Introduction:

The MCS-IE-System is a plugin for DLVHex, which is able to calculate diagnosis and explanations for inconsistency in Multi-Context Systems. A Multi-Context System consists of several contexts which are connected by some bridgerules, and therefore form one system. It is assumed that each context is consistent itself, so the only inconsistency can appear because of the bridgerules. A bridgerule is a rule that says that some atom follows in a specified context, from one or more atoms from other specified contexts.

(For a proper definition of bridgerules and its form please visit:
http://www.kr.tuwien.ac.at/research/systems/mcsie/documentation.html#inputlanguage)

The aim of the MCS-IE Plugins is to find the reason for inconsistency, which means finding those bridgerules that are responsible for this inconsistency. The two constructs we are interested in are:

- diagnoses: A diagnosis defines which bridgerules have to be eliminated and which have to be added unconditionally, in order to restore consistency.
- explanations: an explanation describes which bridgerules must be present and which bridgerules must not be unconditionally added to ensure inconsistency.

So diagnoses are used to repair systems while explanations are ment for finding the source of the inconsistency.

(For a more detailed specification on diagnoses, explanations, and the whole MCS-IE project please visit:http://www.kr.tuwien.ac.at/research/systems/mcsie/literature.html)

Project Description:

Over the course of this project the MCS-IE-System was extended by an additional way of calculation - the calculation over explanations. The aim was to find the explanations immediately by looking on the contexts (so far only calculated from found diagnoses), and then calculating the diagnoses from those found explanations.

The calculation over explanations was implemented but is pretty slow. All benchmark tests have been run successfully, except benchmark8, which has eight bridgerules and would run forever. The computation of medExample, which is the scenario nearest to a real world example, takes about ten hours on an Acer Aspire (Intel Core Duo Processor T2050, 1.6Ghz, 533MHz FSB, 2 MBL2 cache, 1GB RAM).

How the program works:

Example call from directory "examples":

dlvhex plugindir=../src --ieenable --ieexplain=E --iecompoverex benchmark1/dlvctx_master.hex

Parameters:

The parameter **--iecompoverex** initiates the calculation over explanations. When this parameter is set, the plugin will calculate the ouput over explanations rather than over diagnoses when this parameter is not set.

The parameter **--ieexplain** defines what output should be generated. E stands for explanations, Em stands for minimal explanations, Dm stands for minimal diagnosis and D stands for diagnosis. Please be aware that it's not possible to calculate D from the calculation over explanations, therefore this command is restricted, and any calculation will be stopped if D is set, while calculating over explanations. Anyway, you may specify any combination of Dm, E and Em. (for example: --ieexplain=E,Em,Dm in order to calculate all these outputs, which is 3-times faster than calculating them in different runs).

For a description of other parameters (which are not needed for the calculation over explanation) for this plugin please visit:
http://www.kr.tuwien.ac.at/research/systems/mcsie/documentation.html#cmdline

Course of events:

When --iecompoverex is set, the class InputConverterExplanations.cpp will generate a DLVHex-Program which then calculates the explanations from the given contexts with the algorithm from Antonius Weinzierl, presented in the file "naive_explanation.pdf" in the same folder.

When asking for minimal diagnoses, the diagnoses are calculated from the found minimal explanations by the DLV-Program described under Explanatory Notes, and then filtered to match the minimal diagnoses.

Finally the class Outputrewriter.cpp prints out the generated explanations/diagnoses.

Refactoring & Codeextensions:

In order to extend the MCS-IE-System properly several classes have been refactored:

- The Class InputConverter has been refactored, and was therefore split in four classes. The class InputConverter.cpp remains to be the interface to DLV, but the functionality has been shifted to the following two classes: InputConverterDiagnosis.cpp, InputConverterExplanations.cpp. InputConverterDiagnosis contains just the same functionality the old InputConverter used to have, while InputConverterExplanations contains the same basic structure but adapted to the calculation over explanations. Methods used by both classes (parsing the input, generating the syntaxtree, and running through all rules), have been put in a separate class called InputConverterHelper.cpp.

- The class BridgeRule.cpp no longer contains the method named "writeProgram", because both types of calculation (over diagnosis & explanations) need different such methodes. The method "writeProgram" is now contained in the class InputConverterDiagnosis.cpp without changing the functionality. The class InputConverterExplanations.cpp contains a similar method adapted to the calculation over explanations.

- The class Outputrewriter.cpp which is in charge of handling the generated output has been extended to handle the output of the new encoding. Another method has been added: "getDiagnosis", which, with the help of an DLV program, gathers the diagnoses from the found minimal explanations.

- The class ExplanationPrintVisitor.cpp has been created. This class is used for outprinting the found explanations from the calculation over explanation.

- In order to control the two different calculations another parameter has been integrated: --iecompoverex this parameter initializes the computation over explanations by setting the new variable computeOverExplanations in Global.cpp.

Explanatory Notes:

1.) DLV-Program to calculate diagnoses from minimal explanations

```
//for every Bridgerule:
rule(bi).

//for each explanation Expi and all contained bridgerules in E1
inExpi_E1(bj).

//for each explanation Expi and all contained bridgerules in E2
inExpi_E2(bj).

//a bridgerule can be in d1 or not
d1(R) v nd1(R) :- rule(R).

//a bridgerule can be in d2 or not
d2(R) v nd2(R) :- rule(R).

//a bridgerule cannot be in d1 and d2 the same time
:- d1(R), d2(R).

//for minimal explanations Exp1 to Expn, ensure we have a hitting set in E1:
//d1OK follows if the intersection of d1 with all E1 is empty
d1OK :- inExp1_E1(R1), d1(R1) ... inExpn_E1(Rn), d1(Rn).

//for minimal explanations Exp1 to Expn, ensure we have a hitting set in E2:
//d2OK follows if the intersection of d2 with all E2 is empty
d1OK :- inExp1_E2(R1), d2(R1) ... inExpn_E2(Rn), d2(Rn).

//if neither d1OK nor d2OK is calculated, it is no Diagnosis
:- not d1OK, not d2OK.
```

2.) Remark:

In the calculation of the explanations following rules were used in the constructed DLV-Program.

```
//for every bridgerule  bi
r1(bi) :- e1(bi).
r1(bi) v nr1(bi) :- ne1(bi).
r2(bi) v nr2(bi) :- ne2(bi).
```

Rather than:

```
r1(R) :- e1(R).
r1(R) v nr1(R) :- ne1(R).
r2(R) v nr2(R) :- ne2(R).
```

This is due to the fact, that the calculation is twelve times as fast with the grounded rules than with the ungrounded. The cause to this phenomen is unknown. Probably a minor bug in DLVHex.