mergingplugin - User Guide

Christoph Redl*

April 03, 2010

The mergingplugin consists of several external atoms for dlvhex as well as a merging plan compiler which is intended to translate *merging plans* into semantically equivalent HEX programs.

The plugin was developed as part of the master's thesis *Development of a Belief Merging Framework for dlvhex* [1].

1 External Atoms

The merging plugin provides the following external atoms.

1.1 Execution of Nested Programs

1.1.1 &hex

&*hex* is a unary predicate with two input parameters that is intended to execute nested HEX programs.

$$\&hex[Prog, Args](A)$$

An evaluation will execute the hex program in variable *Prog* with the dlvhex arguments given in *Args*. The result is an integer value (*handle*) that *represents* the program's result symbolically. That means, the numeric value is irrelevant, but it can be used to access the result later on (similar to pointers in programming languages).

Note that *Prog* is expected to contain the program to execute directly as string literal and *not* the filename of the program.

Example 1.

$$handle(H) \quad :- \quad \&hex[\text{``}a.b.c : -a.\text{''}, \text{``''}](H).$$

In case the program to embed contains double quotation marks ("), they must be represented with the escape sequence \'. The escape sequence for the backslash character (\\) is \\\\.

*christoph.redl@tuwien.ac.at

Example 2. Assume we want to embed the program:

$$p(constant, \text{“}string\ literal\ containing\ a \setminus backslash\text{”}).$$

Then the host program looks like this:

$$\begin{aligned} handle(H) \quad :- \quad & \&hex[\text{“}p(constant, \\ & \backslash' string\ literal\ containing\ a \backslash\backslash\ backslash \backslash').\text{”}, \text{“”}](H). \end{aligned}$$

1.1.2 &hexfile

&*hexfile* is again a unary predicate with two input parameters that is intended to execute nested HEX programs which are stored within *files* in the file system.

$$\&hexfile[File, Args](A)$$

An evaluation will execture the program in the file named *File* with the dlvhex arguments given in *Args*. The result is a handle to the program's result.

1.2 Investigating the Result

1.2.1 &answersets

&*answersets* is a unary predicate with one input parameter.

$$\&answersets[H](AS)$$

H is expected to be a handle to a program's result (see 1.1). Then the atom will deliver handles AS to each answer-set in this result.

Example 3. The program

$$\begin{aligned} handle(H, AS) \quad :- \quad & \&hex[\text{“}a.b.c :- a.\text{”}, \text{“”}](H), \\ & \&answersets[H](AS). \end{aligned}$$

will have one answer-set, namely $\{handle(0, 0)\}$, where the first 0 is a handle to the embedded progrm's result and the second 0 a handle to the first answer-set of this program.

Note that answer-set handles are only unique *relative* to a certain program answer. Thus, if multiple embedded programs are executed, both the handle to the program's result as well as the handle to an answer-set is required to uniqly identify an answer-set.

1.2.2 &predicates

$\&predicates$ is a binary predicate with two input parameters.

$$\&predicates[H, AS](Pred, Arity)$$

For a given handle to a program's result H and a given handle to an answer-set AS, it returns tuples $(Pred, Arity)$ of all predicates together with their arities that occur within this answer-set.

Example 4. The program

$$\begin{aligned} preds(Pred, Arity) \quad :- \quad & \&hex[\text{“}a.p(x,y).\text{”}, \text{“”}](H), \\ & \&answersets[H](AS), \\ & \&predicates[A, AS](Pred, Arity). \end{aligned}$$

will have one answer-set, namely $\{preds(a, 0), preds(p, 2)\}$.

1.2.3 &arguments

$\&arguments$ is a ternary predicate with three input parameters.

$$\&arguments[H, AS, Pred](I, ArgIndex, Value)$$

For a given predicate $Pred$ within a certain answer-set (identified by H and AS), it will return all the information about this predicate that occurs within this answer-set.

Each triple that is returned tells the $Value$ of the parameter with index $ArgIndex$ in the I-th occurrence of the predicate. I is just a running index that enables the user to distinct different occurrences of the same predicate (since a predicate can occur multiple times with different parameters). All triples with the same value for I describe one occurrence of the predicate. The special index s returns the sign of the predicate: 0 for positive and 1 for (strongly) negated.

Example 5. The program

$$\begin{aligned} val(Pred, I, ArgIndex, Value) \quad :- \quad & \&hex[\text{“}p(a,b).\neg p(x,y).q(f).\text{”}, \text{“”}](H), \\ & \&answersets[H](AS), \\ & \&predicates[A, AS](Pred, Arity), \\ & \&arguments[A, AS, Pred] \\ & \quad (I, ArgIndex, Value). \end{aligned}$$

will have one answer-set, namely

$$\{val(p, 0, s, 0), val(p, 0, 0, a), val(p, 0, 1, b),$$

$\underline{val(p,1,s,1)}, val(p,1,0,x), val(p,1,1,y), val(q,0,s,0), val(q,0,0,f)\}$

This expresses that in the 0-th occurrence of p, the sign is *positive* (0), the 0-th parameter is a and the 1-st parameter is b.

Similar for the 1-st occurrence of p, where the sign is negative. q has just one paramter which is f in the 0-th occurrence.

1.3 Operator Application

The mergingplugin further supports the use of *operators*. Operators get n answers (i.e. sets of answer-sets) as input and compute a further set of answer-sets as output. Additionally they may get key-value pairs (over strings) as input.

The predicate *&operator* is unary with three input parameters. It's output is a handle to the operator's result.

$$\&operator[OpName, Answers, KeyValuePairs](H)$$

OpName is a string containing the name of the operator to apply. *Answers* is a binary predicate, that contains index-handle pairs. They tell the operator *which* answer (identified by it's handle) to pass on *what* parameter position. *KeyValuePairs* is a further binary predicate with key-value pairs.

Example 6. The program

$$\begin{array}{rcl}
input(0,H) & :- & \&hex[\text{``}a.\text{''}, \text{``''}](H). \\
input(1,H) & :- & \&hex[\text{``}b.\text{''}, \text{``''}](H). \\
keyvaluepairs(key1, v1). & & \\
keyvaluepairs(key2, v2). & & \\
output(H) & :- & \&operator[\text{``}union\text{''}, input, keyvaluepairs](H). \\
preds(Pred) & :- & output(H), \&answersets[H](AS), \\
& & \&predicates[H, AS](Pred, Arity).
\end{array}$$

executes two embedded programs, one with answer:

$$\{\{a\}\}$$

and the other one with:

$$\{\{b\}\}$$

Assume that operator "union" is defined with the usual mathematical semantics. Additionally, it includes all values of the key-value pairs in the final answer. Then the evaluation of the *&operator* predicate will pass $\{a\}$ on the 0-th parameter position and $\{b\}$ on the first one to this operator. It further passes the key-value pairs $(key1, v1)$ and $(key2, v2)$.

The operator will compute the result $\{a, b, v1, v2\}$, which is investigated with the &*predicates* evaluation. The final result of the program is therefore

$$\{input(0,0), input(1,1), keyvaluepairs(key1, v1), keyvaluepairs(key2, v2),$$
$$output(3), preds(a), preds(b), preds(v1), preds(v2)\}$$

2 Operator Implementation

2.1 Operator Libraries

Operators are organized as *operator libararies*, where each library can contain arbitrary many operators. An operator library must be compiled as shared object library that is installed either in the system or the user plugin directory of dlvhex. Note: Additional plugin directories that are passed to dlvhex using the command line argument "–plugindir" (or "-p") will *not* be searched for operator libraries. However, the mergingplugin provides an own command line parameter for specifying additional operator locations (see 3).

Entry point of an operator library is a method with the following signature:

std::vector<IOperator*>OPERATORIMPORTFUNCTION()

This method must return a vector with pointers to instances of all the operator implementations in this library (see below). the mergingplugin will call this method on startup and load all operators that are returned by this function.

2.2 Operator Classes

Operators are C++ classes (within operator libraries) that implement the interface IOperator, which is installed in the following subdirectory of the include directory:

"dlvhex/mergingplugin/IOperator.h"

The interface defines two abstract methods, namely:

- std::string getName()
  The operator is expected to return it's desired name. Later, the same name is expected as parameter for the &*operator* predicate to call this operator.

  In case that multiple operators with the same name are defined, the mergingplugin will print a warning on startup and ignore all but the first one.

- HexAnswer apply(int arity, std::vector<HexAnswer*>& answers, OperatorArguments& parameters) throw (OperatorException)
  This method is called when the operator is actually applied. It's input is the number of answers that are passed to the operator (arity) as well as the answers themselves (answers). The answers are passed as vector of HexAnswer, which is defined as vector of AtomSet (since a HEX answer is a set of answer-sets.

  Finally, OperatorArguments is the set of key-value pairs. It is defined as std::pair<string, string>.

  The method is expected to return the operator's result as set of answer-sets (i.e. HexAnswer). In case of an error, an OperatorException can be thrown which will result in a *PluginError* and thus a termination of dlvhex.

3 Command Line Arguments

The mergingplugin recognizes the following command line arguments

3.1 operatorpath or op

Using the syntax

`--operatorpath=path1,path2,...` or `--op=path1,path2,...`

additional paths where operators are loaded from can be specified. A path can point to a directory or a shared object library. In case of a directory, operator libs that are *directly* within this directory will be loaded (*non-recursive*!).

3.2 inputrewriter or irw

The syntax

`--inputrewriter=program` or `--irw=program`

specifies an *input rewriter*. This can be an arbitrary tool that reads from standard input and writes to standard output. The complete dlvhex input will be directed through this program before reasoning starts.

4 Merging Plan Compiler

The merging plan compiler is installed as part of the mergingplugin. It can be called in command line by entering:

`mpcompiler`

with appropriate parameters.

This tool translates a belief merging scenario into a dlvhex program. The merging scenario is defined in one or more input files or is read from standard input.

4.1 Options

The command line options are:

- `-parsetree`
  Generates a parse tree rather than dlvhex code (mostly for debug tasks).

- `-help`
  Prints an online help message.

- `-spirit` or `-bison`
  Forces the compiler to use a *boost spirit* resp. *bison* generated parser. Default is spirit.

If no filenames are passed, the compiler will read from standard input. If at least filename is passed, standard input will *not* be processed by default. However, if −− is passed as additional parameter, standard input will be read additionally to the input files.

4.2 Merging Plan Files

The merging scenario is defined in merging plan files of the following form:

```
[common signature]
predicate: pred1/arity1;
...
predicate: predN/arityN;

[belief base]
name: nameOfBeliefBase1;
mapping: "head1 :- body1."
...
mapping: "headM :- bodyM."

...

[belief base]
name: nameOfBeliefBaseK;
mapping: "head1 :- body1."
...
mapping: "headJ :- bodyJ."

[merging plan]
{
        operator: someOperatorsName;
```

```
        key1: value1;
        ...
        keyN: valueN;
        source: {
                operator: subPlanOperator;
                ...
                source: {nameOfBeliefBase1};
                source: {nameOfBeliefBase2};
        };
        source: {
                ...
        };
}
```

Essentially the file consists of 3 sections.

4.2.1 Common Signature

In statements of form

predicate: pred1/arity1;

all relevant predicates that occur in the belief bases are defined. Those predicates will be output by dlvhex after the merging plan was processed.

4.2.2 Belief Bases

Belief bases can be any data source: relational databases, XML files, etc.. The only requirement is that they are accessable from dlvhex through an appropriate external atom. Belief bases are defined by blocks of form:

```
[belief base]
name: nameOfBeliefBase1;
mapping: "head1 :- body1.";
...
mapping: "headM :- bodyM.";
args: "";
```

where the name defines a legal name for this belief base, followed by an arbitrary number of *mappings*. Mappings can essentially be arbitrary dlvhex code fragments. However, in reasonable applications they access the underlying (prorietary) belief base and map their content onto the common signature (see above). args defines the command line arguments that shall be passed to the reasoner.

Alternatively they can also be defined by

```
[belief base]
name: nameOfBeliefBase1;
source: "externalfile.hex";
```

where "externalfile.hex" is an external file containing (computation source access rules and) mapping rules. Note that *mapping* and *source* cannot be used simultanously.

If an external source shall be evaluated using DLV rather than dlvhex, the definitions dlvmapping or dlvsource can be used, i.e.:

```
[belief base]
name: nameOfDLVBeliefBase1;
mapping: "head1 :- body1.";
...
mapping: "headM :- bodyM.";
```

resp.

```
[belief base]
name: nameOfDLVBeliefBase1;
dlvsource: "externalfile.dl";
```

4.2.3 Merging Plan

The merging plan is a hierarchical structure that combines the belief bases such that only one final result survives at the end of the day. A merging plan section is of form:

```
operator: XYZ.
key1: value1;
...
keyN: valueN;
source: ...;
source: ...;
```

Such a section defines the operator to apply, the key-value pairs that shall be passed to the operator and the sub merging plans (source). A sub merging plan (after a *source* statement) can either be a belief base (denoted as {bbName};) or a *composed merging* plan (i.e. the result of a prior operator application).

4.2.4 Syntax

The following table summarizes the complete syntax of merging task files.

Lexer		
Literal	$\Rightarrow$	$\underline{-}?\ \Sigma_p\ (\underline{(}(\Sigma_c\|\Sigma_v)(\underline{,}\ \Sigma_c\|\Sigma_v)^*\underline{)})?$
PredicateName	$\Rightarrow$	$[\underline{a}-\underline{z}]\ ([\underline{a}-\underline{z}]\|[\underline{A}-\underline{Z}]\|[\underline{0}-\underline{9}])^*$
KBName	$\Rightarrow$	$([\underline{a}-\underline{z}]\|[\underline{A}-\underline{Z}])\ ([\underline{a}-\underline{z}]\|[\underline{A}-\underline{Z}]\|[\underline{0}-\underline{9}])^*$
OPName	$\Rightarrow$	$([\underline{a}-\underline{z}]\|[\underline{A}-\underline{Z}])\ ([\underline{a}-\underline{z}]\|[\underline{A}-\underline{Z}]\|[\underline{0}-\underline{9}])^*$
Variable	$\Rightarrow$	$([\underline{A}-\underline{Z}])\ ([\underline{a}-\underline{z}]\|[\underline{A}-\underline{Z}]\|[\underline{0}-\underline{9}])^*$
Number	$\Rightarrow$	$([\underline{1}-\underline{9}][\underline{0}-\underline{9}]^*)\ \|\ \underline{0}$
General ASP Grammer		
Fact	$\Rightarrow$	$\mathit{RuleHead}\ \underline{.}$
Constraint	$\Rightarrow$	$\underline{:-}\ \mathit{RuleBody}\ \underline{.}$
Query	$\Rightarrow$	$\underline{not}?\ \mathit{Literal}\ (\underline{,}\ \underline{not}?\ \mathit{Literal})^*$
RuleHead	$\Rightarrow$	$\mathit{Literal}\ (\underline{\vee}\ \mathit{Literal})^*$
RuleBody	$\Rightarrow$	*Query*
Rule	$\Rightarrow$	$\mathit{RuleHead}\ \underline{:-}\ \mathit{RuleBody}\ \underline{.}\|\mathit{Fact}\|\mathit{Constraint}$
Merging Plan Specific Grammer		
Program	$\Rightarrow$	*CommonSigDef* *Mappings* *MergingPlan*
CommonSigDef	$\Rightarrow$	$\underline{[common\ signature]}$ $\mathit{PredicateDefinition}^*$
Mappings	$\Rightarrow$	$\mathit{KnowledgeBase}^*$
MergingPlan	$\Rightarrow$	$\underline{[merging\ plan]}\ \mathit{MergingPlanNode}$
MergingPlanNode	$\Rightarrow$	$\underline{\{}$ $\underline{operator}\ \underline{:}\ \mathit{OPName}\ \underline{;}$ $(\mathit{key}\ \underline{:}\ \mathit{value}\ \underline{;})^*$ $(\underline{source}\ \underline{:}\ \mathit{MergingPlanNode}\ \underline{;})^*$ $\underline{\}}\ \|\ \mathit{KBName}$
PredicateDefinition	$\Rightarrow$	$\underline{predicate}\ \underline{:}\ \mathit{PredicateName}\ \underline{/}\ \mathit{Number}\ \underline{;}$
KnowledgeBase	$\Rightarrow$	$\underline{[knowledge\ base]}$ $\underline{name}\ \underline{:}\ \mathit{KBName}\ \underline{;}$ $(\mathit{MappingRule}^*)\|\mathit{ExternalSource}$
MappingRule	$\Rightarrow$	$\underline{mapping}\ \underline{:}\ \underline{“}\mathit{Rule}\underline{”}\underline{;}$
ExternalSource	$\Rightarrow$	$\underline{source}\ \underline{:}\ \mathit{Filename}\underline{;}$
stringliteral	$\Rightarrow$	$\underline{“}\ \{”\}^{c*}\ \underline{”}$ (where S^c is the complement of set S)
Filename	$\Rightarrow$	*stringliteral*
key	$\Rightarrow$	$\Sigma_c\|\mathit{stringliteral}$
value	$\Rightarrow$	$\Sigma_c\|\mathit{stringliteral}$

References

[1] Christoph Redl. Development of a belief merging framewerk for dlvhex. Master's thesis, Vienna University of Technology, Group for Knowledge-

based Systems, A-1040 Vienna, Karlsplatz 13, June 2010.