A model building framework for Answer Set Programming with external computations∗

Thomas Eiter, Michael Fink

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

Giovambattista Ianni

Dipartimento di Matematica, Cubo 30B, Università della Calabria
87036 Rende (CS), Italy
(*e-mail:* ianni@mat.unical.it)

Thomas Krennwallner, Christoph Redl

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

Peter Schüller

Computer Engineering Department, Faculty of Engineering, Marmara University
Goztepe Kampusu, Kadikoy 34722, Istanbul, Turkey
(*e-mail:* peter.schuller@marmara.edu.tr)

submitted 27 January 2015; revised 24 June 2015; accepted 28 June 2015

Abstract

As software systems are getting increasingly connected, there is a need for equipping nonmonotonic logic programs with access to external sources that are possibly remote and may contain information in heterogeneous formats. To cater for this need, HEX programs were designed as a generalization of answer set programs with an API style interface that allows to access arbitrary external sources, providing great flexibility. Efficient evaluation of such programs however is challenging, and it requires to interleave external computation and model building; to decide when to switch between these tasks is difficult, and existing approaches have limited scalability in many real-world application scenarios. We present a new approach for the evaluation of logic programs with external source access, which is based on a configurable framework for dividing the non-ground program into possibly overlapping smaller parts called evaluation units. The latter will be processed by interleaving external evaluation and model building using an evaluation graph and a model graph, respectively, and by combining intermediate results. Experiments with our prototype implementation show a significant improvement compared to previous approaches. While designed for HEX-programs, the new evaluation approach may be deployed to related rule-based formalisms as well.

KEYWORDS: Answer Set Programming, Model Building, External Computation, HEX Programs

∗ This article is a significant extension of (Eiter et al. 2011) and parts of (Schüller 2012). This work has been supported by the Austrian Science Fund (FWF) Grants P24090 and P27730, and the Scientific and Technological Research Council of Turkey (TUBITAK) Grant 114E430.

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-style interface. In particular, HEX programs (Eiter et al. 2005) 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

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

might be used for obtaining pairs of URLs (X, Y), where X actually links Y on the Web, and $\&\mathit{hasHyperlink}$ is an *external predicate* construct. Besides constants (i.e., values) as above, also relational knowledge (predicate extensions) can flow from external sources to the logic program and vice versa, and recursion involving external predicates is allowed under safety conditions. This facilitates a variety of applications that require logic programs to interact with external environments, such as querying RDF sources using SPARQL (Polleres 2007), default rules on ontologies (Hoehndorf et al. 2007; Dao-Tran et al. 2009), complaint management in e-government (Zirtiloğlu and Yolum 2008), material culture analysis (Mosca and Bernini 2008), user interface adaptation (Zakraoui and Zagler 2012), multi-context reasoning (Brewka and Eiter 2007), or robotics and planning (Schüller et al. 2013; Havur et al. 2014), to mention a few.

Despite the absence of function symbols, an unrestricted use of external atoms leads to undecidability, as new constants may be introduced from the sources; in iteration, this can lead to an infinite Herbrand universe for the program. However, even under suitable restrictions like liberal domain-expansion safety (Eiter et al. 2014a) that avoid this problem, the efficient evaluation of HEX-programs is challenging, due to aspects such as nonmonotonic atoms and recursive access (e.g., in transitive closure computations).

Advanced in this regard was the work by Eiter et al. (2012), 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. Compared to previous approaches such as the one by Eiter et al. (2006), it further exploits conflict-driven techniques which were extended to external sources. A generalized notion of Splitting Set (Lifschitz and Turner 1994) was introduced by Eiter et al. (2006) 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. The subprograms are evaluated with various specific techniques, depending on their structure (Eiter et al. 2006; Schindlauer 2006). 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 a modularity property of HEX-programs based on a novel generalization of the Global Splitting Theorem (Eiter et al. 2006), which lifted the Splitting Set Theorem (Lifschitz and Turner 1994) to HEX-programs. In contrast to previous results, the new result is formulated on a *rule splitting set* comprising rules that may be non-ground, moreover it is based on rule dependencies rather than atom dependencies. This theorem allows for defining answer sets of the overall program in terms of the answer sets of program components that may be non-ground.

- Moreover, we present a generalized version of the new splitting theorem which allows for sharing constraints across the split; this helps to prune irrelevant partial models and candidates earlier than in previous approaches. As a consequence — and different from other decomposition approaches— subprograms for evaluation may overlap and also be non-maximal (resp. non-minimal).
- Based on the generalized splitting theorem, we present an evaluation framework that allows for flexible evaluation of HEX-programs. It consists of an *evaluation graph* and a *model graph*; the former captures a modular decomposition and partial evaluation order of the program, while the latter comprises for each node collections of sets of input models (which need to be combined) and output models to be passed on between components. This structure allows us to realize customized divide-and-conquer evaluation strategies. As the method works on non-ground programs, introducing new values by external calculations is feasible, as well as applying optimization based on domain splitting (Eiter et al. 2009).
- A generic prototype of the evaluation framework has been implemented which can be instantiated with different solvers for Answer Set Programming (ASP) (in our suite, with dlv and clasp). It also features *model streaming*, i.e., enumeration of the models 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.

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').

This paper, which significantly extends work in (Eiter et al. 2011) and parts of (Schüller 2012), is organized as follows. In Section 2 we present the HEX-language and consider an example to demonstrate it in an intuitive way; we will use it as a running example throughout the paper. In Section 3 we then introduce necessary restrictions and preliminary concepts that form dependency-based program evaluation. After that, we develop in Section 4 our generalized splitting theorem, which is applied in Section 5 to build a new decomposition framework. Details about the implementation and experimental results are given in Section 6. After a discussion including related work in Section 7, the paper concludes in Section 8. The proofs of all technical results are given in Online Appendix A.

2 Language Overview

In this section, we introduce the syntax and semantics of HEX-programs as far as this is necessary to explain use cases and basic modeling in the language.

2.1 HEX *Syntax*

Let $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ be mutually disjoint sets whose elements are called *constant names*, *variable names*, and *external predicate names*, respectively. Unless explicitly specified, elements from $\mathcal{X}$ (resp., $\mathcal{C}$) are denoted with first letter in upper case (resp., lower case), while elements from $\mathcal{G}$ are prefixed with '&'. Note that constant names serve both as individual and predicate names.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. An *atom* is a tuple $(Y_0, Y_1, \dots, Y_n)$, where $Y_0, \dots, Y_n$ are terms; $n \geq 0$ is the *arity* of the atom. Intuitively, Y_0 is the predicate name, and we thus also use the more familiar notation $Y_0(Y_1, \dots, Y_n)$. The atom is *ordinary* (resp. *higher-order*), if Y_0 is a constant (resp. a variable). An atom is *ground*, if all its terms are constants. Using an auxiliary predicate aux_n for each arity n, we can easily eliminate higher-order atoms by rewriting them to ordinary atoms $aux_n(Y_0, \dots, Y_n)$. We therefore assume in the rest of this article that programs have no higher-order atoms.

An *external atom* is of the form

$$\&g[Y_1, \dots, Y_n](X_1, \dots, X_m), \tag{1}$$

where $Y_1, \dots, Y_n$ and $X_1, \dots, X_m$ are two lists of terms (called *input* and *output* lists, respectively), and $\&g \in \mathcal{G}$ is an external predicate name. We assume that $\&g$ has fixed lengths $in(\&g) = n$ and $out(\&g) = m$ for input and output lists, respectively.

Intuitively, an external atom provides a way for deciding the truth value of an output tuple depending on the input tuple and a given interpretation.

Example 1
(a, b, c), $a(b, c)$, $node(X)$, and $D(a, b)$ are atoms; the first three are ordinary, where the second atom is a syntactic variant of the first, while the last atom is higher-order.

The external atom $\&reach[edge, a](X)$ may be devised for computing the nodes which are reachable in a graph represented by atoms of form $edge(u, v)$ from node a. We have for the input arity $in(\&reach) = 2$ and for the output arity $out(\&reach) = 1$. Intuitively, given an interpretation I, $\&reach[edge, a](X)$ will be true for all ground substitutions $X \mapsto b$ such that b is a node in the graph given by edge list $\{(u, v) \mid edge(u, v) \in I\}$, and there is a path from a to b in that graph.

Definition 1 (*rules and* HEX *programs*)
A *rule* r is of the form

$$\alpha_1 \vee \dots \vee \alpha_k \leftarrow \beta_1, \dots, \beta_n, not\ \beta_{n+1}, \dots, not\ \beta_m, \qquad m, k \geq 0, \tag{2}$$

where all α_i are atoms and all β_j are either atoms or external atoms. We let $H(r) = \{\alpha_1, \dots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \dots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \dots, \beta_m\}$. Furthermore, a *(*HEX*) program* is a finite set P of rules.

We denote by $const(P)$ the set of constant symbols occurring in a program P.

A rule r is a *constraint*, if $H(r) = \emptyset$ and $B(r) \neq \emptyset$; a *fact*, if $B(r) = \emptyset$ and $H(r) \neq \emptyset$; and *nondisjunctive*, if $|H(r)| \leq 1$. We call r *ordinary*, if it contains only ordinary atoms. We call a program P *ordinary* (resp., *nondisjunctive*), if all its rules are ordinary (resp., nondisjunctive). Note that facts can be disjunctive, i.e., contain multiple head atoms.

Example 2 (*Swimming Example*)
Imagine Alice wants to go for a swim in Vienna. She knows two indoor pools called Margarethenbad and Amalienbad (represented by $margB$ and $amalB$, respectively), and she knows that outdoor swimming is possible in the river Danube at two locations called Gänsehäufel and Alte Donau (denoted $gansD$ and $altD$, respectively).[1] She looks up on

[1] To keep the example simple, we assume Alice knows no other possibilities to go swimming in Vienna.

$$P^{EDB}_{swim} = \left\{ \begin{array}{l} location(ind, margB), location(ind, amalB), \\ location(outd, gansD), location(outd, altD) \end{array} \right\}$$

$$P^{IDB}_{swim} = \left\{ \begin{array}{lrl} r_1\colon & swim(ind) \vee swim(outd) & \leftarrow . \\ r_2\colon & need(inoutd, C) & \leftarrow \&rq[swim](C). \\ r_3\colon & goto(X) \vee ngoto(X) & \leftarrow swim(P), location(P, X). \\ r_4\colon & go & \leftarrow goto(X). \\ r_5\colon & need(loc, C) & \leftarrow \&rq[goto](C). \\ c_6\colon & & \leftarrow goto(X), goto(Y), X \neq Y. \\ c_7\colon & & \leftarrow not\ go. \\ c_8\colon & & \leftarrow need(X, money). \end{array} \right\}$$

Fig. 1: Program $P_{swim} = P^{EDB}_{swim} \cup P^{IDB}_{swim}$ to decide swimming location

the Web whether she needs to pay an entrance fee, and what additional equipment she will need. Finally she has the constraint that she does not want to pay for swimming.

The HEX program $P_{swim} = P^{EDB}_{swim} \cup P^{IDB}_{swim}$ shown in Figure 1 represents Alice's reasoning problem. The extensional part P^{EDB}_{swim} contains a set of facts about possible swimming locations (where ind and $outd$ are short for $indoor$ and $outdoor$, respectively). The intensional part P^{IDB}_{swim} incorporates the web research of Alice in an external computation, i.e., using an external atom of the form $\&rq[location\text{-}choice](required\text{-}resource)$, which intuitively evaluates to true iff a given $location\text{-}choice$ requires a certain $required\text{-}resource$ and represents such resources and their origin ($inoutd$, or loc) using predicate $need$. Assume Alice finds out that indoor pools in general have an admission fee, and that one also has to pay at Gänsehäufel, but not at Alte Donau. Furthermore Alice reads some reviews about swimming locations and finds out that she will need her Yoga mat for Alte Donau because the ground is so hard, and she will need goggles for Amalienbad because there is so much chlorine in the water.

We next explain the intuition behind the rules in P_{swim}: r_1 chooses indoor vs. outdoor swimming locations, and r_2 collects requirements that are caused by this choice. Rule r_3 chooses one of the indoor vs. outdoor locations, depending on the choice in r_1, and r_5 collects requirements caused by this choice. By r_4 and c_7 we ensure that some location is chosen, and by c_6 that only a single location is chosen. Finally c_8 rules out all choices that require money. Note that there is no apparent requirement for the first argument of predicate $need$, however this argument ensures, that r_2 and r_5 have different heads, which becomes important in Example 13.

The external predicate $\&rq$ has input and output arity $in(\&rq) = out(\&rq) = 1$. Intuitively $\&rq[\alpha](\beta)$ is true if a resource β is required when swimming in a place in the extension of predicate α. For example, $\&rq[swim](money)$ is true if $swim(ind)$ is true, because indoor swimming pool charge money for swimming. Note that this only gives an intuitive account of the semantics of $\&rq$ which will formally be defined in Example 4.

2.2 HEX *Semantics*

The semantics of HEX-programs (Eiter et al. 2006; Schindlauer 2006) generalizes the answer-set semantics (Gelfond and Lifschitz 1991). Let P be a HEX-program. Then the *Herbrand base* of P, denoted HB_P, is the set of all possible ground versions of atoms and external atoms occurring in P obtained by replacing variables with constants from $\mathcal{C}$. The grounding of a rule r, $grnd(r)$, is defined accordingly, and the grounding of P is given by $grnd(P) = \bigcup_{r \in P} grnd(r)$. Unless specified otherwise, $\mathcal{X}$ and $\mathcal{G}$ are implicitly given by P. Different from the 'usual' ASP setting, the set of constants $\mathcal{C}$ used for grounding a program is only partially given by the program itself; in HEX, external computations may introduce new constants that are relevant for semantics of the program.

Example 3 (*ctd.*)
In P_{swim} the external atom *&rq* can introduce constants $yogamat$ and $goggles$ which are not contained in P_{swim}, but they are relevant for computing answer sets of P_{swim}.

An *interpretation relative to* P is any subset $I \subseteq HB_P$ containing no external atoms. We say that I is a *model* of atom $a \in HB_P$, denoted $I \models a$, if $a \in I$.

With every external predicate name $\&g \in \mathcal{G}$, we associate an $(n{+}m{+}1)$-ary Boolean function (called *oracle function*) $f_{\&g}$ assigning each tuple $(I, y_1 \ldots, y_n, x_1, \ldots, x_m)$ either 0 or 1, where $n = in(\&g)$, $m = out(\&g)$, $I \subseteq HB_P$, and $x_i, y_j \in \mathcal{C}$. We say that $I \subseteq HB_P$ is a *model* of a ground external atom $a = \&g[y_1, \ldots, y_n](x_1, \ldots, x_m)$, denoted $I \models a$, if $f_{\&g}(I, y_1 \ldots, y_n, x_1, \ldots, x_m) = 1$.[2]

Note that this definition of external atom semantics is very general; indeed an external atom may depend on every part of the interpretation. Therefore we will later (Section 3.1) formally restrict external computations such that they depend only on the extension of those predicates in I which are given in the input list. All examples and encodings in this work obey this restriction.

Example 4 (*ctd.*)
The external predicate *&rq* in P_{swim} represents Alice's knowledge about swimming locations as follows: for any interpretation I and some predicate (i.e., constant) α,

$$
\begin{array}{lll}
I \models \&rq[\alpha](money) & \text{iff } f_{\&rq}(I, \alpha, money) = 1 & \text{iff } \alpha(ind) \in I \text{ or } \alpha(gansD) \in I, \\
I \models \&rq[\alpha](yogamat) & \text{iff } f_{\&rq}(I, \alpha, yogamat) = 1 & \text{iff } \alpha(altD) \in I, \text{ and} \\
I \models \&rq[\alpha](goggles) & \text{iff } f_{\&rq}(I, \alpha, goggles) = 1 & \text{iff } \alpha(amalB) \in I.
\end{array}
$$

Due to this definition of $f_{\&rq}$, it holds, e.g., that $\{swim(ind)\} \models \&rq[swim](money)$. This matches the intuition about *&rq* indicated in the previous example.

Let r be a ground rule. Then we say that

(i) I satisfies the head of r, denoted $I \models H(r)$, if $I \models a$ for some $a \in H(r)$;
(ii) I satisfies the body of r ($I \models B(r)$), if $I \models a$ for all $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$; and
(iii) I satisfies r ($I \models r$), if $I \models H(r)$ whenever $I \models B(r)$.

We say that I is a *model* of a HEX-program P, denoted $I \models P$, if $I \models r$ for all $r \in grnd(P)$. We call P *satisfiable*, if it has some model.

Definition 2 (*answer set*)
Given a HEX-program P, the *FLP-reduct* of P with respect to $I \subseteq HB_P$, denoted fP^I, is the set of all $r \in grnd(P)$ such that $I \models B(r)$. Then $I \subseteq HB_P$ is an *answer set of* P if, I is a minimal model of fP^I. We denote by $\mathcal{AS}(P)$ the set of all answer sets of P.

[2] In the implementation, Boolean functions for defining external sources are realized as plugins to the reasoner which exploit a provided interface and can be written either in Python or C++.

Example 5 (*ctd.*)
The HEX program P_{swim} with external semantics as given in the previous example has a single answer set

$$I = \{swim(outd), goto(altD), ngoto(gansD), go, need(loc, yogamat)\}.$$

(Here, and in following examples, we omit P_{swim}^{EDB} from all interpretations and answer sets.) Under I, the external atom $\&rq[goto](yogamat)$ is true and all others ($\&rq[swim](money)$, $\&rq[goto](money)$, $\&rq[swim](yogamat)$, $\ldots$) are false. Intuitively, answer set I tells Alice to take her Yoga mat and go for a swim to Alte Donau.

HEX programs (Eiter et al. 2005) are a conservative extension of disjunctive (resp., normal) logic programs under the answer set semantics: answer sets of *ordinary nondisjunctive* HEX *programs* coincide with stable models of logic programs as proposed by Gelfond and Lifschitz (1988), and answer sets of *ordinary* HEX *programs* coincide with stable models of disjunctive logic programs (Przymusinski 1991; Gelfond and Lifschitz 1991).

2.3 *Using* HEX*-Programs for Knowledge Representation and Reasoning*

While ASP is well-suited for many problems in artificial intelligence and was successfully applied to a range of applications (cf. e.g. (Brewka et al. 2011)), modern trends computing, for instance in distributed systems and the World Wide Web, require accessing other sources of computation as well. HEX-programs cater for this need by its external atoms which provide a bidirectional interface between the logic program and other sources.

One can roughly distinguish between two main usages of external sources, which we will call *computation outsourcing*, *knowledge outsourcing*, and combinations thereof. However, we emphasize that this distinction concerns the usage in an application but both are based on the same syntactic and semantic language constructs. For each of these groups we will describe some typical use cases which serve as usage patterns for external atoms when writing HEX-programs.

2.3.1 *Computation Outsourcing*

Computation outsourcing means to send the definition of a subproblem to an external source and retrieve its result. The input to the external source uses predicate extensions and constants to define the problem at hand and the output terms are used to retrieve the result, which can in simple cases also be a Boolean decision.

On-demand Constraints A special case of the latter case are on-demand constraints of type $\leftarrow \&forbidden[p_1, \ldots, p_n]()$ which eliminate certain extensions of predicates $p_1, \ldots, p_n$. Note that the external evaluation of such a constraint can also return reasons for conflicts to the reasoner in order to restrict the search space and avoid reconstruction of the same conflict (Eiter et al. 2012). This is similar to the CEGAR approach in model checking (Clarke et al. 2003) and can be helpful for reducing the size of the ground program: constraints do not need to be grounded but they are outsourced into an external atom of the above form, which then returns violated constraints as nogoods to the solver. This technique has been used for efficient planning in robotics where external atoms verify the feasibility of a 3D motion (Schüller et al. 2013).

Computations which cannot (easily) be Expressed by Rules Outsourcing computations also allows for including algorithms which cannot easily or efficiently be expressed as a logic program, e.g., because they involve floating-point numbers. As a concrete example, an artificial intelligence agent for the skills and tactics game *AngryBirds* needs to perform physics simulations (Calimeri et al. 2013). As this requires floating point computations which can practically not be done by rules as this would either come at the costs of very limited precision or a blow-up of the grounding, HEX-programs with access to an external source for physics simulations are used.

Complexity Lifting External atoms can realize computations with a complexity higher than the complexity of ordinary ASP programs. The external atom serves than as an 'oracle' for deciding subprograms. While for the purpose of complexity analysis of the formalism it is often assumed that external atoms can be evaluated in polynomial time (Faber et al. 2004)[3], as long as external sources are decidable there is no practical reason for limiting their complexity (but of course a computation with greater complexity than polynomial time lifts the complexity results of the overall formalism as well). In fact, external sources can be other ASP- or HEX-programs. This allows for encoding other formalisms of higher complexity in HEX-programs, e.g., *abstract argumentation frameworks* (Dung 1995).

2.3.2 Knowledge Outsourcing

In contrast, knowledge outsourcing refers to external sources which store information which needs to be imported, while reasoning itself is done in the logic program.

A typical example can be found in Web resources which provide information for import, e.g., *RDF triple stores* (Lassila and Swick 1999) or *geographic data* (Mosca and Bernini 2008). More advanced use cases are *multi-context systems*, which are systems of knowledge-bases (*contexts*) that are abstracted to acceptable belief sets (roughly speaking, sets of atoms) and interlinked by *bridge rules* that range across knowledge bases (Brewka and Eiter 2007); access to individual contexts has been provided through external atoms (Bögl et al. 2010). Also sensor data, as often used when planning and executing actions in an environment, is a form of knowledge outsourcing (cf. ACTHEX (Basol et al. 2010)).

2.3.3 Combinations

It is also possible to combine the outsourcing of computations and of knowledge. A typical example are logic programs with access to description logic knowledge bases (DL KBs), called *DL-programs* (Eiter et al. 2008). A DL KB does not only store information, but also provides a reasoning mechanism. This allows the logic program for formalizing queries which initiate external computations based on external knowledge and importing the results.

[3] Under this assumption, deciding the existence of an answer set of a propositional HEX-program is Σ_2^P-complete.

3 Extensional Semantics and Atom Dependencies

We now introduce additional important notions related to HEX-programs. Some of the following concepts are needed to make the formalism decidable, others prepare the basic evaluation techniques presented in later sections.

3.1 Restriction to Extensional Semantics for HEX *External Atoms*

To make HEX programs computable in practice, it is useful to restrict external atoms, such that their semantics depends only on extensions of predicates given in the input tuple (Eiter et al. 2006). This restriction is relevant for all subsequent considerations.

Syntax Each *&g* is associated with an input type signature $t_1, \dots, t_n$ such that every t_i is the type of input Y_i at position i in the input list of *&g*. A *type* is either $\mathbf{const}$ or a non-negative integer.

Consider *&g*, its type signature $t_1, \dots, t_n$, and a ground external atom *&g*$[y_1, \dots, y_n](x_1, \dots, x_m)$. Then, in this setting, the signature of *&g* enforces certain constraints on $f_{\&g}(I, y_1, \dots, y_n, x_1, \dots, x_m)$ such that its truth value depends only on

(a) the constant value of y_i whenever $t_i = \mathbf{const}$, and
(b) the extension of predicate y_i, of arity t_i, in I whenever $t_i \in \mathbb{N}$.

Note that parameters of type **const** are different from parameters of type 0. In the former case, a parameter is interpreted as a constant that is passed to the external source (essentially as string "p"), while a parameter p with a non-negative integer as type is interpreted as predicate whose extension is passed; in the special case of type 0, the extension reduces to the truth value of the propositional atom p.

Example 6 (*ctd.*)
Continuing Example 1, for *&reach*$[\mathit{edge}, a](x)$, we have $t_1 = 2$ and $t_2 = \mathbf{const}$. Therefore the truth value of *&reach*$[\mathit{edge}, a](x)$ depends on the extension of binary predicate edge, on the constant a, and on x.

Continuing Example 4, the external predicate *&rq* has $t_1 = 1$, therefore the truth value of *&rq*$[\mathit{swim}](x)$ for various x wrt. an interpretation I depends on the extension of the unary predicate swim in the input list.

Note that the truth value of an external atom with only constant input terms, i.e., $t_i = \mathbf{const}$, $1 \leq i \leq n$, is independent of I.

Semantic constraints enforced by signatures are formalized next.

Semantics Let a be a type, I be an interpretation and $p \in \mathcal{C}$. The *projection function* $\Pi_a(I, p)$ is the binary function such that $\Pi_{\mathbf{const}}(I, p) = p$ for $a = \mathbf{const}$, and $\Pi_a(I, p) = \{(p{,}x_1, \dots, x_a) \mid p\,(x_1, \dots, x_a) \in I\}$ for $a \in \mathbb{N}$. Recall that atoms $p(x_1, \dots, x_a)$ are tuples $(p, x_1, \dots, x_a)$. The codomain D_a of $\Pi_a(I, p)$ is $D_a := \mathcal{C}^{a+1}$ for $a \in \mathbb{N}$, i.e., the $a{+}1$-fold cartesian product of $\mathcal{C}$, which contains all syntactically possible atoms with a arguments; furthermore we let $D_{\mathbf{const}} := \mathcal{C}$.

Definition 3 (*extensional evaluation function*)
Let *&g* be an external predicate with oracle function $f_{\&g}$, $in(\&g) = n$, $out(\&g) = m$, and type signature $t_1, \dots, t_n$. Then the *extensional evaluation function* $F_{\&g} : D_{t_1} \times \cdots \times D_{t_n} \to 2^{\mathcal{C}^m}$ of *&g* is defined such that for every $\mathbf{a} = (a_1, \dots, a_m)$

$$\mathbf{a} \in F_{\&g}(\Pi_{t_1}(I, p_1), \dots, \Pi_{t_n}(I, p_n)) \text{ iff } f_{\&g}(I, p_1, \dots, p_n, a_1, \dots, a_m) = 1.$$

Note that $F_{\&g}$ makes the possibility of new constants in external atoms more explicit: tuples returned by $F_{\&g}$ may contain constants that are not contained in P. Furthermore, $F_{\&g}$ is well-defined only under the assertion at the beginning of this section.

Example 7 (*ctd.*)
For I from Example 5, we have $\Pi_1(I, \mathit{swim}) = \{(\mathit{swim}, \mathit{outd})\}$ and $\Pi_1(I, \mathit{goto}) = \{(\mathit{goto}, \mathit{altD})\}$. The extensional evaluation function of *&rq* is

$$\begin{aligned} F_{\&rq}(U) = \{&(\mathit{money}) \mid (X, \mathit{ind}) \in U \text{ or } (X, \mathit{gansD}) \in U\} \cup \\ &\{(\mathit{yogamat}) \mid (X, \mathit{altD}) \in U\} \cup \{(\mathit{goggles}) \mid (X, \mathit{amalB}) \in U\} \end{aligned}$$

Observe that none of the constants $\mathit{yogamat}$ and $\mathit{goggles}$ occurs in P (we have that $\mathit{const}(P) = \{\mathit{swim}, \mathit{goto}, \mathit{ngoto}, \mathit{need}, \mathit{go}, \mathit{inoutd}, \mathit{loc}, \mathit{ind}, \mathit{outd}, \mathit{amalB}, \mathit{gansD}, \mathit{altD}, \mathit{margB}, \mathit{money}, \mathit{location}\}$). These constants are introduced by the external atom semantics. Note that (money) is a unary tuple, as *&rq* has a unary output list.

3.2 Atom Dependencies

To account for dependencies between heads and bodies of rules is a common approach for realizing semantics of ordinary logic programs, as done, e.g., by means of the notions of *stratification* and its refinements like *local stratification* (Przymusinski 1988) or *modular stratification* (Ross 1994), or by *splitting sets* (Lifschitz and Turner 1994). In HEX programs, head-body dependencies are not the only possible source of predicate interaction. Therefore new types of (non-ground) dependencies were considered by Eiter et al. (2006) and Schindlauer (2006). In the following we recall these definitions but slightly reformulate and extend them, to prepare for the following sections where we lift atom dependencies to rule dependencies.

In contrast to the traditional notion of dependency, which in essence hinges on propositional programs, we must consider non-ground atoms; such atoms a and b clearly depend on each other if they unify, which we denote by $a \sim b$.

For analyzing program properties it is relevant whether a dependency is positive or negative. Whether the value of an external atom a depends on the presence of an atom b in an interpretation I depends in turn on the oracle function $f_{\&g}$ that is associated with the external predicate *&g* of a. Depending on other atoms in I, in some cases the *presence* of b might make a true, in some cases its *absence*. Therefore we will not speak of *positive* and *negative* dependencies, as by Eiter et al. (2011), but more adequately of *monotonic* and *nonmonotonic* dependencies, respectively.[4]

[4] Note that anti-monotonicity (i.e., a larger input of an external atom can only make the external atom false, but never true) could be a third useful distinction that was exploited in (Eiter et al. 2012). We here only distinguish monotonic from nonmonotonic external atoms and classify antimonotonic external atoms as nonmonotonic.

Definition 4
An external predicate *&g* is *monotonic*, if for all interpretations I, I' such that $I \subseteq I'$ and all tuples $\mathbf{X}$ of constants, $f_{\&g}(I, \mathbf{X}) = 1$ implies $f_{\&g}(I', \mathbf{X}) = 1$; otherwise *&g* is *nonmonotonic*. Furthermore, a ground external atom a is *monotonic*, if for all interpretations I, I' such that $I \subseteq I'$ we have $I \models a$ implies $I' \models a$; a non-ground external atom is *monotonic*, if each of its ground instances is monotonic.

Clearly, each external atom that involves a monotonic external predicates is monotonic, but not vice versa; thus monotonicity of external atoms is more fine-grained. In the following formal definitions, for simplicity we only consider external predicate monotonicity and disregard external atom monotonicity. However the extension to arbitrary monotonic external atoms is straightforward.

Example 8 (*ctd.*)
Consider $F_{\&rq}(U)$ in Example 7: adding tuples to U cannot remove tuples from $F_{\&rq}(U)$, therefore *&rq* is a monotonic external predicate.

Next we define relations for dependencies from external atoms to other atoms.

Definition 5 (*External Atom Dependencies*)
Let P be a HEX program, let $a = \&g[X_1, \dots, X_k](\mathbf{Y})$ in P be an external atom with the type signature $t_1, \dots, t_k$ and let $b = p(\mathbf{Z})$ be an atom in the head of a rule in P. Then a *depends external monotonically (resp., nonmonotonically) on* b, denoted $a \rightarrow^e_m b$ (resp., $a \rightarrow^e_n b$), if *&g* is monotonic (resp., nonmonotonic), and for some $i \in \{1, \dots, k\}$ we have that $\mathbf{Z}$ has arity $t_i \in \mathbb{N}$and $X_i = p$. We define that $a \rightarrow^e b$ if $a \rightarrow^e_m b$ or $a \rightarrow^e_n b$.

Example 9 (*ctd.*)
In our example we have the three external dependencies $\&rq[swim](C) \rightarrow^e_m swim(ind)$, $\&rq[swim](C) \rightarrow^e_m swim(outd)$, and $\&rq[goto](C) \rightarrow^e_m goto(X)$.

As in ordinary ASP, atoms in HEX programs may depend on each other because of rules in the program.

Definition 6
For a HEX-program P and atoms α, β occurring in P, we say that

(a) α depends monotonically on β ($\alpha \rightarrow_m \beta$), if one of the following holds:
   (i) some rule $r \in P$ has $\alpha \in H(r)$ and $\beta \in B^+(r)$;
   (ii) there are rules $r_1, r_2 \in P$ such that $\alpha \in B(r_1)$, $\beta \in H(r_2)$, and $\alpha \sim \beta$; or
   (iii) some rule $r \in P$ has $\alpha \in H(r)$ and $\beta \in H(r)$.
(b) α depends nonmonotonically on β ($\alpha \rightarrow_n \beta$), if there is some rule $r \in P$ such that $\alpha \in H(r)$ and $\beta \in B^-(r)$.

Note that combinations of Definitions 5 and 6 were already introduced by Schindlauer (2006) and Eiter et al. (2009); however these papers represent nonmonotonicity of external atoms within rule body dependencies and use a single ‘external dependency’ relation that does not contain information about monotonicity. In contrast, we represent nonmonotonicity of external atoms where it really happens, namely in dependencies from external atoms to

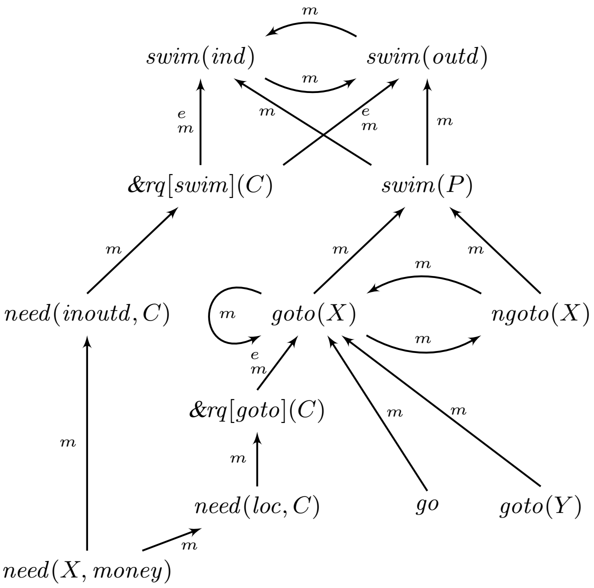

Fig. 2: Atom dependency graph of running example P_{swim}.

ordinary atoms. We therefore obtain a simpler dependency relation between rule bodies and heads.

We say that atom α *depends* on atom β, denoted $\alpha \rightarrow \beta$, if either $\alpha \rightarrow_m \beta$, $\alpha \rightarrow_n \beta$, or $\alpha \rightarrow^e \beta$; that is, $\rightarrow$ is the union of the relations $\rightarrow_m$, $\rightarrow_n$, and $\rightarrow^e$.

We next define the atom dependency graph.

Definition 7
For a HEX-program P, the *atom dependency graph* $ADG(P) = (V_A, E_A)$ of P has as vertices V_A the (possibly non-ground) atoms occurring in non-facts of P and as edges E_A the dependency relations $\rightarrow_m$, $\rightarrow_n$, $\rightarrow^e_m$, and $\rightarrow^e_n$ between them in P.

Example 10 (*ctd.*)
Figure 2 shows $ADG(P_{swim})$. Recall that c_7 is '$\leftarrow$ *not go*'. Note that the nonmonotonic body literal in c_7 does not show up as a nonmonotonic dependency, as c_7 has no head atoms. (The rule dependency graph in Section 4 will make this negation apparent.)

Next we use the dependency notions to define safety conditions on HEX programs.

3.3 Safety Restrictions

To make reasoning tasks on HEX programs decidable (or more efficiently computable), the following potential restrictions were formulated.

Rule safety This is a restriction well-known in logic programming, and it is required to ensure finite grounding of a non-ground program. A rule is safe, if all its variables are safe, and a variable is safe if it is contained in a positive body literal. Formally a rule r is safe iff variables in $H(r) \cup B^-(r)$ are a subset of variables in $B^+(r)$.

Domain-expansion safety In an ordinary logic program P, we usually assume that the set of constants $\mathcal{C}$ is implicitly given by P. In a HEX program, external atoms may invent new constant values in their output tuples. We therefore must relax this to '$\mathcal{C}$ is countable and partially given by P', as shown by the following example.

Example 11
In the Swimming Example, grounding P_{swim} with $const(P_{swim})$ is not sufficient. Further constants 'generated' by external atoms must be considered. For example $yogamat \notin const(P_{swim})$ and $I \models \&rq[goto](yogamat)$, hence we must ground

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

with $C = yogamat$ to obtain the correct answer set.

Therefore grounding P with $const(P)$ can lead to incorrect results. Hence we want to obtain new constants during evaluation of external atoms, and we must use these constants to evaluate the remainder of a given HEX program. However, to ensure decidability, this process of obtaining new constants must always terminate.

Hence, we require programs to be *domain-expansion safe* (Eiter et al. 2006): there must not be a cyclic dependency between rules and external atoms such that an input predicate of an external atom depends on a variable output of that same external atom, if the variable is not guarded by a domain predicate.

With HEX we need the usual notion of rule safety, i.e., a syntactic restriction which ensures that each variable in a rule only has a finite set of relevant constants for grounding.

We first recall the definition of safe variables and safe rules for HEX.

Definition 8 (*Def. 5 by Eiter et al. (2006)*)
The *safe variables* of a rule r is the smallest set of variables X that occur either (i) in some ordinary atom $\beta \in B^+(r)$, or (ii) in the output list $\mathbf{X}$ of an external atom $\&g[Y_1, \dots, Y_n](\mathbf{X})$ in $B^+(r)$ where all $Y_1, \dots, Y_n$ are safe. A rule r *is safe*, if each variable in r is safe.[5]

However, safety alone does not guarantee finite grounding of HEX programs, because an external atom might create new constants, i.e., constants not part of the program itself, in its output list (see Example 7). These constants can become part of the extension of an atom in the rule head, and by grounding and evaluation of other rules become part of the extension of a predicate which is an input to the very same external atom.

Example 12 (*adapted from Schindlauer (2006)*)
The following HEX program is safe according to Definition 8 and nevertheless cannot be finitely grounded:

$$\begin{aligned} source(\text{“}\mathtt{http://some\_url}\text{”}) &\leftarrow . \\ url(X) &\leftarrow \&rdf[source](X, \text{“}rdf{:}subClassOf\text{”}, C). \\ source(X) &\leftarrow url(X). \end{aligned}$$

Suppose the $\&rdf[source](S, P, O)$ atom retrieves all triples (S, P, O) from all RDF triplestores specified in the extension of $source$, and suppose that each triplestore contains a

[5] This is stated by Eiter et al. (2006) as 'if each variable appearing in a negated atom and in any input list is safe, and variables appearing in $H(r)$ are safe', which is equivalent.

triple with a URL S that does not show up in another triplestore. As a result, all these URLs are collected in the extension of source which leads to even more URLs being retrieved and a potentially infinite grounding.

However, we could change the rule with the external atom to

$$\mathit{url}(X) \leftarrow \&\mathit{rdf}[\mathit{source}](X, \text{“}\mathit{rdf}{:}\mathit{subClassOf}\text{”}, C), \mathit{limit}(X) \tag{3}$$

and add an appropriate set of limit facts. This addition of a range predicate $\mathit{limit}(X)$ which does not depend on the external atom output ensures a finite grounding.

To obtain a syntactic restriction that ensures finite grounding for HEX, so called *strong safety* has been introduced for the HEX programs (Eiter et al. 2006). Intuitively, this concept requires all output variables of cyclic external atoms (using the dependency notion from Definition 7) to be bounded by ordinary body atoms of the same rule which are not part of the cycle. However, this condition is unnecessarily restrictive, and therefore, the extensible notion of *liberal domain-expansion safety (lde-safety)* was introduced by Eiter et al. (2014a), which we will use in the following. For the purpose of this article, we may omit the formal details of lde-safety (see Eiter et al. (2014a) and Online Appendix D for an outline); it is sufficient to know that every lde-safe program has a finite grounding that has the same answer sets as the original program.

4 Rule Dependencies and Generalized Rule Splitting Theorem

In this section, we first introduce a new notion of dependencies in HEX-programs, namely between non-ground *rules* in a program (Section 4.1). Based on this notion, we then present a modularity property of HEX-programs that allows us to obtain answer sets of a program from the answer sets of its components (Section 4.2). The property is formulated as a splitting theorem based on dependencies among rules and lifts a similar result for dependencies among atoms, viz. the Global Splitting Theorem (Eiter et al. 2006), to this setting, and it generalizes and improves it. This result is exploited in a more efficient HEX-program evaluation algorithm, which we show in Section 5.

4.1 Rule Dependencies

We define rule dependencies as follows.

Definition 9 (*Rule dependencies*)
Let P be a program and a, b atoms occurring in distinct rules $r, s \in P$. Then r depends on s according to the following cases:

(i) if $a \sim b$, $a \in B^+(r)$, and $b \in H(s)$, then $r \rightarrow_m s$;
(ii) if $a \sim b$, $a \in B^-(r)$, and $b \in H(s)$, then $r \rightarrow_n s$;
(iii) if $a \sim b$, $a \in H(r)$, and $b \in H(s)$, then both $r \rightarrow_m s$ and $s \rightarrow_m r$;
(iv) if $a \rightarrow^e b$, $a \in B(r)$ is an external atom, and $b \in H(s)$, then
- $r \rightarrow_m s$ if $a \in B^+(r)$ and $a \rightarrow^e_m b$, and
- $r \rightarrow_n s$ otherwise.

Intuitively, conditions (i) and (ii) reflect the fact that the applicability of a rule r depends on the applicability of a rule s with a head that unifies with a literal in the body of rule

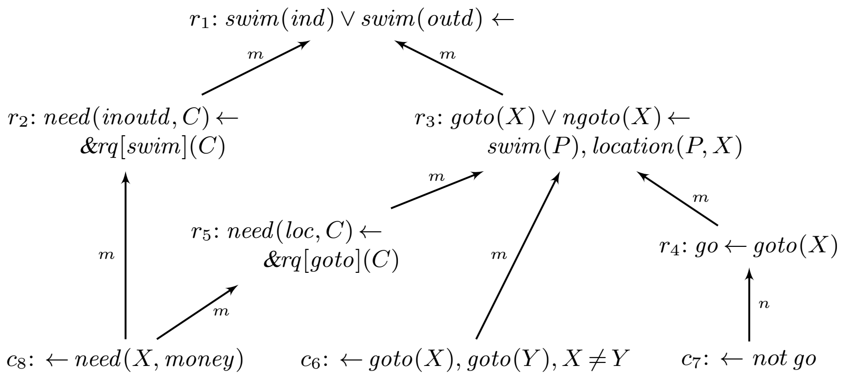
r_1: *swim*(*ind*) ∨ *swim*(*outd*) ←
r_2: *need*(*inoutd*, C) ← &*rq*[*swim*](C)
r_3: *goto*(X) ∨ *ngoto*(X) ← *swim*(P), *location*(P, X)
r_5: *need*(*loc*, C) ← &*rq*[*goto*](C)
r_4: *go* ← *goto*(X)
c_8: ← *need*(X, *money*)
c_6: ← *goto*(X), *goto*(Y), $X \neq Y$
c_7: ← *not go*
m
m
m
m
m
m
m
m
n

Fig. 3: Rule dependency graph of running example P_{swim}.

r; condition (iii) exists because r and s cannot be evaluated independently if they share a common head atom (e.g., $u \vee v \leftarrow$ cannot be evaluated independently from $v \vee w \leftarrow$); and (iv) defines dependencies due to predicate inputs of external atoms.

In the sequel, we let $\rightarrow_{m,n} = \rightarrow_m \cup \rightarrow_n$ be the union of monotonic and nonmonotonic rule dependencies. We next define graphs of rule dependencies.

Definition 10
Given a HEX-program P, the *rule dependency graph* $DG(P) = (V_D, E_D)$ of P is the labeled graph with vertex set $V_D = P$ and edge set $E_D = \rightarrow_{m,n}$.

Example 13 (*ctd.*)
Figure 3 depicts the rule dependency graph of our running example. According to Definition 9, we have the following rule dependencies in P^{IDB}_{swim}:

- due to (i) we have $r_3 \rightarrow_m r_1$, $r_4 \rightarrow_m r_3$, $c_6 \rightarrow_m r_3$, $c_8 \rightarrow_m r_2$, and $c_8 \rightarrow_m r_5$;
- due to (ii) we have $c_7 \rightarrow_n r_4$;
- due to (iii) we have no dependencies; and
- due to (iv) we have $r_2 \rightarrow_m r_1$ and $r_5 \rightarrow_m r_3$.

Note that if we would omit the first argument of predicate *need*, we would have in addition $r_2 \rightarrow_m r_5$ and $r_5 \rightarrow_m r_2$ due to (iii). Also note that &*rq* is monotonic (see Example 8).

4.2 Splitting Sets and Theorems

Splitting sets are a notion that allows for describing how a program can be decomposed into parts and how semantics of the overall program can be obtained from semantics of these parts in a divide-and-conquer manner.

We lift the original HEX splitting theorem (Eiter et al. 2006, Theorem 2) and the according definitions of global splitting set, global bottom, and global residual (Eiter et al. 2006, Definitions 8 and 9) to our new definition of dependencies among rules.

A *rule splitting set* is a part of a (non-ground) program that does not depend on the rest of the program. This corresponds in a sense with global splitting sets by Eiter et al. (2006).

Definition 11 (*Rule Splitting Set*)
A *rule splitting set* R for a HEX-program P is a set $R \subseteq P$ of rules such that whenever $r \in R$, $s \in P$, and $r \rightarrow_{m,n} s$, then $s \in R$ holds.

Example 14 (*ctd.*)
The following are some rule splitting sets of P_{swim}: $\{r_1\}$, $\{r_1, r_2\}$, $\{r_1, r_3\}$, $\{r_1, r_2, r_3\}$, $\{r_1, r_2, r_3, r_5, c_8\}$. The set $R = \{r_1, r_2, c_8\}$ is not a rule splitting set, because $c_8 \rightarrow_m r_5$ but $r_5 \notin R$.

Because of possible constraint duplication, we no longer partition the input program, and the customary notion of splitting set, bottom, and residual, is not appropriate for sharing constraints between bottom and residual. Instead, we next define a *generalized bottom* of a program, which splits a non-ground program into two parts which may share certain constraints.

Definition 12 (*Generalized Bottom*)
Given a rule splitting set R of a HEX-program P, a *generalized bottom* B of P wrt. R is a set B with $R \subseteq B \subseteq P$ such that all rules in $B \setminus R$ are constraints that do not depend nonmonotonically on any rule in $P \setminus B$.

Example 15 (*ctd.*)
A rule splitting set R of P_{swim} (e.g., those given in Example 14) is also a generalized bottom of P_{swim} wrt. R. The set $\{r_1, r_2, c_8\}$ is not a rule splitting set, but it is a generalized bottom of P_{swim} wrt. the rule splitting set $\{r_1, r_2\}$, as c_8 is a constraint that depends only monotonically on rules in $P_{swim} \setminus \{r_1, r_2, c_8\}$.

Next, we describe how interpretations of a generalized bottom B of a program P lead to interpretations of P without re-evaluating rules in B. Intuitively, this is a relaxation of the previous non-ground HEX splitting theorem: a constraint may be put both in the bottom and in the residual if it has no nonmonotonic dependencies to the residual. The benefit of such constraint sharing is a smaller number of answer sets of the bottom, and hence of fewer evaluations of the residual program.

Notation. For any set I of ground ordinary atoms, we denote by $facts(I)$ the corresponding set of ground facts; furthermore, for any set P of rules, we denote by $gh(P)$ the set of ground head atoms occurring in $grnd(P)$.

Theorem 1 (*Splitting Theorem*)
Given a HEX-program P and a rule splitting set R of P, $M \in \mathcal{AS}(P)$ iff $M \in \mathcal{AS}(P \setminus R \cup facts(X))$ with $X \in \mathcal{AS}(R)$.

Using the definition of generalized bottom, we generalize the above theorem.

Theorem 2 (*Generalized Splitting Theorem*)
Let P be a HEX-program, let R be a rule splitting set of P, and let B be a generalized bottom of P wrt. R. Then

$$M \in \mathcal{AS}(P) \text{ iff } M \in \mathcal{AS}(P \setminus R \cup facts(X)) \text{ where } X \in \mathcal{AS}(B).$$

Note that $B \setminus R$ contains shareable constraints that are used twice in the Generalized Splitting Theorem, viz. in computing X and in computing M.

The Generalized Splitting Theorem is useful for early elimination of answer sets of the bottom thanks to constraints which depend on it but also on rule heads outside the bottom. Such constraints can be shared between the bottom and the remaining program.

Example 16 (*ctd.*)

We apply Theorems 1 and 2 to P_{swim} and compare them. Using the rule splitting set $\{r_1, r_2\}$, we can obtain $\mathcal{AS}(P_{swim})$ by first computing $\mathcal{AS}(\{r_1, r_2\}) = \{I_1, I_2\}$ where $I_1 = \{swim(ind), need(inoutd, money)\}$, $I_2 = \{swim(outd)\}$, and by then using Theorem 1: $X \in \mathcal{AS}(P_{swim})$ iff it holds that $X \in \mathcal{AS}(\{r_3, r_4, r_5, c_6, c_7, c_8\} \cup facts(I_1))$ or $X \in \mathcal{AS}(\{r_3, r_4, r_5, c_6, c_7, c_8\} \cup facts(I_2))$. Note that the computation with I_1 yields no answer set, as $need(inoutd, money) \in I_1$ satisfies the body of c_8 and 'kills' any model candidate. In contrast, if we use the generalized bottom $\{r_1, r_2, c_8\}$, we have $\mathcal{AS}(\{r_1, r_2, c_8\}) = \big\{\{swim(outd)\}\big\}$ and can use Theorem 2 to obtain $\mathcal{AS}(P_{swim})$ with only one further answer set computation: $X \in \mathcal{AS}(P_{swim})$ iff $X \in \mathcal{AS}(\{r_3, r_4, r_5, c_6, c_7, c_8\} \cup \{swim(outd) \leftarrow\})$. Note that we use c_8 in both computations, i.e., c_8 is shared between the generalized bottom and the remaining computation.

Armed with the results of this section, we proceed to program evaluation in the next section. A discussion of the new splitting theorems that compares them to previous related theorems and argues for their advantage is given in Section 7.1.

5 Decomposition and Evaluation Techniques

We now introduce our new HEX evaluation framework, which is based on selections of sets of rules of a program that we call *evaluation units* (or briefly *units*).

The traditional HEX evaluation algorithm (Eiter et al. 2006) uses a dependency graph over (non-ground) atoms, and gradually evaluates sets of rules (the 'bottoms' of a program) that are chosen based on this graph. In contrast our new evaluation algorithm exploits the rule-based modularity results for HEX-programs in Section 4.

While previously a constraint can only kill models once all its dependencies on rules are fulfilled, the new algorithm increases evaluation efficiency by sharing non-ground constraints, such that they may kill models earlier; this is safe if all their nonmonotonic dependencies are fulfilled. Moreover, units no longer must be maximal. Instead, we require that partial models of units, 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.

In the following, we first define a decomposition of a HEX-program into evaluation units that are organized in an evaluation graph (Section 5.1). Then we define an interpretation graph which contains input and output interpretations of each evaluation unit (Section 5.2). We next extend this definition to answer set graphs which are related with answer sets of the program (Section 5.3). Finally Section 5.4 uses these definitions in an algorithm for enumerating answer sets of the HEX-program.

5.1 Evaluation Graph

Using rule dependencies, we next define the notion of evaluation graph on evaluation units. We then relate evaluation graphs to splitting sets (Lifschitz and Turner 1994) and show how to use them to evaluate HEX-programs by evaluating units and combining the results.

We define evaluation units as follows.

Definition 13
An *evaluation unit* (in short 'unit') is any lde-safe HEX-program.

The formal definition of lde-safety (see Online Appendix D and Eiter et al. (2014a)) is not crucial here, merely the property that a unit has a finite grounding with the same answer sets as the original unit which can be effectively computed; lde-safe HEX-programs are the most general class of HEX-programs with this property and computational support.

An important point of the notion of evaluation graph is that rule dependencies $r \rightarrow_x s$ lead to different edges, i.e., unit dependencies, depending on the dependency type $x \in \{n, m\}$ and whether r resp. s is a constraint; constraints cannot (directly) make atoms true, hence they can be shared between units in certain cases, while sharing non-constraints could violate modularity.

Given a rule $r \in P$ and a set U of units, we denote by $U|_r = \{u \in U \mid r \in u\}$ the set of units that contain rule r.

Definition 14 (*Evaluation graph*)
An *evaluation graph* $\mathcal{E} = (U, E)$ of a program P is a directed acyclic graph whose vertices U are evaluation units and which fulfills the following properties:

(a) $P = \bigcup_{u \in U} u$, i.e., every rule $r \in P$ is contained in at least one unit;
(b) every non-constraint $r \in P$ is contained in exactly one unit, i.e., $\left|U|_r\right| = 1$;
(c) for each nonmonotonic dependency $r \rightarrow_n s$ between rules r, $s \in P$ and for all $u \in U|_r$, $v \in U|_s$, $u \neq v$, there exists an edge $(u, v) \in E$ (intuitively, nonmonotonic dependencies between rules have corresponding edges everywhere in $\mathcal{E}$); and
(d) for each monotonic dependency $r \rightarrow_m s$ between rules r, $s \in P$, there exists some $u \in U|_r$ such that E contains all edges (u, v) with $v \in U|_s$ for $v \neq u$ (intuitively, for each rule r there is (at least) one unit in $\mathcal{E}$ where all monotonic dependencies from r to other rules have corresponding outgoing edges in $\mathcal{E}$).

We remark that Eiter et al. (2011) and Schüller (2012) defined evaluation units as *extended pre-groundable* HEX-programs; later, Redl (2014) and Eiter et al. (2014a) defined *generalized evaluation units* as lde-safe HEX-programs, which subsume *extended pre-groundable* HEX-programs, and *generalized evaluation graphs* on top as in Definition 14. As more the grounding properties of units matter than the precise fragment, we dropped here 'generalized' to avoid complex terminology.

As a non-constraint can occur only in a single unit, the above definition implies that all dependencies of non-constraints have corresponding edges in $\mathcal{E}$, which is formally expressed in the following proposition.

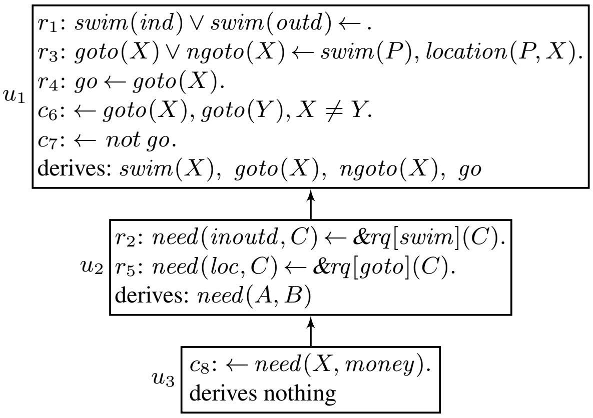
u_1
r_1: $swim(ind) \vee swim(outd) \leftarrow .$
r_3: $goto(X) \vee ngoto(X) \leftarrow swim(P), location(P, X).$
r_4: $go \leftarrow goto(X).$
c_6: $\leftarrow goto(X), goto(Y), X \neq Y.$
c_7: $\leftarrow$ $not\ go.$
derives: $swim(X),\ goto(X),\ ngoto(X),\ go$
u_2
r_2: $need(inoutd, C) \leftarrow \&rq[swim](C).$
r_5: $need(loc, C) \leftarrow \&rq[goto](C).$
derives: $need(A, B)$
u_3
c_8: $\leftarrow need(X, money).$
derives nothing

Fig. 4: Evaluation graph $\mathcal{E}_1$ for running example HEX program P_{swim}.

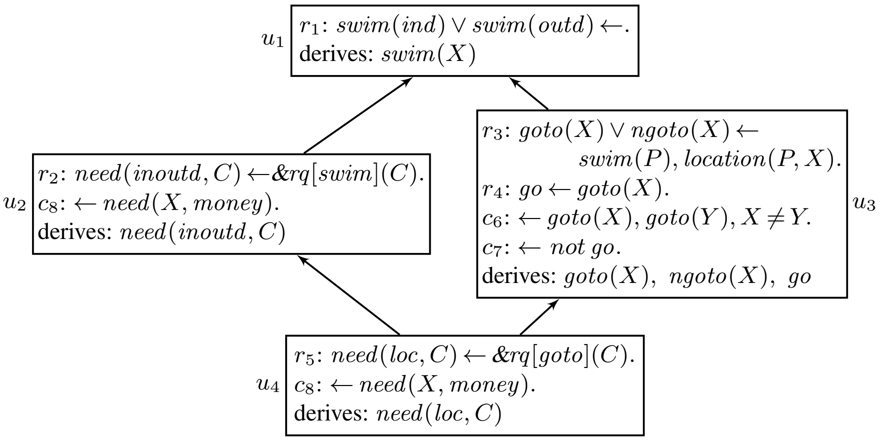
u_1
r_1: $swim(ind) \vee swim(outd) \leftarrow .$
derives: $swim(X)$
u_2
r_2: $need(inoutd, C) \leftarrow \&rq[swim](C).$
c_8: $\leftarrow need(X, money).$
derives: $need(inoutd, C)$
u_3
r_3: $goto(X) \vee ngoto(X) \leftarrow$
$swim(P), location(P, X).$
r_4: $go \leftarrow goto(X).$
c_6: $\leftarrow goto(X), goto(Y), X \neq Y.$
c_7: $\leftarrow$ $not\ go.$
derives: $goto(X),\ ngoto(X),\ go$
u_4
r_5: $need(loc, C) \leftarrow \&rq[goto](C).$
c_8: $\leftarrow need(X, money).$
derives: $need(loc, C)$

Fig. 5: Evaluation graph $\mathcal{E}_2$ for running example HEX program P_{swim}.

Proposition 1
Let $\mathcal{E} = (U, E)$ be an evaluation graph of a program P, and assume $r \rightarrow_{m,n} s$ is a dependency between a non-constraint $r \in P$ and a rule $s \in P$. Then $\{(u, v) \mid u \in U|_r, v \in U|_s\} \subseteq E$ holds.

Example 17 (*ctd.*)
Figures 4 and 5 show two possible evaluation graphs for our running example. The evaluation graph $\mathcal{E}_1$ contains every rule of P_{swim} in exactly one unit. In contrast, $\mathcal{E}_2$ contains c_8 both in u_2 and in u_4. Condition (d) of Definition 14 is particularly interesting for these two graphs; it is fulfilled as follows. Graph $\mathcal{E}_1$ can be obtained by contracting rules in the rule dependency graph $DG(P_{swim})$ into units, i.e., $\mathcal{E}_1$ is a (graph) minor of $DG(P_{swim})$ and therefore all rule dependencies are realized as unit dependencies and Conditions (c) and (d) are satisfied. In contrast, $\mathcal{E}_2$ is not a minor of $DG(P_{swim})$ because dependency $c_8 \rightarrow_m r_5$ is not realized as a dependency from u_2 to u_4. Nonetheless, all dependencies from c_8 are realized at u_4 and thus $\mathcal{E}_2$ conforms with condition (d), which merely requires that rule

dependencies have edges corresponding to all monotonic rule dependencies at *some* unit of the evaluation graph.

Evaluation graphs have the important property that partial models of evaluation units do not intersect, i.e., evaluation units do not mutually depend on each other. This is achieved by acyclicity and because rule dependencies are covered in the graph.

In fact, due to acyclicity, mutually dependent rules of a program are contained in the same unit; thus each strongly connected component of the program's dependency graph is fully contained in a single unit. Furthermore, a unit can have in its rule heads only atoms that do not unify with atoms in the rule heads of other units, as rules which have unifiable heads mutually depend on one another. This ensures that under any grounding, the following property holds.

Proposition 2 (*Disjoint unit outputs*)
Let $\mathcal{E} = (U, E)$ be an evaluation graph of a program P. Then for each distinct units $u_1, u_2 \in U$, it holds that $gh(u_1) \cap gh(u_2) = \emptyset$.[6]

Example 18 (*ctd.*)
Figures 4 and 5 show for each unit which atoms can become true due to rule heads in them, denoted as 'derived' atoms. Observe that both graphs have strictly non-intersecting atoms in rule heads of distinct units.

As units of evaluation graphs can be arbitrary lde-safe programs, we clearly have the following property.

Proposition 3
For every lde-safe HEX program P, some evaluation graph $\mathcal{E}$ exists.

Indeed, we can simply put P into a single unit to obtain a valid evaluation graph. Thus the HEX evaluation approach based on evaluation graphs is applicable to all domain-expansion safe HEX programs.

5.1.1 Evaluation Graph Splitting

We next show that units and their predecessors in an evaluation graph correspond to generalized bottoms. We then use this property to formulate an algorithm for unit-based, efficient evaluation of HEX-programs.

Given an evaluation graph $\mathcal{E} = (U, E)$, we write $u < w$, if a path from u to w exists in $\mathcal{E}$, and $u \leq w$ if either $u < w$ or $u = w$.

For a unit $u \in U$, we denote by $preds_{\mathcal{E}}(u) = \{v \in U \mid (u, v) \in E\}$ the set of units on which u (directly) depends and by $u^{<} = \bigcup_{w \in U, u < w} w$ the set of rules in all units on which u transitively depends; furthermore, we let $u^{\leq} = u^{<} \cup u$. Note that for a leaf unit u (i.e., u has no predecessors) we have $preds_{\mathcal{E}}(u) = u^{<} = \emptyset$ and $u^{\leq} = u$.

[6] See page 16 for the definition of notation $gh(P)$.

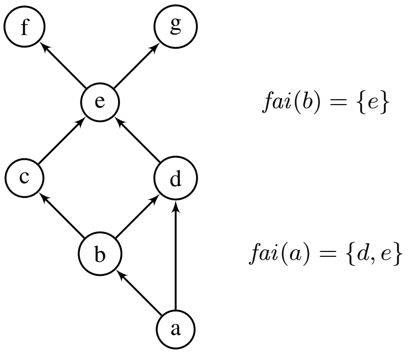

Fig. 6: First Ancestor Intersection units (FAIs) in an evaluation graph.

Theorem 3
For every evaluation graph $\mathcal{E} = (U, E)$ of a HEX-program Q and unit $u \in U$, it holds that $u^{<}$ is a generalized bottom of $u^{\leq}$ wrt. $R = \{r \in u^{<} \mid H(r) \neq \emptyset\}$.

Example 19 (*ctd.*)
In $\mathcal{E}_1$, $u_2^{<} = u_1$ and $u_2^{\leq} = u_1 \cup u_2$ and $u_2^{<}$ is a generalized bottom of $u_2^{\leq}$ wrt. $R = \{r_1, r_3, r_4\}$. In $\mathcal{E}_2$, we have $u_4^{<} = u_1 \cup u_2 \cup u_3$ and $u_4^{\leq} = P_{swim}$ and $u_4^{<}$ is a generalized bottom of P_{swim} wrt. $R = \{r_1, r_2, r_3, r_4\}$. We can verify this on Definition 12: we have $P = P_{swim}$, $B = u_4^{<} = \{r_1, r_2, r_3, r_4, c_6, c_7, c_8\}$, and R as above. Then $R \subseteq B \subseteq P$, and furthermore $B \setminus R = \{c_6, c_7, c_8\}$ consists of constraints none of which depends nonmonotonically on a rule in $P \setminus B = \{r_5\}$.

Theorem 4
Let $\mathcal{E} = (U, E)$ be an evaluation graph of a HEX-program Q and $u \in U$. Then for every unit $u' \in \mathit{preds}_{\mathcal{E}}(u)$, it holds that $u'^{\leq}$ is a generalized bottom of the subprogram $u^{<}$ wrt. the rule splitting set $R = \{r \in u'^{\leq} \mid H(r) \neq \emptyset\}$.

Example 20 (*ctd.*)
In $\mathcal{E}_1$, we have $u_1 \in \mathit{preds}_{\mathcal{E}_1}(u_2)$; hence $u_1^{\leq} = u_1$ is by Theorem 4 a generalized bottom of $u_2^{<} = u_1$ wrt. $R = \{r_1, r_3, r_4\}$. Furthermore, $u_2 \in \mathit{preds}_{\mathcal{E}_1}(u_3)$ and hence $u_2^{\leq} = u_1 \cup u_2$ is a generalized bottom of $u_3^{<} = u_1 \cup u_2$ wrt. $R = \{r_1, r_2, r_3, r_4, r_5\}$. The case of $\mathcal{E}_2$ and u_4 is less clear. We have $u_2 \in \mathit{preds}_{\mathcal{E}_2}(u_4)$, thus by Theorem 4 $u_2^{\leq} = u_1 \cup u_2 = \{r_1, r_2, c_8\}$ is a generalized bottom of $u_4^{<} = u_1 \cup u_2 \cup u_3$ wrt. $R = \{r_1, r_2\}$. Comparing against Definition 12, we have $P = u_1 \cup u_2 \cup u_3$ and $B = u_1 \cup u_2$; thus indeed $R \subseteq B \subseteq P$ and no constraint in $B \setminus R = \{c_8\}$ depends nonmonotonically on any rule in $P \setminus B = \{r_3, r_4, c_6, c_7\}$.

5.1.2 First Ancestor Intersection Units

We will use the evaluation graph for model building; as syntactic dependencies reflect semantic dependencies between units, multiple paths between units require attention. Of particular importance are *first ancestor intersection units,* which are units where distinct paths starting at some unit meet first. More formally,

Definition 15
Given an evaluation graph $\mathcal{E} = (U, E)$ and units $v \neq w \in U$, we say that unit w *is a first ancestor intersection unit (FAI) of* v, if paths $p_1 \neq p_2$ from v to w exist in E that overlap only in v and w. By $\mathit{fai}(v)$ we denote the set of all FAIs of v.

Example 21
Figure 6 sketches an evaluation graph with dependencies $a \rightarrow b \rightarrow c \rightarrow e \rightarrow f$, $a \rightarrow d \rightarrow e \rightarrow g$, and $b \rightarrow d$. We have that $\mathit{fai}(a) = \{d, e\}$, $\mathit{fai}(b) = \{e\}$, and $\mathit{fai}(u) = \emptyset$ for each $u \in U \setminus \{a, b\}$. In particular, f and g are not FAIs of b, because all pairs of distinct paths from b to f or g overlap in more than two units.

Note that for tree-shaped evaluation graphs, $\mathit{fai}(v) = \emptyset$ for each unit v as paths between nodes in a tree are unique.

Example 22 (*ctd.*)
The evaluation graph $\mathcal{E}_1$ of P_{swim} is a tree (see Fig. 4), thus $\mathit{fai}(u) = \emptyset$ for $u \in \{u_1, u_2, u_3\}$. In contrast, the evaluation graph $\mathcal{E}_2$ of P_{swim} (see Fig. 5) is not a tree; we have that $\mathit{fai}(u_4) = \{u_1\}$ and no other unit in $\mathcal{E}_2$ has FAIs.

We can build an evaluation graph $\mathcal{E}$ for a program P based on the dependency graph $DG(P)$. Initially, the units are set to the maximal strongly connected components of $DG(P)$, and then units are iteratively merged while preserving acyclicity and the conditions (a)-(d) of an evaluation graph; we will discuss some existing heuristics in Section 6.2, while for details we refer to Redl (2014).

5.2 Interpretation Graph

We now define the Interpretation Graph (short i-graph), which is the foundation of our model building algorithm. An i-graph is a labeled directed graph defined wrt. an evaluation graph, where each vertex is associated with a specific evaluation unit, a type (input resp. output interpretation) and a set of ground atoms.

We do not use interpretations themselves as vertices, as distinct vertices may be associated with the same interpretation; still we call vertices of the i-graph interpretations.

Towards defining i-graphs we first define an auxiliary concept called *interpretation structure*. We then define i-graphs as the subset of interpretation structures that obey certain topological and uniqueness conditions. Finally we present an example (Example 23 and Figure 8).

Definition 16 (*Interpretation Structure*)
Let $\mathcal{E} = (U, E)$ be an evaluation graph for a program P. An *interpretation structure* $\mathcal{I}$ *for* $\mathcal{E}$ is a directed acyclic graph $\mathcal{I} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ with nodes $M \subseteq \mathcal{I}_{id}$ from a countable set $\mathcal{I}_{id}$ of identifiers, edges $F \subseteq M \times M$, and total node labeling functions $\mathit{unit}\colon M \rightarrow U$, $\mathit{type}\colon M \rightarrow \{\text{I}, \text{O}\}$, and $\mathit{int}\colon M \rightarrow 2^{HB_P}$.

The following notation will be useful. Given unit $u \in U$ in the evaluation graph associated with an i-graph $\mathcal{I}$, we denote by $\textit{i-ints}_{\mathcal{I}}(u) = \{m \in M \mid \mathit{unit}(m) = u \text{ and } \mathit{type}(m) = \text{I}\}$ the input (i-)interpretations, and by $\mathit{o\text{-}ints}_{\mathcal{I}}(u) = \{m \in M \mid \mathit{unit}(m) = u \text{ and } \mathit{type}(m) = \text{O}\}$ the output (o-)interpretations of $\mathcal{I}$ at unit u. For every vertex $m \in M$, we denote by

$$\mathit{int}^{+}(m) = \mathit{int}(m) \cup \bigcup\{\mathit{int}(m') \mid m' \in M \text{ and } m' \text{ is reachable from } m \text{ in } \mathcal{I}\}$$

the *expanded interpretation* of m.

Given an interpretation structure $\mathcal{I} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ for $\mathcal{E} = (U, E)$ and a unit $u \in U$, we define the following properties:

(IG-I) *I-connectedness:* for every $m \in \mathit{o\text{-}ints}_{\mathcal{I}}(u)$, it holds that $|\{m' \mid (m, m') \in F\}| = |\{m' \in \mathit{i\text{-}ints}_{\mathcal{I}}(u) \mid (m, m') \in F\}| = 1$;

(IG-O) *O-connectedness:* for every $m \in \mathit{i\text{-}ints}_{\mathcal{I}}(u)$, $|\{m_i \mid (m, m_i) \in F\}| = |\mathit{preds}_{\mathcal{E}}(u)|$ and for every $u_i \in \mathit{preds}_{\mathcal{E}}(u)$ we have $|\{m_i \in \mathit{o\text{-}ints}_{\mathcal{I}}(u_i) \mid (m, m_i) \in F\}| = 1$;

(IG-F) *FAI intersection:* let $\mathcal{E}'$ be the subgraph of $\mathcal{E}$ on the units reachable from u[7] and for every $m \in \mathit{i\text{-}ints}_{\mathcal{I}}(u)$, let $\mathcal{I}'$ be the subgraph of $\mathcal{I}$ reachable from m. Then $\mathcal{I}'$ contains exactly one o-interpretation at each unit of $\mathcal{E}'$. (Note that both $\mathcal{I}$ and $\mathcal{E}$ are acyclic, hence $\mathcal{I}'$ does not include m and $\mathcal{E}'$ does not include u.)

(IG-U) *Uniqueness:* for every $m_1 \neq m_2 \in M$ such that $\mathit{unit}(m_1) = \mathit{unit}(m_2) = u$, we have $\mathit{int}^{+}(m_1) \neq \mathit{int}^{+}(m_2)$ (the expanded interpretations differ).

Definition 17 (*Interpretation Graph*)
Let $\mathcal{E} = (U, E)$ be an evaluation graph for a program P. then an *interpretation graph (i-graph)* for $\mathcal{E}$ is an interpretation structure $\mathcal{I} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ that fulfills for every unit $u \in U$ the conditions (IG-I), (IG-O), (IG-F), and (IG-U).

Intuitively, the conditions make every i-graph 'live' on its associated evaluation graph: an i-interpretation must conform to all dependencies of the unit it belongs to, by depending on exactly one o-interpretation at that unit's predecessor units (IG-O); moreover an o-interpretation must depend on exactly one i-interpretation at the same unit (IG-I). Furthermore, every i-interpretation depends directly or indirectly on exactly one o-interpretation at each unit it can reach in the i-graph (IG-F); this ensures that no expanded interpretation $\mathit{int}^{+}(m)$ 'mixes' two or more i-interpretations resp. o-interpretations from the same unit. (The effect of condition (IG-F) is visualized in Figure 7.) Finally, redundancies in an i-graph are ruled out by the uniqueness condition (IG-U).

Example 23 (*ctd.*)
Figure 8 shows an interpretation graph $\mathcal{I}_2$ for $\mathcal{E}_2$. The unit label is depicted as dashed rectangle labeled with the respective unit. The type label is indicated after interpretation names, i.e., m_1/I denotes that interpretation m_1 is an input interpretation. For $\mathcal{I}_2$ the set $\mathcal{I}_{id}$ of identifiers is $\{m_1, \ldots, m_{15}\}$. The symbol $\not{\imath}$ in a unit u pointing to an i-interpretation m indicates that there is no o-interpretation wrt. input m of unit u. Section 5.4 describes an algorithm for building an i-graph given an evaluation graph.

Dependencies are shown as arrows between interpretations. Observe that I-connectedness (IG-I) is fulfilled, as every o-interpretation depends on exactly one i-interpretation at the same unit. For example m_9 and m_{10} depend on m_7. O-connectedness (IG-O) is similarly

[7] I.e., $\mathcal{E}'$ is the subgraph of $\mathcal{E}$ induced by the set of units reachable from u, including u; in abuse of terminology, we briefly say 'the subgraph (of $\mathcal{E}$) reachable from'

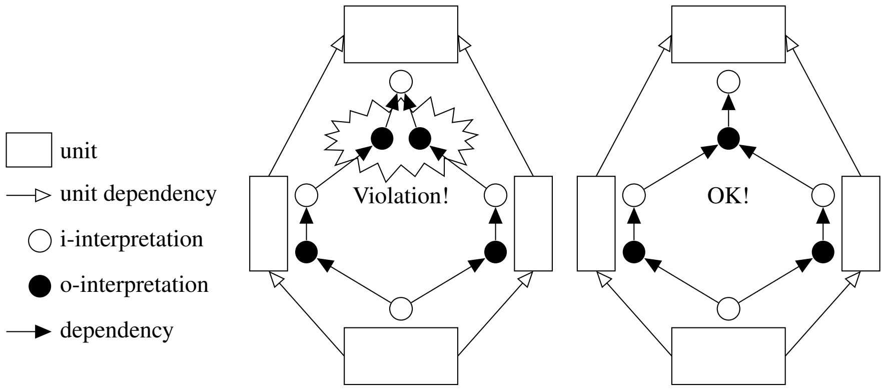

Fig. 7: Interpretation Graphs: violation of the FAI condition on the left, correct situation on the right.

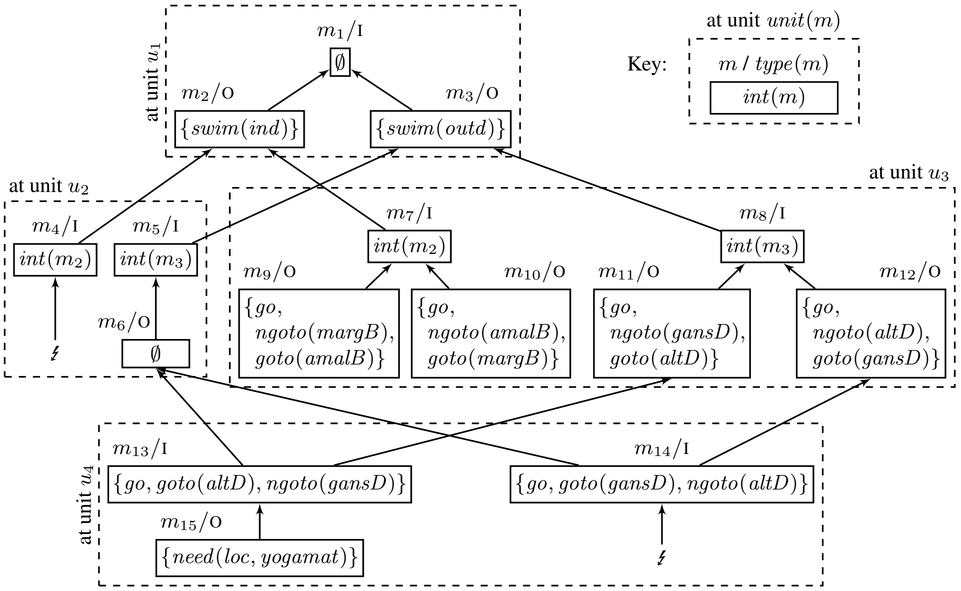

Fig. 8: Interpretation graph $\mathcal{I}_2$ for $\mathcal{E}_2$: dashed areas group interpretations according to their $unit(\cdot)$ value.

fulfilled, in particular consider i-interpretations of u_4 in $\mathcal{I}_2$: u_4 has two predecessor units (u_2 and u_3) and every i-interpretation at u_4 depends on exactly one o-interpretation at u_2 and exactly one o-interpretation at u_3. The condition on FAI intersection (IG-F) could only be violated by i-interpretations at u_4, concretely it would be violated if two different o-interpretations are reachable at u_1 from one i-interpretation at u_4. We can verify that from both m_{13} and m_{14} we can reach exactly one o-interpretation at each unit; hence the condition is fulfilled. An example for a violation would be an i-interpretation at u_4 that depends on m_6 and m_9: in this case we could reach two distinct o-interpretations m_2 and

m_3 at u_1, thereby violating (IG-F). Uniqueness (IG-U) is satisfied, as in both graphs no unit has two output models with the same content.

Note that the empty graph is an i-graph. This is by intent, as our model building algorithm will progress from an empty i-graph to one with interpretations at every unit, precisely if the program has an answer set.

5.2.1 Join

We will build i-graphs by adding one vertex at a time, always preserving the i-graph conditions. Adding an o-interpretation requires to add a dependency to one i-interpretation at the same unit. Adding an i-interpretation similarly requires addition of dependencies. However this is more involved because condition (IG-F) could be violated. Therefore, we next define an operation that captures all necessary conditions.

We call the combination of o-interpretations which yields an i-interpretation a '*join*'. Formally, the join operation ' $\bowtie$ ' is defined as follows.

Definition 18
Let $\mathcal{I} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ be an i-graph for an evaluation graph $\mathcal{E} = (V, E)$ of a program P. Let $u \in V$ be a unit, let $\mathit{preds}_{\mathcal{E}}(u) = \{u_1, \dots, u_k\}$ be the predecessor units of u, and let $m_i \in \mathit{o\text{-}ints}_{\mathcal{I}}(u_i)$, $1 \leq i \leq k$, be an o-interpretation at u_i. Then *the join* $m_1 \bowtie \cdots \bowtie m_k = \bigcup_{1 \leq i \leq k} \mathit{int}(m_i)$ *at* u *is defined* iff for each $u' \in \mathit{fai}(u)$ the set of o-interpretations at u' that are reachable (in F) from some o-interpretation m_i, $1 \leq i \leq k$, contains exactly one o-interpretation $m' \in \mathit{o\text{-}ints}_{\mathcal{I}}(u')$.

Intuitively, a set of interpretations can only be joined if all interpretations depend on the same (and on a single) interpretation at every unit.

Example 24 (*ctd.*)
In $\mathcal{I}_2$, i-interpretations m_1, m_4, m_5, m_7, and m_8 are created by trivial join operations with none or one predecessor unit. For m_{13} and m_{14}, we have a nontrivial join: $\mathit{int}(m_{13}) = \mathit{int}(m_6) \cup \mathit{int}(m_{11})$ and the join is defined because $\mathit{fai}(u_4) = \{u_1\}$, and from m_6 and m_{11} we can reach in $\mathcal{I}_2$ exactly one o-interpretation at u_1. Observe that the join $m_6 \bowtie m_9$ is not defined, as we can reach in $\mathcal{I}_2$ from $\{m_6, m_9\}$ the o-interpretations m_2 and m_3 at u_1, and thus more than exactly one o-interpretation at some FAI of u_4. Similarly, the join $m_6 \bowtie m_{10}$ is undefined, as we can reach m_2 and m_3 at u_1.

The result of a join is the union of predecessor interpretations; this is important for answer set graphs and join operations on them, which comes next. Note that each leaf unit (i.e., without predecessors) has exactly one well-defined join result, viz. $\emptyset$.

If we add a new i-interpretation from the result of a join operation to an i-graph and dependencies to all participating o-interpretations, the resulting graph is again an i-graph; thus the join is sound wrt. to the i-graph properties. Moreover, each i-interpretation that can be added to an i-graph while preserving the i-graph conditions can be synthesized by a join; that is, the join is complete for such additions. This is a consequence of the following result.

Proposition 4
Let $\mathcal{I} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ be an i-graph for an evaluation graph $\mathcal{E} = (V, E)$ and $u \in V$ with $\mathit{preds}_{\mathcal{E}}(u) = \{u_1, \ldots, u_k\}$. Furthermore, let $m_i \in \mathit{o\text{-}ints}_{\mathcal{I}}(u_i)$, $1 \leq i \leq k$, such that no vertex $m \in \mathit{i\text{-}ints}_{\mathcal{I}}(u)$ exists such that $\{(m, m_1), \ldots, (m, m_k)\} \subseteq F$. Then the join $J = m_1 \bowtie \cdots \bowtie m_k$ is defined at u iff $\mathcal{I}' = (M', F', \mathit{unit}', \mathit{type}', \mathit{int}')$ is an i-graph for $\mathcal{E}$ where (a) $M' = M \cup \{m'\}$ for some new vertex $m' \in \mathcal{I}_{id} \setminus M$, (b) $F' = F \cup \{(m', m_i) \mid 1 \leq i \leq k\}$, (c) $\mathit{unit}' = \mathit{unit} \cup \{(m', u)\}$, (d) $\mathit{type}' = \mathit{type} \cup \{(m', \textsc{i})\}$, and (e) $\mathit{int}' = \mathit{int} \cup \{(m', J)\}$.

Note that the i-graph definition specifies topological properties of an i-graph wrt. an evaluation graph. In the following we extend this specification to the contents of interpretations.

5.3 Answer Set Graph

We next restrict i-graphs to *answer set graphs* such that interpretations correspond with answer sets of certain HEX programs that are induced by the evaluation graph.

Definition 19 (*Answer Set Graph*)
An *answer set graph* $\mathcal{A} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ for an evaluation graph $\mathcal{E} = (U, E)$ is an i-graph for $\mathcal{E}$ such that for each unit $u \in U$, it holds that

(a) $\{\mathit{int}^{+}(m) \mid m \in \mathit{i\text{-}ints}_{\mathcal{I}}(u)\} \subseteq \mathcal{AS}(u^{<})$, i.e., every expanded i-interpretation at u is an answer set of $u^{<}$;
(b) $\{\mathit{int}^{+}(m) \mid m \in \mathit{o\text{-}ints}_{\mathcal{I}}(u)\} \subseteq \mathcal{AS}(u^{\leq})$, i.e., every expanded o-interpretation at u is an answer set of $u^{\leq}$; and
(c) for each $m \in \mathit{i\text{-}ints}_{\mathcal{I}}(u)$, it holds that $\mathit{int}(m) = \bigcup_{(m,m_i) \in F} \mathit{int}(m_i)$.

Note that each leaf unit u, has $u^{<} = \emptyset$, and thus $\emptyset$ is the only i-interpretation possible. Moreover, condition (c) is necessary to ensure that an i-interpretation at unit u contains all atoms of answer sets of predecessor units that are relevant for evaluating u. Furthermore, note that the empty graph is an answer set graph.

Example 25 (*ctd.*)
The example i-graph $\mathcal{I}_2$ is in fact an answer set graph. First, $\mathit{int}^{+}(m_1) = \emptyset$ and $u_1^{<} = \emptyset$ and indeed $\emptyset \in \mathcal{AS}(\emptyset)$ which satisfies condition (a). Less obvious is the case of o-interpretation m_6 in $\mathcal{I}_2$: $\mathit{int}^{+}(m_6) = \{\mathit{swim}(\mathit{outd})\}$ and $u_2^{\leq} = \{r_1, r_2, c_8\}$; as c_8 kills all answer sets where money is required, $\mathcal{AS}(\{r_1, r_2, c_8\}) = \{\{\mathit{swim}(\mathit{outd})\}\}$; hence $\mathit{int}^{+}(m_6)$ is the only expanded interpretation of an o-interpretation possible at u_2. Furthermore, the condition (IG-U) on i-graphs implies that m_6 is the only possible o-interpretation at u_2. Consider next m_{13}:

$$u_4^{<} = \{r_1, r_2, r_3, r_4, c_6, c_7, c_8\} \text{ and}$$
$$\mathit{int}^{+}(m_{13}) = \{\mathit{go}, \mathit{goto}(\mathit{altD}), \mathit{ngoto}(\mathit{gansD}), \mathit{swim}(\mathit{outd})\}.$$

The two answer sets of $u_4^{<}$ are $\{\mathit{go}, \mathit{goto}(\mathit{altD}), \mathit{ngoto}(\mathit{gansD}), \mathit{swim}(\mathit{outd})\}$, and $\{\mathit{go}, \mathit{goto}(\mathit{gansD}), \mathit{ngoto}(\mathit{altD}), \mathit{swim}(\mathit{outd})\}$, and $\mathit{int}^{+}(m_{13})$ is one of them; the other one is $\mathit{int}^{+}(m_{14})$. Finally

$$\mathit{int}^{+}(m_{15}) = \{\mathit{swim}(\mathit{outd}), \mathit{goto}(\mathit{altD}), \mathit{go}, \mathit{ngoto}(\mathit{gansD}), \mathit{need}(\mathit{loc}, \mathit{yogamat})\},$$

which is the single answer set of $u_4^{\leq} = P_{\mathit{swim}}$.

Similarly as for i-graphs, the join is a sound and complete operation to add i-interpretations to an answer set graph.

Proposition 5
Let $\mathcal{A} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ be an answer set graph for an evaluation graph $\mathcal{E} = (V, E)$ and let $u \in V$ with $\mathit{preds}_{\mathcal{E}}(u) = \{u_1, \dots, u_k\}$. Furthermore, let $m_i \in \mathit{o\text{-}ints}_{\mathcal{A}}(u_i)$, $1 \leq i \leq k$, such that no $m \in \mathit{i\text{-}ints}_{\mathcal{A}}(u)$ with $\{(m, m_1), \dots, (m, m_k)\} \subseteq F$ exists. Then the join $J = m_1 \bowtie \cdots \bowtie m_k$ is defined at u iff $\mathcal{A}' = (M', F', \mathit{unit}', \mathit{type}', \mathit{int}')$ is an answer set graph for $\mathcal{E}$ where (a) $M' = M \cup \{m'\}$ for some new vertex $m' \in \mathcal{I}_{id} \setminus M$, (b) $F' = F \cup \{(m', m_i) \mid 1 \leq i \leq k\}$, (c) $\mathit{unit}' = \mathit{unit} \cup \{(m', u)\}$, (d) $\mathit{type}' = \mathit{type} \cup \{(m', \mathrm{I})\}$, and (e) $\mathit{int}' = \mathit{int} \cup \{(m', J)\}$.

Example 26 (ctd.)
Imagine that $\mathcal{I}_2$ has no interpretations at u_4. The following candidate pairs of o-interpretations exist for creating i-interpretations at u_4: $m_6 \bowtie m_9$, $m_6 \bowtie m_{10}$, $m_6 \bowtie m_{11}$, and $m_6 \bowtie m_{12}$. A seen in Example 24, $m_{13} = m_6 \bowtie m_{11}$ and $m_{14} = m_6 \bowtie m_{12}$ are the only joins at u_4 that are defined. In Example 25 we have seen that $\mathcal{AS}(u_4^{<}) = \{\mathit{int}^+(m_{13}), \mathit{int}^+(m_{14})\}$, and due to (IG-U), we cannot have additional i-interpretations with the same content.

5.3.1 Complete Answer Set Graphs

We next introduce a notion of completeness for answer set graphs.

Definition 20
Let $\mathcal{A} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ be an answer set graph for an evaluation graph $\mathcal{E} = (U, E)$ and let $u \in U$. Then

- *$\mathcal{A}$ is input-complete for u,* if $\{\mathit{int}^+(m) \mid m \in \mathit{i\text{-}ints}_{\mathcal{A}}(u)\} = \mathcal{AS}(u^{<})$, and
- *$\mathcal{A}$ is output-complete for u,* if $\{\mathit{int}^+(m) \mid m \in \mathit{o\text{-}ints}_{\mathcal{A}}(u)\} = \mathcal{AS}(u^{\leq})$.

If an answer set graph is complete for all units of its corresponding evaluation graph, answer sets of the associated program can be obtained as follows.

Theorem 5
Let $\mathcal{E} = (U, E)$, where $U = \{u_1, \dots, u_n\}$, be an evaluation graph of a program P, and let $\mathcal{A} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ be an answer set graph that is output-complete for every unit $u \in U$. Then

$$\mathcal{AS}(P) = \Big\{ \textstyle\bigcup_{i=1}^{n} \mathit{int}(m_i) \mid m_i \in \mathit{o\text{-}ints}_{\mathcal{A}}(u_i), 1 \leq i \leq n, |\mathit{o\text{-}ints}_{\mathcal{A}'}(u_i)| = 1 \Big\}, \quad (4)$$

where $\mathcal{A}'$ is the subgraph of $\mathcal{A}$ consisting of all interpretations that are reachable in $\mathcal{A}$ from some interpretation $m_1, \dots, m_n$.

Example 27 (ctd.)
In $\mathcal{I}_2$ we first choose $m_{15} \in \mathit{o\text{-}ints}(u_4)$, which is the only o-interpretation at u_4. The

subgraph reachable from m_{15} must contain exactly one o-interpretation at each unit; we thus must choose every o-interpretations m such that $m_{15} \rightarrow^{+} m$. Hence we obtain

$$\begin{aligned}&\big\{\mathit{int}(m_3) \cup \mathit{int}(m_6) \cup \mathit{int}(m_{11}) \cup \mathit{int}(m_{15})\big\}\\ &= \big\{\{\mathit{swim}(\mathit{outd})\} \cup \emptyset \cup \{\mathit{goto}(\mathit{altD}), \mathit{ngoto}(\mathit{gansD}), \mathit{go}\} \cup \{\mathit{need}(\mathit{loc}, \mathit{yogamat})\}\big\}\\ &= \big\{\{\mathit{swim}(\mathit{outd}), \mathit{goto}(\mathit{altD}), \mathit{ngoto}(\mathit{gansD}), \mathit{go}, \mathit{need}(\mathit{loc}, \mathit{yogamat})\}\big\}\end{aligned}$$

which is indeed the set of answer sets of P_{swim}.

The rather involved set construction in (4) establishes a relationship between answer sets of a program and complete answer set graphs that resembles condition (IG-F) of i-graphs. To obtain a more convenient way to enumerate answer sets, we can extend an evaluation graph always with a single void unit u_{final} that depends on all other units in the graph (i.e., $(u_{\mathit{final}}, u) \in E$ for each $u \in U \setminus \{u_{\mathit{final}}\}$), which we call a *final unit*; the answer sets of P correspond then directly to i-interpretations at u_{final}. Formally,

Proposition 6
Let $\mathcal{A} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ be an answer set graph for an evaluation graph $\mathcal{E} = (U, E)$ of a program P, where $\mathcal{E}$ contains a final unit u_{final}, and assume that $\mathcal{A}$ is input-complete for U and output-complete for $U \setminus \{u_{\mathit{final}}\}$. Then

$$\mathcal{AS}(P) = \{\mathit{int}(m) \mid m \in \textit{i-ints}_{\mathcal{A}}(u_{\mathit{final}})\}. \tag{5}$$

Expanding i-interpretations at u_{final} is not necessary, as u_{final} depends on all other units; thus for every $m \in \textit{i-ints}_{\mathcal{A}}(u_{\mathit{final}})$ it holds that $\mathit{int}^{+}(m) = \mathit{int}(m)$.

We will use the technique with u_{final} for our model enumeration algorithm; as the join condition must be checked anyways, this technique is an efficient and simple method for obtaining all answer sets of a program using an answer set graph without requesting an implementation of the conditions in Theorem 5.

5.4 Answer Set Building

Thanks to the results above, we can obtain the answer sets of a HEX-program from any answer set graph for it. To build an answer set graph, we proceed as follows. We start with an empty graph, obtain o-interpretations by evaluating a unit on an i-interpretation, and then gradually generate i-interpretations by joining o-interpretations of predecessor units in an evaluation graph at hand.

Towards an algorithm for evaluating a HEX-program based on an evaluation graph, we use a generic grounding algorithm GROUNDHEX for lde-safe programs, and a solving algorithm EVALUATEGROUNDHEX which returns for a ground HEX-program P its answer sets $\mathcal{AS}(P)$. We assume that they satisfy the following properties.

Property 1
Given an lde-safe program P, GROUNDHEX(P) returns a finite ground program such that $\mathcal{AS}(P) = \mathcal{AS}(\text{GROUNDHEX}(P))$.

Property 2
Given a finite ground HEX-program P, EVALUATEGROUNDHEX$(P) = \mathcal{AS}(P)$.

Algorithm 1: EvaluateLDESafe

```
Input: A liberally de-safe HEX-program $P$, an input interpretation $I$
Output: All answer sets of $P \cup \mathit{facts}(I)$ without $I$
// add input facts and ground, cf. (Eiter et al. 2014a)
$P' \leftarrow$ GroundHEX($P \cup \mathit{facts}(I)$)
// evaluate the ground program, cf. (Eiter et al. 2014b),
// and perform output projection
return $\{I' \setminus I \mid I' \in$ EvaluateGroundHEX$(P')\}$
```

Concrete such algorithms are given in (Eiter et al. 2014a) and (Eiter et al. 2014b), respectively. Since the details of these algorithms are not relevant for the further understanding of this paper, we give here only an informal description and refer the interested reader to the respective papers. The idea of the grounding algorithm is to iteratively extend the grounding by expanding the set of constants until it is large enough to ensure that it has the same answer sets as the original program. To this end, the algorithm starts with the constants in the input program only, and in each iteration of the algorithm it evaluates external atoms a (finite) number of relevant inputs in order to determine additional relevant constants. Under the syntactic restrictions recapitulated in the preliminaries, this iteration will reach a fixpoint after finitely many steps. The solving algorithm is based on conflict-driven clause learning (CDCL) and lifts the work of Gebser et al. (2012) from ordinary to HEX programs. The main idea is to learn not only conflict clauses, but also (parts of) the behavior of external sources while the search space is traversed. The behavior is described in terms of input-output relations, i.e., certain input atoms and constants lead to a certain output of the external atom. This information is added to the internal representation of the program such that guesses for external atoms that violate the known behavior are eliminated in advance.

By composing the two algorithms, we obtain Algorithm 1 for evaluating a single unit. Formally, it has the following property.

Proposition 7
Given an lde-safe HEX-program P and an input interpretation I, Algorithm 1 returns the set $\{I' \setminus I \mid I' \in \mathcal{AS}(P \cup \mathit{facts}(I))\}$, i.e., the answer sets of P augmented with facts for the input I, projected to the non-input.

We are now ready to formulate an algorithm for evaluating HEX programs that have been decomposed into an evaluation graph.

To this end, we build first an evaluation graph $\mathcal{E}$ and then compute gradually an answer set graph $\mathcal{A} = (M, F, \mathit{unit}, \mathit{type}, \mathit{int})$ based on $\mathcal{E}$, proceeding along already evaluated units towards the unit u_{final}. Algorithm 2 shows the model building algorithm in pseudo-code, in which the positive integers $\mathbb{N} = \{1, 2, \dots\}$ are used as identifiers $\mathcal{I}_{id}$ and $\max(M)$ is maximum in any set $M \subseteq \mathbb{N}$ where, by convention, $\max(\emptyset) = 0$. Intuitively, the algorithm works as follows. The set U contains units for which $\mathcal{A}$ is not yet output-complete (see Definition 20); we start with an empty answer set graph $\mathcal{A}$, thus initially $U = V$. In each iteration of the while loop (a), a unit u that is not output-complete and depends only on output-complete units is selected. The first for loop (c) makes u input-complete; if u is the final unit, the answer sets are returned in (d), otherwise the second for loop (e) makes u

```
Algorithm 2: BuildAnswerSets
  Input: $\mathcal{E} = (V, E)$: evaluation graph for HEX program $P$, which contains a unit $u_{\mathit{final}}$
         that depends on all other units in $V$
  Output: a set of all answer sets of $P$
  $M := \emptyset$, $F := \emptyset$, $\mathit{unit} := \emptyset$, $\mathit{type} := \emptyset$, $\mathit{int} := \emptyset$, $U := V$
(a) while $U \neq \emptyset$ do
      choose $u \in U$ s.t. $\mathit{preds}_{\mathcal{E}}(u) \cap U = \emptyset$
      let $\{u_1, \ldots, u_k\} = \mathit{preds}_{\mathcal{E}}(u)$
      if $k = 0$ then
(b)       $m := \mathit{max}(M) + 1$
          $M := M \cup \{m\}$
          $\mathit{unit}(m) := u$, $\mathit{type}(m) := \mathrm{I}$, $\mathit{int}(m) := \emptyset$
      else
(c)       for $m_1 \in \mathit{o\text{-}ints}(u_1), \ldots, m_k \in \mathit{o\text{-}ints}(u_k)$ do
              if $J = m_1 \bowtie \cdots \bowtie m_k$ is defined then
                  $m := \mathit{max}(M) + 1$
                  $M := M \cup \{m\}$, $F := F \cup \{(m, m_i) \mid 1 \leq i \leq k\}$
                  $\mathit{unit}(m) := u$, $\mathit{type}(m) := \mathrm{I}$, $\mathit{int}(m) := J$
(d)   if $u = u_{\mathit{final}}$ then
          return $\mathit{i\text{-}ints}(u_{\mathit{final}})$
(e)   for $m' \in \mathit{i\text{-}ints}(u)$ do
          $O :=$ EvaluateLDESafe$(u, \mathit{int}(m'))$
          for $o \in O$ do
              $m := \mathit{max}(M) + 1$
              $M := M \cup \{m\}$, $F := F \cup \{(m, m')\}$
              $\mathit{unit}(m) := u$, $\mathit{type}(m) := \mathrm{O}$, $\mathit{int}(m) := o$
(f)   $U := U \setminus \{u\}$
```

output-complete, and then u is removed from U. Each iteration makes one unit input- and output-complete; hence when the algorithm reaches u_{final} and makes it input-complete, all answer sets can directly be returned in (d). Formally, we have

Theorem 6
Given an evaluation graph $\mathcal{E} = (V, E)$ of a HEX program P, BuildAnswerSets$(\mathcal{E})$ returns $\mathcal{AS}(P)$.

A run of the algorithm on our running example using the evaluation graph $\mathcal{E}_2$ extended with a final unit is given in Online Appendix B.

5.4.1 Model Streaming

Algorithm BuildAnswerSets as described above keeps all answer sets in memory, and it evaluates each unit only once wrt. every possible i-interpretation. This may lead to a resource bound excess, as in general an exponential number of answer sets respectively interpretations at evaluation units are possible. However, keeping the whole answer set graph in memory is not necessary for computing all answer sets.

We have realized a variant of Algorithm BUILDANSWERSETS that uses the same principle of constructing an answer set graph, interpretations are created at a unit *on demand* when they are requested by units that depend on it; furthermore, the algorithm keeps basically only one interpretation at each evaluation unit in memory at a time, which means that interpretations are provided in a *streaming fashion* one by one, and likewise the answer sets of the program at the unit u_{final}, where the model building starts. Such answer set streaming is particularly attractive for applications, as one can terminate the computation after obtaining sufficiently many answer sets. On the other hand, it comes at the cost of potential re-evaluation of units wrt. the same i-interpretation, as we need to trade space for time. However, in practice this algorithm works well and is the one used in the dlvhex prototype. We describe this algorithm in Online Appendix C.

6 Implementation

In this section we give some details on the implementation of the techniques. Our prototype system is called dlvhex; it is written in C++ and online available as open-source software.[8] The current version 2.4.0 was released in September 2014.

We first describe the general architecture, the major components, and their interplay (Section 6.1). Then we give an overview about the existing heuristics for building evaluation graphs (Section 6.2). Experimental results are presented and discussed in Section 6.3. For details on the usage of the system, we refer to the website; an exhaustive description of the supported command-line parameters is output when the system is called without parameters.

6.1 System Architecture

The dlvhex system architecture is shown in Figure 9. The arcs model both control and data flow within the system. The evaluation of a HEX-program works as follows.

First, the input program is passed to the *evaluation framework* ①, which creates an *evaluation graph* depending on the chosen evaluation heuristics. This results in a number of interconnected *evaluation units*. While the interplay of the units is managed by the evaluation framework, the individual units are handled by *model generators* of different kinds.

Each instance of a model generator realizes EVALUATELDESAFE (Algorithm 1) for a single evaluation unit, receives *input interpretations* from the framework (which are either output by predecessor units or come from the input facts for leaf units), and sends output interpretations back to the framework ②, which manages the integration of the latter to final answer sets and realizes BUILDANSWERSETS (Algorithm 2).

Internally, the model generators make use of a *grounder* and a *solver* for ordinary ASP programs. The architecture of our system is flexible and supports multiple concrete backends that can be plugged in. Currently it supports dlv, gringo 4.4.0 and clasp 3.1.0, as well as an

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

internal grounder and a solver that were built from scratch (mainly for testing purposes); they use basically the same core algorithms as gringo and clasp, but without optimizations. The reasoner backends gringo and clasp are statically linked to our system; thus no interprocess communication is necessary. The model generator within the dlvhex core sends a non-ground evaluation unit to the HEX-grounder which returns a ground evaluation unit ③. The HEX-grounder in turn uses one of the above mentioned ordinary ASP grounders as backend ④ and accesses external sources to handle newly introduced constants that are not part of the input program (called *value invention*) ⑤. The ground evaluation unit is then sent to the ASP solver and answer sets of the ground unit are returned ⑥.

Intuitively, model generators evaluate evaluation units by replacing external atoms by ordinary 'replacement' atoms, guessing their truth value, and making sure that the guesses are correct with respect to the external oracle functions. To achieve that, the solver backend needs to make callbacks to the *Post Propagator* in the dlvhex core during model building. The Post Propagator checks guesses for external atoms against the actual semantics and checks the minimality of the answer set. It processes a complete or partial model candidate, and returns learned nogoods to the external solver ⑦ as formalized in (Eiter et al. 2012). The dlv backend calls the Post Propagator only for complete model candidates, the internal solver and the clasp backend also call it for partial model candidates of evaluation units. For the clasp backend, we exploit its SMT interface, which was previously used for the special case of constraint answer set solving (Gebser et al. 2009b). Verifying guesses of replacement atoms requires calling *plugins* that implement the external sources (i.e., the oracle functions $F_{\&g}$ from Definition 3) ⑧. Moreover, the Post Propagator also ensures answer set minimality by eliminating unfounded sets that are caused by external sources and therefore can not be detected by the ordinary ASP solver backend (as shown by Eiter et al. (2014b)). Finally, as soon as the evaluation framework obtains an i-interpretation of the final evaluation unit u_{final}, this i-interpretation (which is an answer set according to Proposition 6) is returned to the user ⑨.

6.2 Heuristics

As for creating evaluation graphs, several heuristics have been implemented. A heuristics starts with the rule dependency graph as by Definition 10 and then acyclically combines nodes into units.

Some heuristics are described in the following.

H0 is a 'trivial' heuristics that makes units as small as possible. This is useful for debugging, however it generates the largest possible number of evaluation units and therefore incurs a large overhead. As a consequence *H0* performs clearly worse than other heuristics and we do not report its performance in experimental results.

H1 is the evaluation heuristics of the dlvhex prototype version 1. *H1* makes units as large as possible and has several drawbacks as discussed above.

H2 is a simple evaluation heuristics which has the goal of finding a compromise between the *H0* and *H1*. It places rules into units as follows:

(i) it puts rules r_1, r_2 into the same unit whenever $r_1 \rightarrow_{m,n} s$ and $r_2 \rightarrow_{m,n} s$ for some rule s and there is no rule t such that exactly one of r_1, r_2 depends on t;

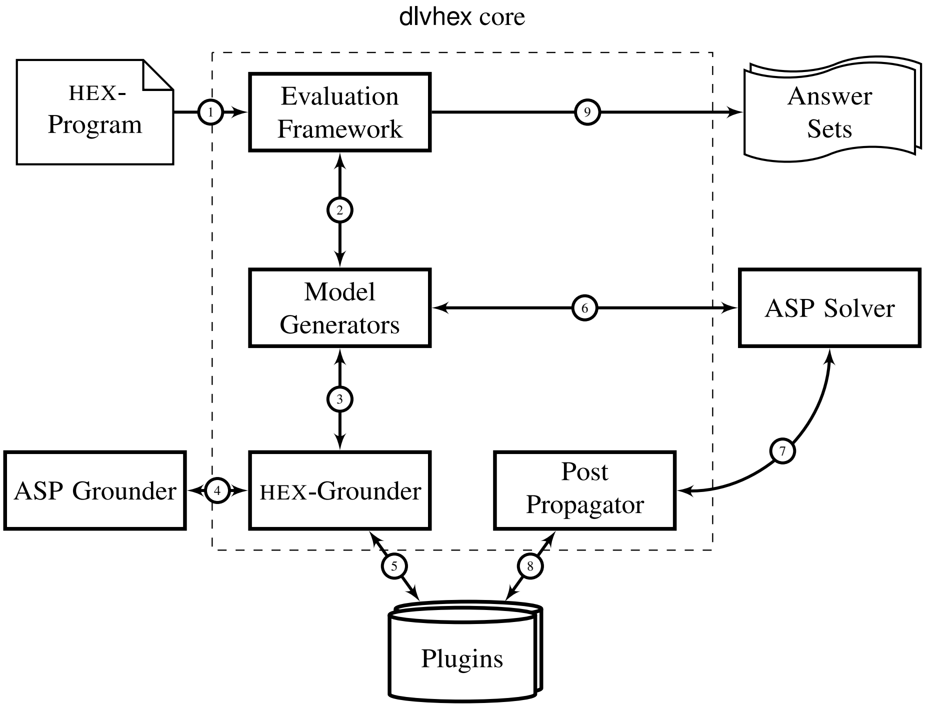

Fig. 9: Architecture of dlvhex

(ii) it puts rules r_1, r_2 into the same unit whenever $s \rightarrow_{m,n} r_1$ and $s \rightarrow_{m,n} r_2$ for some rule s and there is no rule t such that t depends on exactly one of r_1, r_2; but

(iii) it never puts rules r, s into the same unit if r contains external atoms and $r \rightarrow_{m,n} s$.

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 interpretations.

H3 is a heuristics for finding a compromise between (1) minimizing the number of units, and (2) splitting the program whenever a de-relevant nonmonotonic external atom would receive input from the same unit. We mention this heuristics only as an example, but disregard it in the experiments since it was developed in connection with novel 'liberal' safety criteria (Eiter et al. 2013) that are beyond the scope of this paper. *H3* greedily gives preference to (1) and is motivated by the following considerations. The grounding algorithm by Eiter et al. (2014a) evaluates the external sources under all interpretations such that the set of observed constants is maximized. While monotonic and antimonotonic input atoms are not problematic (the algorithm can simply set all to true resp. false), nonmonotonic parameters require an exponential number of evaluations in general. Thus, although program decomposition is not strictly necessary for evaluating liberally safe HEX-programs, it is still useful in such cases as it restricts grounding to those interpretations that are actually relevant in some answer set. However, on the other hand it can be disadvantageous for propositional solving algorithms such as those in (Eiter et al. 2012).

Program decomposition can be seen as a hybrid between traditional and lazy grounding (cf. e.g. Palù et al. (2009)), as program parts are instantiated that are larger than single rules but smaller than the whole program.

6.3 Experimental Results

In this section, we evaluate the model-building framework empirically. To this end, we compare the following configurations. In the *H1* column, we use the previous state-of-the-art evaluation method (Schindlauer 2006) before the framework in Section 5 was developed. This previous method also makes use of program decomposition. However, in contrast to our new framework, the decomposition is based on atom dependencies rather than rule dependencies, and the decomposition strategy is hard-coded and not customizable. This evaluation method corresponds to heuristics *H1* in our new framework.

In the *w/o framework* column, we present the results without application of the framework using the HEX-program evaluation algorithm by Eiter et al. (2014a) which allows to first instantiate and then solve the instantiated HEX-program. Note that before this algorithm was developed, such a 'two-phase' evaluation was not possible since program decomposition was necessary for grounding purposes. With the algorithm in (Eiter et al. 2014a), decomposition is not necessary anymore, but can still be useful as the results in the *H2* column shows, which correspond to the results when applying the heuristics *H2* described above.

The configuration of the grounding algorithm and the solving algorithm (e.g., conflict-driven learning strategies) also influence the results. Moreover, in addition to the default heuristics of framework, other heuristics have been developed as well and the best selection of the heuristics often depends on the configuration of the grounding and the solving algorithm. Since they were used as black boxes in Algorithm 1, an exhaustive experimental analysis of the system is beyond the scope of this paper and would require an in-depth description of these algorithms. Thus, we confine the discussion to the default settings, which suffices to show that the new framework can speed up the evaluation significantly. The only configuration difference between the result columns *H1* and *H2* is the evaluation heuristics, all other parameters are equal. Evaluating the *w/o framework* column requires the grounding algorithm from (Eiter et al. 2014a) instead of evaluation via decomposition, therefore *w/o framework* does not use any heuristics. The solver backend (clasp) configuration is the same in *H1*, *H2*, and *w/o framework*. We use the streaming algorithm (see Online Appendix C) in all experiments. For an in depth discussion, we refer to Eiter et al. (2014b; 2014a) and Redl (2014), where the efficiency was evaluated using a variety of applications including planning tasks (e.g., robots searching an unknown area for an object, tour planning), computing extensions of abstract argumentation frameworks, inconsistency analysis in multi-context systems, and reasoning over description logic knowledge bases.

We discuss here two benchmark problems, which we evaluated on a Linux server with two 12-core AMD 6176 SE CPUs with 128GB RAM running dlvhex version 2.4.0. and an *HTCondor* load distribution system[9] that ensures robust runtimes. The HTCondor system ensures that multiple runs of the same instance have negligible deviations in the order of fractions of a second, thus we can restrict the experiments to one run. The grounder and solver backends for all benchmarks are gringo 4.4.0 and clasp 3.1.1. For each instance, we

[9] `http://research.cs.wisc.edu/htcondor`

Topology and	First Answer Set			All Answer Sets		
Instance Size	H1	w/o framework	H2	H1	w/o framework	H2
d-7-7-3-3 (10)	1.23 (0)	0.29 (0)	0.38 (0)	4.93 (0)	0.76 (0)	0.79 (0)
d-7-7-4-4 (10)	18.43 (0)	1.09 (0)	0.76 (0)	50.78 (0)	3.39 (0)	1.80 (0)
d-7-7-5-5 (10)	94.18 (1)	3.60 (0)	1.52 (0)	289.35 (4)	20.21 (0)	4.97 (0)
h-9-9-3-3 (10)	83.17 (1)	3.77 (0)	0.70 (0)	300.96 (4)	28.67 (0)	2.11 (0)
h-9-9-4-4 (10)	389.74 (6)	30.56 (0)	2.14 (0)	555.94 (9)	335.11 (5)	12.56 (0)
r-7-7-4-4 (10)	39.27 (0)	2.82 (0)	0.33 (0)	366.17 (5)	57.26 (0)	2.06 (0)
r-7-7-5-5 (10)	389.88 (6)	105.80 (1)	0.93 (0)	600.00 (10)	377.37 (5)	4.39 (0)
r-7-8-5-5 (10)	226.04 (3)	25.11 (0)	0.57 (0)	541.80 (9)	317.64 (5)	3.99 (0)
r-7-9-5-5 (10)	355.37 (5)	145.99 (2)	0.87 (0)	600.00 (10)	458.14 (7)	5.42 (0)
r-8-7-5-5 (10)	502.64 (8)	329.47 (5)	1.21 (0)	555.26 (9)	443.15 (7)	5.84 (0)
r-8-8-5-5 (10)	390.81 (6)	201.08 (3)	1.00 (0)	600.00 (10)	495.41 (8)	5.38 (0)
z-7-7-3-3 (10)	2.34 (0)	0.32 (0)	0.44 (0)	9.17 (0)	1.13 (0)	1.00 (0)
z-7-7-4-4 (10)	33.32 (0)	1.58 (0)	1.07 (0)	182.44 (2)	9.00 (0)	2.67 (0)
z-7-7-5-5 (10)	164.33 (2)	12.69 (0)	3.52 (0)	502.49 (8)	89.01 (1)	6.90 (0)

Table 1: MCS experiments: variable topology (d, h, r, z) and instance size.

limited the CPU usage to two cores and 8GB RAM. The timeout for each instance was 600 seconds. Each line shows the average runtimes over all instances of a certain size, where each timeout counts as 600 seconds. While instances usually become harder with larger size, there might be some exceptions due to the randomly generated instances; however, the overall trend shows that runtimes increase with the instance size. Numbers in parentheses are the numbers of instances of respective size in the leftmost column and the numbers of timeout instances elsewhere. The generators, instances and external sources are available at `http://www.kr.tuwien.ac.at/research/projects/hexhex/hexframework`.

*6.3.1 Multi-Context Systems (*MCS*)*

The MCS benchmarks originate in the application scenario of enumerating output-projected equilibria (i.e., global models) of a given multi-context system (MCS) (cf. Section 2.3.2). Each instance comprises 7–9 contexts (propositional knowledge bases) whose local semantics is modeled by external atoms; roughly speaking, they single out assignments to the atoms of a context occurring in bridge rules such that local models exist. For each context, 5–10 such atoms are guessed and bridge rules, which are modeled by ordinary rules, are randomly constructed on top. The MCS instances were generated using the DMCS (Bairakdar et al. 2010) instance generator, with 10 randomized instances for different link structure between contexts (diamond (d), house (h), ring (r), zig-zag (z)) and system size; they have between 4 and about 20,000 answer sets, with an average of 400. We refer to (Bairakdar et al. 2010) and (Schüller 2012) for more details on the benchmarks and the HEX-programs.

Table 1 shows the experimental results: computation with the old method *H1* often exceeds the time limit, while the new method *H2* manages to enumerate all solutions of all instances. Monolithic evaluation without decomposition shows a performance between the old and new method. These results show that our new evaluation method is essential for using HEX to computationally realize the MCS application.

Instance Size	First Answer Set			All Answer Sets		
	H1	w/o framework	H2	H1	w/o framework	H2
1 (1)	2.84 (0)	3.14 (0)	2.78 (0)	2.73 (0)	3.14 (0)	2.79 (0)
2 (1)	6.13 (0)	7.18 (0)	4.90 (0)	6.05 (0)	7.17 (0)	4.88 (0)
3 (1)	10.18 (0)	12.30 (0)	8.32 (0)	10.25 (0)	12.35 (0)	8.37 (0)
4 (1)	15.92 (0)	18.66 (0)	12.12 (0)	15.86 (0)	18.85 (0)	12.16 (0)
5 (1)	26.06 (0)	28.47 (0)	17.17 (0)	26.23 (0)	28.35 (0)	17.06 (0)
6 (1)	47.06 (0)	45.71 (0)	23.39 (0)	46.84 (0)	45.62 (0)	23.26 (0)
7 (1)	92.76 (0)	79.41 (0)	31.19 (0)	96.56 (0)	79.82 (0)	31.04 (0)
8 (1)	198.59 (0)	155.10 (0)	37.85 (0)	199.74 (0)	155.26 (0)	38.06 (0)
9 (1)	600.00 (1)	600.00 (1)	46.61 (0)	600.00 (1)	600.00 (1)	46.75 (0)
10 (1)	600.00 (1)	600.00 (1)	57.48 (0)	600.00 (1)	600.00 (1)	57.40 (0)
11 (1)	600.00 (1)	600.00 (1)	68.98 (0)	600.00 (1)	600.00 (1)	69.45 (0)
12 (1)	600.00 (1)	600.00 (1)	84.41 (0)	600.00 (1)	600.00 (1)	84.11 (0)
13 (1)	600.00 (1)	600.00 (1)	99.55 (0)	600.00 (1)	600.00 (1)	99.52 (0)
14 (1)	600.00 (1)	600.00 (1)	117.39 (0)	600.00 (1)	600.00 (1)	117.15 (0)
15 (1)	600.00 (1)	600.00 (1)	138.45 (0)	600.00 (1)	600.00 (1)	137.51 (0)
16 (1)	600.00 (1)	600.00 (1)	163.12 (0)	600.00 (1)	600.00 (1)	158.43 (0)
17 (1)	600.00 (1)	600.00 (1)	184.99 (0)	600.00 (1)	600.00 (1)	181.94 (0)
18 (1)	600.00 (1)	600.00 (1)	208.83 (0)	600.00 (1)	600.00 (1)	210.82 (0)
19 (1)	600.00 (1)	600.00 (1)	236.98 (0)	600.00 (1)	600.00 (1)	237.45 (0)
20 (1)	600.00 (1)	600.00 (1)	267.54 (0)	600.00 (1)	600.00 (1)	268.60 (0)
21 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)	600.00 (1)

Table 2: RSTrack experiments: variable number of conference tracks, single answer set.

*6.3.2 Reviewer Selection (*RS*)*

Our second benchmark is Reviewer Selection (RS): we represent c conference tracks, r reviewers and p papers. Papers and reviewers are assigned to conference tracks, and there are conflicts between reviewers and papers, some of which are given by external atoms. We consider two scenarios: RSTrack and RSPaper. They are designed to measure the effect of external atoms on the elimination of a large number of answer set candidates; in contrast to the MCS experiments we can control this aspect in the RS experiments.

In RSTrack we vary the number c of conference tracks, where each track has 20 papers and 20 reviewers. Each paper must get two reviews, and no reviewer must get more than two papers. Conflicts are dense such that only one valid assignment exists per track, hence each instance has exactly one answer set, and in each track two conflicts are external. For each number c there is only one instance because RSTrack instances are not randomized. The results of RSTrack are shown in Table 2: runtimes of the old evaluation heuristics (*H1*) grow fastest with size, without using decomposition grows slightly slower but also reaches timeout at size 9. Only the new decomposition (*H2* heuristics) can deal with size 20 without timeout. Finding the first answer set and enumerating all answer sets show very similar times, as RSTrack instances have a single answer set and finding it seems hard.

In RSPaper we fix the number of tracks to $c = 5$; we vary the number p of papers in each track and set the number of reviewers to $r = p$. Each paper must get three reviews and each reviewer must not get more than three papers assigned. Conflicts are randomized and less dense than in RSTrack: the number of answer sets is greater than one and does not grow with the instance size. Over all tracks and papers, $2p$ randomly chosen conflicts are external, and we generate 10 random instances per size and report results averaged per instance size in Table 3. As clearly seen, our new method is always faster than the other methods, and

Instance Size	First Answer Set			All Answer Sets		
	H1	w/o framework	H2	H1	w/o framework	H2
5 (10)	1.06 (0)	0.28 (0)	0.21 (0)	2.25 (0)	0.43 (0)	0.23 (0)
8 (10)	8.76 (0)	2.73 (0)	0.38 (0)	14.73 (0)	4.54 (0)	0.44 (0)
11 (10)	108.70 (1)	83.26 (1)	0.98 (0)	171.01 (2)	104.84 (1)	1.28 (0)
14 (10)	180.99 (2)	125.83 (1)	2.08 (0)	299.22 (4)	245.62 (3)	2.67 (0)
17 (10)	418.92 (6)	364.95 (5)	5.15 (0)	549.01 (9)	513.21 (8)	8.14 (0)
20 (10)	485.35 (8)	453.39 (7)	7.32 (0)	507.66 (8)	501.74 (8)	14.45 (0)
23 (10)	542.03 (9)	508.75 (8)	13.91 (0)	600.00 (10)	600.00 (10)	23.16 (0)
26 (10)	600.00 (10)	600.00 (10)	33.20 (0)	600.00 (10)	600.00 (10)	154.51 (2)
29 (10)	600.00 (10)	600.00 (10)	60.78 (0)	600.00 (10)	600.00 (10)	108.03 (0)
32 (10)	600.00 (10)	600.00 (10)	129.95 (0)	600.00 (10)	600.00 (10)	315.56 (4)
35 (10)	600.00 (10)	600.00 (10)	136.84 (0)	600.00 (10)	600.00 (10)	302.90 (3)
38 (10)	600.00 (10)	600.00 (10)	308.92 (3)	600.00 (10)	600.00 (10)	441.06 (6)
41 (10)	600.00 (10)	600.00 (10)	421.69 (6)	600.00 (10)	600.00 (10)	529.80 (8)
44 (10)	600.00 (10)	600.00 (10)	470.61 (7)	600.00 (10)	600.00 (10)	553.19 (9)
47 (10)	600.00 (10)	600.00 (10)	485.60 (7)	600.00 (10)	600.00 (10)	529.00 (8)
50 (10)	600.00 (10)	600.00 (10)	485.07 (7)	600.00 (10)	600.00 (10)	526.66 (8)

Table 3: RSPaper experiments: variable number of papers/reviewers, multiple answer sets, randomized.

evaluation without a decomposition framework performs slightly better than the old method. Different from RSTrack, we can see a clear difference between finding the first answer set and enumerating all answer sets as RSPaper instances have more than one answer set.

To confirm that the new method is geared towards handling many external atoms, we conducted also experiments with instances that had few external atoms for eliminating answer set candidates but many local constraints. For such highly constrained instances, the new decomposition framework is not beneficial as it incurs an overhead compared to the monolithic evaluation that increases runtimes.

6.3.3 Summary

The results demonstrate a clear improvement using the new framework; they can often be further improved by fine-tuning the grounding and solving algorithm, and by customizing the default heuristics of the framework, as discussed by Eiter et al. (2014b; 2014a), and Redl (2014). However, already the default settings yield results that are significantly better than using the previous evaluation method or using no framework at all; note that the latter requires an advanced grounding algorithm as by Eiter et al. (2014a), which was not available at the time the initial evaluation approach as by Schindlauer (2006) was developed.

In conclusion, the evaluation framework in Section 5 pushes HEX-programs towards scalability for realistic instance sizes, which previous evaluation techniques missed.

7 Related Work and Discussion

We now discuss our results in the context of related work, and will address possible optimizations.

7.1 Related Work

We first discuss related approaches for integrating external reasoning into ASP formalisms, then we discuss work related to the notion of rule dependencies that we introduced in Section 4.1, we discuss related notions of modularity and program decomposition. Finally we relate our splitting theorems to other splitting theorems in the literature.

7.1.1 External Sources

The dlv-ex system (Calimeri et al. 2007) was a pioneering work on value invention through external atoms in ASP. It supported VI-restricted programs, which amount to HEX-programs under extensional semantics without higher-order atoms and a strong safety condition that is subsumed by lde-safety. Answer set computation followed the traditional approach on top of dlv, but used a special progressive grounding method (thus an experimental comparison to solving, i.e., model building as in the focus of this paper, is inappropriate).

With respect to constraint theories and ASP, several works exist. The ACsolver system (Mellarkod et al. 2008), the Ezcsp system (Balduccini 2009), and the clingcon system (Ostrowski and Schaub 2012) divide the program into ASP-literals and constraint-literals, which can be seen as a special case of HEX-programs that focuses on a particular external source. As for evaluation, an important difference to general external sources is that constraint atoms do not use value invention. The modularity techniques from above are less relevant for this setting as grounding the overall program in one shot is possible. However, this also fits into our framework as disabling decomposition in fact corresponds to a dedicated (trivial) heuristics which keeps the whole program as a single unit. For a detailed comparison between ACsolver, Ezcsp, and clingcon see (Lierler 2014).

Balduccini and Lierler (2013) also experimentally compared Ezcsp and clingcon while varying the degree of integration between the constraint solver and the ASP solver backend. Their 'black-box integration' corresponds with dlvhex's integration of the dlv backend: external atom semantics are verified by plugins callbacks only when a full answer set candidates has been found in the backend; moreover their 'clear-box integration' corresponds with dlvhex's integration of the clasp solver backend: plugin callbacks are part of the CDCL propagation and can operate on partial answer set candidates. Note that constraint answer set programs can be realized as a dlvhex-plugin (such an effort is currently ongoing).[10]

We also remark that gringo and clasp use a concept called 'external atoms' for realizing various applications such as constraint ASP solving as in clingcon and incremental solving (Gebser et al. 2014). However, despite their name they are different from external atoms in HEX-programs. In the former case, external atoms are excluded from grounding-time optimization such that these atoms are not eliminated even if their truth value is deterministically false during grounding. This allows to add rules that found truth of such atoms in later incremental grounding steps. In case of HEX the truth value is determined by external sources. Moreover gringo contains an interface for Lua and Python functions that can perform computations during grounding. HEX external atoms are more expressive: they cannot

[10] `http://github.com/hexhex/caspplugin`

always be evaluated during grounding because their semantics is defined with respect to the answer set.

7.1.2 Rule Dependencies

In the context of answer set programming, dependency graphs over rules have been used earlier, e.g., by Linke (2001) and Linke and Sarsakov (2004). However, these works consider only ordinary ground programs, and furthermore the graphs are used for characterizing and computing the answer sets of a program from these graphs. In contrast, we consider nonground programs with and external atoms, and we use the graph to split the program into evaluation units with the goal of modularly computing answer sets.

7.1.3 Modularity

Our work is naturally related to work on program modularity under stable model semantics, as targeted by splitting sets (Lifschitz and Turner 1994) and descendants, with the work by Oikarinen and Janhunen (2008) and Janhunen et al. (2009) a prominent representative that lifted them to modular programs with choice rules and disjunctive rules, by considering 'symmetric splitting'. Other works, e.g., by Lierler and Truszczynski (2013) go further to *define* semantics of systems of program modules, departing from a mere semantics-preserving decomposition of a larger program into smaller parts, or consider multi-language systems that combine modules in possibly different formalisms on equal terms (cf. e.g. Järvisalo et al. (2009) and Tasharrofi and Ternovska (2011)).

Comparing the works by Oikarinen and Janhunen (2008) and Janhunen et al. (2009) as, from a semantic decomposition perspective, the closest in this group to ours, an important difference is that our approach works for non-ground programs and explicitly considers possible overlaps of modules. It is tailored to efficient evaluation of arbitrary programs, rather than to facilitate module-style logic programming with declarative specifications, or to provide compositional semantics for modules beyond uni-lateral evaluation, as done by Järvisalo et al. (2009) and Tasharrofi and Ternovska (2011); for them, introducing values outside the module domain (known as *value invention*) does not play a visible role. In this regard, it is in line with previous HEX-program evaluation (Eiter et al. 2006) and decomposition techniques to ground ordinary programs efficiently (Calimeri et al. 2008).

7.1.4 Splitting Theorems

Our new splitting theorems compare to related splitting theorems as follows.

Theorem 1 is similar to Theorem 4.6.2 by Schindlauer (2006); however, we do not use splitting sets on atoms, but splitting sets on rules. Furthermore, Schindlauer (2006) has no analog to Theorem 2.

The seminal Splitting Set Theorem by Lifschitz and Turner (1994) divides the interpretation of P into disjoint sets X and Y, where X is an answer set of the 'bottom' $gb_A(P) \subseteq P$ and Y is an answer set of a 'residual' program obtained from $P \setminus gb_A(P)$ and X. In the residual program, all references to atoms in X are removed, in a way that it semantically behaves as if facts X were added to $P \setminus gb_A(P)$, while the answer sets of the residual do not contain any atom in X. This works nicely for answer set programs, but it is problematic when applied to HEX programs, because external atoms may depend on the bottom and on atoms in heads of the residual program; hence, they cannot be eliminated from rule bodies. The only way to eliminate bottom facts from the residual program would be to 'split' external atoms semantically into a part depending on the bottom and the program remainder, and by replacing external atoms in rules with external atoms that have been partially evaluated wrt. a bottom answer set. Technically, this requires to introduce new external atoms, and formulating a splitting theorem for HEX programs with two disjoint interpretations X and Y is not straightforward. Furthermore, such external atom splitting and partial evaluation might not be possible in a concrete application scenario.

Different from the two splitting theorems recalled above, the Global Splitting Theorem by Eiter et al. (2006) does not split an interpretation of the program P into disjoint interpretations X and Y, and thus should be compared to our Theorem 2. However, the Global Splitting Theorem does not allow constraint sharing, and it involves a residual program which specifies how external atoms are evaluated via 'replacement atoms', which lead to extra facts D in the residual program that must be removed from its answer sets. Both the specification of replacement atoms and the extra facts make the Global Splitting Theorem cumbersome to work with when proving correctness of HEX encodings. Moreover, the replacement atoms are geared towards a certain implementation technique which however is not mandatory and can be avoided.

Lemma 5.1 by Eiter et al. (1997) is structurally similar to our Theorem 2: answer sets of the bottom program are evaluated together with the program depending on the bottom (here called the residual), hence answer sets of the residual are answer sets of the original program. However, the result was based on atom dependencies and did neither consider negation nor external atoms.

In sumary our new Generalized Splitting Theorem has the following advantages.

- By moving from atom to rule splitting sets, no separate definition of the bottom is needed, which just becomes the (rule) splitting set.
- As regards HEX-programs, splitting is simple (and not troubled) if all atoms that are true in an answer set of the bottom also appear in the residual program. Typically, this is not the case in results from the literature.
- Finally, also the residual program itself is simpler (and easier to construct), by just dropping rules and adding facts. No rule rewriting needs to be done, and no extra facts need to be introduced in the residual program nor in the bottom.

The only (negligible) disadvantage of the new theorems is that the answer sets of the bottom and the residual program may no longer be disjoint; however, each residual answer set includes some (unique) bottom answer set.

7.2 Possible Optimizations

Evaluation graphs naturally encode parallel evaluation plans. We have not yet investigated the potential benefits of this feature in practice, but this property allows us to do parallel solving based on solver software that does not have parallel computing capabilities itself ('parallelize from outside'). This applies both to programs with external atoms, as well as to ordinary ASP programs (i.w., without external atoms). Improving reasoning performance by decomposition has been investigated by Amir and McIlraith (2005), however, only wrt. monotonic logics.

Improving HEX evaluation efficiency by using knowledge about domain restrictions of external atoms has been discussed by Eiter et al. (2009). These rewriting methods yield partially grounded sets of rules which can easily be distributed into distinct evaluation units by an optimizer. This directly provides efficiency gains as described in the above work.

As a last remark on possible optimizations, we observe that the data flow between evaluation units can be optimized using proper notions of model projection, such as in (Gebser et al. 2009a). 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.

8 Conclusion

HEX-programs extend answer set programs with access to external sources through an API-style interface, which has been fruitfully deployed to various applications. Providing efficient evaluation methods for such programs is a challenging but important endeavor, in order to enhance the practicality of the approach and to make it eligible for a broader range of applications. In this direction, we have presented in this article a novel evaluation method for HEX-programs based on modular decomposition. We have presented new results for the latter using special splitting sets, which are more general than previous results and use rule sets as a basis for splitting rather than sets of atoms as in previous approaches. Furthermore, we have presented an evaluation framework which employs besides a traditional evaluation graph that consists of program components and reflects syntactic dependencies among them, also a model graph whose nodes collect answer sets that are combined and passed on between components. Using decomposition techniques, evaluation units can be dynamically formed and evaluated in the framework using different heuristics, Moreover, the answer sets of the overall program can be produced in a streaming fashion. The new approach leads in combination with other techniques to significant improvements for a variety of applications, as demonstrated by Eiter et al. (2014a; 2014b) and Redl (2014). Notably, while our results target HEX-programs, the underlying concepts and techniques are not limited to them (e.g., to separate the evaluation and the model graph) and may be fruitfully transferred to other rule-based formalisms.

8.1 Outlook

The work we presented can be continued 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 and evaluation strategies. Indeed, the generic notions of evaluation unit, evaluation graph 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 (a combination of) time, space, or parallelization conditions. Third, the framework is amenable to a form of coarse-grained distributed computation at the level of evaluation units (in the style of Perri et al. (2010)).

While modular evaluation is advantageous in many applications, it can also be counterproductive, as currently the propagation of knowledge learned by conflict-driven techniques into different evaluation units is not possible. In such cases, evaluating the program as a single evaluation unit is often also infeasible due to the properties of the grounding algorithm, as observed by Eiter et al. (2014a). Thus, another starting point for future work is a tighter integration of the solver instances used to evaluate different units, e.g., by exchanging learned knowledge. In this context, also the interplay of the grounder and the solver is an important topic.

Acknowledgements

We would like to thank the anonymous reviewers and Michael Gelfond for their constructive feedback.

References

AMIR, E. AND MCILRAITH, S. A. 2005. Partition-based logical reasoning for first-order and propositional theories. *Artificial Intelligence 162,* 1-2, 49–88.

BAIRAKDAR, S. E.-D., DAO-TRAN, M., EITER, T., FINK, M., AND KRENNWALLNER, T. 2010. The DMCS solver for distributed nonmonotonic multi-context systems. In *European Conference on Logics in Artificial Intelligence (JELIA)*. Springer, 352–355.

BALDUCCINI, M. 2009. Representing Constraint Satisfaction Problems in Answer Set Programming. In *Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP).*

BALDUCCINI, M. AND LIERLER, Y. 2013. Hybrid Automated Reasoning Tools: from Black-box to Clear-box Integration. In *Answer Set Programming and Other Computing Paradigms (ASPOCP)*. 17–31.

BASOL, S., ERDEM, O., FINK, M., AND IANNI, G. 2010. HEX programs with action atoms. In *Technical Communications of the International Conference on Logic Programming (ICLP)*. 24–33.

BÖGL, M., EITER, T., FINK, M., AND SCHÜLLER, P. 2010. The MCS-IE system for explaining inconsistency in multi-context systems. In *European Conference on Logics in Artificial Intelligence (JELIA)*. 356–359.

BREWKA, G. AND EITER, T. 2007. Equilibria in heterogeneous nonmonotonic multi-context systems. In *AAAI Conference on Artificial Intelligence*. AAAI Press, 385–390.

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

CALIMERI, F., COZZA, S., AND IANNI, G. 2007. External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence 50,* 3–4, 333–361.

CALIMERI, F., COZZA, S., IANNI, G., AND LEONE, N. 2008. Computable functions in ASP: Theory and implementation. In *ICLP*. LNCS. Springer, 407–424.

CALIMERI, F., FINK, M., GERMANO, S., IANNI, G., REDL, C., AND WIMMER, A. 2013. AngryHEX: an artificial player for angry birds based on declarative knowledge bases. In *National Workshop and Prize on Popularize Artificial Intelligence*. 29–35.

CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM 50,* 5, 752–794.

DAO-TRAN, M., EITER, T., AND KRENNWALLNER, T. 2009. Realizing default logic over description logic knowledge bases. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*. Springer, 602–613.

DUNG, P. M. 1995. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence 77,* 2, 321–357.

EITER, T., FINK, M., IANNI, G., KRENNWALLNER, T., AND SCHÜLLER, P. 2011. Pushing efficient evaluation of HEX programs by modular decomposition. In *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. 93–106.

EITER, T., FINK, M., AND KRENNWALLNER, T. 2009. Decomposition of Declarative Knowledge Bases with External Functions. In *International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI Press, 752–758.

EITER, T., FINK, M., KRENNWALLNER, T., AND REDL, C. 2012. Conflict-driven ASP solving with external sources. *Theory and Practice of Logic Programming 12,* 4-5, 659–679.

EITER, T., FINK, M., KRENNWALLNER, T., AND REDL, C. 2013. Liberal Safety Criteria for HEX-Programs. In *Twenty-Seventh AAAI Conference (AAAI 2013), July 14–18, 2013, Bellevue, Washington, USA* (July 14–18, 2013), M. desJardins and M. Littman, Eds. AAAI Press. To appear.

EITER, T., FINK, M., KRENNWALLNER, T., AND REDL, C. 2014a. Domain expansion for ASP-programs with external sources. Tech. Rep. INFSYS RR-1843-14-02, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria.

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

EITER, T., GOTTLOB, G., AND MANNILA, H. 1997. Disjunctive datalog. *ACM Transactions on Database Systems 22,* 3, 364–418.

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

EITER, T., IANNI, G., SCHINDLAUER, R., AND TOMPITS, H. 2005. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In *International Joint Conference on Artificial Intelligence (IJCAI)*. Professional Book Center, 90–96.

EITER, T., IANNI, G., SCHINDLAUER, R., AND TOMPITS, H. 2006. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In *European Semantic Web Conference (ESWC)*. Springer, 273–287.

FABER, W., LEONE, N., AND PFEIFER, G. 2004. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *European Conference on Logics in Artificial Intelligence (JELIA)*. Springer, 200–212.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2014. Clingo = ASP + control: Preliminary report. *CoRR abs/1405.3694*.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2009a. Solution enumeration for projected boolean search problems. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*. Springer, 71–86.

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

GEBSER, M., OSTROWSKI, M., AND SCHAUB, T. 2009b. Constraint answer set solving. In *International Conference on Logic Programming (ICLP)*. Springer, 235–249.

GELFOND, M. AND LIFSCHITZ, V. 1988. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings of the 5th International Conference and Symposium*, R. Kowalski and K. Bowen, Eds. MIT Press, 1070–1080.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9,* 3/4, 365–386.

HAVUR, G., OZBILGIN, G., ERDEM, E., AND PATOGLU, V. 2014. Geometric Rearrangement of Multiple Movable Objects on Cluttered Surfaces: A Hybrid Reasoning Approach. In *International Conference on Robotics and Automation (ICRA)*. 445–452.

HOEHNDORF, R., LOEBE, F., KELSO, J., AND HERRE, H. 2007. Representing default knowledge in biomedical ontologies: Application to the integration of anatomy and phenotype ontologies. *BMC Bioinformatics 8,* 1, 377.

JANHUNEN, T., OIKARINEN, E., TOMPITS, H., AND WOLTRAN, S. 2009. Modularity Aspects of Disjunctive Stable Models. *Journal of Artificial Intelligence Research 35*, 813–857.

JÄRVISALO, M., OIKARINEN, E., JANHUNEN, T., AND NIEMELÄ, I. 2009. A module-based framework for multi-language constraint modeling. In *Logic Programming and Nonmonotonic Reasoning (LPNMR)*. 155–168.

LASSILA, O. AND SWICK, R. 1999. Resource description framework (RDF) model and syntax specification. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222`.

LIERLER, Y. 2014. Relating constraint answer set programming languages and algorithms. *Artificial Intelligence 207*, 1–22.

LIERLER, Y. AND TRUSZCZYNSKI, M. 2013. Modular answer set solving. In *Late-Breaking Developments in the Field of Artificial Intelligence, Bellevue, Washington, USA, July 14-18, 2013*. AAAI Workshops, vol. WS-13-17. AAAI.

LIFSCHITZ, V. AND TURNER, H. 1994. Splitting a Logic Program. In *Proceedings ICLP-94*. MIT-Press, Santa Margherita Ligure, Italy, 23–38.

LINKE, T. 2001. Graph Theoretical Characterization and Computation of Answer Sets. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 641–645.

LINKE, T. AND SARSAKOV, V. 2004. Suitable graphs for answer set programming. In *LPAR*, F. Baader and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 3452. Springer, 154–168.

MELLARKOD, V. S., GELFOND, M., AND ZHANG, Y. 2008. Integrating Answer Set Programming and Constraint Logic Programming. *Annals of Mathematics and Artificial Intelligenc 53,* 1-4, 251–287.

MOSCA, A. AND BERNINI, D. 2008. Ontology-driven geographic information system and dlvhex reasoning for material culture analysis. In *Italian Workshop RiCeRcA at ICLP*.

NIEMELÄ, I. 1999. Logic programming with stable model semantics as constraint programming paradigm. *Annals of Mathematics and Artificial Intelligenc 25,* 3–4, 241–273.

OIKARINEN, E. AND JANHUNEN, T. 2008. Achieving compositionality of the stable model semantics for smodels programs. *TPLP 8,* 5-6, 717–761.

OSTROWSKI, M. AND SCHAUB, T. 2012. ASP modulo CSP: the clingcon system. *Theory and Practice of Logic Programming (TPLP) 12,* 4-5, 485–503.

PALÙ, A. D., DOVIER, A., PONTELLI, E., AND ROSSI, G. 2009. Gasp: Answer set programming with lazy grounding. *Fundamenta Informaticae 96,* 3, 297–322.

PERRI, S., RICCA, F., AND SIRIANNI, M. 2010. A parallel ASP instantiator based on DLV. In *Declarative Aspects of Multicore Programming (DAMP'10)*. LNCS. Springer, 73–82.

POLLERES, A. 2007. From SPARQL to rules (and back). In *International Conference on World Wide Web (WWW)*. ACM, 787–796.

PRZYMUSINSKI, T. C. 1988. On the Declarative Semantics of Deductive Databases and Logic Programs. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufman, 193–216.

PRZYMUSINSKI, T. C. 1991. Stable semantics for disjunctive programs. *New Generation Computing 9*, 401–424.

REDL, C. 2014. Answer set programming with external sources: Algorithms and efficient evaluation. Ph.D. thesis, Vienna University of Technology.

ROSS, K. 1994. Modular Stratification and Magic Sets for Datalog Programs with Negation. *Journal of the ACM 41,* 6, 1216–1267.

SCHINDLAUER, R. 2006. Answer set programming for the semantic web. Ph.D. thesis, Vienna University of Technology, Vienna, Austria.

SCHÜLLER, P. 2012. Inconsistency in multi-context systems: Analysis and efficient evaluation. Ph.D. thesis, Vienna University of Technology, Vienna, Austria.

SCHÜLLER, P., PATOGLU, V., AND ERDEM, E. 2013. A Systematic Analysis of Levels of Integration between Low-Level Reasoning and Task Planning. In *Workshop on Combining Task and Motion Planning at IEEE International Conference on Robotics and Automation (ICRA)*.

SHEN, Y., WANG, K., EITER, T., FINK, M., REDL, C., KRENNWALLNER, T., AND DENG, J. 2014. FLP answer set semantics without circular justifications for general logic programs. *Artificial Intelligence 213*, 1–41.

TASHARROFI, S. AND TERNOVSKA, E. 2011. A semantic account for modularity in multi-language modelling of search problems. In *International Symposium on Frontiers of Combining Systems (FroCoS)*. 259–274.

WANG, Y., YOU, J., YUAN, L., SHEN, Y., AND ZHANG, M. 2012. The loop formula based semantics of description logic programs. *Theor. Comput. Sci. 415*, 60–85.

ZAKRAOUI, J. AND ZAGLER, W. L. 2012. A method for generating CSS to improve web accessibility for old users. In *Int. Conf. on Computers Helping People with Special Needs (ICCHP)*. 329–336.

ZIRTILOĞLU, H. AND YOLUM, P. 2008. Ranking semantic information for e-government: complaints management. In *International Workshop on Ontology-supported business intelligence (OBI)*. ACM.