Problem Solving Using the HEX Family*

Thomas Eiter[1], Christoph Redl[1], and Peter Schüller[2]

[1] Technische Universität Wien, Institute für Informationssysteme, Knowledge Based Systems Group, Vienna, Austria
{eiter,redl}@kr.tuwien.ac.at
[2] Marmara University, Faculty of Engineering, Department of Computer Engineering, Istanbul, Turkey
peter.schuller@marmara.edu.tr

Abstract. The HEX formalism has been designed as an extension of answer set programs that offers an abstract interface to access external sources of information and computation, such as the World Wide Web or description logics reasoners. The generic nature makes the extension powerful, which has been exploited in different ways: as an end user problem solving language, as a backend formalism, or as the basis of a richer formalism with possibly increased expressiveness. The increasing spread of HEX is paralleled with the frequently asked questions of what HEX is, and how it can be used for problem solving. In this paper, we aim to answer these questions; we consider different scenarios and provide a methodology for applying HEX from a user perspective. Furthermore, we briefly present a collection of applications based on HEX or derived from it, including sample snippets from associated HEX programs.

1 Introduction

Answer Set Programming (ASP) is a declarative problem solving approach [43,45,48], in which a problem is described by the rules of a nonmonotonic logic program, such that the answer sets [33] (i.e., specific models) of the program correspond to the solutions of the problem; the latter can be extracted from the answer sets computed using an ASP solver. With the advent of efficient and expressive such solvers (e.g., smodels [58], dlv [41], ASSAT [44], and Gringo plus Clasp [30, 31]), this approach has been fruitfully deployed to a growing range of applications in different areas and disciplines, cf. [8].

However, the World Wide Web and trends in distributed systems have created a need for accessing external information sources in a program, ranging from light-weight data access (e.g., XML, RDF, or data bases) to knowledge-intensive formalisms (e.g., description logics reasoners), and even to information sources not based on logical grounds (e.g., dictionaries or route planning services). To cater for this need, *HEX programs* have been introduced in [13] as an extension to nonmonotonic logic programs in which access to external

* This research has been supported by the Austrian Science Fund (FWF) project P27730 and the Scientific and Technological Research Council of Turkey (TUBITAK) Grant 114E430.

sources is possible via designated external atoms, which abstractly define external predicates whose valuation is determined by external computation.

For a simple example, consider the rule

$$pointsTo(X, Y) \leftarrow \&hasHyperlink[X](Y), url(X); \tag{1}$$

informally, it obtains pairs (X, Y) of URLs, where X actually links Y on the Web. Here, *&hasHyperlink* is an *external predicate* associated with an external computation function; X is the input for the latter and Y is a result. Besides single values, also relational information (predicate extensions) can flow from the program to external sources and back; e.g., an extended input $[X, skip]$ in (1) may provide a relation *skip* containing pairs of URLs whose linkage should be omitted. Notably, the output Y may involve values not occurring in the program (known as *value invention*), and Y may influence the input of the atom where in practice, certain safety conditions must be obeyed [15]. This makes the efficient evaluation of HEX programs challenging, for which advanced techniques have been developed [13, 14, 16, 18].

The abstract concept of an external atom has been realized in the open-source software DLVHEX[3] as an API, which allows the user via a plugin mechanism to tailor external atoms for her needs using Python or C++. This makes the system very powerful; depending on the external evaluation cost, HEX programs offer a range of problem solving capacity, from Σ_2^p for polynomial-time external atoms to Turing-completeness in general.

HEX programs and DLVHEX have been used for solving diverse kinds of problems. This frequently raises the questions of prospective users what HEX is after all, and how HEX programs or DLVHEX can be exploited for solving their applications; notably DLVHEX offers a growing suite of library plugins that have been used in different applications.

This paper addresses these questions and focuses on HEX programs as a KR tool for problem solving, which they support at different levels of abstraction, namely as an end user problem solving language, as a backend formalism or as the basis of a richer formalism with possibly increased expressiveness. Besides a methodology for using HEX programs, we further present some examples of HEX applications. More in detail, we proceed on this as follows.

- After recalling in Section 2 HEX programs and briefly addressing some aspects of the DLVHEX system, we show in Section 3 how HEX programs can be used for problem solving. Besides a basic methodology, which is a strict generalization of the ASP methodology, we present typical kinds of external sources and HEX use scenarios.
- In Section 4 we then consider some end user applications which have been realized on top of HEX programs.
- In Section 5 we show how several extensions of the HEX formalism and some of their internals. While some extensions increase the expressiveness and need extensions of the internal evaluation algorithms, many others are in fact only syntactic shortcuts and can be compiled to pure HEX programs.

[3] www.kr.tuwien.ac.at/research/systems/dlvhex

– In Section 6 we show applications that use HEX as a backend formalism for the sake of evaluation, sometimes completely hiding HEX from the user.

After a discussion of related work in Section 7, we conclude in Section 8 with an outlook on future topics.

This paper is in honor of Gabriele Kern-Isberner, who has worked extensively on Artificial Intelligence and in particular on knowledge representation and reasoning, and made over many years numerous important contributions to belief change, conditional and nonmonotonic reasoning, reasoning with uncertainty, argumentation, agents etc., based on solid and deep mathematical foundations. Gabriele's excellent book "Methods of Knowledge based Systems – Foundations, Algorithms and Applications" [5], co-authored by Christoph Beierle and now in its fifth edition, demonstrates her comprehension of the field, and provides a coherent and formally guided introduction to the ingredients of knowledge based systems. Unfortunately, the book is only available in German; but in the style of Cato the Elder, Thomas keeps asking her ever since the first edition to have an English translation, so that a much larger audience can read and enjoy it, which would be well-deserved – and that we can use it in our English classes! In turn, we hope that Gabriele may discover the wealth of the HEX family as a tool box for building advanced knowledge based systems, as envisaged in her book and further supported by her many research results.

2 HEX Programs

In this section, we formally introduce the syntax and semantics of HEX programs; for more details and background, see e.g. [13, 23, 24, 55].

2.1 HEX Program Syntax

Let $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ be mutually disjoint sets of *constants*, *variables*, and *external predicates*, respectively. Usually constants (resp., variables) are denoted with first letter in upper case (resp., lower case), while external predicates start with '&'. Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. An *atom* is a tuple $(Y_0, Y_1, \ldots, Y_n)$, where $Y_0, \ldots, Y_n$ are terms and $n \geq 0$ is the *arity* of the atom. Intuitively, Y_0 is the predicate name, and we often use the more familiar notation $Y_0(Y_1, \ldots, Y_n)$. An atom is *ordinary* (resp., *higher-order*) if Y_0 is a constant (resp., a variable), and it is *ground*, if all its terms are constants.

An *external atom* is of the form

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m),$$

where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called *input* and *output* lists, resp.), 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. In the ground case, the input terms $Y_1, \ldots, Y_n$ intuitively consist of individual constants (e.g. *joe*) and predicate names (e.g. *edge*). An external atom provides a way for deciding the truth value of an output tuple depending on the input tuple and a given interpretation.

A *rule* r is of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, \textbf{not } \beta_{n+1}, \ldots, \textbf{not } \beta_m, \qquad m, k \geq 0,$$

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

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.

Example 1. Consider the following program Π_{goto} to decide where to go for a city trip, but exclude cities where the (external) weather report is bad.

$$\begin{aligned} &\mathit{badweather}(\mathit{rain}). \qquad \mathit{badweather}(\mathit{snow}). \\ &\mathit{goto}(\mathit{paris}) \vee \mathit{goto}(\mathit{london}). \\ &\leftarrow \&\mathit{weatherreport}[\mathit{goto}](W), \mathit{badweather}(W). \end{aligned}$$

We guess where to go and forbid that the externally obtained weather report $\&\mathit{weatherreport}$ indicates bad weather for a city in the extension of *goto*. □

2.2 HEX Program Semantics

The semantics of HEX programs generalizes the well-known answer-set semantics of ordinary programs [33]. Given a HEX program P, its *Herbrand base*, 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 $\mathcal{C}$ of constants 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.

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 $(I, \vec{y}, \vec{x})$ where $\vec{y} = y_1, \ldots, y_n$ and $\vec{x} = x_1, \ldots, x_m$ 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 $a = \&g[\vec{y}](\vec{x})$, denoted $I \models a$, if $f_{\&g}(I, \vec{y}, \vec{x}) = 1$. This definition of external atom semantics is very general; indeed an external atom may depend on every part of the interpretation. For practical reasons, external atom semantics is usually restricted such that it depends only on the extension of those predicates in I that are given in the input list.

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.

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.

HEX programs are a conservative extension of disjunctive [33] (resp., normal [32]) logic programs under the answer set semantics.

Example 2 (cont'd). Assume that the weather report for *paris* is *sun* and for *london* it is *rain*, then $I \models \&weatherreport[goto](sun)$ if $I \models goto(paris)$, moreover $I \models \&weatherreport[goto](rain)$ if $I \models goto(london)$, and Π_{goto} has one answer set $\{goto(paris)\}$ (we omit atoms in facts of Π_{goto}). If weather reports of both cities are sunny, we additionally obtain the answer set $\{goto(london)\}$. Finally if the weather report for both cities is *snow*, there is no answer set. □

2.3 Usability Issues

When realizing a project with HEX and the DLVHEX reasoner, besides writing the HEX program it is usually also necessary to write a plugin which implements the semantics of external atoms to be used (unless an already existing plugin can be reused). Plugins may be implemented either in Python or C++ using a reasoner API provided by DLVHEX; for details, see [26].

For the sake of performance improvements, external atom semantics implementations can further inject *nogoods* into the solver process, i.e., combinations of truth values of atoms which are inconsistent with the external atom semantics and cannot occur in any answer set. This allows for eliminating inconsistent guesses earlier and might speed up the solving process.

Besides defining external atoms, plugins may also (i) rewrite the input program, and (ii) post-process answer sets. Rewriting the input is useful for creating language extensions (see Section 5) or for changing the behavior of an input program by modifying its code (e.g., for performance reasons or for debugging). Post-processing answer sets allows for translating the answer sets into a more application-specific presentation. This is useful when HEX is used as a backend for other KR formalisms, cf. Section 6.

The DLVHEX user manual [26] presents examples for HEX programs and the corresponding implementations of external atom semantics. Furthermore, it describes different ways of obtaining, building, and installing the DLVHEX solver for Linux (in particular Ubuntu), Mac OS X, and soon Windows.

3 KR Problem Solving using HEX

In this section, we show how the HEX family can be used for declarative problem solving. To this end, we first present the basic methodology in Section 3.1, and

show how modeling techniques from ordinary ASP can be generalized to HEX. We provide methodology for using external atoms in Section 3.2 and further distinguish typical kinds of external sources. Roughly, one can classify them as outsourcing of either computation or information, or as a combination thereof. Afterwards, we present three kinds of use case scenarios in Section 3.3. HEX programs can either directly be used as a formalism for modeling end user applications, as a basis for language extensions (i.e., extensions of HEX which are compiled into plain HEX), or as backend for the realization of other KR formalisms. The three use cases are orthogonal to the types of external sources as in any scenario one may use all types of external sources.

3.1 Basic Methodology

Because the HEX family is an extension of ASP, all modeling techniques from ASP may also be used in HEX programs. One of the most important examples is the *guess and check paradigm*, where default negation or disjunctive rules are used to generate a superset of the intended solutions (*guessing part*), and constraints are used to eliminate spurious candidates (*checking part*). For instance, if we assume that facts over predicates *node* and *edge* define a graph, then the well-known graph 3-colorability problem can be solved by guessing all possible colorings of the nodes of a graph using the disjunctive rule

$$g\colon\ color(red, X) \vee color(green, X) \vee color(blue, X) \leftarrow node(X), \qquad (2)$$

and eliminating all colorings which assign the same color to adjacent nodes using the constraint

$$c\colon\ \leftarrow color(C, X), color(C, Y), edge(X, Y). \qquad (3)$$

However, unlike in ASP, HEX programs allow for using external atoms in addition. They can occur both in the guessing and in the checking part. In the former case, they may be used to import individuals over which guessing is performed. For instance, one may replace the atom $node(X)$ in the body of rule (2) by $\&node[](X)$ to import the nodes of the graph. In the latter case, external atoms can be used in the body of constraints to check given conditions. For instance, rule c may be replaced by

$$c'\colon\ \leftarrow \mathbf{not}\ \&check[color, edge](), \qquad (4)$$

where $\&check[color, edge]()$ is true if *color* is a valid 3-coloring wrt. *edge* and false otherwise.

The *saturation technique* is an advanced modeling technique for solving problems up to Σ_2^P-completeness, by exploiting the subset-minimality of answer sets for checking whether a property holds *for all* guesses in a search space [20]. A typical example is the check if a graph is *not* 3-colorable, i.e., all possible colorings are invalid. Also here, the checking part may employ external atoms.

For more details about ASP modeling techniques we refer to [20, 29].

3.2 Methodology for Using External Atoms

In general, one can roughly distinguish between two main usages of external sources that we call *computation outsourcing* and *information outsourcing*, respectively, and combinations thereof. We stress that this distinction concerns the usage in applications, as both usages are based on the same language constructs. For each of them we will describe some typical use cases that serve as usage patterns for external atoms when writing HEX programs.

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 are of the form $\leftarrow \&forbidden[p_1, \ldots, p_n]()$ eliminate certain extensions of predicates $p_1, \ldots, p_n$ and are a special case of computational outsourcing, see also the 3-colorability example above. The external evaluation of such a constraint can return reasons for conflicts to the reasoner in order to restrict the search space and avoid reconstruction of the same conflict [14]. This technique avoids explicitly grounding the forbidden combinations of atoms as constraints and reduces the size of the ground program. On-demand constraints have been used for efficient planning in robotics where external atoms verify the feasibility of a 3D motion [35, 56].

Computations that cannot (easily) be expressed by rules. Outsourcing computations also allows for including algorithms which cannot (easily or efficiently) be expressed by rules. As a concrete example, an artificial intelligence agent for the skills and tactics game *AngryBirds* needs to perform physics simulations [10]. This requires floating point computations which can not be done by rules in a practical way (this would either come at the costs of very limited precision or a blow-up of the grounding) therefore the physics simulations are integrated with game playing rules as external atoms in a HEX program.

Complexity lifting. This is another kind of computational outsourcing that allows for realizing 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 [27][4], as long as external sources are decidable there is no practical reason for limiting their complexity. External sources can also be other ASP or HEX programs, which allows for encoding other formalisms of higher complexity in HEX programs, e.g., *abstract argumentation frameworks* [12].

Information Outsourcing refers, in contrast to computational outsourcing, to external sources which import information, 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* [40] or *geographic data* [47]. More advanced

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

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 [7]; access to individual contexts has been provided through external atoms [6]. Also sensor data, as often used when planning and executing actions in an environment, is a form of information outsourcing (cf. ACTHEX [4]).

Combinations. It is also possible to outsource computation and information at the same time. A typical example are logic programs with access to Description Logic knowledge bases (DL KB), called *DL-programs* [22]. A DL KB not only stores information, but also provides reasoning services. This allows for interleaving reasoning within the DL KB and the logic program with information that flows across the external atom API in both directions.

3.3 Use Scenarios

One can distinguish between three main types of usages of the HEX formalism. Note that the following classification is orthogonal to the types of external sources above, i.e., each of the following scenarios may make use of various types of external atoms.

End user applications based on HEX. The first scenario is the modeling of *end-user applications*. The HEX language is directly used for modeling a problem at hand and computing its solutions. Note that the problem instance formally consists both of the HEX program and the external sources, but external sources may be reused for different applications if suitable.

The typical procedure when modeling an end user application starts with identifying and realizing the required external sources, followed by writing a HEX program which makes use of these external sources. The two steps may be repeated in order to refine the encoding, i.e., while writing the HEX program, the need for further or modified external sources may arise. In some cases, external atoms of other applications can be reused. Some existing plugins are generic and useful for different applications, e.g., string manipulation functions and an interface to RDF triple stores. We present such applications in Section 4.

HEX language extensions. It turns out that some advanced applications call for additional language features as they can not or not easily by realized in pure HEX programs. A possible relief are language extensions, of which some may be compiled to pure HEX syntax, while others actually increase expressiveness. However, even in cases where language extensions are only syntactic shortcuts, they still not only increase the user comfort but also give the reasoner more specific information about the user's intents (compare this with a constraint $\leftarrow Body$ vs. an equivalent rule $p \leftarrow Body, \mathbf{not}\, p$ in ordinary ASP). This can be exploited to improve efficiency. We present such extensions in Section 5.

HEX as backend formalism. Finally, other formalisms may be implemented on top of HEX programs (or extensions thereof) using an appropriate translation. Instead of encoding the end user application directly, its encoding as a HEX program is automatically generated from a different representation. This

step can either be hidden from the user, or can be transparent such that modifications (e.g., extensions or improvements) can be made prior to evaluation. We present existing applications using HEX as a backend in Section 6.

4 End User Applications Based on HEX

In this section, we consider some end applications of HEX programs, which have been conceived in different domains.

In the context of the Semantic Web, HEX was applied to connect SPARQL and RDF querying with logic programming rules [51]. Moreover, HEX was used for archaeological research in order to combine geographical and cultural knowledge from various ontologies [47], and for adapting user interfaces targeted at elderly and disabled people by combining ontologies about user profiles with rules about potential user interface styles [59].

As described above, an important use case of HEX is planning: the DLV$^{\mathcal{C}}$ planning language can use external atoms for determining effects of actions [49]; moreover in robotic planning, external atoms have been used to perform checks on feasibility of actions or action costs [35, 56]. In the following we describe a specific planning application realized with HEX in more detail.

Route planning. While many commercial and free route planning applications exist (Google Maps is currently perhaps the most popular), the supported query types are usually limited. In contrast, an implementation in HEX programs allows for an easy addition of side constraints and thus tailoring to very specific use cases. As a concrete use-case, [16] considered tours with multiple stops (e.g. at shops, a pharmacy, kindergarden, etc) using an external source that supports only point-to-point queries. Side constraints may include restrictions on the order of stops, the tour length, or opening hours at the stops.

Related to route planning is a trip planning scenario. When planning a holiday trip with multiple stops, the order of the stops is often irrelevant, but one wants to spend a certain number of days at each location. However, due to shifts of the dates, the overall price often differs significantly with different sequences. In addition to the sequence of the locations, also other considerations affect the price. E.g. instead of a multi-stop flight through all locations, one may book a return flight to one of them plus local flights from there to the others; sometimes special offers for two-way-tickets make this more attractive. A logic program can automatically generate flight plans according to the constraints and enquire their ticket prices by an external atom that internally uses an online flight booking service. An additional weak constraint can select the cheapest.

AngryHEX. The annual *AIBirds Competition*[5] is a competition for AI agents based on the popular *Angry Birds*[6] game, which is about using a slingshot to shoot birds of different types at pigs placed on a scene in order to destroy them. The pigs are usually protected by obstacles of different types. The game uses a realistic physics simulation, including gravity and statics. In the competition,

[5] https://aibirds.org
[6] https://www.angrybirds.com

agents are given the positions and dimensions of the objects in the scene and need to return the angle and velocity for shooting the next bird.

The *AngryHEX* agent [38] is implemented on top of HEX programs. The basic strategy is to maximize the estimated damage to obstacles and pigs for all possible targets. Plain ASP is ill-suited for this application as the computation involves physics simulation and floating point numbers. Therefore, a HEX program was used to realize the basic strategy including the optimal selection of the target, while low-level numeric computations have been outsourced. The agent participated in the competition since 2012 and ranked second in 2015.

HEX programs with nested program calls. Notably, DLVHEX can be used to 'call' HEX programs from other HEX programs, which we refer to as the *called program* and the *host program*, respectively. Specifically, one can process the collection of answer sets of a different program, and e.g. reason about it. To this end, dedicated external atoms for evaluating subprograms and inspecting their answer sets are available, cf. [25, 53].

When a subprogram call (corresponding to the evaluation of a special external atom) is encountered in the host program, the external atom internally creates another instance of DLVHEX to evaluate the subprogram. The result is then stored in an *answer cache* and gets a unique *handle* which can be later used to reference the result and access its components (e.g., predicate names, literals, arguments) via other external atoms. The subprogram can either be *directly embedded* in the host program, or *stored in a separate file*. In the latter case, code reuse is easy and libraries for solving re-occurring subproblems in ASP applications, e.g., graph problems or combinatorial optimization problems, can be built, where updates are automatically reflected in the call program.

To this end, we use external atoms $\&callhex_n$, $\&callhexfile_n$, $\&answersets$, $\&predicates$, and $\&arguments$, where

$$\&callhex_n[\mathtt{P}, p_1, \ldots, p_n](H) \quad \text{and} \quad \&callhexfile_n[\mathtt{FN}, p_1, \ldots, p_n](H)$$

$n \geq 0$, allow to execute a subprogram given by a string `P` or in a file `FN`, respectively; here n specifies the number of predicate names p_i, $1 \leq i \leq n$, used to define the input facts. When evaluating such an external atom on an interpretation I, the system adds all atoms $p_i(\vec{t})$ in I as facts to the specified program, creates another DLVHEX instance to evaluate it, and returns a symbolic handle H as result. A *handle* is a unique integer that represents a certain program answer cache entry. For convenience, we omit the subscript n in $\&callhex_n$ and $\&callhexfile_n$ as it is clear from the context.

Example 3. We use two predicates p_1 and p_2 to define the input to the subprogram `sub.hex` ($n = 2$), i.e., all atoms over these predicates are added to the subprogram prior to evaluation. The call derives a handle H as result.

$$p_1(x, y); \quad p_2(a); \quad p_2(b);$$
$$handle(H) \leftarrow \&callhexfile[\mathtt{sub.hex}, p_1, p_2](H)$$

In the implementation, handles are consecutive numbers starting at 0. The unique answer set of the program is $\{handle(0), p_1(x, y), p_2(a), p_2(b)\}$. □

Formally, given an interpretation I, $f_{\&callhexfile_n}(I, file, p_1, \ldots, p_n, h) = v$ with $v = 1$ if h is the handle to the result of the program in file *file* augmented with the facts over predicates $p_1, \ldots, p_n$ that are true in I, and $v = 0$ otherwise. The formal notion and use of $\&callhex_n$ to call embedded subprograms is analogous to $\&callhexfile_n$.

Example 4. Consider the following program:

$$h_1(H) \leftarrow \&callhexfile[\texttt{sub.hex}](H)$$
$$h_2(H) \leftarrow \&callhexfile[\texttt{sub.hex}](H)$$
$$h_3(H) \leftarrow \&callhex[\texttt{a};\ \texttt{b} \leftarrow \textbf{not}\ \texttt{c}](H)$$ □

The rules execute the program `sub.hex` and the embedded program $P_e = \{a;\ b \leftarrow \textbf{not}\ \ c\}$, with no facts being added. The single answer set is $\{h_1(0), h_2(0), h_3(1)\}$ or $\{h_1(1), h_2(1), h_3(0)\}$ depending on the order in which the subprograms are executed (which is irrelevant). Note that the program in `sub.hex` is called in two places but executed only once; P_e is (possibly) different from `sub.hex` and thus evaluated separately.

Now we want to determine how many answer sets a program has. For this purpose, we design an external atom $\&answersets[PH](AH)$ that associates subprograms with their answer set handles. Formally, for an interpretation I, we have $f_{\&answersets}(I, h_{Prog}, h_{AS}) = v$ with $v = 1$, if h_{AS} is a handle to an answer set of the program with program handle h_{Prog}, and $v = 0$ otherwise.

Example 5. The single rule

$$ash(PH, AH) \leftarrow \&callhex[\text{“}\texttt{a} \vee \texttt{b} \leftarrow\text{”}](PH), \&answersets[PH](AH)$$

calls the embedded subprogram $P_e = \{a \vee b \leftarrow\}$ and retrieves pairs (PH, PA) of handles to its answer sets; here $\&callhex[\text{”}\texttt{a} \vee \texttt{b} \leftarrow\text{”}](PH)$ returns a handle $PH = 0$ to the result of P_e, which is passed to the atom $\&answersets[PH](AH)$. The latter returns the handles 0 and 1, as P_e has two answer sets ($\{a\}$ and $\{b\}$). The overall program has thus the single answer set $\{ash(0, 0), ash(0, 1)\}$. As for each program the answer set handles start at 0, only a pair of a program and an answer set handle uniquely identifies an answer set. □

Using the external atoms from above, it is now easy e.g. to count the answer sets of a subprogram by determining the largest valid handle to an answer set. Similarly, external atoms *&predicates* and *&arguments* can be used to inspect answer sets; we refer to [25] for details.

5 HEX Language Extensions

We now turn to some extensions of HEX programs that are motivated by application needs.

HEX programs with function symbols. Uninterpreted function symbols, as for instance $do(a, s)$ to represent the follow up of a situation s after executing an action a, can be easily realized in HEX using external atoms; we thus can extend the language by such function symbols as syntactic sugar.

Formally, the set $\mathcal{X} \cup \mathcal{C}$ of terms is enriched, given a set $\mathcal{F}$ of function symbols, to the smallest superset $\mathcal{T}$ of $\mathcal{X} \cup \mathcal{C}$ such that for each $f \in \mathcal{F}$ of arity n, it holds that $\{f(t_1, \ldots, t_n) \mid t_1, \ldots, t_n \in \mathcal{T}\} \subseteq \mathcal{T}$; technically, it is possible to let $\mathcal{F} \subseteq \mathcal{C}$.

Using external atoms, it is possible to simulate composition and decomposition of function terms, as described in [9]. For every $k \geq 0$, two external predicates $\&comp_k$ and $\&decomp_k$ are defined that have $k+1$ (resp., 1) input arguments and 1 (resp., $k+1$) output arguments; the oracle functions are

$$f_{\&comp_k}(I, f, X_1, \ldots, X_k, T) = f_{\&decomp_k}(I, T, f, X_1, \ldots, X_k) = v,$$

with $v = 1$ if $T = f(X_1, \ldots, X_k)$ and $v = 0$ otherwise. Intuitively, $\&comp_n$ constructs a nested term from a function symbol and its term arguments (possibly nested themselves), and $\&decomp_n$ extracts the function symbol and the term arguments from a nested term.

Concrete occurrences of function terms in rules can now be eliminated by using auxiliary variables and adding appropriate $\&comp_n$ and $\&decomp_n$ atoms to the rule bodies. This will be clear from an example.

Example 6. Consider the following HEX program P with function symbols and its rewriting $T_f(P)$ to a plain HEX program:

$$\begin{array}{lll} P\colon & q(z);\ q(y) & \\ & p(f(f(X))) \leftarrow q(X) & \\ & r(X) \leftarrow p(X) & \\ & r(X) \leftarrow r(f(X)) & \end{array} \qquad \begin{array}{ll} T_f(P)\colon & q(z);\ q(y) \\ & p(V) \leftarrow q(X), \&comp_1[f, X](U), \\ & \qquad \&comp_1[f, U](V) \\ & r(X) \leftarrow p(X) \\ & r(X) \leftarrow r(V), \&decomp_1[V](f, X) \end{array}$$

Intuitively, $T_f(P)$ first builds $f(f(X))$ for all X on which q holds using two atoms over $\&comp_1$, and then extracts X from derived $r(f(X))$ facts using a $\&decomp_1$-atom. □

Realizing function symbols on top of external atoms allows for a better control of their processing. For example, the construction of new nested terms may be subject to additional conditions which are integrated into the semantics of the external predicates $\&comp_k$ and $\&decomp_k$. A concrete example is *data type checking*, i.e., checking whether the arguments of a function term are in a given domain. Another example is automatic computation of some argument from others; e.g., in building $roman(8, viii)$ from 8, the first argument is converted to Roman number representation.

HEX programs with action atoms. ACTHEX [4] is an extension of HEX programs which allows for the execution of declaratively scheduled actions. To this end, *action atoms* are introduced to rule heads, which operate on an *environment* and may modify it. The environment can be seen as an abstraction of realms outside the logic program. Thus, in contrast to ASP and HEX programs, which are stateless, ACTHEX allows for modifications of the external environment without wrapping the solver in a procedural language. We here review ACTHEX programs at a glance and refer to [4, 28] for details.

Intuitively, the evaluation of an ACTHEX program starts with evaluating it as an ordinary HEX program. Answer sets that optimize an associated objective

function are *best models*. The evaluation algorithm selects a single best model which determines a sequence of executable action atoms called the *execution schedule*. These actions are then executed and possibly modify the environment.

The ACTHEX language provides a set $\mathcal{A}$ of *action predicate names*, which start with #. An action atom is of the form $\#g[\vec{Y}]\{o,p\}[w:l]$, where $\#g \in \mathcal{A}$ is an action predicate name, $\vec{Y} = Y_1, \ldots, Y_n$ is the input list, $o \in \{b, c, c_p\}$ is the *action option* which declares actions as one of *brave*, *cautious* or *preferred cautious*, and the optional integer attributes p, w, and l are called *precedence*, *weight*, and *level*. Rules and programs are then defined as in ordinary HEX programs but may contain action atoms in rule heads.

The semantics of external atoms is generalized such that the environment may influence its truth value. To this end, a ground external atom $\&g[\vec{y}](\vec{x})$ with k-ary input and l-ary output has an associated a $2{+}k{+}l$-ary Boolean *oracle function* $f_{\&g}$; the atom $\&g[\vec{y}](\vec{x})$ is true wrt. assignment I and environment E, if $f_{\&g}(I, E, \vec{y}, \vec{x}) = 1$. *Best models* are defined based on level and weight of actions, and actions are *executable* wrt. a best model depending on their action option o. An *execution schedule* S_I for a best model I is a sequence of all actions executable in I that respects action precedence. The effect of executing a ground action $\#b[y_1, \ldots, y_n]\{o, r\}[w : i]$ on an environment E is modeled by a $(2{+}n)$-ary function $f_{\#b}$ that determines a follow-up environment $E' = f_{\#b}(I, E, \vec{y})$.

Example 7 (from [28]). The following ACTHEX-program controls a robot capable of executing a parameterized action *#robot*, where an external *&sensor* predicate enables to access sensor data.

$$\begin{aligned} \#robot[clean, kitchen]\{c, 2\}[1:1] &\leftarrow night \\ \#robot[clean, bedroom]\{c, 2\}[1:1] &\leftarrow day \\ \#robot[goto, charger]\{b, 1\}[1:1] &\leftarrow \&sensor[bat](low) \\ night \vee day &\leftarrow \end{aligned}$$

Informally, in the night the kitchen should be cleaned, and during daytime the bedroom; if the battery is low, the robot needs to go to the charger. The option b makes this action mandatory, while the other actions are by option c only taken if they occur in every answer set; by the disjunctive fact, this is not the case. Note that precedence 1 of $\#robot[goto, charger]\{b, 1\}$ makes the robot recharge its battery (if needed) before any cleaning. □

Use-cases of ACTHEX programs. ACTHEX has been used in several applications; for a more elaborative discussion, we refer to [4] and [28].

Action languages, such as the one by [34], are used to describe the relations between *actions* that modify the state of the world which is described by *fluents*, i.e., predicate that can change over time. Such languages can be captured by ACTHEX, exploiting the precedence attribute of action atoms to model time.

Related to this is *knowledge base update*, as adding and removing statements in knowledge bases maintenance can be modeled by action atoms. The ACTHEX-programmer can in this way reason over knowledge bases and modify

them declaratively depending on the current content. Since ACTHEX supports iterative solving, it can be exploited for various use cases such as belief revision, belief merging or implementing the observe-think-act cycle of agents [39].

Agents with *iterative strategies* do not compute solutions in a single shot using an appropriate encoding, but in multiple steps using intermediate solutions. This can be advantageous, in particular if the grounding of a monolithic problem encoding is very large, as holds e.g. for the logic puzzles Sudoku and Reversi. An ACTHEX Sudoku agent that iteratively adds numbers to a cell or excludes them from the set of possible values has the potential to solve larger instances than pure ASP can handle [28].

Constraint HEX programs. *Constraint Answer Set Programming (CASP)* (see e.g. [42,46]) combines ASP with constraint programming [1]. A well-known implementation is the clingcon system [50], which integrates GRINGO, CLASP and the constraint solver GECODE. Constraints can be encoded in plain ASP using builtin predicates, but this quickly produces groundings of unmanageable size; hence, a genuine support of constraints in ASP is reasonable, which can hide instances of constraint variables in the constraint solver.

Dedicated CASP solvers, however, do not allow to integrate background theories other than constraints. This motivated an integration of CASP with HEX programs to *constraint HEX programs*. Such programs are strictly more general than CASP programs, as besides constraints arbitrary background theories can be accessed via external atoms. Technically, the integration uses a translation of constraint HEX programs into native HEX programs, where constraints are handled using an *SMT-like* [3] approach (also used by clingcon).

Informally, a constraint HEX program may contain besides ordinary and external atoms also *constraint atoms*. The latter are comparisons of arithmetic expressions that consist of (constraint) variables and constants, such as $x + y < 10$. Here, x and y are constraint variables which range over a certain domain. Different from ASP variables, constraint variables are global, i.e., each occurrence in a program is bound to the same value; thus, the atoms $x < 10$ and $x > 20$ can never be jointly true, even if they occur in different rules. Notably, (upper-case) ASP variables can occur in constraint atoms (they are eliminated by the grounder); e.g., in $x + Y > 5$, the ASP variable Y is substituted by ground terms yielding ground constraint atoms.

We omit here a formal definition of constraint HEX programs, but illustrate them by an example.

Example 8. Suppose Alice's restaurant offers daily menus. The menus can be selected based on the price of food and drink, where drink should be cheaper than food and each menu should cost at most 20 Euros; menus of the limit cost

are called *exclusive*. This knowledge is encoded by the following program:

$$r_1\colon\ food(P) \leftarrow \&sql[\text{“Select price from Food”}](P)$$
$$r_2\colon\ drink(P) \leftarrow \&sql[\text{“Select price from Drink”}](P)$$
$$r_3\colon\ max\_price(20)$$
$$r_4\colon\ inMenu(F,D) \vee outMenu(F,D) \leftarrow drink(D), food(F)$$
$$r_5\colon\ \leftarrow D\ > F, inMenu(F,D)$$
$$r_6\colon\ F + D\ \leqslant P \leftarrow inMenu(F,D), max\_price(P)$$
$$r_7\colon\ exclusive\_menu \leftarrow inMenu(F,D), max\_price(P), F + D \equiv P$$

Here, food and drink prices are represented by atoms $food(\cdot)$ and $drink(\cdot)$, respectively; via the external atoms $\&sql[\cdot](\cdot)$, all prices from the database of the restaurant are loaded. Rule r_4 generates all price combinations of menus, while rule r_5 checks that food is more expensive than drink and rule r_6 that the maximum price is not exceeded. The rule r_7 checks for the existence of an exclusive menu. Note that the constraint atom in the head of rules r_6 is not *derived* to be true if the body is true, but it must *evaluate* to true in this case.

If the database contains prices 18 and 9 for food and 5 for drink, the single answer set contains $inMenu(9,5)$, and $outMenu(18,5)$, encoding a single menu of 9 Euros for food and 5 Euros for drink; there is no exclusive menu. □

Constraint HEX programs can be translated into plain HEX programs using a dedicated external atom for constraint checking. The idea is to guess the truth values of all constraint atoms, which are represented using a special predicate $con(\cdot)$, in the program and pass the guess to external constraint checking; the answer set candidate is eliminated if the guess is not compatible, i.e., the corresponding constraints are not satisfiable. This is best illustrated on the previous example.

Example 9 (cont'd). The constraint atoms $D > F$, $F + D \leqslant P$, and $F + D \equiv P$ are represented by $con(D, >, F)$, $con(F, +, D, \leqslant, P)$, and $con(F, +, D, \equiv, P)$, and their negations by $con(D, \leqslant, F)$, $con(F, +, D, >, P)$, and $con(F, +, D, \not\equiv, P)$. The rules r_5–r_7 are now replaced by the following rules:

$$r_5'\colon\ \leftarrow con(D, >, F), inMenu(F,D)$$
$$r_6'\colon\ con(F, +, D, \leqslant, P) \leftarrow inMenu(F,D), max\_price(P)$$
$$r_7'\colon\ exclusive\_menu \leftarrow inMenu(F,D), max\_price(P), con(F, +, D, \equiv, P)$$
$$g_1\colon\ con(D, >, F)\ \vee\ con(D, \leqslant, F) \leftarrow inMenu(F,D)$$
$$g_2\colon\ con(F, +, D, \leqslant, P) \vee con(F, +, D, >, P) \leftarrow inMenu(F,D), max\_price(P)$$
$$g_3\colon\ con(F, +, D, \equiv, P) \vee con(F, +, D, \not\equiv, P) \leftarrow inMenu(F,D), max\_price(P)$$
$$c\colon\ \leftarrow \mathbf{not}\ \&check[con, sum]()$$

The rule r_i' results from r_i by replacing the constraint atom with its guess atom. The rules g_1, g_2 and g_3 guess the truth values of all (ground) constraint atoms, and c checks compatibility of the guess via the constraint solver. Here *sum* is like *con* a special predicate for sums in constraint expressions that is void. □

For more details and discussion, we refer to [54].

HEX$^\exists$ programs. An important feature of HEX programs is that they are capable of value invention, i.e., that new constants are introduced into a program. This relates to existential quantification, as, given an external atom $\&p[\vec{y}](\vec{x})$ that evaluates to true, the output values $\vec{x}$ witness that the formula $\exists \vec{X}\, \&p(\vec{y}, \vec{X})$ is true. If we are just interested in some (arbitrary) such witness $\vec{x}$, we might write rules that choose one of them; alternatively, one may delegate this choice to the external source, i.e., use a variant $\&p'$ of $\&p$ such that $\&p'[\vec{y}](\vec{x})$ holds for a unique $\vec{x}$. As the choice of $\vec{x}$ depends on the external source, we obtain in this way *domain-specific existential quantification*. If, as in pure logic, we want to leave concrete witnesses open, we can use a tuple $\vec{x}' = x_1 \ldots, x_m$ of fresh constants x_i (or *null values*) as generic witness; this amounts to Skolemization for the elimination of function symbols.

Such logical existential quantification is supported in the language of *HEX$^\exists$ programs*, which are finite sets of rules of the form

$$\exists \vec{X} : p(\vec{Y'}, \vec{X}) \leftarrow \mathbf{conj}[\vec{Y}], \tag{5}$$

where $\vec{X}$ and $\vec{Y}$ are disjoint sets of variables, $\vec{Y'} \subseteq \vec{Y}$, $p(\vec{Y'}, \vec{X})$ is an ordinary atom, and $\mathbf{conj}[\vec{Y}]$ is a conjunction of (possibly default-negated) atoms and external atoms containing all and only the variables $\vec{Y}$. Semantically, this rule assigns for each ground instance $\mathbf{conj}[\vec{y}]$ that evaluates to true a tuple $\vec{x} = x_1, \ldots, x_m$ of *new null values* as above.[7]

A HEX$^\exists$ program Π can be transformed to an equivalent HEX program $T(\Pi)$ by rewriting each rule r of form (5) to the rule

$$p(\vec{Y'}, \vec{X}) \leftarrow \mathbf{conj}[\vec{Y}], \&exists^{|\vec{Y'}|,|\vec{X}|}[r, \vec{Y'}](\vec{X}),$$

where the existential quantifier is replaced by a new external atom $\&exists$ of appropriate input and output arity which uses value invention.

Example 10. Consider the following HEX$^\exists$ program Π, which expresses that each employee X has some office Y:

$$\begin{array}{rl} & employee(john). \quad employee(joe). \\ r_1 : \exists X : & office(Y, X) \leftarrow employee(Y). \\ r_2 : & room(X) \leftarrow office(Y, X) \end{array}$$

In the translated program $T_\exists(\Pi)$, r_1 is replaced by

$$r_1' : office(Y, X) \leftarrow employee(Y), \&exists^{1,1}[r_1, Y](X).$$

One can use HEX$^\exists$ programs to model query answering from existential rules (i.e., $\mathbf{conj}[\vec{Y}]$ in (5) consists of ordinary atoms). Even if the answer set may be infinite, a finite fragment may suffice for this purpose. For more details see [17].

[7] By the underlying unique name assumption, these values do not match with any other values; to model this, equality reasoning would need to be imposed on top.

6 HEX as Backend Formalism

For the third use scenario, we consider some data and knowledge-based formalisms that use (extended) HEX programs as backend.

Multi-context systems. Multi-context systems (MCSs) [7] are a formalism for interlinking multiple knowledge based systems called *contexts*. The formalism abstracts from the knowledge representation language and models context semantics in terms of accepted *belief sets*. The latter are abstractly modeled as naked sets whose elements (i.e., the beliefs) need not bear logical structure. The contexts are interlinked by so called *bridge rules* which add formulas to the knowledge base of a context depending on the presence and/or absence of beliefs from the belief sets of other contexts. The semantics of an MCS is given in terms of *equilibria*, which are global states that consist of acceptable belief sets for each context, such that all bridge rules are satisfied.

Besides computing equilibria, an important reasoning task for MCSs is *inconsistency analysis*; e.g., given a MCS M that lacks equilibria, compute a reason for this inconsistency [19]. Inconsistency explanations can be computed using a HEX program encoding [6] in which external atoms *outsource contextual reasoning* and check whether a context accepts a certain belief set. The use of external atoms in the program is highly cyclic as the saturation technique is employed; the latter is required as the problem is beyond NP and co-NP.

Description Logics plus rules. *Description logics (DLs)* provide a logical formalism for ontologies that are well-suited for the Semantic Web [36] or in medical applications [37]. Ontologies represent classes of objects, referred to as *concepts*, and the relations between objects, called *roles*. Concepts and roles correspond to unary and binary predicates in first-order logic, respectively. A *description logic knowledge base* consists of a *Tbox* (*the terminology*) that defines concepts and roles and represents relations between them, and an *Abox* (*assertions*), that contains specific information on membership of individuals in concepts resp. of pairs of individuals in roles.

Example 11. Suppose *PhDStudent*, *Student* and *Professor* are concepts and *isAssistantOf* is a role. The Tbox may contain the *concept inclusion axiom* $PhDStudent \sqsubseteq Student$, which states that the class of PhD students is a subclass of all students. The Abox contains concept membership assertions like $Professor(smith)$ and $PhDStudent(johnson)$, representing that *smith* is a professor and *johnson* a PhD student. An assertion $isAssistantOf(johnson, smith)$ states that *johnson* is an assistant of professor *smith*. □

Typical reasoning tasks over description logic knowledge bases include concept and role retrieval, i.e., listing all individuals or pairs of individuals which are members of a given concept or role, respectively. In the example above one may ask for all members of *Student* and expects as answer *johnson* as he is a *PhDStudent* and thus, by the terminological knowledge, also a *Student*.

Combining ontologies and answer set programming is especially valuable as existing domain knowledge can be accessed from logic programs. To this end, *DL-programs* have been developed by [21] which have been implemented on top of HEX programs with dedicated external atoms; where the external source

features external atoms for concept and role queries. Prior to query evaluation, concepts and/or roles are enriched by individuals from the ASP program. This allows for advanced reasoning tasks such as terminological default reasoning or closed world reasoning on description logic knowledge bases [11].

As description logics are monotonic, default reasoning can only be realized by the (cyclic) interaction of rules and the DL knowledge base. To this end, appropriate encodings and an implementation were developed [11]. DL-programs have, e.g., been applied in complaint management for e-government [60].

The MELD belief merging system deals with merging *collections of belief sets* [52, 53], which are roughly sets of classical ground literals. A merging strategy is defined by tree-shaped *merging plans*, whose leaves are the collections of belief sets to be merged, and whose inner nodes are *merging operators* (provided by the user). The structure is akin to syntax trees of terms. The automatic evaluation of tree-shaped merging plans is based on nested HEX programs; it proceeds bottom-up, where every step requires inspection of the subresults, i.e., accessing the answer sets of subprograms. In fact, the need for such processing has led to develop nested HEX program.

Interactive ASP. The Answer Set Application Programming (ASAP) framework [57] allows for creating interactive applications based on ASP. In this framework, incoming events (e.g., keyboard) are processed by ASP and the application state is managed using fluents (as in planning). An ASAP program is rewritten to a HEX program where each evaluation obtains fluent values and event information via HEX external atoms. Answer sets determine future fluent values by atoms $fl{=}val@t$, which intuitively means that fluent fl has value val at time t. Programs can contain actions (atoms starting with '@') as in ACTHEX, for example to display the user interface or to quit the program.

Example 12. The following ASAP-program displays a help text and the state (on or off) of a switch. The user can change the state using cursor keys and quit with the Q key.

$$
\begin{aligned}
\#\mathit{initial}\ \mathit{switch}{=}\mathit{off} &\leftarrow .\\
\mathit{switch}{=}\mathit{off}@\mathit{next} &\leftarrow \&\mathit{event}[\texttt{"key.special"}](\texttt{"Down"}).\\
\mathit{switch}{=}\mathit{on}@\mathit{next} &\leftarrow \&\mathit{event}[\texttt{"key.special"}](\texttt{"Up"}).\\
@\mathit{exit}(0) &\leftarrow \&\mathit{event}[\texttt{"key.normal"}](\texttt{"q"}).\\
@\mathit{drawText}(2,2,\texttt{"Up/Down: switch, Q: quit"}) &\leftarrow .\\
@\mathit{drawText}(5,4,\mathit{State}) &\leftarrow \mathit{switch}{=}\mathit{State}@\mathit{next}.
\end{aligned}
$$

Here the time *next* refers to the state after the currently processed event. □

ASAP is a *hybrid* HEX use scenario: an ASAP-program is rewritten into a HEX program, transforming fluent atoms into regular atoms and adding rules containing external atoms. At the same time an ASAP program can use arbitrary external atoms, e.g., for string processing. ASAP combines computation outsourcing (string processing) with information outsourcing (events and fluents) and uses HEX as a clean interface to the real world.

7 Discussion

In the previous sections, we have considered different types of use case scenarios for formalisms from the HEX family. We now briefly discuss some advantages and drawbacks of these types. However, we remain at the surface and the considerations only serve as a general guideline to select the way for realizing a concrete application using HEX. After that, we consider related work.

7.1 Comparison of the use case types

Unsurprisingly, using the HEX formalism directly provides maximal flexibility to the user, as this allows one to formulate arbitrary rules and to access arbitrary external sources from the rules, provided that the external predicates are defined and implementations are provided as plugins. Depending on the conceptual complexity of the application, this might come at the price of a low user convenience due to the need for heavy-weight and repetitive syntax in the problem encodings. Furthermore, the manual implementation of external source access through a plugin requires some effort (as in general, if one aims at coupling systems via interfaces), and depending on experience and technical skills may be time-consuming; besides development also proper testing and validation of the plugin have to be considered.

In contrast to this, HEX program extensions as well as automated translations of frontends in other KR formalisms into HEX programs provide one with the possibility to use specific language features for expressing particular aspects of a problem. This comes with the advantage of eliminating some repetitive work, where the same rule patterns need not be written over and over again, and in this way also helps to reduce errors that are made by spoiled copy and paste. Furthermore, using designated language constructs gives the solver as in ordinary ASP more insight into the user's intention in writing a certain part of the program, which may be exploited for performance enhancements; guessing such intention respectively discovering program parts or rule patterns that may amount to an intention is expensive in general. For a simple example, the constraint c' in (4) is equivalent to the rule

$$a \leftarrow \mathbf{not}\ \&check[color, edge](), \mathbf{not}\ a; \tag{6}$$

where a is a fresh atom. The explicit form of c' allows us to immediately conclude that $\&check[color, edge]()$ must be true in every answer set; given (6), further analysis of the program is needed.

On the other hand, the use of front end translations provides limited flexibility and one is in general committed to a specific encoding. This can be disadvantageous, if for some targeted application that encoding is not working well (e.g., for performance reasons), or some additional features or aspects should be respected (e.g., some simple preference of alternatives). As a compromise, one can then resort to encodings that are automatically generated, but then manually customized by the user; this has been advocated and used e.g. in [21]. However, this approach requires a proper grasp and understanding of the encoding produced by the translation, which may require considerable

effort, the more if—as happens often in practice—optimizations are applied to the code (sometimes without proper documentation, and based on implicit assumptions).

7.2 Related Work

Despite the terminological similarity, customizable functions as supported by GRINGO and so called *frozen atoms* supported by CLASP (sometimes called external atoms) are different from external atoms as supported by HEX. GRINGO supports custom functions (implemented in the scripting languages Lua or Python) which are evaluated during the program grounding and thus compiled away prior to the solving step. They are intended to be used as customizable built-in atoms, but no cyclic dependencies are possible. The frozen atoms of CLASP are sometimes also called *external atoms*; they are protected from optimization, and their truth values can be determined from code (e.g., in Python) that controls the grounding and solving process, for example in advanced techniques such as incremental solving. Using frozen atoms requires that the solver and the truth values of atoms are controlled "from outside" using imperative code. HEX inverts the roles of ASP and imperative code: the DLVHEX solver engine controls the ASP evaluation and evaluates external atom semantics "inside" the ASP evaluation on demand and whenever needed. That is, with HEX external semantics is evaluated within evaluation of ASP semantics, while with CLASP, ASP semantics is evaluated within imperative code that configures the truth values of frozen atoms.

Besides GRINGO and CLASP, there are extensions of ASP towards the integration of specific external sources. Examples are constraint ASP as an integration of ASP with constraint programming as realized e.g. in clingcon [50] and EZCSP [2], or DL-programs as a native combination of ASP with ontologies [22]. In contrast, HEX allows for the integration of arbitrary external sources through a general interface and their flexible combination; the other use cases correspond to special cases thereof.

8 Conclusion

Arriving at the end, we give a brief summary and an outlook on future work.

8.1 Summary

HEX programs extend answer set programs with access to external sources through an API-style interface, which has been fruitfully deployed to various applications. In this paper, we briefly discussed how the formalism can be used for problem solving in KR.

To this end, we first presented the general methodology as a strict generalization of ASP. In particular, the prominent guess and check paradigm can be seamlessly combined with external sources, both in the guessing and in the checking part. Also other ASP techniques, such as saturation, can be used with external sources. We then presented two typical types of external sources for

computation outsourcing and for information outsourcing, respectively, and for combinations thereof. We further distinguished three typical use case scenarios of HEX programs, namely for encoding end user applications, as a basis for language extensions, and as a backend for other KR formalisms. For each of the scenarios, we have briefly presented existing applications.

8.2 Open Issues and Future Work

Ongoing work includes the extension of the interface for external sources. While the current interface is convenient, it turns out that the integration of external sources as black boxes inhibits efficient evaluation in many cases. Low-level interfaces might be less convenient, but give the reasoner more insights into the semantics and properties of external atoms, which might be exploited to increase efficiency. This includes the possibility for evaluating external sources under partial interpretations and retrieving partial answers. Such interfaces are currently under development and will be an alternative to the existing ones without replacing them.

Another issue concerns robustness of performance. Currently, in some cases small changes in the encoding and in the run options of the solver influence the efficiency considerably. While such effects are understandable to users who know the algorithms underlying the systems well, inexperienced users may face difficulties in crafting efficient encodings for their applications. The same issue applies to ordinary ASP as well, and thus is not a genuine issue for HEX programs; indeed, [29] states that "crafting an [plain] ASP encoding that also leads to the best possible system performance is yet not as obvious as it might seem." In general, the highly declarative nature of ASP and its intrinsic intractability comes at a computational price, and handling different inputs smoothly requires complex and sophisticated algorithms, where also the use of heuristics is indispensable; the presence of external atoms adds to this difficulty.

This naturally leads to two directions for potential future work. On the solver side, the problem might be mitigated, but may not be expected to be eliminated, by more advanced optimization methods and heuristics, which automatically optimize the input program and determine optimal solver options to process the current input program. On the modeling side, an interesting issue are methodologies for deciding when it makes sense to outsource computation for efficiency reasons, and methodologies for encoding problems with external atoms to ensure the best possible efficiency.

References

1. Krzysztof Apt. *Principles of Constraint Programming.* Cambridge University Press, New York, NY, USA, 2003.
2. Marcello Balduccini. Representing constraint satisfaction problems in answer set programming. In *Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP) at ICLP*, 2009.
3. Clark Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 185, chapter 26, pages 825–885. IOS Press, 2009.

4. Selen Basol, O. Erdem, M. Fink, and G. Ianni. HEX programs with action atoms. In *Technical Communications of the International Conference on Logic Programming (ICLP)*, pages 24–33, 2010.
5. Christoph Beierle and G. Kern-Isberner. *Methoden wissensbasierter Systeme - Grundlagen, Algorithmen, Anwendungen (5. Aufl.)*. Computational Intelligence. Springer/Vieweg, 2014.
6. Markus Bögl, T. Eiter, M. Fink, and P. Schüller. The MCS-IE system for explaining inconsistency in multi-context systems. In *European Conference on Logics in Artificial Intelligence (JELIA)*, pages 356–359, 2010.
7. Gerd Brewka and T. Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *AAAI Conference on Artificial Intelligence*, pages 385–390. AAAI Press, 2007.
8. Gerhard Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.
9. Francesco Calimeri, S. Cozza, and G. Ianni. External sources of knowledge and value invention in logic programming. *Annals Math. Artif. Intell.*, 50(3–4):333–361, 2007.
10. Francesco Calimeri, M. Fink, S. Germano, G. Ianni, C. Redl, and A. Wimmer. AngryHEX: an artificial player for angry birds based on declarative knowledge bases. In *National Workshop and Prize on Popularize Artificial Intelligence*, pages 29–35, 2013.
11. Minh Dao-Tran, T. Eiter, and T. Krennwallner. Realizing default logic over description logic knowledge bases. In *European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pages 602–613, 2009.
12. Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321–357, 1995.
13. Thomas Eiter, M. Fink, G. Ianni, T. Krennwallner, C. Redl, and P. Schüller. A model building framework for answer set programming with external computations. *Theory and Practice of Logic Programming*, 2015. doi:10.1017/S1471068415000113, http://arxiv.org/abs/1507.01451.
14. Thomas Eiter, M. Fink, T. Krennwallner, and C. Redl. Conflict-driven ASP solving with external sources. *Theory and Practice of Logic Programming*, 12(4-5):659–679, 2012.
15. Thomas Eiter, M. Fink, T. Krennwallner, and C. Redl. Liberal Safety Criteria for HEX-Programs. In Marie des Jardins and Michael Littman, editors, *AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2013.
16. Thomas Eiter, M. Fink, T. Krennwallner, and C. Redl. Domain expansion for ASP-programs with external sources. Tech. Rep. INFSYS RR-1843-14-02, Inst. f. Informationssysteme, TU Wien, Austria, Sept. 2014.
17. Thomas Eiter, M. Fink, T. Krennwallner, and C. Redl. HEX-programs with existential quantification. In *International Conference on Applications of Declarative Programming and Knowledge Management (INAP)*, 2014.
18. Thomas Eiter, M. Fink, T. Krennwallner, C. Redl, and P. Schüller. Efficient HEX-program evaluation based on unfounded sets. *Journal of Artificial Intelligence Research*, 49:269–321, 2014.
19. Thomas Eiter, M. Fink, P. Schüller, and A. Weinzierl. Finding Explanations of Inconsistency in Multi-Context Systems. *Artificial Intelligence*, 216:233–274, November 2014.
20. Thomas Eiter, G. Ianni, and T. Krennwallner. Answer set programming: A primer. In *Reasoning Web Summer School*, pages 40–110, 2009.

21. Thomas Eiter, G. Ianni, T. Krennwallner, and R. Schindlauer. Exploiting conjunctive queries in description logic programs. *Annals Math. Artif. Intell.*, 53(1–4):115–152, 2008.
22. Thomas Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.
23. Thomas Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In *Proc. IJCAI 2005*, pages 90–96. Professional Book Center, 2005.
24. Thomas Eiter, G. Ianni, R. Schindlauer, and H. Tompits. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In *European Semantic Web Conference*, pages 273–287, 2006.
25. Thomas Eiter, T. Krennwallner, and C. Redl. HEX-Programs with Nested Program Calls. In *Applications of Declarative Programming and Knowledge Management (INAP 2011)*, pages 1–10. Springer, 2013.
26. Thomas Eiter, M. Mehuljic, C. Redl, and P. Schüller. User guide: dlvhex 2.x. Tech. Rep. INFSYS RR-1843-15-05, Inst. f. Informationssysteme, TU Wien, Austria, Sept. 2015.
27. Wolfgang Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *European Conference on Logics in Artificial Intelligence (JELIA)*, pages 200–212. Springer, 2004.
28. Michael Fink, S. Germano, G. Ianni, C. Redl, and P. Schüller. ActHEX: implementing HEX programs with action atoms. In Pedro Cabalar and TranCao Son, editors, *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 317–322. Springer, 2013.
29. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.
30. Martin Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M.T. Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.*, 24(2):107–124, 2011.
31. Martin Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187–188:52–89, 2012.
32. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proceedings of the 5th International Conference and Symposium*, pages 1070–1080. MIT Press, 1988.
33. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *Next Generation Computing*, 9(3–4):365–386, 1991.
34. Enrico Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153:2004, 2004.
35. Giray Havur, G. Ozbilgin, E. Erdem, and V. Patoglu. Geometric Rearrangement of Multiple Movable Objects on Cluttered Surfaces: A Hybrid Reasoning Approach. In *International Conference on Robotics and Automation (ICRA)*, pages 445–452, 2014.
36. J. Heflin and H. Munoz-Avila. Lcw-based agent planning for the semantic web. In A. Pease, editor, *Ontologies and the Semantic Web*, number WS-02-11 in AAAI Technical Report, pages 63–70, Menlo Park, CA, 2002. AAAI Press.
37. Robert Hoehndorf, F. Loebe, J. Kelso, and H. Herre. Representing default knowledge in biomedical ontologies: Application to the integration of anatomy and phenotype ontologies. *BMC Bioinformatics*, 8(1):377, 2007.
38. Giovambattista Ianni, F. Calimeri, S. Germano, A. Humenberger, C. Redl, D. Stepanova, A. Tucci, and A. Wimmer. Angry-HEX: an artificial player for

angry birds based on declarative knowledge bases. *IEEE Trans. Computational Intelligence and AI in Games*, 2015. Accepted for publication.
39. R. Kowalski and F. Sadri. From logic programming towards multi-agent systems. *Annals Math. Artif. Intell.*, 25(3-4):391–419, 1999.
40. Ora Lassila and R.R. Swick. Resource description framework (RDF) model and syntax specification, 1999. www.w3.org/TR/1999/REC-rdf-syntax-19990222.
41. Nicola Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562, July 2006.
42. Yuliya Lierler. Relating constraint answer set programming languages and algorithms. *Artificial Intelligence*, 207:1–22, February 2014.
43. Vladimir Lifschitz. Answer Set Programming and Plan Generation. *Artificial Intelligence*, 138:39–54, 2002.
44. Fangzhen Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1–2):115–137, 2004.
45. Victor W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer, 1999.
46. Veena S Mellarkod, M. Gelfond, and Y. Zhang. Integrating Answer Set Programming and Constraint Logic Programming. *Annals Math. Artif. Intell.*, 53(1-4):251–287, 2008.
47. Alessandro Mosca and D. Bernini. Ontology-driven geographic information system and dlvhex reasoning for material culture analysis. In *Italian Workshop RiCeRcA at ICLP*, 2008.
48. Ilkka Niemelä. Logic programming with stable model semantics as constraint programming paradigm. *Annals Math. Artif. Intell.*, 25(3–4):241–273, 1999.
49. Davy Van Nieuwenborgh, T. Eiter, and D. Vermeir. Conditional planning with external functions. In *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 214–227. Springer, 2007.
50. Max Ostrowski and T. Schaub. ASP modulo CSP: the clingcon system. *Theory and Practice of Logic Programming (TPLP)*, 12(4-5):485–503, 2012.
51. Axel Polleres. From SPARQL to rules (and back). In *International Conference on World Wide Web (WWW)*, pages 787–796. ACM, 2007.
52. Christoph Redl. Development of a belief merging framewerk for dlvhex. Master's thesis, Vienna University of Technology, A-1040 Vienna, Karlsplatz 13, 2010.
53. Christoph Redl, T. Eiter, and T. Krennwallner. Declarative belief set merging using merging plans. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 99–114. Springer, 2011.
54. Alessandro De Rosis, T. Eiter, C. Redl, and F. Ricca. Constraint answer set programming based on HEX-programs. In *Eighth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2015)*. https://sites.google.com/site/aspocp15/accepted.
55. Roman Schindlauer. *Answer Set Programming for the Semantic Web*. PhD thesis, Vienna University of Technology, Vienna, Austria, 2006.
56. Peter Schüller, V. Patoglu, and E. Erdem. A Systematic Analysis of Levels of Integration between Low-Level Reasoning and Task Planning. In *Workshop on Combining Task and Motion Planning at ICRA*, 2013.
57. Peter Schüller and A. Weinzierl. Answer Set Application Programming: a Case Study on Tetris. In Marina De Vos, T. Eiter, Yuliya Lierler, and Francesca Toni, editors, *International Conference on Logic Programming (ICLP), Technical Communications*, volume 1433. CEUR-WS.org, 2015.

58. Patrik Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.
59. Jesia Zakraoui and W.L. Zagler. A method for generating CSS to improve web accessibility for old users. In *International Conference on Computers Helping People with Special Needs (ICCHP)*, pages 329–336, 2012.
60. Hande Zirtiloğlu and P. Yolum. Ranking semantic information for e-government: complaints management. In *International Workshop on Ontology-supported business intelligence (OBI)*. ACM, 2008.