# Declarative Belief Set Merging using Merging Plans*

Christoph Redl, Thomas Eiter, and Thomas Krennwallner

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{redl,eiter,tkren}@kr.tuwien.ac.at

**Abstract.** We present a declarative framework for belief set merging tasks over (possibly heterogeneous) knowledge bases, where belief sets are sets of literals. The framework is designed generically for flexible deployment to a range of applications, and allows to specify complex merging tasks in tree-structured *merging plans*, whose leaves are the possible belief sets of the knowledge bases that are processed using *merging operators*. A prototype is implemented in MELD (MErging Library for Dlvhex) on top of the dlvhex system for HEX-programs, which are nonmonotonic logic programs with access to external sources. Plans in the task description language allow to formulate different conflict resolution strategies, and by shared object libraries, the user may also develop and integrate her own merging operators. MELD supports rapid prototyping of merging tasks, providing a computational backbone such that users can focus on *operator optimization and evaluation*, and on *experimenting* with merging strategies; this is particularly useful if a best merging operator or strategy is not known. Example applications are combining multiple decision diagrams (e.g., in biomedicine), judgment aggregation in social choice theory, and ontology merging.

## 1 Introduction

Merging knowledge from multiple knowledge bases has gained increasing attention over the years, given that more and more knowledge from (possibly heterogeneous) different sources must be combined into a coherent view. As knowledge bases are associated with sets of beliefs, i.e., statements an agent believes to be true (which need not to be the case), in particular merging the belief sets of knowledge bases into a single belief set is an issue. This problem has been widely studied, and there are many different approaches, e.g., [10]; for an introduction and a distinction from belief revision, see [11].

Roughly, the merging approaches fall into two classes. The one class adheres to base-oriented, syntactic strategies where the result of merging is a knowledge base, such that its belief sets are the merged belief sets (e.g., [8]). The other class performs merging at the semantic level, i.e., at the level of *models* of the knowledge bases, and aims to construct a merged set of models with associated syntactic belief sets (e.g., [13, 16]). Several approaches are based on measuring distances between models resp. formulas [9]; however, appropriate distance functions are usually application dependent.

Apparently there is no single approach which is superior to all others in arbitrary scenarios and applications. Lack of domain knowledge may make it very hard to predict

* This research has been supported by the Austrian Science Fund (FWF) project P20841.

which choice will work out best. It is then reasonable, or also necessary, to experiment with various choices and to evaluate the results empirically. Furthermore, it may be necessary to combine different merging operators, taking the specific needs and criteria of some of the knowledge bases into account. However, despite many theoretical frameworks for belief merging, support for merging in practice is scarce, and the user has the burden to develop merging procedures and implement a workflow (e.g. perform syntactic alignment of the knowledge bases, apply a binary merging operator repeatedly, etc), as well as to cope with issues of heterogeneity. Changes for experimenting with different operators and workflows are cumbersome and require major efforts.

To alleviate this problem, we have developed a practical framework for belief set merging. It allows the declarative specification of a merging task in a formal and machine-readable way, using *merging plans* in a dedicated language. Application-dependent parts of the specification are defined by the user, i.e., the application developer, while routine tasks are managed by our framework. To encompass wide applicability, the framework is generically based on beliefs that are literals, i.e., possibly negated atomic formulas, following the semantic direction; via suitable encodings and operators, also sources with non-logical content may be handled (e.g., decision diagrams as we show).

Our main contributions are briefly summarized as follows.

- We define a simple, generic framework for belief set merging tasks where belief sets are sets of ground literals in predicate logic; they may also be viewed as models of the knowledge bases, which are sets of formulas (we will use the term *belief bases* synonymously) (Section 2). We provide the formal syntax and semantics of merging plans in a dedicated *merging task language* (Section 3). A merging plan is, like an arithmetic expression, a hierarchical arrangement of *merging operators* of arity $n \geq 1$ which describe how to merge $n$ sets of belief sets into a single one; allowing $n = 1$ is convenient to accommodate also transformations (conversion, data cleaning, etc.) on sets of belief sets. An operator is either applied on merging sub-plans, i.e., the result of previous operator applications, or on the input knowledge bases.
- We have implemented the formal framework in the MELD system (*<u>ME</u>rging <u>L</u>ibrary for <u>D</u>lvhex*) [14] (Section 4), which has been developed as plugin for the dlvhex reasoner.[1] The systems allows the automatic evaluation of merging plans written in our merging language, i.e., the computation of the merged belief sets according to the merging plans. MELD is based on HEX-programs [5], which are non-monotonic logic programs that allow to access external sources (for our concerns, knowledge bases at an extensional level). In fact, we extended HEX-programs to nested HEX-programs that allow to evaluate HEX-programs and access the resulting models as first class citizens; such an extension is novel and of independent interest for non-monotonic logic programs in general. Via abstract interfacing, also merging of heterogeneous knowledge bases can be handled in a flexible way.
- To explore the usefulness of the approach, we have considered various applications, which currently include decision diagram merging in life sciences (e.g., for DNA classification or screening tests), judgment aggregation, and merging of knowledge bases in the Semantic Web (Section 6). We focus here on decision diagrams, which are encoded to belief sets via a natural encoding into a factual representation. The

[1] www.kr.tuwien.ac.at/research/systems/dlvhex/mergingplugin.html

support for rapid prototyping and experimenting with merging scenarios could be fruitfully exploited to arrive for real-world data at a merging result that outperforms other results, and could have hardly been obtained without automated support.

To our knowledge, no comparable framework for belief merging in practice exists. MELD aims at providing a user-friendly interface for rapid prototyping of belief set merging tasks with large flexibility, such that the application developer can focus on the selection, optimization, and workflow of the merging strategy. The merging operators can be selected from a predefined library or defined by the user, using a simple plugin interface. We believe implementations of our framework like MELD will greatly alleviate to determine the right merging strategy in prototyping for a range of applications.

## 2 Preliminaries

We consider merging of belief sets that are close to model-based semantics of classical logic, in a finite setting. In our view, we abstract from a concrete language for knowledge bases and identify the latter with associated sets of belief sets. In this context, the term *belief bases* is used as a synonym for knowledge bases. To formulate beliefs, we assume a signature $\Sigma = (\Sigma_c, \Sigma_p)$ of a set $\Sigma_c$ of constant symbols and a set $\Sigma_p$ of predicate symbols of arity $\geq 0$. For practical concerns, $\Sigma$ is finite.

**Definition 1.** *A* belief *is an atomic formula $p(c_1, \ldots, c_n)$ or negated atomic formula $\neg p(c_1, \ldots, c_n)$ (i.e., a literal) over $\Sigma$. The set of all beliefs over $\Sigma$ is denoted by $Lit_\Sigma$ (i.e., the set of all literals over $\Sigma$). A* belief set *is a set $B \subseteq Lit_\Sigma$ of literals. The set of all belief sets is denoted by $\mathcal{A}(\Sigma) = 2^{Lit_\Sigma}$.*

The semantic abstraction of knowledge bases is then as follows.

**Definition 2.** *Given a knowledge base $KB$ (in some language), it has associated belief sets $BS(KB) \subseteq \mathcal{A}(\Sigma)$.*

Intuitively, each belief set $B \in BS(KB)$ coherently collects conclusions from the knowledge base. There might be different possibilities, e.g., in a model-based view, or as common in non-monotonic logics. The following examples illustrate this.

*Example 1.* Consider the knowledge base $KB = \{dog(sue) \vee cat(sue), male(sue)\}$ in classical logic. Adopting as belief sets the maximal sets of literals consistent with $KB$ (i.e., the Herbrand models of $KB$), we have $BS(KB) = \{\, \{dog(sue), \neg cat(sue), male(sue)\}, \{\neg dog(sue), cat(sue), male(sue)\}\,\}$. Alternatively, if a belief set consists of all classically entailed literals, we obtain $BS(KB) = \{\, \{male(sue)\}\,\}$.

*Example 2.* Consider the logic program $P = \{dog(sue) \vee cat(sue).,\ eat_fish(X) \leftarrow cat(X), \text{not}\ abnormal(X).\}$. Adopting as belief sets the answer sets $AS(P)$ of this program [7], we obtain $BS(P) = AS(P) = \{\{dog(sue)\}, \{cat(sue), eat_fish(sue)\}\}$.

While we abstract from concrete languages, it will be convenient to refer with $KB^\Sigma$ to the implicitly defined signature of $BS(KB)$.

**HEX-programs.** Our implementation employs HEX-programs [5], which consist of rules

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

where each $a_i$ is a classical literal and each $b_j$ is either a classical literal or an *external* literal of the form $\&p[q_1, \ldots, q_k](t_1, \ldots, t_l)$, where $p$ is the name of an external predicate, the $q_i$ are predicate names, and the $t_j$ are terms;[2] intuitively, $p$ is evaluated externally, where the value of $q_1, \ldots, q_k$ is passed as input. The atom succeeds for variable binding if the external evaluation succeeds. Via such atoms, in particular abstract belief set computation is conveniently facilitated, also across the Web.

*Example 3.* Suppose an external knowledge base consists of an RDF file located on the web at "$\mathit{http}://\ldots/\mathit{data.rdf}$." Using an external atom $\&\mathit{rdf}[<\mathit{url}>](X, Y, Z)$, we may access all RDF triples $(s, p, o)$ at the URL specified with $<\mathit{url}>$. To form belief sets of pairs that drop the third argument from RDF triples, we may use the rule

$$\mathit{bel}(X, Y) \leftarrow \&\mathit{rdf}[\text{“}\mathit{http}://\ldots/\mathit{data.rdf}\text{”}](X, Y, Z).$$

The semantics of HEX-programs generalizes the answer set semantics of logic programs [7], but we omit a further account (as it is less relevant) and refer to [5, 6] for background and details. For execution, we use the dlvhex system [6], which implements HEX-programs providing a plugin mechanism for library and user defined external atoms.

## 3 Belief Set Merging using Merging Plans

We now develop our formal framework for merging belief sets, which introduces merging plans and merging tasks. In the following, we suppose to have a collection $KB = KB_1, \ldots, KB_n$ of knowledge bases with associated sets of belief sets $BS(KB_1), \ldots, BS(KB_n)$. For illustration, we use logic programs under answer set semantics.

Recall that we aim at merging the belief sets $BS(KB_i)$ as such, rather than the underlying knowledge bases $KB_i$. This may be necessary if the knowledge base access is limited, for instance in the Web context. There also frequently the source formats may be not aligned, such that beliefs and belief sets are similar but not identical. Possible mismatches have to be overcome and conflicts resolved.

A closer look at the problem reveals that two basic types of mismatches need attention, viz. *language* (*syntactic*) *incompatibilities* and *logical inconsistencies*.

**Syntactic Incompatibilities.** A first problem is that the belief sets may use different vocabularies to encode the same information. For example, the programs $P_1 = \{\mathit{degree}(\mathit{john}, \text{“}\mathit{MSc}\text{”}) \leftarrow\}$ and $P_2 = \{\mathit{deg}(\mathit{john}, \text{“}\mathit{Master\ of\ Science}\text{”}) \leftarrow\}$ have a single answer set with a single fact encoding the same information, but syntactically their answer sets are different. The problem may concern constants or predicate names.

We resolve this problem by introducing a so called common signature, which acts as a vocabulary shared by all sources, and applying mapping functions.

The *common signature* $\Sigma^C = (\Sigma_c^C, \Sigma_p^C)$ is a signature which suffices to define mappings from the collections of belief sets $BS(KB_i)$ over $\Sigma^{KB_i}$ to new collections of belief sets $\mathcal{B}'_i$ over $\Sigma^C$, such that $\mathcal{B}'_i = \mathcal{B}'_j$ if and only if the user considers $\mathcal{B}'_i$ and $\mathcal{B}'_j$ to represent equivalent information, with respect to the application in mind. A *belief set conversion* is then a function $\mu_i : 2^{\mathcal{A}(\Sigma^{KB_i})} \to 2^{\mathcal{A}(\Sigma^C)}$. Informally, $\mu_i$ maps the

[2] Strictly, [5] considers only positive literals but the extension to negative literals is trivial; furthermore, [5], also allows variables for predicate names which we do not need.

semantics of $KB_i$, expressed in the signature $\Sigma^{KB_i}$, to a semantics in the common signature $\Sigma^C$. The mapping has to provided by the user, who must ensure that converted sets of belief sets are identical only if she wishes them to be treated the same for merging.

Continuing our previous example, suitable mapping functions are

$$\mu_1(\mathcal{B}) = \mathcal{B},$$
$$\mu_2(\mathcal{B}) = \{\{degree(X, \text{“}MSc\text{”}) \mid deg(X, \text{“}Master\ of\ Science\text{”}) \in B\} \cup \{degree(X,Y) \mid deg(X,Y) \in B, Y \neq \text{“}Master\ of\ Science\text{”}\} \mid B \in \mathcal{B}\};$$

i.e., the belief sets of $P_1$ are unchanged while all occurrences of “$Master\ of\ Science$” in the belief sets of $P_2$ are changed to “$MSc$”, and predicate $deg$ is changed to $degree$.

The above notion of conversion is very general, but as in the example, often simple modular conversions at the level of belief sets ($\mu_i(\mathcal{B}) = \bigcup_{B \in \mathcal{B}} \mu'_i(B)$) or even at the level of atoms ($\mu'_i(B) = \{\tau_i(b) \mid b \in B\}$), for instance by mapping $\Sigma^{KB_i}$ via $\tau_i$ to $\Sigma^C$) may be used. More involved mappings may exploit schema matching and alignment (if possible), which however we omit here. After the mappings have been applied, we can safely assume that all sources are given over the same vocabulary.

**Logical Inconsistencies.** The second and more complicated type of conflicts concerns logical mismatches. While syntactic incompatibilities could be resolved by translating each source *independently* into the common language, logical inconsistencies only appear when multiple belief sets with contradicting contents are united.

We abstractly model application-dependent integrity constraints on sets of belief sets $\mathcal{B}$ as a set $\mathcal{C} \subseteq 2^{\mathcal{A}(\Sigma^C)}$, such that $\mathcal{B} \subseteq \mathcal{A}(\Sigma^C)$ satisfies the integrity constraints iff $\mathcal{B} \in \mathcal{C}$. Then, collections $\mathcal{B}_1, \mathcal{B}_2 \subseteq \mathcal{A}(\Sigma^C)$ of belief sets over the common signature $\Sigma^C$ violate the constraints (i.e., are inconsistent) iff $(\mathcal{B}_1 \cup \mathcal{B}_2) \notin \mathcal{C}$.

The resolution of such inconsistencies is only possible during the incorporation of the sources. For this purpose we introduce the concept of *merging operators*.

**Definition 3.** *An $n$-ary operator with parameters from $\mathcal{D}_1, \ldots, \mathcal{D}_m$, $m \geq 0$, is a function*

$$\circ^{n,m} : \underbrace{\left(2^{\mathcal{A}(\Sigma^C)}\right)^n}_{collections\ of\ belief\ sets} \times \underbrace{\mathcal{D}_1 \times \ldots \times \mathcal{D}_m}_{additional\ parameters} \rightarrow 2^{\mathcal{A}(\Sigma^C)} \ .$$

The first $n$ arguments are the collections of belief sets the operator is applied on. We assume that they have already been mapped to the common signature by applying the functions $\mu_i$. The other $m$ arguments over arbitrary domains $\mathcal{D}_i$ (like integers, enum types or strings) may provide additional information to control the behavior of the operator, e.g., by guiding it in special cases. The result of the operator is a further set of belief sets over the common signature $\Sigma^C$.

*Example 4.* The naive union operator, which has no parameter ($m = 0$), is defined as

$$\circ_{\cup}^{2,0}(\mathcal{B}_1, \mathcal{B}_2) = \{B_1 \cup B_2 \mid B_1 \in \mathcal{B}_1, B_2 \in \mathcal{B}_2, \nexists A : \{A, \neg A\} \subseteq (B_1 \cup B_2)\} \ ,$$

where the parameters $\mathcal{B}_1$ and $\mathcal{B}_2$ are sets of belief sets. The operator unions the belief sets of both sources pairwise, where classically inconsistent pairs are skipped.

If this operator is applied on the belief sets of the programs $P_1 = \{p \vee \neg p \leftarrow;\ q \leftarrow\}$ and $P_2 = \{p \vee r \leftarrow;\ s \leftarrow\}$ under the answer set semantics, the result is

$$\circ_{\cup}^{2,0}(AS(P_1), AS(P_2)) = \{\{p,\ q,\ s\}, \{p,\ q,\ r,\ s\}, \{\neg p,\ q,\ r,\ s\}\} \ .$$

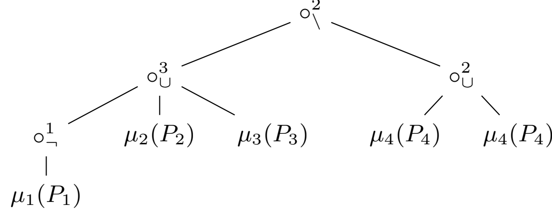


Fig. 1: Merging plan

The belief sets of the knowledge bases are $AS(P_1) = \{\{p,\ q\}, \{\neg p,\ q\}\}$ and $AS(P_2) = \{\{p,\ s\}, \{r,\ s\}\}$. This yields four pairs $B_1 \cup B_2$; one is inconsistent and thus skipped.

### 3.1 Merging Plans and Tasks

The result of an operator could be input to a further operator, similar as sub-expressions and numbers in complex arithmetic expressions. This leads to a hierarchical tree-structure with the converted belief sets $\mu_i(BS(KB_i))$ at the leaf nodes, and merging operators at the inner nodes. We call such a structure a *merging plan*, formally defined next.

**Definition 4.** *The set $\mathcal{M}_{KB,\Omega}$ of merging plans over knowledge bases $KB = KB_1, \ldots, KB_n$ and a set $\Omega = \{\circ_1, \ldots, \circ_n\}$ of operators is the smallest set such that*
*(i) each $M \in KB$, called* atomic *merging plan, is in $\mathcal{M}_{KB,\Omega}$;*
*(ii) if $\circ_i^{n,m} \in \Omega, s_j \in \mathcal{M}_{KB,\Omega}$ and $a_k \in \mathcal{D}_i$ for $1 \leq j \leq n, 1 \leq k \leq m$, then $(\circ_i^{n,m}, s_1, \ldots, s_n, a_1, \ldots, a_m) \in \mathcal{M}_{KB,\Omega}$.*

*Example 5.* Fig. 1 shows a graphical representation of a merging plan over logic programs $P_1, \ldots, P_5$ with primitive merging operators of different arities. It informally computes the negation of $P_1$ using the unary operator $\circ^1_\neg$, and unions this with $P_2$ and $P_3$ (using a ternary version of the operator). It then subtracts from this the union of $P_4$ and $P_5$, using set difference. The formal expression for this merging plan is

$$M = (\circ^2_\setminus,\ (\circ^3_\cup,\ (\circ^1_\neg,\ P_1),\ P_2,\ P_3), (\circ^2_\cup,\ P_4,\ P_5)).$$

With merging plans available, we now formalize *merging tasks*.

**Definition 5.** *A* merging task *is a quintuple $T = \langle KB, \Sigma^C, \mu, \Omega, M\rangle$, where $KB = KB_1, \ldots, KB_n$ are knowledge bases, $\Sigma^C$ is a common signature, $\mu = \mu_1, \ldots, \mu_n$ are belief set conversions $\mu_i : 2^{\mathcal{A}(\Sigma^{KB_i})} \rightarrow 2^{\mathcal{A}(\Sigma^C)}$, $\Omega$ is a set of operators, and $M \in \mathcal{M}_{KB,\Omega}$ is a merging plan over $KB$ and $\Omega$.*

The set of merging operators $\Omega$ is the only component that is, even though it is part of the formal task definition, usually not specific for a certain merging task in practice. It rather consists of approved operators which are probably useful in many different scenarios. The knowledge bases $KB$ will mostly exist before merging is planned and are often provided by third parties. The components $\Sigma^C$, $\mu_i$ and $M$ must be defined by the user as part of he merging scenario formalization.

Using our previous definitions, we define the outcome of a merging task next.

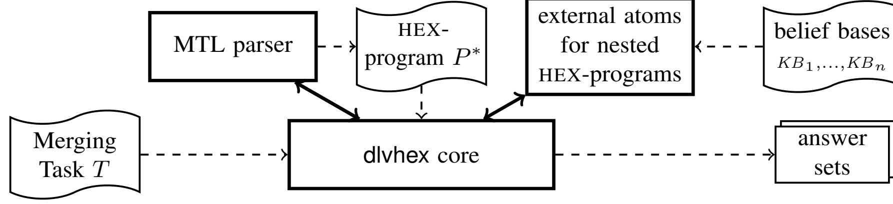


Fig. 2: MELD System Architecture (control flow ⟶ , data flow - - → )

**Definition 6.** *The* result *of a merging task $T = \langle KB, \Sigma^C, \mu, \Omega, M\rangle$, denoted as $[\![T]\!]$, is*

$$[\![T]\!] = \begin{cases} [\mu_i(BS(M))]_{\Sigma_p^C}, & \text{if } M \in KB, \\ [\circ^{n,m}([\![T_1]\!], \ldots, [\![T_n]\!], a_1, \ldots, a_m)]_{\Sigma_p^C}, & \text{if } M = (\circ^{n,m}, s_1, \ldots, s_n, a_1, \ldots, a_m), \end{cases}$$

*where $[\mathcal{B}]_{\Sigma_p^C} = \{\{p(a_1, \ldots, a_n) \in BS \mid p = (\neg)p',\ p' \in \Sigma_p^C\} \mid BS \in \mathcal{B}\}$ denotes the projection of $\mathcal{B}$ to the atoms over $\Sigma_p^C$, and $T_i = \langle KB, \Sigma^C, \mu, \Omega, s_i\rangle$, $1 \leq i \leq n$.*

Informally, if $M$ is an atomic merging plan (i.e., a knowledge base), then it can be evaluated directly and the result is just the associated set of belief sets, mapped to the common signature. Otherwise, $M$ contains at least one operator application, and the result is the one of the topmost operator, applied on the results of the merging sub-plans.

*Example 6 (cont'd).* Let $M$ be the merging plan from Example 5 and consider programs

$$P_1 = \{a.,\ b.\}, P_2 = \{x.,\ y.\}, P_3 = \{\neg a.,\ c.\}, P_4 = \{a.,\ x.\}, P_5 = \{c.,\ x.,\ y.\}$$

that consist only of facts (rules $p \leftarrow$, omitting $\leftarrow$). The complete merging task is $T = \big\langle \{P_1, \ldots, P_5\}, \Sigma^C, \mu_{id}, \Omega, M \big\rangle$, where the common signature contains the propositional (0-ary) atoms $a$, $b$, $c$, $x$ and $y$, all mappings in $\mu_{id}$ are identity functions (since all knowledge bases use already the common vocabulary), and $\Omega = \{\circ_{\cup}^3, \circ_{\setminus}^2\, \circ_{\neg}^1\}$.

Now we compute the result $[\![T]\!]$ of this merging task as follows. For the sake of readability, we use $[\![M]\!]$ as an abbreviation for $[\![\{P_1, \ldots, P_5\}, \Sigma^C, \mu_{id}, \Omega, M]\!]$:

$$\begin{aligned} &[\![\langle\{P_1, \ldots, P_5\}, \Sigma^C, \mu_{id}, \Omega, M\rangle]\!] = \\ &\quad \circ_{\setminus}^2 \big( [\![(\circ_{\cup}^3, (\circ_{\neg}^1,\ P_1),\ P_2,\ P_3)]\!],\ [\![(\circ_{\cup}^2, P_4, P_5)]\!] \big) = \\ &\quad \circ_{\setminus}^2 \big( \circ_{\cup}^3([\![(\circ_{\neg}^1,\ P_1)]\!],\ [\![P_2]\!],\ [\![P_3]\!]),\ [\![(\circ_{\cup}^2, P_4, P_5)]\!] \big) = \\ &\quad \cdots = \circ_{\setminus}^2 (\{\{\neg a, \neg b, c, x, y\}\},\ \{\{a, c, x, y\}\}) = \{\{\neg a, \neg b\}\}. \end{aligned}$$

We may view a merging task $T$ as a knowledge base per se by casting it into a knowledge base in some logic language; e.g., $T$ may be cast to the classical formula $\neg a \wedge \neg b$.

The examples above are trivial and involve only simple set operations, but still illustrate the principles. We clearly can use advanced belief merging operators, showing the usefulness of the framework. We will see this in the following sections.

## 4 The MELD System

We have implemented our framework in MELD (*<u>ME</u>rging <u>L</u>ibrary for <u>D</u>lvhex*), using the infrastructure and possibilities of the dlvhex system to transparently access heteroge-

neous (possibly dispersed) knowledge sources. In MELD, we assume that the knowledge bases $KB_i$ are given as HEX-programs [6]. This has the great advantage that by using external atoms, the belief sets (which then are answer sets) may contain information from virtually any knowledge source, e.g. the tuples of a relational database, the triplets of an RDF ontology, or a model of a propositional formula. The HEX-programs only serve as an interface to these sources, enabling access to their belief sets (in fact, using nested HEX-programs described below).

The architecture of MELD is shown in Fig. 2. Essentially the system consists of three major components: (1) a language for specifying merging tasks in a machine-readable format; (2) a compiler which translates declarative task descriptions into semantically equivalent *nested* HEX-*programs*, i.e., HEX-programs with program nesting. The program constructed computes the merged belief sets in its answer sets; and (3) a suite of specific external atoms developed in dlvhex which allow for executing nested HEX-programs.

The system is realized as a plugin to dlvhex, and provides a user-friendly interface for the specification of a merging task $T = \langle KB, \Sigma^C, \mu, \Omega, M \rangle$ as defined above. The knowledge bases $KB$ are given as (nested) HEX-programs, and the merging operators $\Omega$ are implemented as C++ classes. MELD comes with a few predefined operators, and can be easily extended, using a plugin-mechanism, with user-defined operators in C++. The components $\Sigma^C$, $\mu_i$ and $R$ are the most specific parts of a certain merging scenario. All components are declaratively specified in an INI-style text file (we use here filename extension `.mt`), in a dedicated *merging task language (MTL)*, which we discuss below. Then MELD can be used to compute merging task result according to the semantics above by executing the command: `dlvhex --merging task.mt`. The evaluation of merging tasks will be described below.

**Merging Task Language (MTL).** To specify merging tasks in a machine-readable way, we defined the *merging task language* (MTL). For space reasons, here an in depth presentation is not possible, and we will illustrate it on examples. More details, including the complete syntax, can be found on the system's website (see footnote 1).

*Example 7.* Consider the merging task description in Fig. 3. It consists of three parts.

- `[common signature]`: The first part is the common signature, which is a list of all predicate names with associated arities. Only atoms over these predicates will be regarded during belief set merging.
- `[belief base]`: The second part is the declaration of the belief bases. For each belief base, a unique name and the mapping function to the common signature are specified, via arbitrarily many mapping rules under the HEX-semantics. The mapping may be done directly, as for belief bases `input1` and `input3`, or outsourced as in case of `input2`. Note that the actual belief sources are *not* defined directly, as they are given implicitly and accessed by queries in the rule bodies. E.g., in `input1` we access an RDF file on the web using the external atom `&rdf`. The mapping rules can derive arbitrary atoms in the heads (also intermediate atoms), but only those using predicates listed in the common signature will be regarded during merging.
- `[merging plan]`: The third part is a tree-structured merging plan, which defines how to combine the sources, described as a nested expression, with names of belief bases at the bottom and operators applied to inputs (`source`). In the example, we first compute the union of `input1` and `input2`, and subsequently subtract `input3`.

```
[common signature]
predicate:foo/1; predicate:bar/3;

[belief base]
name: input1;
mapping: "bar(X, Y, Z) :- &rdf[\'http://...\'](X, Y, Z).";
mapping: "foo(Y) :- &rdf[\'http://...\'](X, Y, Z).";

[belief base]
name: input2;
source: "P2.lp";

[belief base]
name: input3;
mapping: "foo(X) :- &dlC[\'http://...\',a,b,c,d,\'student\'](X)";

[merging plan] {
     operator: setminus;
     source: {
         operator: setunion;  source: {input1};  source: {input2};
     };
     source: {input3};
}
```

Fig. 3: Merging Task Description

MELD allows the automatic computation of the merged belief sets according to a merging plan of this kind a more elaborate example is discussed in Section 5.

**Translation to Nested HEX-Programs.** We briefly describe how merging task description are evaluated in MELD. The key concept are *nested* HEX*-programs*.

We designed a suite of external atoms which allow to evaluate a (possibly nested) HEX-program $P$ given as input, and to access each answer set of $P$ like an object in the host program. Thus, processing the answer sets and reasoning over them, inside another program, is possible. To our knowledge, this is the first ASP language featuring this and of independent interest. The sub-programs can be executed *independently* of the host program, such that their answer is imported into the main program and computation continued afterwards. We realized nested HEX-programs using *handles* that refer to sub-programs, answer sets and their constituents; this is best explained with an example.

*Example 8.* Consider the following two rules.

$$h(H, S) \leftarrow \&hex[\text{“}node(a).\ node(b).\ edge(a, b).\text{”}, \text{“”}](H), \&answersets[H](S).$$
$$p(P, A) \ \leftarrow h(H, S), \&predicates[H, S](P, A).$$

The external atom $\&hex$ in the first rule is used to execute the sub-program $Q$ given as string literal “$node(a).\ node(b).\ edge(a, b).$”. It will “return” a unique integer value $H$ that can be used later on to investigate the answer to $Q$. Here, this done in the evaluation of the external atom $\&answersets$, which in turn returns, one by one, a set of unique handles $S$ that point to the answer sets of $Q$. In the second rule, we pass each pair $(H, S)$ retrieved by the first rule to the external atom $\&predicates$ which finds out the names and arities of the predicates contained in the respective answer set. This well lead to the atoms $p(node, 1)$ and $p(edge, 2)$. We could go a step further and also retrieve the

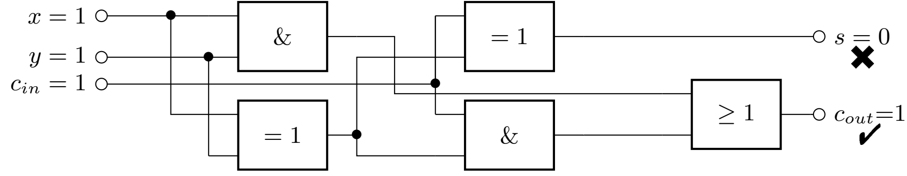


Fig. 4: Malfunctioning full adder (expected output: $s = 1$ and $c_{out} = 1$)

arguments of the atoms in $Q$'s answer sets, using further external atoms provided by our plugin. Moreover, by using $\&$*hexfile*, sub-programs in external files can be included.

**Evaluation of merging tasks**. We have implemented a transformation which parses a declarative merging task, specified in MTL, and assembles a semantically equivalent HEX-program $P^*$ that uses program nesting, reflecting the merging plan structure. The translation is complex and we omit the details here.

Briefly, $\&hex$ resp. $\&$*hexfile* atoms serve as starting point for evaluating atomic merging plans, i.e., merging plans which consist of a single belief base without operator applications. For non-atomic merging plans, we compute the result bottom-up like an arithmetic expression. To this end, we realized an external atom $\&operator$ which allows us to call operators implemented as C++ classes. As answer sets are accessible objects in our extension, we can pass them from operator to operator until we finally retrieve the result of the topmost operator, which yields the outcome of the merging plan.

Our implementation automatically assembles and evaluates $P^*$ when dlvhex is started with the `--merging` option. The input files must contain a merging task description. For details of the transformation and a proof of the correctness, we refer to [14].

## 5 Belief Merging in Action

We now consider a more realistic belief merging example in fault diagnosis, which is a classical KR problem. In the course of this, we consider different merging operators, which are based on distance functions and give rise to a hierarchically constructed a family of such operators, and we report how the problem can be solved in MELD.

*Example 9 (**Circuit Diagnosis**).* Consider the full adder circuit shown in Fig. 4. Given input values $x = y = 1$ and carry input $c_{in} = 1$, the value of the output carry $c_{out} = 1$ is correct, but the output sum $s$ should be $1$ instead of $0$. Any component in the circuit may be broken, leading to different possible abductive explanations (i.e., fault assumptions that logically entail the observation): either (1) the XOR gate on the lower left ($xor_1$), (2) the XOR gate located middle top ($xor_2$), or (3) both $xor_1$ and $xor_2$ are malfunctioning; the result is not unique.

Here our framework comes into play: different experts may find different explanations. For a collective diagnosis, we must integrate the individual opinions such that (i) the group explanation is still a valid explanation, and (ii) it is close to the individual opinions (under a suitable notion).

Suppose we have three experts inspecting the malfunctioning full adder, with individual explanations $AS(P_1) = \{ab(xor_1)\}$, $AS(P_2) = \{ab(xor_2)\}$ and $AS(P_3) =$

```
[common signature]
predicate:ab/1;

[belief base]
name: expert1;
source: "P1.lp";

[belief base]
name: expert2;
source: "P2.lp";
```

```
[belief base]
name: expert3;
source: "P3.lp";

[merging plan] {
    operator: dbo; bsdistance: "ignoring";
    constraintfile: "fulladder.lp";
    constraintfile: "fault.obs";
    {expert1}; {expert2}; {expert3};
}
```

Fig. 5: Group decision problem

$\{ab(xor_1), ab(xor_2)\}$. Informally, expert 1 believes that $xor_1$ is broken, expert 2 suspects $xor_2$ is broken, and expert 3 believes both are broken. Clearly, besides on these opinions, the overall result depends on the merging operator.

**Distance-based merging operators.** A popular class of operators is defined using a distance function for between two interpretations resp. sets of literals. We call such operators *distance-based operators*. An example is an adaption of Dalal's distance between interpretations [2] (which is their Hamming distance) to $|B_1 \triangle B_2|$, the cardinality of the symmetric difference of belief sets $B_1$ and $B_2$. For more discussion, see [9].

We build a distance function $D_{d,\underline{d}}(B, KB)$ between a belief set $B$ and knowledge bases $KB = KB_1, \ldots, KB_n$ in three steps:

1. We start from a distance function $\underline{d}(B_1, B_2)$ between two belief sets. Besides the adapted Dalal distance, which we denote with $\underline{dal}$, many other choices are possible (e.g., weighted distance; we consider two other options below).
2. Next, we define a distance function $d$ on top of $\underline{d}$ to measure the distance of belief set $B$ to a knowledge base $KB$, denoted $d_{\underline{d}}(B, KB)$, by aggregating the values $\underline{d}(B, B')$ for all belief sets of $B' \in BS(KB)$. Here a popular choice is $d = min$:
$$min_{\underline{d}}(B, KB_i) = \min_{B' \in BS(KB_i)} \underline{d}(B, B')$$
i.e., we informally take the distance of $B$ to the closed belief set of $KB$.
3. For each $1 \leq i \leq n$, the function $d_{\underline{d}}(B, KB_i)$ yields a distance value; these $n$ values are aggregated into a single value $D_{d,\underline{d}}(B, KB)$. A popular choice for $D$ is the sum:
$$sum_{\underline{d},d}(B, KB) = \Sigma_{i=1}^{n} d_{\underline{d}}(B, KB_i)$$

Summarizing, in each step some parameter ($\underline{d}$, $d$, $D$) can be chosen to arrive at $D_{\underline{d}}(B, KB)$. Given the latter, the following merging operator is straightforward:

$$\Delta_{\underline{d},d,D}(KB, \mathcal{C}) = \arg\min_{B \in \mathcal{C}} D_{\underline{d},d}(B, KB),$$

i.e., selecting among all possible belief sets that satisfy the (application dependent) constraints $\mathcal{C}$, a belief set $B$ which is at minimum overall distance to $KB$.

**Solving the group decision problem.** In MELD, a group decision of the three experts can be obtained using the above operator $\Delta_{\underline{d},d,D}(KB, \mathcal{C})$ (named `dbo` for *d̲istance-b̲ased o̲perator* there), where the constraints $\mathcal{C}$ ensure that the result is still an abductive explanation. The merging task is specified as follows.

The belief bases with respective belief sets (as answer sets) are in external files `Pj.lp`. with answer sets as described above (either hard-coded or suitable computed). As all

external programs deliver belief sets over predicate ab, no mapping functions are specified; this makes the implementation use identity mappings. The merging plan applies the previously defined operator on the three individual belief sets, where in MELD $d_{\underline{d}} = min_{\underline{d}}$ and $D_{d,\underline{d}} = sum_{d,\underline{d}}$ by default, and $\underline{d}$ (the distance between two belief sets) is defined in the `bsdistance`-statement. Setting `bsdistance` to `dal` gives us the adapted Dalal distance $\underline{dal}$. The group decision is then $\{\{ab(xor_1)\}, \{ab(xor_2)\}\}$; either $xor_1$ or $xor_2$ is defect. Omitting formal details, the value `ignoring` for `bsdistance` penalizes situations where atoms from individual belief sets are missing in the group decision candidate. This results in the group decision $\{\{ab(xor_1), ab(xor_2)\}\}$: it satisfies all three experts completely - no beliefs are ignored, thus has distance $0$.

Besides `ignoring` and `dal`, MELD supports the option `unfounded`, which penalizes situations where the group decision candidate contains atoms *not* occurring in an individual belief set. This will yield the result $\{\{ab(xor_1)\}, \{ab(xor_2)\}\}$. Each explanation is minimal, as its single atom is unfounded for only one of the experts.

This example demonstrates the advantages of the framework and its implementation compared to hard-coding: It is easy to prototype merging scenarios and quickly change merging operators and parameters. If we would like to add further expert opinions or sub-divide the group into sub-groups and aggregate the decisions hierarchically, this could be easily done by modifying the `.mt` file accordingly.

## 6 Evaluation and Experiments

We take a closer look at MELD regarding performance and usefulness in practice. The runtime behavior is less an issue for two reasons. First, the system is intended to serve as a rapid prototyping tool to support the user when experimenting and evaluating different merging strategies. For a production version, a hard-coded implementation can be considered after an optimal setting has been found. Second, the behavior is largely determined by the merging operators in use, as the information flow in between and the translation of formal merging tasks to nested HEX-programs are both linear in the number of belief bases and the sizes of their answer sets. Also evaluating the assembled program $P^*$ is not a big issue as its structure is fairly simple. The merging operators are application dependent and can be implemented and optimized by the user, so our framework does not cause notable overhead. We now describe real world tasks which can be solved by MELD with merging task descriptions similar to Example 9.

**Decision diagram merging.** Decision diagrams are an important tool for decision making in many fields of science. This is because compared to other formalisms (e.g. production rules) they are intuitive even for non-professionals in knowledge engineering. Biomedical examples include severity ratings of diseases depending on patient data (in particular tumor staging systems [19]), DNA classification (coding vs. junk DNA), and aids for therapy selection. Another frequent application domain is business and economy (e,g., liquidity appraisal in economics [1]).

Informally, a *decision diagram* is are rooted directed acyclic graph $D = (V, R)$. Each edge in $E$ is labeled with a condition $X \diamond Y$, where $\diamond$ is a comparison operator and $X$ and $Y$ are variables or values from suitable domains. For example, $b \leq 12.5$ may compare a blood value $b$ of a patient to the maximum value for healthy people, and each

leaf (a node without out-edges) of $D$ is tagged with a class label. Clearly, $D$ must satisfy further structural and semantic conditions, but we simply omit this here.

To classify an instance, one starts at the root (the only node without in-edges) and follows an edge iff its condition is satisfied. This is repeated until a leaf is reached; there one reads the assigned class. Sometimes we have multiple similar but non-equivalent diagrams, e.g. due to different expert opinions or different training sets if the diagrams stem from machine-learning tools. It becomes then necessary to incorporate the input classifiers into a single one. If we encode decision diagrams as sets of facts (e.g., over predicates *leaf*(X, C), $innernode(X, Y)$, etc.) and provide merging operators tailored for decision diagrams (and decode the encodings internally), the merging can clearly be done automatically using MELD. This is a great advantage if it is not clear right from the beginning which training algorithms and merging strategies behave best.

As a concrete example we consider a popular approach to classify protein-coding DNA sequences [17]. One first computes numerical features of the sequences in a large annotated training set $T$, which are known to vary significantly between coding and non-coding DNA for biological reasons. We used 20 numerical features proposed by [12]. They are computed for a set of sequences of 54 bases each. Subsequently, one trains a decision diagram $D$ over these features; we will work with decision *trees* here, which are a special form of decision diagrams. When a new sequence needs to be classified, one computes the feature values and runs through $D$.

Here our framework comes into play, realizing an idea discussed in [18]. Instead of training a single decision tree over $T$, first split $T$ into subsets $T_1, \dots, T_n$ and train classifiers $D_1, \dots, D_n$ on them (thus fostering parallelization); then merge the $D_i$ into a single tree $D$. We found that combining several classifiers trained by different machine-learning algorithms may significantly improve the final result. This is not always true, but depends on the training set $T$, the selection of learners, and the merging procedure. Some combinations may increase the accuracy compared to a single tree trained over $T$, while others decrease it. However, this exactly demonstrates the strengths of our framework: it is easy to try out several different scenarios and evaluate the results empirically, while the technical details of the merging process are managed by MELD.

For training data extracted from the Human-Genome Project, we achieved our optimal result using a merging operator inspired by the algorithm in [1] and three input trees trained over only 10 sequences each (hence each tree queried only the feature with highest entropy), gaining an accuracy increase from 48.85% to 65.25%; see [15] for details. Many other approaches based on statistical features were developed; in comparison, our result is fairly good. Most of them produce a classifier with accuracy only slightly above 70%, see e.g., [17], including recent ones [20, 21]; this suggests that there may be a close by natural limit for statistical features.

A further finding of our experiments is that by training multiple classifiers and merging them afterwards, mostly a much smaller training set (in total) suffices to gain the same accuracy as by training a single classifier; e.g., for an accuracy of 65.25%, the latter needed several hundred sequences (compared to 30). Furthermore, the merged decision tree usually has a lower depth than a tree created over a single, but larger training set. Obtaining these results would have been much harder without the framework, since the merging of the classifiers had to be done by hand after each change of parameters.

Our merging operator implements an algorithm developed in [18] and realized in the MORGAN system. But in contrast to MORGAN, where the algorithm is implemented directly as part of the main system, MELD sources it out into an operator library. This simplifies the implementation of further merging strategies and exchanging them easily.

**Judgment aggregation.** In Section 4 we have seen how to incorporate individual beliefs into a group decision; this is a common problem in social choice theory [3]. Realistic applications include planning of group activities with individual preferences, and diagnosis making by teams of several doctors.

**Syntactic belief merging.** We distinguished syntactic belief merging approaches, i.e., merging sets of formulas or programs, and semantic approaches. Even if our framework is essentially semantic, we may also use it for syntactic tasks by using an appropriate encoding of formulas or programs. That is, we encode the knowledge base as sets of literals. This allows us to use MELD for (e.g. talking about the same domain but focusing on different details), we need to combine them into a single ontology [22]. Given appropriate merging operators which decode the ontologies represented by literal sets internally, this task can clearly be supported by MELD. To vary the ordering of the sources is then easy. If the merging operator is not commutative and associative or if we have multiple alternative operators, the quality of the final ontology can vary as well, and we may find the best one by empirically using MELD.

## 7 Related Work and Conclusion

While the theory of belief merging has a rich literature, only few implemented systems are available. In [8], the authors present an implementation of Removed Set Fusion based merging of logic programs by translating sets of logic programs into a single logic program, whose answer sets correspond to removed sets. Compared to our approach, this method uses a syntactic strategy for merging belief bases and has a fixed semantics.

COBA [4] handles belief *revision* rather than merging. Its approach to finding consistent models is similar to our implementation of distance-based operators in Section 4. In contrast to our generic framework, COBA uses a fixed semantics.

The MORGAN system [18] was used to merge DNA classification trees. The merging operator we used for our experiments is almost equivalent to this system. However, while we implemented the algorithm as merging operator for our flexible framework, MORGAN is hard-coded. Therefore it is easy to modify parameters and make experiments with different settings in our system. The changes only concern the declarative task description, while in MORGAN one needs to rewrite the main source files.

Our approach works semantically, i.e., it does not merge logic programs but interpretations of the programs. Nevertheless, by encoding formulas as literals one can also implement syntactic strategies within the framework. Future work will include addressing performance issues which was neglected so far since the main goal was flexibility. Another possible extension is the implementation of additional merging operators. This increases the chance that the user will find a suitable one and avoids that she needs to implement it on his own. Also support for syntactic strategies for belief merging may be fruitfully deployed in our framework. Additionally, the merging task language may be extended by further language constructs like iterative application of operators.

## References


1. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: ICCAD'93. pp. 188–191 (1993)
2. Dalal, M.: Investigations into a theory of knowledge base revision. In: AAAI. 475–479 (1988)
3. Dasgupta, P.S., Hammond, P.J., Maskin, E.S.: The implementation of social choice rules: Some general results on incentive compatibility. Rev. Econ. Stud. 46(2), 185–216 (1979)
4. Delgrande, J.P., Liu, D.H., Schaub, T., Thiele, S.: COBA 2.0: A Consistency-Based Belief Change System. In: ECSQARU'07. pp. 78–90. Springer (2007)
5. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: IJCAI'05. 90–96 (2005)
6. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: dlvhex: A system for integrating multiple semantics in an answer-set programming framework. In: WLP'06. pp. 206–210 (2006)
7. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generat. Comput. 9(3–4), 365–385 (1991)
8. Hué, J., Papini, O., Würbel, E.: Merging belief bases represented by logic programs. In: ECSQARU'09. pp. 371–382. Springer (2009)
9. Konieczny, S., Lang, J., Marquis, P.: $DA^2$ merging operators. AIJ 157(1-2), 49–79 (2004)
10. Konieczny, S., Pérez, R.P.: On the logic of merging. In: KR'98. pp. 488–498 (1998)
11. Liberatore, P., Schaerf, M.: Arbitration (or how to merge knowledge bases). IEEE Trans. Knowl. Data Eng. 10(1), 76–90 (1998)
12. Liew, A.W.C., Wu, Y., Yan, H.: Selection of statistical features based on mutual information for classification of human coding and non-coding DNA sequences. In: ICPR. 766–769 (2004)
13. Lin, J., Mendelzon, A.: Knowledge base merging by majority. In: Dynamic Worlds: From the Frame problem to Knowledge Management. Kluwer (1999)
14. Redl, C.: Development of a belief merging framewerk for dlvhex. Master's thesis, Vienna University of Technology, A-1040 Vienna, Karlsplatz 13 (June 2010), `http://media.obvsg.at/AC07808210-2001`
15. Redl, C.: Merging of biomedical decision diagrams. Master's thesis, Vienna University of Technology, A-1040 Vienna, Karlsplatz 13 (October 2010), `http://media.obvsg.at/AC07808795-2001`
16. Revesz, P.: On the semantics of arbitration. Intl. J. Algebra Comput. 7(2), 133–160 (1997)
17. Salzberg, S.: Locating protein coding regions in human DNA using a decision tree algorithm. J. Comput. Biol. 2, 473–485 (1995)
18. Salzberg, S., Delcher, A.L., Fasman, K.H., Henderson, J.: A decision tree system for finding genes in DNA. J. Comput. Biol. 5(4), 667–680 (1998)
19. Sobin, L., Gospodarowicz, M., Wittekind, C.: TNM Classification of Malignant Tumours. Wiley, 7 edn. (2009)
20. Sree, P.K., Babu, I.R., Murty, J.V.R., Rao, P.S.: Towards an artificial immune system to identify and strengthen protein coding region identification using cellular automata classifier. Intl. J. Comput. Commun. 1(2), 26–34 (2007)
21. Sree, P.K., Babu, I.R.: Identification of protein coding regions in genomic DNA using unsupervised FMACA based pattern classifier. Intl. J. Comp. Sci. Netw. Secur. 8(1), 305–309 (2008)
22. Stumme, G., Maedche, A.: FCA-MERGE: Bottom-Up Merging of Ontologies. In: IJCAI'01, pp. 225–230 (2001)