL-programs, which provide ASP programs with query access to an ontology.

**Allow to access sources of whatever type (abstract modeling)**

**Features** (cf. [E_ et al., 2005b]):

- **Hilog-style atoms**: variables for predicate names (syntactic sugar)
  
  E.g., \( P(X, Z) \leftarrow P(X, Y), P(Y, Z) \)  
  (transitivity of \( P \))

  \[ \Rightarrow \text{reify atoms } p(t_1, \ldots, t_n) \text{ to } (p, t_1 \ldots, t_n) \]

- **External atoms**: access to external sources, e.g.,
  - ontologies (RDF, OWL, ...)
  - planners,
  - data structures (libraries, built-ins)
  - ... (user definable)
HEX Programs: Idea

Example: import parts of external RDF triples into the program

\[ p(X, Y) \leftarrow url(U), \, \text{\&rdf}[U](X, Y, Z). \]

- \( \text{\&rdf}[U](X, Y, Z) \): *external atom* with “input” \( U \) and “output” \( X, Y, Z \)
- Intuitively, for concrete input (a constant, or predicate name), evaluate the external source and return output

**Issues:**
- model external atoms
- semantics of programs (in spirit of ASP)
- *value invention* (constants not occurring in the program)
Key Construct: External Atoms

\[ p(X, Y) \leftarrow url(U), \&rdf[U](X, Y, Z) \]

**External Atom**

An external atom \( a \) is of the form \( \&g[\vec{Y}](\vec{X}) \), where \( \vec{Y} = Y_1, \ldots, Y_n \) and \( \vec{X} = X_1, \ldots, X_m \) are two lists of terms (input / output list), and \( \&g \) is an external predicate name.

- External atoms may occur only in rule bodies
- \( \&g \) has an associated function

\[ f_{\&g} : 2^{HB_P} \times C^{n+m} \rightarrow \{0, 1\} \]

mapping each \((I, \vec{y}, \vec{x})\), where \( \vec{y} = y_1 \ldots, y_n \), \( \vec{x} = x_1, \ldots, x_m \), to 0 or 1, where \( I \) is an interpretation and \( x_i, y_j \) are from a fixed set \( C \) of constants.

- In practice:
  - \( y_i \) are predicate names or individuals, \( x_j \) are individuals
  - \( HB_P \) are the Herbrand interpretations over \( C = C_{\text{pred}} \cup C_{\text{ind}} \)
- Embraces e.g. aggregates \( \&\text{min}[sal](X) \), string functions \( \&\text{concat}['jim', 'doe'](N) \), etc
HEX Programs: Formal Notions

**Definition (HEX program, omitting higher order)**

A HEX program $P$ is a set of rules $r$ of the form

$$a_1 \lor \cdots \lor a_l \leftarrow b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_n$$

where all $a_i$ and $b_j$ are first-order atoms, and all $c_k$ are first-order or external atoms over $C$ (plus variables $\mathcal{V}$).

Semantics is defined in terms of grounding over $C$ as before

**Definition (Satisfaction)**

A (Herbrand) interpretation $I$ satisfies an (ground) external atom $\&g[\vec{y}](\vec{x})$, denoted $I \models \&g[\vec{y}](\vec{x})$, if $f_{\&g}(I, \vec{y}, \vec{x}) = 1$.

Satisfaction of (ground) $\neg c_k$, body $B(r) = b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_n$, head $H(r) = a_1 \lor \cdots \lor a_l$, rule $r = H(r) \leftarrow B(r)$, and program $P$ are then as usual.
HEX Programs /2

Answer sets are defined via the FLP-reduct [Faber et al., 2011]

**Definition (HEX answer set)**

An interpretation $I$ is an answer set of a (ground) HEX program $P$, if $I$ is a minimal model (w.r.t. $\subseteq$) of $fP^I = \{ r \in P \mid I \models B \}$.

**Note:**

- Gelfond -Lifschitz style reduct (treating external atoms like *not*-_literals) does not ensure minimality of models

  $$P = \{ p(a) \leftarrow \text{\&id}[p](X) \}$$

  where $f_{\&id}(I, p, x) = 1$ iff $I \models p(x)$ (identity)

  single answer set $\{\}$, but another GL-style answer set $\{p(a)\}$

- *not*, $\lor$-free HEX-programs can have multiple answer sets

  $$P = \{ p(a), s_1(X) \leftarrow \text{\&diff}[p, s_2](X). \quad s_2(X) \leftarrow \text{\&diff}[p, s_1](X) \}$$

  where $f_{\&diff}(I, p, q, x) = 1$ iff $I \models p(x)$ and $I \not\models q(x)$ (set difference)

  answer sets $\{p(a), s_1(a)\}$ and $\{p(a), s_1(a)\}$
Implementation

DLVHEX

http://www.kr.tuwien.ac.at/research/systems/dlvhex

- **Challenges:**
  - value invention: in principle, infinitely many constants
  - black box nature: API-style interface abstract
  - minimality checking: source of complexity (from the definition)

- **dlvhex1:**
  - **rewriting approach**
    - use replacement atoms and value guessing ($P \rightarrow \hat{P}$):
      
      $\&g[\bar{y}](\bar{x}) \rightsquigarrow e_{\&g[\bar{y}]}(\bar{x}) \quad e_{\&g[\bar{y}]}(\bar{x}) \lor ne_{\&g[\bar{y}]}(\bar{x}) \leftarrow$
    
    - a-posteriori *compatibility check:* $I \models \&g[\bar{y}](\bar{x})$ iff $e_{\&g[\bar{y}]}(\bar{y}) \in I$
    
    - minimality checking: as $I$ is wrt. $\hat{P}$ is founded, can be often skipped
  - **plugin architecture** (C++ code), use dlv
Implementation /2

- **dlvhex2**

![Diagram]

- **conflict-driven clause learning**
- **plugin architecture** (C++, Python), gringo and clasp
- value invention: liberal domain-expansion safety (2013)
- minimality: unfounded set checking (2012)
- partial input evaluation (2016)
Grounding and Safety

- External atoms may introduce new constants: **value invention**.
- Can lead to programs which **cannot be finitely grounded**.

**Example**

\[
P = \left\{ r_1 : \text{start}(s). \quad r_2 : \text{scc}(X) \leftarrow \text{start}(X). \quad r_3 : \text{scc}(Y) \leftarrow \text{scc}(X), \text{edge}[X](Y). \right\}
\]

**Solution**: syntactic restrictions (safety)

- allow output-variables of external atoms not occurring in ordinary positive atoms \((Y)\)
- traditionally: **strong safety**: no involvement in positive cycles
- essentially, this means **no recursive value invention**!
- but: overly restrictive (e.g. if external graph known to be finite)

**New approach**: **liberal domain expansion (lde) safety**

- exploit **semantic properties** of external atoms
- aim to identify **finite groundability even under recursive value invention**
- a **modular framework in which different properties can be combined**
- embraces e.g. \(\omega\)-, \(\lambda\)-, argument-restrictedness, finite domain, \(\Gamma\)-acyclicity
Grounding and Safety

- **Ide-safety: main ideas**
  - focus on *attribute positions* \((\text{pred}, i)\): *Ide-safe*, if the number of different terms in some grounding \(P'\) that preserves \(AS(P)\) is finite.
  - *program P is Ide-safe*, if all its attribute positions are Ide-safe
  - exploit both syntactic and semantic criteria to find Ide-safe attributes
  - use *term bounding function (TBF)* as parameters:
    - a TBF identifies variables in rules with finitely many instantiations in \(P'\)
  - use a *monotone grounding operator* to create an envelope of \(P'\) and to qualify TBFs;
  - define Ide-safe \((\text{pred}, i)\) in mutual induction with TBFs
  - Example (ctd): in rule \(r_3\), the input/output attribute positions \((\&edge_I[X], 1)\) and \((\&edge_O[X], 1)\) are Ide-safe (for a finite graph, trivial TBFs exist)

- **Grounding Ide-safe programs**: interleave an ordinary ASP grounder with external source evaluation
  - Technically involved, for details see [E_ et al., 2016a]
Model Building Framework

- For program evaluation, developed a model building framework [E_ et al., 2016b]
  1. divide program $P$ into units $u_i$ with partial order
     - employ a **Generalized Splitting Theorem**
     - respect rule dependencies and atom dependencies
  2. compute $u_i$’s answer sets from those of all $u_j$, $j = i_1, \ldots, i_n$, s.t. $u_i \rightarrow u_j$
     - combine answer sets $m_{i_1}, \ldots, m_{i_n}$ to an “input” answer set $m_i$ of $u_i$, added as facts $F(m_i)$
     - output all answer sets of $m_i$
  3. obtain the answer sets of $P$ as those of (unique) maximal unit $u_n$

- Maintain a program graph and a model graph
  - Model combination requires some provenance condition on the $m_{i_1}, \ldots, m_{i_n}$
  - Enables parallelization and model streaming
Example: deciding about swimming location

\[ P^{EDB}_{\text{swim}} = \{ \text{location}(\text{ind}, \text{margB}), \text{location}(\text{ind}, \text{amalB}), \text{location}(\text{outd}, \text{gansD}), \text{location}(\text{outd}, \text{altD}) \} \]

\[ P^{IDB}_{\text{swim}} = \begin{cases} r_1: \text{swim}(\text{ind}) \lor \text{swim}(\text{outd}) \leftarrow . \\ r_2: \quad \text{need}(\text{inoutd}, C) \leftarrow \&rq[\text{swim}](C). \\ r_3: \quad \text{goto}(X) \lor \text{ngoto}(X) \leftarrow \text{swim}(P), \text{location}(P, X). \\ r_4: \quad \text{go} \leftarrow \text{goto}(X). \\ r_5: \quad \text{need}(\text{loc}, C) \leftarrow \&rq[\text{goto}](C). \\ c_6: \quad \leftarrow \text{goto}(X), \text{goto}(Y), X \neq Y. \\ c_7: \quad \leftarrow \text{not go}. \\ c_8: \quad \leftarrow \text{need}(X, \text{money}). \end{cases} \]

- swimming locations indoors and outdoors
- external atom \&rq[L](A): informally, a given location-choice \( L \) has certain required-resource claims \( A \)
Model Building Framework

- atom dependencies: $\alpha \rightarrow \beta \quad \text{“}\alpha\ \text{depends on}\ \beta\text{”}$
  - $m$ = monotonically, $n$ = nonmonotonically, $e$ = externally
- atom dependency graph $AD(P)$

$r_1: \text{swim}(\text{ind}) \lor \text{swim}(\text{outd}) \leftarrow .$
$r_2: \quad \text{need}(\text{inoutd}, C) \leftarrow \&\text{rq}[\text{swim}](C).$
$r_3: \quad \text{goto}(X) \lor \text{ngoto}(X) \leftarrow \text{swim}(P), \text{location}(P, X).$
$r_4: \quad \text{go} \leftarrow \text{goto}(X).$
$r_5: \quad \text{need}(\text{loc}, C) \leftarrow \&\text{rq}[\text{goto}](C).$
$c_6: \quad \leftarrow \text{goto}(X), \text{goto}(Y), X \neq Y.$
$c_7: \quad \leftarrow \text{not go}.$
$c_8: \quad \leftarrow \text{need}(X, \text{money}).$
Model Building Framework

- For program decomposition, use rule dependencies $r \rightarrow s$ (more convenient)

  “application of $r$ depends on the one of $s$”
  - $m = \text{monotonically}$, $n = \text{nonmonotonically}$
  - for external atom $\alpha$ in $B(r)$, atom dependencies $\alpha \rightarrow \beta$ matter

- rule dependency graph $RG(P)$

```
r1: swim(ind) \lor swim(outd) \leftarrow

r2: need(inoutd, C) \leftarrow
&rq[swim](C)

r3: goto(X) \lor ngoto(X) \leftarrow
swim(P), location(P, X)

r5: need(loc, C) \leftarrow
&rq[goto](C)

r6: \leftarrow goto(X), goto(Y), X \neq Y

r7: \leftarrow not go

r8: \leftarrow need(X, money)
```

T. Eiter / TU Wien
VTSA 2016 08-09/2013
Model Building Framework /5

- decompose \( P \) via \( RG(P) \) into \textit{units}
  - closed wrt. cycles
  - may share constraints
  - respect each \( r \rightarrow_n s \) always, and \( r \rightarrow_m s_1, \ldots, r \rightarrow_m s_k \) at some unit

\[\begin{align*}
  u_1 & : \quad \text{swim}(\text{ind}) \lor \text{swim}(\text{outd}) \leftarrow. \\
  \text{derives: } & \quad \text{swim}(X) \\
  r_1 : & \quad \text{swim}(\text{ind}) \lor \text{swim}(\text{outd}) \leftarrow. \\
  \text{derives: } & \quad \text{swim}(X) \\
  u_2 & : \quad \text{need}(\text{inoutd}, C) \leftarrow \&rq[\text{swim}](C). \\
  \text{c8: } & \quad \leftarrow \text{need}(X, \text{money}). \\
  \text{derives: } & \quad \text{need}(\text{inoutd}, C) \\
  r_2 : & \quad \text{need}(\text{inoutd}, C) \leftarrow \&rq[\text{swim}](C). \\
  \text{c8: } & \quad \leftarrow \text{need}(X, \text{money}). \\
  \text{derives: } & \quad \text{need}(\text{inoutd}, C) \\
  u_3 & : \quad \text{goto}(X) \lor \text{ngoto}(X) \leftarrow \text{swim}(P), \text{location}(P, X). \\
  \text{r3: } & \quad \text{goto}(X) \lor \text{ngoto}(X) \leftarrow \text{swim}(P), \text{location}(P, X). \\
  \text{c6: } & \quad \leftarrow \text{goto}(X), \text{goto}(Y), X \neq Y. \\
  \text{c7: } & \quad \leftarrow \text{not go}. \\
  \text{derives: } & \quad \text{goto}(X), \text{ngoto}(X), \text{go} \\
  r_3 : & \quad \text{goto}(X) \lor \text{ngoto}(X) \leftarrow \text{swim}(P), \text{location}(P, X). \\
  \text{c6: } & \quad \leftarrow \text{goto}(X), \text{goto}(Y), X \neq Y. \\
  \text{c7: } & \quad \leftarrow \text{not go}. \\
  \text{derives: } & \quad \text{goto}(X), \text{ngoto}(X), \text{go} \\
  u_4 & : \quad \text{need}(\text{loc}, C) \leftarrow \&rq[\text{goto}](C). \\
  \text{c8: } & \quad \leftarrow \text{need}(X, \text{money}). \\
  \text{derives: } & \quad \text{need}(\text{loc}, C) \\
  r_5 : & \quad \text{need}(\text{loc}, C) \leftarrow \&rq[\text{goto}](C). \\
  \text{c8: } & \quad \leftarrow \text{need}(X, \text{money}). \\
  \text{derives: } & \quad \text{need}(\text{loc}, C)
\end{align*}\]
Model Building Framework /4

- Model Evaluation: final unit $u_4$

answer set $\{\text{need(loc, yogamat)}, \text{go, goto(altD), ngoto(gansD), swim(outd)}\}$
From Black-box to Grey-box

**Previous Evaluation Bottleneck**
- External sources were seen as **black boxes**.
- Behavior under an interpretation did **not** allow for drawing conclusions about other interpretations.
- Algorithms must be very general $\Rightarrow$ room for optimizations **limited**.

**Idea**
- Developers of external sources and/or implementer of HEX-program might have useful additional information.
- Provide a (predefined) list of possible properties of external sources.
- Let the developer and/or user **specify** which properties are satisfied.
- Algorithms **exploit** them for various purposes, most importantly:
  - **efficiency improvements** and
  - **language flexibility** (reducing syntactic restrictions).

**Important:**
User specifies them but does **not** need to know how they are exploited!
Specifying Properties

How to specify them?

- During development of external source using the plugin API.
- As part of the HEX-program using property tags \langle \cdots \rangle.

Example: \&greaterThan[p, 10]() is true, if \( \sum_{I \models p(c)} c > 10 \), for positive integers, it is monotonic

Available properties (samples)

- **Functionality:** \&add[X, Y](Z)\langle functional\rangle
  
  Adds integers \( X \) and \( Y \) and is true for their sum \( Z \).
  
  It provides exactly one output for a given input.

- **Monotonicity in a parameter:** \&diff[p, q](X)\langle monotonic p\rangle
  
  Computes the difference of the extensions of \( p \) and \( q \).
  
  It is monotonic in predicate parameter \( p \).

- **Well-ordering:** \&decrement[X](Z)\langle wellordering 0 0\rangle
  
  Decrements a given integer.
  
  The 0-th output is no greater than the 0-th input (wrt. some ordering).

- **Three-valued semantics:**
  
  The external source can be evaluated under partial interpretations.
  
  \( \cdots \)
Exploiting Properties for Efficiency Improvement

**Conflict-driven Solving**
- ASP program is internally represented by nogoods
- Additional nogoods are learned from conflicting interpretations
- HEX-solver further learns nogoods from external sources which describe parts of their behavior to avoid future wrong guesses.

**Example**
- We evaluate \( \text{diff}[p, q](X) \) under \( I = \{p(a), \neg q(a), \neg p(b), q(b)\} \).
- It is true for \( X = a \) (and false otherwise), i.e., \( I \models \text{diff}[p, q](a) \).
- \( \Rightarrow \) Learn nogood \( N = \{p(a), \neg q(a), \neg p(b), q(b), \neg \text{diff}[p, q](a)\} \).

**Exploiting Properties**
- Use known properties to shrink nogoods to their essential part.

**Example**
- \( \text{diff}[p, q](X) \) is monotonic in \( p \)
- Shrink \( N \) to \( N' = \{p(a), \neg q(a), q(b), \neg \text{diff}[p, q](a)\} \).
  (If \( p(b) \) turns true, \( \text{diff}[p, q](a) \) is still true \( \Rightarrow \neg p(b) \) not needed.)
Exploiting Properties for Language Flexibility

- Ide-Safety: use properties

**Example (ctd)**

\[
P = \left\{\begin{array}{l}
    r_1 : \text{start}(s) . \\
    r_2 : \text{scc}(X) \leftarrow \text{start}(X) . \\
    r_3 : \text{scc}(Y) \leftarrow \text{scc}(X) , \text{edge}[X](Y) .
\end{array}\right\}
\]

**Finite domain:** \text{edge}[X](Y) \langle \text{finitedomain 0} \rangle

Has only finitely many distinct output values; use directive \text{finitedomain} (0 is first argument).

**Example: set splitting**

\[
P = \left\{\begin{array}{l}
    r_1 : p(a) . \\
    r_2 : s_1(X) \leftarrow \text{diff}[p, s_2](X) . \\
    r_3 : s_2(X) \leftarrow \text{diff}[p, s_1](X) .
\end{array}\right\}
\]

**Monotonicity in a parameter:** \text{diff}[p, q](X) \langle \text{monotonic p} \rangle

The source is monotonic in parameter \( p \).

**Antimonotonicity in a parameter:** \text{diff}[p, q](X) \langle \text{antimonotonic q} \rangle

The source is antimonotonic in \( q \) (i.e., shrinking \( q \) does not shrink the output).

- Based on this, Ide-safety is concluded by the solver
Python Programming Interface

- More convenient interface
  Previously only C++ support, but Python preferred by many developers:
  - No overhead due to makefiles, compilation, linking, etc.
  - High-level features.
  - Negligible overhead compared to plugins implemented in C++.

- Implementation of $\text{edge}[X](Y)$:

```python
def edge(x):
    graph = [(1,2), (1,3), (2,3)]  # simplified implementation
    for edge in graph:
        if edge[0] == x.intValue():
            dlvhex.output((edge[1],))  # output edge target

def register():
    prop = dlvhex.ExtSourceProperties()  # inform dlvhex about
    prop.addFiniteOutputDomain(0)  # finiteness of the graph
    dlvhex.addAtom("edge", (dlvhex.CONSTANT, ), 1, prop)
```
Implementation of \( \&\text{diff}[p, q](X) \)

```python
import dlvhex

def diff(p, q):
    for x in dlvhex.getTrueInputAtoms():
        if x.tuple()[0] == p:
            if dlvhex.isFalse(dlvhex.storeAtom((q, x.tuple()[1]))):
                dlvhex.output((x.tuple()[1], ))

def register():
    prop = dlvhex.ExtSourceProperties()
    prop.addMonotonicInputPredicate(0)  # monotonicity/antimonotonicity
    prop.addAntimonotonicInputPredicate(1)  # in the first/second parameter
    dlvhex.addAtom("\text{diff}", (dlvhex.PREDICATE, dlvhex.PREDICATE), 1, prop)
```
Further Improvements

### Availability

- **Pre-compiled binaries** for major platforms available (previously distributed only as sourcecode).

- **Online demo:**
  
  http://www.kr.tuwien.ac.at/research/systems/dlvhex/demo.php

### Interoperability

- Support for all features of the **ASP-Core-2** standard.

- Support for input/output in **CSV format** (interoperability with tools and spreadsheet programs).

More information: [Redl, 2016], [E_ et al., 2015]
Constraint HEX-programs (C-HEX)

**Combining HEX (thus also ordinary ASP) with CP**
- Extension of HEX-Programs to access a constraint solver
- Similar in spirit to clingcon
- *Easy to combine with other external sources*

**Evaluation techniques**
- translation of C-HEX programs to (standard) HEX-programs
- search space pruning techniques
  - conflict-driven learning
  - theory propagation

**Implementation**
- plugin for dlvhex, using CP solver GECODE
CP Syntax and Semantics

Syntax

A CSP problem is a triplet $\langle V, D, C \rangle$, where:
- $V$ is a set of variables, $D$ is a domain and
- $C$ is a set of constraints of form $\langle X, R \rangle$, $X \in V^k$ and $R \subseteq D^k$.

Semantics

- An interpretation is a mapping $I : V \rightarrow D$
- A solution is an interpretation $I$ such that for all $\langle \vec{X}, R \rangle \in C$ it holds that $(I(X_1), \ldots, I(X_n)) \in R$.

Example

Consider the CSP problem $\langle V, D, C \rangle$, where:

$V = \{x, y\}$, $D = \{1, \ldots, 10\}$,

$C = \{ c_1 : \langle y, \{7, 8\}\rangle, c_2 : \langle x, \{1, 2\}\rangle, c_3 : \langle y, \{9, 10\}\rangle \}$

The CSP has solution, e.g., $I(x) = 1$ and $I(y) = 7$. The CSP has no solution (i.e., is inconsistent) due to $c_1$ and $c_3$. 
Constraint HEX-Programs: Formal Notions

HEX showcase: adding constraints [De Rosis et al., 2015]

**Definition (C-HEX program)**

A C-HEX program $P$ is a set of rules $r$ of the form

$$a_1 \lor \cdots \lor a_l \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n$$

- the $a_i$ are (ordinary) atoms or a constraint expressions
- the $b_j, c_k$ are atoms, external atoms or a constraint expressions
- a constraint expression has the form $l \circ r$, where
  - $l, r$ are arith. expressions, atoms, terms and variables built with
    \{$+,-,\ast,/\}$
  - $\circ \in \{\equiv, \neq, >, \geq, \leq, <\}$ is a comparison operator
  - domain is the integers by default

**Intuitive Semantics:**

- Constraint expressions are translated to constraints
- $\Gamma(P, I)$ denotes the CSP associated to interpretation $I$ of $P$
- “$I$ is a constraint answer set of $P$, if it is an answer set of $P$ and the CSP denoted by $\Gamma(P, I)$ is satisfiable”
Example (Alice’s Menu)

Alice’s restaurant menus are such that

1. food should cost more than drink
2. a menu should cost no more than 20

This can be encoded as:

\[
\begin{align*}
    r_1 & : food(P) \leftarrow \text{sql}[	ext{"Select price from Food"](P) \\
    r_2 & : drink(P) \leftarrow \text{sql}[	ext{"Select price from Drink"](P) \\
    r_3 & : \text{max\_price}(20) \\
    r_4 & : \text{inMenu}(F, D) \lor \text{outMenu}(F, D) \leftarrow \text{drink}(D), \text{food}(F) \\
    r_5 & : D > F, \text{inMenu}(F, D) \\
    r_6 & : F + D \leq P \leftarrow \text{inMenu}(F, D), \text{max\_price}(P)
\end{align*}
\]
Realization in HEX

- **Basic idea:**
  1. encode constraint expressions in $P$ by ordinary atoms
     - $l \circ r$ is encoded as $\text{con}(l, \text{“} \circ \text{”}, r)$
  2. guess the truth of ordinary atoms corresponding to constraint atoms
     - e.g., $\text{con}(l, \circ, r) \lor \text{con}(l, \overline{\circ}, r)$, where $\overline{\circ}$ is the negation of $\circ$
  3. check consistency with the CSP $\Gamma(P, A)$ using an external atom
     - $\leftarrow \text{not} \ & \text{check}[\text{con}]()$

- **Implementation**
  - no need for a specialized grounder!
  - develop $\text{check}[\text{con}]()$ external atom using GECODE
  - package it as a back-end of DLVHEX
  - extend nogood learning to inconsistent CSPs, theory propagation

- slower than clingcon (price of generality, as recursive external atoms)
Clingo

- clingo family offers external atoms (cf. http://potassco.sourceforge.net/)
  - targets more logic/reasoners,
  - minimality or models restricted, no dynamic input/recursion through external atoms
  - clingcon is particular "instance" (outgrow)

- clingo 4: Lua + Python interface, multishot solving [Gebser et al., 2014]

- clingo 5 (upcoming) [Gebser et al., 2016]
  - uses extended gringo
  - can specify theory grammars
  - define then concrete "nonground" theories that are handled by gringo
Clingo /2

```lisp
1  #theory difference {
2     constant   { - : 0, unary };  
3     diff_term  { - : 0, binary, left };  
4     linear_term { + : 2, unary; - : 2, unary;  
5             * : 1, binary, left;  
6             + : 0, binary, left; - : 0, binary, left };  
7     domain_term { .. : 1, binary, left };  
8     show_term { / : 1, binary, left };  
9     &dom/0 : domain_term, {=} , linear_term, any;  
10    &sum/0 : linear_term, {=, =, =},<=,>=,<>,!,=}, linear_term, any;  
11    &diff/0 : diff_term, {<=}, constant, any;  
12    &show/0 : show_term, directive  
13   }.
15  #const n=2.  #const m=1000.
16  task(1..n). duration(T,200*T) :- task(T).
17  &dom { 1..m } = start(T) :- task(T).
18  &dom { 1..m } = end(T) :- task(T).
19  &diff { end(T)-start(T) } <= D :- duration(T,D).
20  &sum { end(T) : task(T); -start(T) : task(T) } <= m.
21  &show { start/1; end/1 }.
```

Listing 1: Logic program enhanced with difference and linear constraints (diff.lp)

- (1)–(13) is a theory definition with theory (t-) terms (2)-(8) and atoms (9)–(12)
- t-terms have symbol, priority, arity and associativity (if applicable)
- t-atoms have predicate/arity, names of t-terms, t-operators, and occurrence
- rules (17)–(21) use concrete instance of the generic theory terms
Clingo /3

- **High level interfaces for theory propagation**
  - augment clasp’s native propagation with conclusions from theory $T$
  - user can add custom propagation algs externally (like in dlvhex), both *stateful* and *stateless*

- **Semantics: "LP modulo theories"**
  - distinguish strict and non-strict (program-defined) atoms
  - $T$-stable model concept
    - is a regular stable model relative to extension determined by a "solution" $S$ to theory $T$
      - informally, $S$ is a set of atoms compliant with $T$, where strict atoms must exactly match, true ones may be turned to false
    - augment program $P$ with facts/constraints/choice facts for $S$
  - note: minimality of models is *not* ensured, even for strict atoms, in case of cyclic reference

Example: $P = \{p(a) \leftarrow \text{&id}[p](X)\}$
Clingo’s CDCL Modulo Theories

(I) \textit{initialize} \quad \text{// register theory propagators and initialize watches} \\
\textbf{loop} \\
\textit{propagate} completion, loop, and recorded nogoods \quad \text{// deterministically assign literals} \\
\textbf{if} \text{ no conflict} \textbf{then} \\
\textbf{if} \text{ all variables assigned} \textbf{then} \\
(C) \textbf{if} \text{ some } \delta \in \Delta_T \text{ is violated for } T \in \mathbb{T} \textbf{ then} \text{ record } \delta \quad \text{// theory propagators check } \Delta_T \\
\textbf{else} \textbf{return} \text{ variable assignment} \quad \text{// } \mathbb{T}\text{-stable model found} \\
\textbf{else} \\
(P) \textbf{propagate} \text{ theories } T \in \mathbb{T} \quad \text{// theory propagators may record theory nogoods from } \Delta_T \\
\textbf{if} \text{ no nogood recorded} \textbf{ then} \textbf{ decide} \quad \text{// non-deterministically assign some literal} \\
\textbf{else} \\
\textbf{if} \text{ top-level conflict} \textbf{ then} \textbf{ return} \text{ unsatisfiable} \\
\textbf{else} \\
\textit{analyze} \quad \text{// resolve conflict and record a conflict constraint} \\
(U) \textit{backjump} \quad \text{// undo assignments until conflict constraint is unit}

extends basic CDCL in 4 places:

1. initialization
2. propagation (on partial assignments):
   - "watched literals" trigger the theory propagator (user defined)
3. final check of total assignments
   - lazy theory propagation possible: delay to complete assignments
4. undo steps upon backjumping
Evaluation

- Experiments [Gebser et al., 2016]
  - Scheduling problems, using difference logic constraints
  - Use different graph algorithms for satisfiability checking, stateless and stateful
  - Clear profile: stateful < stateless < native encoding (though details are lacking)

- Purpose of clingo 5: very much for developing domain-specific solvers, "lower level"

- dlvhex, instead is more to the user-level, and targets general external sources (and their combination)
Rehan Abdul Aziz, Geoffrey Chu, and Peter J. Stuckey.
Stable model semantics for founded bounds.

Marcello Balduccini.
Integrating grounding in the search process for answer set computing.
Online available at https://www.mat.unical.it/ASPOCP09/.

Pedro Cabalar, Roland Kaminski, Max Ostrowski, and Torsten Schaub.
An ASP semantics for default reasoning with constraints.
In Kambhampati [2016], pages 1015–1021.

Alessandro Francesco De Rosis, Thomas Eiter, Christoph Redl, and Francesco Ricca.
Constraint answer set programming based on hex-programs.
In Daniela Inclezan and Marco Maratea, editors, Informal Proceedings 8th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2015), 2015.
https://sites.google.com/site/aspocep15/accepted.

Christian Drescher and Toby Walsh.
A translational approach to constraint answer set solving.
Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. 
NLP-DL: A KR system for coupling nonmonotonic logic programs with description logics. 
System poster.

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. 
A uniform integration of higher-order reasoning and external evaluations in answer set programming. 

Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. 
Combining Answer Set Programming with Description Logics for the Semantic Web. 


Thomas Eiter, Giovambattista Ianni, Michael Fink, Thomas Krennwallner, Christoph Redl, and Peter Schüller.
A model building framework for ASP with external computations.

Thomas Eiter, Tobias Kaminski, Christoph Redl, and Antonius Weinzierl.
Exploiting partial assignments for efficient evaluation of answer set programs with external source access.

Wolfgang Faber, Nicola Leone, and Gerald Pfeifer.
Semantics and complexity of recursive aggregates in answer set programming.

Onofrio Febbraro, Nicola Leone, Giovanni Grasso, and Francesco Ricca.
JASP: A framework for integrating answer set programming with java.
In Brewka et al. [?].
Michael Fink, Stefano Germano, Giovambattista Ianni, Christoph Redl, and Peter Schüller.  
Acthex: Implementing hex programs with action atoms.  

M. Gebser, M. Ostrowski, and T. Schaub.  
Constraint answer set solving.  

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub.  
Clingo = ASP + control: Preliminary report.  

Theory solving made easy with clingo 5.  
To appear. Extended version  
M. Gelfond, H. Przymusinska, and T. Przymusinski.
The Extended Closed World Assumption and its Relationship to Parallel Circumscription.

Tomi Janhunen, Guohua Liu, and Ilkka Niemelä.
Tight integration of non-ground answer set programming and satisfiability modulo theories.

Subbarao Kambhampati, editor.

Yuliya Lierler and Benjamin Susman.
Constraint answer set programming versus satisfiability modulo theories.
In Kambhampati [2016], pages 1181–1187.

Yuliya Lierler.
Relating constraint answer set programming languages and algorithms.
Guohua Liu, Tomi Janhunen, and Ilkka Niemelä.
Answer set programming via mixed integer programming.
In Gerhard Brewka, Thomas Eiter, and Sheila A. McIlraith, editors, Principles of Knowledge
Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR

Veena S Mellarkod, Michael Gelfond, and Yuanlin Zhang.
Integrating Answer Set Programming and Constraint Logic Programming.

Max Ostrowski and Torsten Schaub.
ASP modulo CSP: the clingcon system.

Christoph Redl.
The DLVHEX system for knowledge representation: Recent advances (system description).
Benjamin Susman and Yuliya Lierler.  
System description: Smt-based constraint answer set solver.  
To appear.  