Pushing Efficient Evaluation of HEX Programs by Modular Decomposition*

Thomas Eiter[1], Michael Fink[1], Giovambattista Ianni[2],
Thomas Krennwallner[1], and Peter Schüller[1]

[1] Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,fink,tkren,ps}@kr.tuwien.ac.at
[2] Dipartimento di Matematica, Cubo 30B, Università della Calabria
87036 Rende (CS), Italy
ianni@mat.unical.it

Abstract. The evaluation of logic programs with access to external knowledge sources requires to interleave external computation and model building. Deciding where and how to stop with one task and proceed with the next is a difficult problem, and existing approaches have severe scalability limitations in many real-world application scenarios. We introduce a new approach for organizing the evaluation of logic programs with external knowledge sources and describe a configurable framework for dividing the non-ground program into overlapping possiblysmaller parts called evaluation units. These units will then be processed by interleaving external evaluations and model building according to an evaluation and a model graph, and by combining intermediate results. Experiments with our prototype implementation show a significant improvement of this technique compared to existing approaches. Interestingly, even for ordinary logic programs (with no external access), our decomposition approach speeds up existing state of the art ASP solvers in some cases, showing its potential for wider usage.

1 Introduction

Motivated by a need for knowledge bases to access external sources, extensions of declarative KR formalisms have been conceived that provide this capability, which is often realized via an API like interface. In particular, HEX programs [6] extend nonmonotonic logic programs under the stable model semantics, with the possibility to bidirectionally access external sources of knowledge and/or computation. E.g., a rule

$$pointsTo(X, Y) \leftarrow \&hasHyperlink[X](Y), url(X)$$

might be used for obtaining pairs of URLs (X, Y), where X actually links Y on the Web, and $\&hasHyperlink$ is an *external predicate* construct. Besides constant values, as above, also relational knowledge (predicate extensions) can flow from external sources to the logic program at hand and vice versa, and recursion involving external predicates is allowed under suitable safety conditions. This facilitates a variety of applications which

* This research has been supported by the Austrian Science Fund (FWF) project P20841 and the Vienna Science and Technology Fund (WWTF) project ICT08-020.

require logic programs to interact with external environments, such as querying RDF sources using SPARQL [17], bioinformatics [11], combining rules and ontologies [5], e-government [21], planning [14], and multi-contextual reasoning [2], to mention a few.

Despite the lack of function symbols, an unrestricted use of external atoms leads to undecidability, as new constants may be introduced, yielding a potentially infinite Herbrand universe. However, even under suitable restrictions like domain-expansion safety [7], the efficient evaluation of HEX-programs is challenging, due to aspects like nonmonotonic atoms and recursive access (e.g., in transitive closure computations).

Advanced in this regard was [7], which fostered an evaluation approach using a traditional LP system. Roughly, the values of ground external atoms are guessed, model candidates are computed as answer sets of a rewritten program, and then those discarded which violate the guess. A generalized notion of Splitting Set [13] was introduced in [7] for non-ground HEX-programs, which were then split into subprograms with and without external access, where the former are as large and the latter as small as possible. They are evaluated with various specific techniques, depending on their structure [7, 20]. However, for real-world applications this approach has severe scalability limitations, as the number of ground external atoms may be large, and their combination causes a huge number of model candidates and memory outage without any answer set output.

To remedy this problem, we reconsider model computation and make several contributions, which are summarized as follows.

- We present an evaluation framework for HEX-programs, which allows for flexible program evaluation. It comprises an *evaluation graph*, which captures a modular decomposition and partial evaluation order, and a *model graph*, which comprises for each node, sets of input models (which need to be combined) and output models to be passed on. This structure allows us to realize customized divide-and-conquer evaluation strategies, using a further generalization of the Splitting Theorem. As the method works on non-ground programs, value introduction by external calculations and applying optimization techniques based on domain splitting [4] are feasible.
- The nodes in the evaluation graph are *evaluation units* (program sub-modules), which noticeably — and different from other decomposition approaches — may overlap and be non-maximal resp. minimal. In particular, constraint sharing can prune irrelevant partial models and candidates earlier than in previous approaches.
- A prototype of the evaluation framework has been implemented, which is generic and can be instantiated with different ASP solvers (in our case, with dlv and clasp). It features also *model streaming*, i.e., computation one by one. In combination with early model pruning, this can considerably reduce memory consumption and avoid termination without solution output in a larger number of settings.
- In contrast to the previous approach, the new one allows for generating parallelizable evaluation plans. Applying it to ordinary programs (without external functions) allows us to do parallel solving with a solver software that does not have parallel computing capabilities itself ("parallelize from outside").

In order to assess the new approach, we conducted a series of experiments which clearly demonstrate its usefulness. The new implementation outperforms the current dlvhex system significantly, using (sometimes exponentially) less memory and running much faster. Interestingly, also on some ordinary test programs it compared well to state of the

art ASP solvers: apart from some overhead on fast solved instances, our decomposition approach showed a speed up on top of dlv and clasp. The results indicate a potential for widening the optimization techniques of ordinary logic programs, and possibly also other formalisms like multi-context systems. Due to space limitation, only selected experiments are shown and proofs omitted. The full experimental outcome with benchmark instances is available at http://www.kr.tuwien.ac.at/research/systems/dlvhex/experiments.html.

2 Preliminaries

HEX programs [7] are built on mutually disjoint sets $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ of *constant*, *variable*, and *external predicate names*, respectively. By default, the elements of $\mathcal{X}$ (resp., $\mathcal{C}$) start with a letter in upper (resp., lower) case; elements of $\mathcal{G}$ are prefixed with '&'. Constant names serve both as individual and predicate names. Noticeably, $\mathcal{C}$ may be infinite. *Terms* are elements of $\mathcal{C} \cup \mathcal{X}$. A (*higher-order*) *atom* is of form $Y_0(Y_1, \ldots, Y_n)$, where $Y_0, \ldots, Y_n$ are terms; $n \geq 0$ is its *arity*. The atom is *ordinary*, if $Y_0 \in \mathcal{C}$. In this paper, we assume that all atoms are ordinary, i.e., of the form $p(Y_1, \ldots, Y_n)$. An *external atom* is of the form $\&g[\mathbf{X}](\mathbf{Y})$, where $\mathbf{X} = X_1, \ldots, X_n$ and $\mathbf{Y} = Y_1, \ldots, Y_m$ are lists of terms (called *input* and *output list*, resp.), and $\&g$ is an *external predicate name*.

HEX-programs (or simply *programs*) are finite sets of *rules* r of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, not\ \beta_{n+1}, \ldots, not\ \beta_m, \tag{1}$$

where $m, k \geq 0$, all α_i are atoms, all β_j are atoms or external atoms, and "not" is *negation as failure* (or *default negation*). If $k = 0$, r is a *constraint*, otherwise a *non-constraint*. If r is variable-free, $k = 1$, and $m = 0$, it is a *fact*.

We call $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ the *head* of r and $B(r) = B^+(r) \cup B^-(r)$ the *body* of r, where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$ are the (sets of) *positive* and *negative body atoms*, respectively. We write $a \sim b$ when two atoms a and b unify.

Semantics. Answer sets of ordinary programs [10] are extended to HEX-programs P, using the FLP reduct [8]. The *Herbrand base* HB_P of P, is the set of all ground instances of atoms and external atoms occurring in P, obtained by a variable substitution over $\mathcal{C}$. The grounding of a rule r, $grnd(r)$, and of P, $grnd(P) = \bigcup_{r \in P} grnd(r)$, is analogous.

An *interpretation relative to* P is any subset $I \subseteq HB_P$ containing only atoms. Satisfaction is defined as follows: I is a *model* of (i) an atom $a \in HB_P$ respectively (ii) a ground external atom $a = \&g[\mathbf{x}](\mathbf{y})$, denoted $I \models a$, iff (i) $a \in I$ respectively (ii) $f_{\&g}(I, \mathbf{x}, \mathbf{y}) = 1$, where $f_{\&g} \colon 2^{HB_P} \times \mathcal{C}^{n+m} \to \{0, 1\}$ is a (fixed) function associated with $\&g$, called *oracle function*; intuitively, $f_{\&g}$ tells whether $\mathbf{y}$ is in the output computed by the external source $\&g$ on input $\mathbf{x}$.

For a ground rule r, (i) $I \models H(r)$ iff $I \models a$ for some $a \in H(r)$, (ii) $I \models B(r)$ iff $I \models a$ for every $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$, and (iii) $I \models r$ iff $I \models H(r)$ or $I \not\models B(r)$. Then, I is a *model* of P, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$.

The *FLP-reduct* [8] of P w.r.t. an interpretation I, denoted fP^I, is the set of all $r \in grnd(P)$ such that $I \models B(r)$. Finally, $I \subseteq HB_P$ is an *answer set of* P, iff I is a subset-minimal model of fP^I. By $\mathcal{AS}(P)$ we denote the set of all answer sets of P.

A ground external atom a is called *monotonic*, iff $I \models a$ implies $I' \models a$ for all interpretations I, I' such that $I \subseteq I'$. A non-ground external atom is *monotonic*, if all its

ground instances are. For practical concerns, we assume that each input argument X_i of an external atom $a = \&g[\mathbf{X}](\mathbf{Y})$, has a type label *predicate* or *constant*, which is unique for $\&g$. We then assume that $\forall I, I', \mathbf{x}, \mathbf{y}. f_{\&g}(I, \mathbf{x}, \mathbf{y}) = f_{\&g}(I', \mathbf{x}, \mathbf{y})$ holds for all I and I' which coincide on all the extensions of predicates x_i such that X_i is of type *predicate*; hence, $f_{\&g}$ depends only on the input of a given by predicate extensions and individuals.

3 Formal Framework

The former HEX evaluation algorithm [7] is based on a dependency graph between non-ground atoms; depending on the above, piecewise evaluation is carried out on appropriate selections of sets of rules (the 'bottoms' of a program). In contrast with that, we base evaluation on dependency between, possibly overlapping, subsets of rules of the program at hand. We call such groups of rules *evaluation units* (in short: units); with respect to the former approach, units are not necessarily maximal. Instead, when forming units we require that their partial models (i.e., atoms in heads of their rules) do not interfere with those of other units. This allows for independence, efficient storage, and easy composition of partial models of distinct units.

Creating an *evaluation graph* (a graph of units) for a given HEX program is done by an *evaluation heuristics*. Note that several possible evaluation graphs exist with highly varying evaluation performance. Here, we concentrate on the evaluation framework, leaving the design of optimal heuristics subject to future work. In Section 5 we informally describe the old heuristics *H1* and a simple new heuristics *H2* used in the experiments.

For illustrating our contribution, we make use of the following running example.

Example 1. Consider the HEX program P with facts $choose(a, c, d)$ and $choose(b, e, f)$:

$$
\begin{array}{lrl}
r_1\colon & plan(a) \vee plan(b) & \leftarrow \\
r_2\colon & need(p,C) & \leftarrow \&cost[plan](C) \\
r_3\colon & use(X) \vee use(Y) & \leftarrow plan(P), choose(P,X,Y) \\
r_4\colon & need(u,C) & \leftarrow \&cost[use](C) \\
c_5\colon & & \leftarrow need(\_,money)
\end{array}
$$

External atom $\&cost$ has one predicate type input: for any interpretation I and some constant (predicate) q, $f_{\&cost}(I, q, C) = 1$ iff $C = money$ and $I \cap \{q(a), q(f)\} \neq \emptyset$, or $C = time$ and $I \cap \{q(b), q(c), q(d), q(e)\} \neq \emptyset$; otherwise 0.

The program P informally expresses to guess a $plan$ and a sub-plan use; resource usage is evaluated using $\&cost$, solutions that require $money$ are forbidden by a constraint, choosing a or f in r_1 or r_3 requires $money$, other choices require $time$. The answer set of P is $\{plan(b), use(e), need(p, time), need(u, time)\}$ (omitting facts). □

We next introduce a new notion of dependencies in HEX programs.

Dependency Information. To account for dependencies between heads and bodies of rules is a common approach for devising an operational semantics of ordinary logic programs, as done e.g. by *stratification* and its refinements like *local* [18] or *modular stratification* [19], or by *splitting sets* [13]. New types of dependencies were considered in [7], as in HEX programs, head-body dependencies are not the only possible source of predicate interaction. In contrast to the traditional notion of dependency that in essence

hinges on propositional programs, we consider relationships between non-ground atoms. We lift the definition of atom dependency in [7] to dependencies among rules.

Definition 1 (Rule dependencies). *Let P be a program with rules $r, s \in P$. We denote by $r \rightarrow_p s$ (resp. $r \rightarrow_n s$) that r* depends positively *(resp.* negatively*)* on *s whenever:*

(i) $a \in B^+(r)$, $b \in H(s)$, *and* $a \sim b$, *then* $r \rightarrow_p s$;
(ii) $a \in B^-(r)$, $b \in H(s)$, *and* $a \sim b$, *then* $r \rightarrow_n s$;
(iii) $a \in H(r)$, $b \in H(s)$, *and* $a \sim b$, *then both* $r \rightarrow_p s$ *and* $s \rightarrow_p r$;
(iv) $a \in B(r)$ *is an external atom of form* $\&g[\mathbf{X}](\mathbf{Y})$ *where* $\mathbf{X} = X_1, \ldots, X_n$, *the input* $X_i = p$ *is of type predicate, and* $b \in H(s)$ *is an atom of form* $p(\mathbf{Z})$, *then*
  - $r \rightarrow_p s$ *if* $\&g$ *is monotonic and* $a \in B^+(r)$, *and*
  - $r \rightarrow_n s$ *otherwise.*

Example 2 (ctd.). According to Definition 1, due to (i) we have dependencies $r_3 \rightarrow_p r_1$, $c_5 \rightarrow_p r_2$, and $c_5 \rightarrow_p r_3$; and due to (iv) we have dependencies $r_2 \rightarrow_p r_1$ and $r_4 \rightarrow_p r_3$. □

We generically say that r *depends on* s ($r \rightarrow s$), if either $r \rightarrow_p s$ or $r \rightarrow_n s$.

Evaluation Graph. Using the above notion of rule dependencies, we define the structure of an evaluation graph consisting of evaluation units depending on one another.

We handle constraints separately from non-constraints. Constraints cannot infer additional atoms, hence they can be shared between evaluation units in many cases, while sharing non-constraints could violate the modularity of partial models. In the former evaluation algorithm, a constraint could only kill answer sets once all its dependencies were fulfilled. The new algorithm increases evaluation efficiency by duplicating nonground constraints, allowing them to kill models earlier.

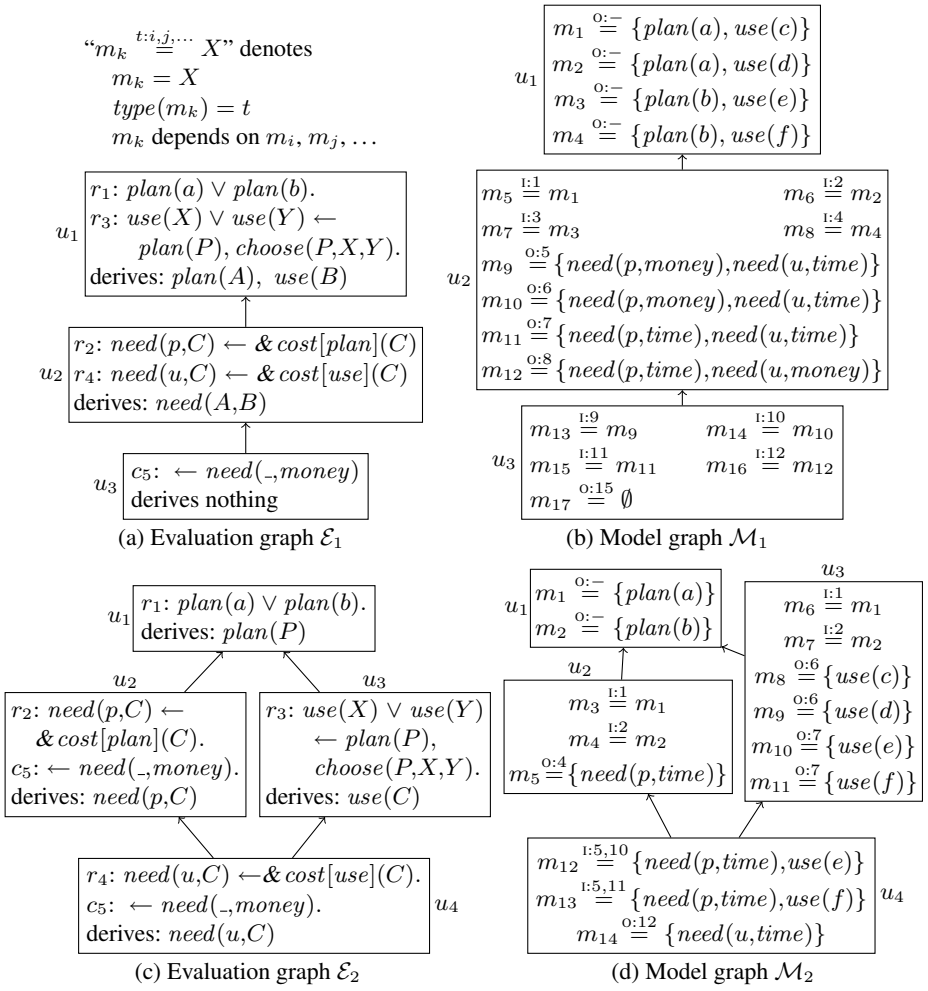
In the following, an *evaluation unit* is any nonempty subset of the rules of a program. An *evaluation unit graph* is a directed graph where each vertex is a unit. Let $G = (U, E)$ be an evaluation unit graph of program P, $v \in U$, and $r \in v$. We say that the dependencies of r *are covered by* G *at unit* v iff for all rules s of P, if $r \rightarrow s$ holds for $s \in w$, $w \in U$, and $w \neq v$, then there is an edge from v to w in G.

Definition 2 (Evaluation graph). *An* evaluation graph $\mathcal{E} = (V, E)$ *of a program P is an acyclic evaluation unit graph such that (a) $\bigcup V = P$, (b) for each $r \in P$ and each $v \in V$ with $r \in v$, negative dependencies of r are covered by $\mathcal{E}$ at v, and (c) for each $r \in P$ its positive dependencies are covered by $\mathcal{E}$ at every (resp., some) unit $v \in V$ with $r \in v$ if r is a non-constraint (resp., constraint).*

Note that by acyclicity of $\mathcal{E}$, mutually dependent rules must be in the same unit. Furthermore, a unit can have in its rule heads only atoms which do not match atoms derivable by other units, due to dependencies between rules deriving unifiable heads.

Let $\mathcal{E} = (V, E)$ be an evaluation graph. We write $v < w$ iff there exists a path from v to w in $\mathcal{E}$, and $v \leq w$ iff either $v < w$ or $v = w$. For a unit $v \in V$, let $v^< = \bigcup_{w \in V, v < w} w$ be the set of rules in 'preceding' units on which v depends, and let $v^\leq = v^< \cup \{v\}$. Furthermore, for each unit $v \in V$, we define $preds_E(v) = \{w \in V \mid (v, w) \in E\}$.

Example 3 (ctd.). We focus on two specific evaluation graphs of program P. Graph $\mathcal{E}_1$ in Fig. 1a corresponds to the former HEX evaluation method. Intuitively, u_1 guesses inputs for u_2, which then evaluates rules with external atoms; finally u_3 checks constraints.



(a) Evaluation graph $\mathcal{E}_1$ (b) Model graph $\mathcal{M}_1$

(c) Evaluation graph $\mathcal{E}_2$ (d) Model graph $\mathcal{M}_2$

Fig. 1: Old vs New strategy: evaluation and model graphs

Another (possibly more efficient) evaluation graph is $\mathcal{E}_2$ in Fig. 1c: guesses of r_1 and r_3 are split into separate units u_1 and u_3, reducing redundancy of external atom evaluation. The constraint c_5 is shared between u_2 and u_4, and it prunes models in u_2, again reducing redundancy. Note that units with multiple inputs and constraint duplication do not exist in the former HEX algorithm. □

Model Graph. We now define the model graph, which interrelates models at evaluation units. It is the foundation of our model building algorithm. In the following, a *model* m is a set of ground atoms. Each model belongs to a specific unit $unit(m)$, and has a type $type(m)$ which is either input (I) or output (O). Given a set of models M and a unit u, we denote by $imods_M(u) = \{m \in M \mid unit(m) = u,\ type(m) = \text{I}\}$ and $omods_M(u) = \{m \in M \mid unit(m) = u,\ type(m) = \text{O}\}$ the sets of input and output

Algorithm 1: BUILDMODELGRAPH ($\mathcal{E} = (V, E)$: evaluation graph)

Output: $\mathcal{M} = (M, F, unit, type)$: model graph
$M := \emptyset$, $F := \emptyset$, $U := V$
while $U \neq \emptyset$ **do**
  choose $u \in U$ s.t. $preds_E(u) \cap U = \emptyset$
  let $preds_E(u) = \{u_1, \ldots, u_k\}$ and $M' := \emptyset$
  for $m_1 \in omods_M(u_1), \ldots, m_k \in omods_M(u_k)$ **do**
(a) **if** $m := m_1 \bowtie \cdots \bowtie m_k$ *is defined* **then**
(b) $M' :=$ EVALUATEUNIT(u, m)
    set $unit(m) := u$ and $type(m) :=$ I
    set $unit(m') := u$ and $type(m') :=$ O for all $m' \in M'$
    $F := F \cup \{(m', m) \mid m' \in M'\} \cup \{(m, m_i) \mid 1 \leq i \leq k\}$
    $M := M \cup M' \cup \{m\}$
  $U := U \setminus \{u\}$
return $(M, F, unit, type)$

models of u, resp. Intuitively, when computing answer sets of a program P under an evaluation graph $\mathcal{E}$, each unit u might have a number of input models: each input model determines a particular set of input assertions for unit u, and is built in turn by merging a number of output models m_i, one per each unit $u_i \in preds_E(u)$. Given an input model m for u, the evaluation of u might 'produce' a number of output models depending on m.

Definition 3 (Model graph). *Given an evaluation graph $\mathcal{E}$=(U,E) for a program P, a* model graph $\mathcal{M}$=$(M, F, unit, type)$ *is a labelled directed acyclic graph, where each vertex $m \in M$ is a model, $F \subseteq M \times M$, and $unit\colon M \to U$ and $type\colon M \to \{\text{I,O}\}$ are vertex labelling functions. F consists of the following edges for each model m: (a) (m, m') with $m \in omods_M(u)$, $m' \in imods_M(u)$ for some $u \in U$ s.t. $preds_E(u) \neq \emptyset$; and (b) $(m, m_1), \ldots, (m, m_k)$ if $m \in imods_M(u)$, $u \in U$, and $\{u_1, \ldots, u_k\} = preds_E(u)$, such that $m_i = f(m, u_i) \in omods_M(u_i)$, $1 \leq i \leq k$ is some (unique) output model of u_i.*

Note that the empty graph is a model graph, and that evaluation units may have no models in the model graph. This is by intent, as our model building algorithm progresses from an empty model graph to one with models at each unit (iff the program has an answer set). Given a model m, we denote by m^+ the *expanded model* of m, which is the union of m and all output models on which m transitively depends. Note, that given m at unit u, m^+ is a union of one output model from each unit in $u^{\leq}$.

Example 4 (ctd.). In $\mathcal{M}_2$, some expanded models are $m_5^+ = \{need(p,time), plan(b)\}$, $m_{11}^+ = \{use(f), plan(b)\}$, and $m_{13}^+ = \{need(p,time), use(f), plan(b)\}$. □

4 Evaluation

Roughly speaking, answer sets of a program can be built by first obtaining an evaluation graph, and then computing a model graph accordingly. We next demonstrate model building on our example, informally discussing the main operations BUILDMODELGRAPH,

Algorithm 2: EVALUATEUNIT(u: evaluation unit, m: input model at u)

Output: output models at u
// determine external atoms that get input only from m
$A_{in} := \{\&g[\mathbf{x}](\mathbf{y}) \mid r \in u \text{ and } x \cap \left(\bigcup_{r' \in u} H(r')\right) = \emptyset\}$
$m_{aux} := \{d_{\&g}(\mathbf{x}, \mathbf{y}) \mid \&g[\mathbf{x}](\mathbf{y}) \in A_{in} \text{ and } f_{\&g}(m, \mathbf{x}, \mathbf{y}) = 1\}$ // get replacement facts
$u' := u$ with external atoms A_{in} replaced by their corresponding auxiliaries
choose $ES \in \{\text{PLAIN}, \text{WELLF}, \text{GNC}\}$ according to the structure of u'
return $ES(u', m \cup m_{aux})$ // return set of models for u' w.r.t. m and m_{aux} using ES

model join '$\bowtie$', and EVALUATEUNIT, which are later defined formally. To simplify our algorithm, we assume empty dummy input models at units without predecessors.

Example 5 (ctd.). Fig. 1b and 1d show model graphs $\mathcal{M}_1$ and $\mathcal{M}_2$ resulting from the evaluation of $\mathcal{E}_1$ and $\mathcal{E}_2$, resp. On $\mathcal{E}_1$, unit u_1 is evaluated first, yielding output models $m_1, \ldots, m_4$ containing guesses over $plan$ and use. These models are also input models for u_2; for each of them we first evaluate the external atoms $\&cost[plan](C)$ and $\&cost[use](C)$ and then evaluate $\{r_2, r_4\}$ using an external solver; we then obtain output models $m_9, \ldots, m_{12}$ which are also input models for $u_3 = \{c_5\}$. Evaluation of u_3 yields a model m_{17} for input model m_{15}, and $m_{17}^+ = \{need(p, time), need(u, time), plan(b), use(e)\}$ is the only answer set of P. For evaluation graph $\mathcal{E}_2$, we start with u_1, which yields output models m_1 and m_2. Then we process u_2 and u_3 in arbitrary order (or even in parallel). One external atom will be evaluated for each input model of u_2. Then, we evaluate $\{r_2, c_5\}$, which yields output model m_5 for input m_4, and no model for m_3. For each input model of u_3 two models are generated by r_3. Input models for u_4 are built by joining output models from u_2 and u_3 (cf. Ex. 7). Finally, u_4 evaluates one external atom per input model and then gets the models m_{14} for input m_{12} and no model for input m_{13} for $\{r_4, c_5\}$. We again get a single answer set m_{14}^+ of P. □

Model Joining. Input models of a unit u are built by combining one output model m_i for each unit u_i on which u depends. Only combinations with common ancestry in the model graph are allowed. To formalize this condition, we introduce the following notion. Unit w *is a common ancestor unit (cau) of* v in an evaluation graph $\mathcal{E} = (V, E)$ iff $v, w \in V$, $v \neq w$, and there exist distinct paths p_1, p_2 from v to w in E s.t. p_1 and p_2 overlap only in vertices v and w. We denote by $caus(v)$ the set of all caus of unit v.

Example 6. In an evaluation graph sketched by dependencies $a \to b \to c \to d \to e$, $a \to c$, and $a \to d$, we have that $caus(a) = \{c, d\}$ and no other unit besides a has caus. □

We next formally define the join operator '$\bowtie$' on models.

Definition 4. *Let $\mathcal{M} = (M, F)$ be a model graph for an evaluation graph $\mathcal{E} = (V, E)$ of a program P, and let $u \in V$ be a unit. Let $u_1, \ldots, u_k$ be all the units u_i on which u depends, and let $m_i \in omods_M(u_i)$, $1 \leq i \leq k$. Then the* join $m = m_1 \bowtie \cdots \bowtie m_k = \bigcup_{1 \leq i \leq k} m_i$ *is defined iff for each $u' \in caus(u)$ there exists exactly one model $m' \in omods_M(u')$ reachable (in $\mathcal{M}$) from some model m_i, $1 \leq i \leq k$.*

Example 7 (ctd.). Building input models for u_4 in Figure 1d requires a join operation: from all pairs of output models at u_2 and u_3, only those with a common ancestor at u_1

yield a valid input model: u_4 has two input models $m_5 \bowtie m_{10}$ and $m_5 \bowtie m_{11}$; they have m_2 as a common ancestor at u_1. For other combinations, the join is undefined. □

Evaluation Algorithm. Alg. 1 builds our model graph: U contains units for which models still have to be calculated; in each iteration step (a) determines all input models m for unit u, step (b) calculates output models originating in m.

EVALUATEUNIT (Alg. 2) evaluates unit u; it creates output models for given input model m. Given a possibly non-ground external atom $\&g[\mathbf{x}](\mathbf{y})$, we denote by the ordinary atom $d_{\&g}(\mathbf{x}, \mathbf{y})$ its *corresponding replacement atom*. These replacement atoms are instrumental for evaluating rules containing external atoms; we apply the approach of "HEX component evaluation" introduced in [20]: intuitively, we evaluate external atom functions wrt. a given input model m, augment m with replacement facts for inputs where $f_{\&g}$ evaluates to 1, replace external atoms by corresponding replacement atoms in all rule bodies, and then evaluate the resulting program R wrt. augmented input model m'. Depending on the structure of R, we can choose between different *evaluation strategies* as described in [20]; PLAIN: if R contains no external atoms, we create output models $\mathcal{AS}(R \cup m')$ by an external solver; WELLF: if external atoms in R are monotonic, and none is contained in a negative dependency cycle, we use a fixpoint algorithm; and GNC: in all other cases, we use a guess-and-check algorithm.

Soundness and Completeness. Because of constraint duplication, the evaluation graph does not partition the input program, and the customary notion of splitting set does not apply to evaluation units. We define a *generalized bottom* of a program, that is a way to split a program into two parts with a nonempty intersection containing certain constraints. We prove that generalized bottoms behave similar as bottoms gb_A created by global splitting sets and $EVAL$ [7], to which we only refer here.

Definition 5. *Given a* HEX *program P, a* generalized bottom $P' \subseteq P$ *is a subset of P such that there exists a global splitting set A and the set $C = P' \setminus gb_A(P)$ is a set of constraints with $B^-(C) \subseteq A$.*

Definition 6 (as in [7]). *For an interpretation I and a program Q, the* global residual*, $gres(Q, I)$, is the program obtained from Q as follows: (i) add all the atoms in I as facts, and (ii) for each "resolved" external atom $a = \&g[\mathbf{X}](\mathbf{Y})$ occurring in some rule of Q, replace a with a fresh ordinary atom $d_{\&g}(\mathbf{c})$ for each tuple $\mathbf{c}$ output by $EVAL(\&g, Q, I)$.*

Theorem 1. *Let P be a domain-expansion safe* HEX *program over atoms U, and let P' be a generalized bottom of P with global splitting set A and constraints C. Then $M \setminus D \in \mathcal{AS}(P)$ iff $M \in \mathcal{AS}(gres(P'', I))$ with $I \in \mathcal{AS}(P')$ and $P'' = (P \setminus P') \cup C'$, where D is the set of additional atoms in $gres(P'')$ with predicate name of form $d_{\&g}$, and $C' = \{c \in C \mid B^+(c) \cap (U \setminus A) \neq \emptyset\}$ is the set of constraints in P' with body atoms unifying with atoms in A as well as with atoms in $U \setminus A$.*

Intuitively, this is a relaxation of the previous nonground HEX splitting theorem regarding constraints: those matching atoms derived in the splitting set as well as in the residual program may be added to P' iff they are not removed from the residual program. The benefit of sharing such constraints is a reduced set of models $\mathcal{AS}(P')$.

The following lemma applies the above splitting theorem to the evaluation graph and the model graph, and is instrumental for showing correctness of the algorithm.

Lemma 1. *Given an evaluation graph $\mathcal{E} = (V_E, E_E)$ of a* HEX *program P, and an evaluation unit $u \in V_E$, it holds that (i) the subprogram $u^<$ is a generalized bottom of the subprogram $u^\leq$; furthermore if for each predecessor $u' \in preds_E(u)$ we have that models $\{m'^+ \mid m' \in omods_M(u')\}$ are the models of $u'^\leq$, it holds that (ii) step (a) of* BUILDMODELGRAPH *creates the models of bottom $u^<$ as $imods_M(u)$, and (iii) step (b) builds models $omods_M(u)$ of u s.t. $\{m^+ \mid m \in omods_M(u)\}$ are the models of $u^\leq$.*

Using this lemma, we can inductively prove that each iteration of BUILDMODELGRAPH chooses a unit u without models, creates input models and then output models at u, such that all expanded models at unit u are answer sets of subprogram $u^\leq$. In this manner, the model graph is extended until no longer possible. We have the following result.

Theorem 2. *Given an evaluation graph $\mathcal{E} = (V, E)$ of a* HEX *program P,* BUILDMODELGRAPH *returns the model graph $\mathcal{M} = (M, F)$ such that $\{m_1 \bowtie \cdots \bowtie m_n \mid m_i \in omods_M(u_i),\ u_i \in V\} = \mathcal{AS}(P)$.*

5 Implementation and Experiments

The presented framework has been implemented to become the next version of the dlvhex solver: dlvhex 2.0 (http://www.kr.tuwien.ac.at/research/systems/dlvhex/). The current implementation supports dlv (http://www.dlvsystem.com/) and (for a non-disjunctive fragment of HEX) clasp+gringo (http://potassco.sourceforge.net/) as back-end ASP solvers.

In addition to the framework described above, an online model calculation algorithm has been implemented that can easily be extended to add query support. So far, the evaluation strategy PLAIN has been implemented; implementing other strategies just requires adapting legacy code to new C++ data structures. Two evaluation heuristics are implemented: the former dlvhex evaluation heuristics *H1* and a new heuristics *H2* (cf. Ex. 3). *H1* was ported for comparing dlvhex 1.x to 2.x; it splits a given program into strongly connected components and external components (which are as small as possible). The new *H2* places rules into units as follows: (i) combine rules r_1, r_2 whenever $r_1 \rightarrow s$ and $r_2 \rightarrow s$ and there is no rule t s.t. exactly one of r_1, r_2 depends on t; (ii) combine rules r_1, r_2 whenever $s \rightarrow r_1$ and $s \rightarrow r_2$ and there is no rule t s.t. t depends on exactly one of r_1, r_2; but (iii) never combine rules r, s if r contains external atoms and $r \rightarrow s$. Intuitively, *H2* builds an evaluation graph that puts all rules with external atoms and their successors into one unit, while separating rules creating input for distinct external atoms. This avoids redundant computation and joining unrelated models.

Experimental Setup and Benchmarks. A series of 6 concurrent tests were run on a Linux machine with two quad-core Intel Xeon 3GHz CPUs and 32GB RAM. The system resources were limited to a maximum of 3GB memory usage and 600 secs execution time for each run. The computation task for all experiments was to compute all answer sets of two kinds of benchmark instances:

• MCS. The first kind of benchmark instances, motivating this research, are HEX programs capturing multi-context systems (MCS)—a formalism for interlinking distributed knowledge sources with possibly nonmonotonic "bridge rules" (see [2]). Each instance consists of 5–10 guessed atoms of input and output interpretations for each of 7–9 knowledge sources, which are realized by external atoms in constraints. Most guesses

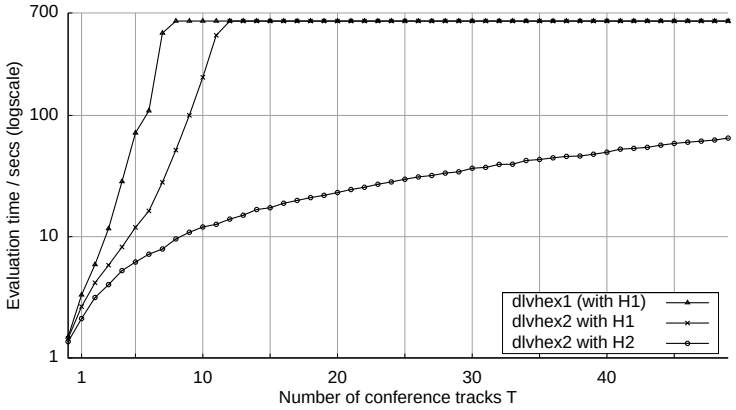


Fig. 2: REVSEL 1 ($P = 20$), out-of-memory for $T \geq 12$

are eliminated by these constraints, the remaining guesses are linked by HEX rules representing bridge rules of the modeled system. These benchmarks come in 14 different flavors (bridge rule topologies and sizes), each with 10 randomized single instances. Instances have an average of 400 models, with values ranging from 4 to ∼20,000 models.

• REVIEWER SELECTION (REVSEL). The second class of benchmark instances encode the selection of reviewers for conference papers—taking conflicts into account, some of which are encoded by external atoms. For these instances, we vary the number T of conference tracks and the number P of papers per track. The number of reviewers available for each track equals P and there is one reviewer assigned to all tracks (establishing a dependency between conference track assignments). Each paper must have 2 reviews and no reviewer gets more than 2 papers assigned. We generated conflicts such that we limit the number of overall models, as well as the number of candidate models per conference track, before checking conflicts modeled via external atoms.

We consider two special classes of reviewer selection. In REVSEL 1, we first compared the old and the new evaluation approach for a very specific program structure, as well as the old and new implementation with the ported (old) evaluation heuristics *H1*. For that we used $P = 20$ papers per conference track and varied the number of tracks T. External atoms and conflicts were configured such that all conference tracks have two solutions before evaluating constraints with external atoms, and one overall model after program evaluation. The REVSEL 2 experiment involved no external atoms: we used $T = 5$ conference tracks and varied the number of papers per track. Conflicts are generated such that there are 1-2 solutions per conference track, with a shared reviewer such that each program Q has 9 answer sets in total.

Results. It turned out that on the considered problems, the new evaluation approach outperforms the old one significantly, using less memory (sometimes exponentially less). For the MCS benchmark instances, the old approach had 34 timeouts and 83 memory outages; thus only 16% of all instances triggered some output. The average time and memory usage for successful termination was 86 seconds and 623MB, resp. Both values

have a high standard deviation. In contrast, the new approach successfully calculated all models for all instances with an average solve time of 3 seconds and an average memory usage of 32MB, both with a small standard deviation. This big improvement makes usage of HEX programs in this problem domain feasible for the first time. Note that this problem domain was originally not generated for benchmarking HEX programs, so it is not specifically geared towards showing beneficial effect of our new approach.

Results for REVSEL 1 are shown in Fig. 2: an exponential increase of runtime is visible in the old approach, compared to linear time growth of the new one. Memory usage behaves similarly. In general, increasing T causes timeouts, yet bigger T's exhaust memory. Under *H1*, dlvhex 2 acts better, which may be explained with technical improvements. As a surprising result of REVSEL 2, our divide-and-conquer approach performs better than solving Q directly with native solvers. This has been observed for both dlv and clasp as a backend. Our prototype incurs a small overhead for decomposing the program: small instances with running time < 2 sec are slower than native solvers, while big instances using *H2* were solved faster and with significantly less memory usage.

6 Discussion and Conclusion

We illustrated a new general technique for evaluating nonmonotonic logic programs with external sources of computation. Our work is clearly related to work on program modularity under stable model semantics, including, e.g., the seminal paper [13] on splitting sets, and [15, 12], which lifted them to modular programs with choice rules and disjunctive rules and allow for "symmetric splitting." An important difference is that our decomposition approach works for nonground programs and explicitly considers the possibility that modules overlap. It is tailored to efficient evaluation of arbitrary programs, rather than to facilitate module-style logic programming with declarative specifications. In this regard, it is in line with previous work on HEX program evaluation [7] and decomposition techniques for efficient grounding of ordinary programs [3].

The work presented here can be furthered in different directions. As for the prototype reasoner, a rather straightforward extension is to support brave and cautious reasoning on top of HEX programs, while incorporating constructs like aggregates or preference constraints requires more care and efforts. Regarding program evaluation, our general evaluation framework provides a basis for further optimizations that, as indicated by our experiments, are also of interest for ordinary logic programs. Indeed, the generic notions of evaluation unit, evaluation plan and model graph allow to specialize and improve our framework in different respects: first, evaluation units (which may contain duplicated constraints), can be chosen according to a proper estimate of the number of answer sets (the fewer, the better); second, evaluation plans can be chosen by ad-hoc optimization modules, which may give preference to time, space, or parallelization requirements, or to a combination of the three. Furthermore, our framework is ready to a form of coarse-grained distributed computation at the level of evaluation units (in the style of [16]): evaluation graphs naturally encode parallel evaluation plans. Independent units can in fact be evaluated in parallel, while our 'model streaming' architecture lends itself to pipelined evaluation of subsequent modules. Improving reasoning performance by decomposition has been investigated in [1], however, only wrt. monotonic logics.

As a last remark on possible optimizations, we observe that the data flow (constituted by intermediate answer sets) between evaluation units can be optimized using proper notions of model projection, such as in [9]. Model projections would tailor input data of evaluation units to necessary parts of intermediate answer sets; however, given that different units might need different parts of the same intermediate input answer set, a space-saving efficient projection technique is not straightforward.

References

1. Amir, E., McIlraith, S.: Partition-based logical reasoning for first-order and propositional theories. Artif. Intell. 162(1-2), 49–88 (2005)
2. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: AAAI'07. pp. 385–390. AAAI Press (2007)
3. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in ASP: Theory and implementation. In: ICLP'08. pp. 407–424. Springer (2008)
4. Eiter, T., Fink, M., Krennwallner, T.: Decomposition of Declarative Knowledge Bases with External Functions. In: IJCAI'09. pp. 752–758. AAAI Press (2009)
5. Eiter, T., Ianni, G., Krennwallner, T., Schindlauer, R.: Exploiting conjunctive queries in description logic programs. Ann. Math. Artif. Intell. 53(1–4), 115–152 (2008)
6. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: IJCAI'05. pp. 90–96. (2005)
7. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Effective integration of declarative rules with external evaluations for semantic-web reasoning. In: ESWC'06. pp. 273–287 (2006)
8. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. 175(1), 278–298 (2011)
9. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected boolean search problems. In: CPAIOR'09. pp. 71–86. Springer (2009)
10. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. Next Generat. Comput. 9(3–4), 365–386 (1991)
11. Hoehndorf, R., Loebe, F., Kelso, J., Herre, H.: Representing default knowledge in biomedical ontologies: Application to the integration of anatomy and phenotype ontologies. BMC Bioinf. 8(1), 377 (2007) doi:10.1186/1471-2105-8-377
12. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity Aspects of Disjunctive Stable Models. J. Artif. Intell. Res. 35, 813–857 (2009) doi:10.1613/jair.2810
13. Lifschitz, V., Turner, H.: Splitting a Logic Program. In: ICLP'94. pp. 23–38. MIT-Press (1994)
14. Nieuwenborgh, D.V., Eiter, T., Vermeir, D.: Conditional planning with external functions. In: LPNMR'07. pp. 214–227. Springer (2007)
15. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for smodels programs. Theory Pract. Log. Prog. 8(5-6), 717–761 (2008)
16. Perri, S., Ricca, F., Sirianni, M.: A parallel ASP instantiator based on DLV. In: DAMP'10. pp. 73–82. Springer (2010)
17. Polleres, A.: From SPARQL to rules (and back). In: WWW'07. pp. 787–796. ACM (2007)
18. Przymusinski, T.C.: On the Declarative Semantics of Deductive Databases and Logic Programs. In: Foundations of Deductive Databases and Logic Programming, pp. 193–216. (1988)
19. Ross, K.: Modular Stratification and Magic Sets for Datalog Programs with Negation. J. ACM 41(6), 1216–1267 (1994)
20. Schindlauer, R.: Answer-set programming for the Semantic Web. Ph.D. thesis, Vienna University of Technology (2006)
21. Zirtiloğlu, H., Yolum, P.: Ranking semantic information for e-government: complaints management. In: OBI'08. pp.1 7. No. 5, ACM (2008) doi:10.1145/1452567.1452572