VU Einführung in Wissensbasierte Systeme
ASP Project: ASP Modelling & Model-based Diagnosis

Winter term 2016

October 21, 2016

You have two turn-in possibilities for this project. The procedure is as follows: your solutions will be tested with automatic test cases after the 1st turn-in deadline and tentative points will be made available in TUWEL. You may turn-in (repeatedly) until the 2nd turn-in deadline 2 weeks later (p_i are the achieved points for i-th turn-in, $i \in \{1, 2\}$), where the total points for your project are calculated as follows:

- if you deliver your project at both turn-in 1 and 2, you get the maximum of the points and the weighted mean:

$$\max\left\{p_1, \quad 0.8 \cdot p_2, \quad \frac{p_1 + 0.8 \cdot p_2}{1.8}\right\} \ ;$$

- if you only deliver your project at turn-in 1: p_1;
- if you only deliver your project at turn-in 2: $0.8 \cdot p_2$; and
- 0, otherwise.

The deadline for the first turn-in is **Friday, November 25, 23:55**. You can submit multiple times, please submit early and do not wait until the last moment. The deadline for the second turn-in is **Friday, December 9, 23:55**.

This project is divided into two parts, the first one is about modelling a computational problem in core ASP (Section 1), while the second part consists of modelling and diagnosis (Section 2). Up to **8.5 points** can be scored each. More specifically, the first part consists of two subparts, where **4 points** and **4.5 points** are achievable. The second part is divided into 3 subparts, where **2.5 points**, **3 points**, and **3 points** are achievable. Thus the maximum score amounts to 17 points. In order **to pass** this project you are required to attain at least **9 points**. **(Please note that solving only one part (i.e. completing only Section 1 or only Section 2 will NOT be sufficient for a positive grade!)** Detailed information on how to submit your project is given in Section 3. Make sure that you haved named all files and all predicates in your encodings according to the specification. Failing to comply with the naming requirements will result in point reductions.

For general questions on the assignment please consult the TISS forums. Questions that reveal (part of) your solution should be discussed privately during the tutor sessions (see the timeslots listed in TUWEL) or send an email to

ewbs-2016w@kr.tuwien.ac.at .

1 ASP Modelling (8.5 pts.)

1.1 Problem Specification of the Course Schedule Problem

The *Course Schedule Problem* is defined as follows. You are given a time table description of several courses. For each row i on the grid there is an associated number m_i between 0 and the number of rows on the grid, and for each column j on the grid there is an associated number n_j between 0 and the number of columns on the grid. Rows represent neighboring rooms, columns represent days, and each cell therefore represents the schedule of a particular room on a particular day. The courses have their own constraints which are shown on the timetable in different forms and sizes, and if a cell belongs to some course, the corresponding room on the corresponding day can be scheduled for

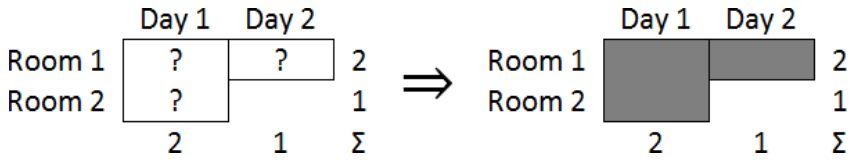

Figure 1: Example with two courses.

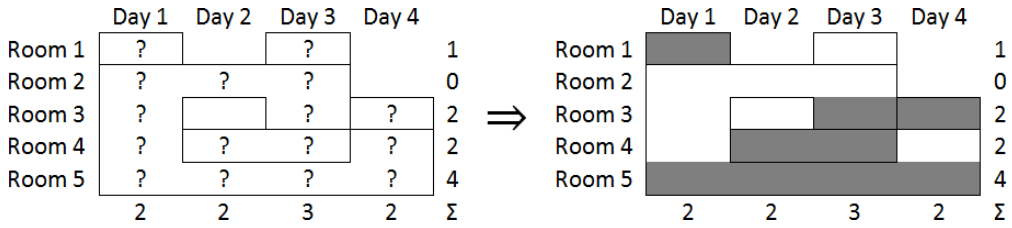

Figure 2: An example involving five courses.

that course (if a cell does not belong to any course, no schedule is possible for that cell). Courses will never overlap, but they can have gaps, i.e., a course does not necessarily have to be a set of neighbouring cells. Now a solution to this problem consists of all those cells on the grid that are scheduled, such that for all rows i on the grid, the sum of scheduled cells along i matches the given number m_i of cells for i, and for all columns j on the grid, the sum of scheduled cells along j matches the given number n_j of cells for j.

See Figure 1 for a simple example with two courses. The subfigure on the right shows the correct solution, that is, all cells belonging to courses are scheduled. Figure 2 shows a more involved example.

1.2 Problem Encoding

Your assignment is to write an ASP encoding (i.e., a logic program) for `DLV` that is using the representation of a problem instance for the course schedule problem (the layout of the grid, the possible schedule of the courses, and the constraints on the scheduled rows and columns) as a set of logic program facts. The answer sets of your program must describe all possible schedules of the courses such that the constraints for each row and each column are satisfied. That is, the answer sets of your ASP encoding correspond one-to-one to the solutions of the problem instance.

Use the following predicates for the input specification:

`nextday(X1,X2)` represents that column `X1` is followed by column `X2`.
Ex.: `nextday(1,2).`

`nextroom(Y1,Y2)` designates row `Y1` to be followed by row `Y2`.
Ex.: `nextroom(1,2).`

`sumrooms(X,S)` designates that in column `X` there are `S` scheduled cells.
Ex.: `sumrooms(1,2).`

`sumdays(Y,S)` expresses that row `Y` exhibits `S` scheduled cells.
Ex.: `sumdays(1,1).`

`course(C,X,Y)` designates that the cell at column `X` and row `Y` can be scheduled for the course labelled `C` (i.e., all cells with the same `C` together form the timeframe for a single course).
Ex.: `course(c1,1,2).`

Use the binary predicate `scheduled` as output of your encoding; `scheduled(X,Y)` then indicates that the cell located at column `X` and row `Y` is scheduled.

Note that the grid always has its origin at the absolute column-row-position $(1, 1)$, which specifies the upper-left corner of the grid. This means that cells located at (c, r) such that $r > 1$ are below of $(1, 1)$, and cells at (c, r) with $c > 1$ are to the right of the origin.

1.2.1 Writing tests

Think of some test cases before you start to encode the problem. Once you have written the actual program you will be able to check whether it works as expected. You should design at least 5 test cases and save them in separate files, name these files `course_test`n`.dl`, where n is a number ($1 \leq n \leq 5$).

As an example, consider the respective test case for grid and constraints depicted in Figure 1:

```
nextday(1,2). nextroom(1,2).
course(c1,1,1). course(c1,1,2). course(c2,2,1).
sumdays(1,2). sumdays(2,1). sumrooms(1,2). sumrooms(2,1).
```

Now you are ready to write the actual answer-set program by using the *Guess & Check* methodology. A *Guess & Check* program consists of two parts, namely a *guessing* part and a *checking* part. The first being responsible for defining the search space, i.e. generating solution candidates, whereas the latter ensures that all criteria are met and filters out inadmissible candidates. You should create two files — `course_guess.dl` and `course_check.dl` — each consisting of the corresponding part of the ASP encoding.

Important

Creating and submitting test cases is mandatory, i.e. if you do not submit your test files you will get no points for Section 1.

1.2.2 Writing the guessing program (4 pts.)

Let us have a look at the example in Figure 1, the *guessing* part of your ASP encoding should generate $2^3 = 8$ *potential solutions* (taking only those cells into account which belong to some course):

1. `{}`
2. `{scheduled(1,1)}`
3. `{scheduled(1,2)}`
4. `{scheduled(2,1)}`
5. `{scheduled(1,1), scheduled(1,2)}`
6. `{scheduled(1,1), scheduled(2,1)}`
7. `{scheduled(1,2), scheduled(2,1)}`
8. `{scheduled(1,1), scheduled(1,2), scheduled(2,1)}`

Note that we can instruct `DLV` to display only the predicates `p1`, `p2`, ... by passing `-pfilter=p1,p2,...` on the command line. E.g., when you are only interested in the extension of the `scheduled` predicate you might use `-pfilter=scheduled` and receive the output above.

After you created the guessing part of your encoding you can run it along with your test cases using `DLV`. E.g., if we store the problem instance shown in Figure 1 to a file called `course_test1.dl`, we may call

```
$ dlv course_test1.dl course_guess.dl
```

to generate all possible solution candidates.

Hint

During the construction of the program, you can limit the number of generated answer sets by passing the command line argument `-n=K` to compute only the first `K` answer sets.

1.2.3 Writing the checking program (4.5 pts.)

You might have noticed that not all of the potential solutions shown above are actual solutions, i.e., not all of them match the required number of schedules per row respectively column. For instance, the first candidate cannot be a solution since the number of schedules in each row and column is 0. On the other hand, the last candidate is indeed a solution and it corresponds to the solution shown in the right grid of Figure 1.

Thus, the computation of inadmissible solutions has to be avoided, this is what the *checking* part is for. The file `course_check.dl` should contain *constraints* which ensure that the solutions meet the given criteria. This is usually done by adding integrity constraints to your encoding, each describing non-solutions of the given problem.

Once you have completed the checking part of your encoding, run your test cases you wrote before along with both the guessing and the checking part of your ASP encoding and compare carefully whether the expected schedules will be computed. This can be done with

```
$ dlv course_test1.dl course_guess.dl course_check.dl
```

which should produce the approriate answer set(s). One of them is, e.g.,

```
{course(c1,1,1), course(c1,1,2), course(c2,2,1), nextroom(1,2),
  sumdays(1,2), sumdays(2,1), sumrooms(1,2), sumrooms(2,1),
  nextday(1,2), scheduled(1,1), scheduled(1,2), scheduled(2,1)} .
```

Hint

Again, the use `-pfilter=p1,p2,...` and `-n=K` options help to restrict the output to a managable size.

Now the answer sets of your program given a set of facts that describe the problem instance should match the solutions of this instance.

Important

Note that it is also possible that a given problem instance does not have a solution. The simplest example would be a timetable made up of only one cell $(1, 1)$ and only one course at $(1, 1)$, but the grid has conflicting numbers for the row- and column-scheduling. Can you imagine how those numbers may look like? What should be the output of `DLV` be in this situation?

2 Model-based Diagnosis (8.5 pts.)

2.1 Problem Specification

You are the boss of a counterfeit money network which can be seen in Figure 3.

Your task is to model your network as an answer-set program. Your network has three source deposits ($\mathtt{D}_1$, $\mathtt{D}_2$, $\mathtt{D}_3$) and two target deposits ($\mathtt{D}_4$, $\mathtt{D}_5$), and all of them are either used for fake or real money, represented by `0` or `1`, respectively.

Since your network should not be debunked, you do not transfer the money directly, but you instruct strawmen to do it in a stepwise manner, and each of them is supposed to know exactly his own subtask.

Strawman `c1` is instructed to act credulously, passing real money if at least one of his sources provides real money, and otherwise passing fake money.

Strawmen `w1` and `w2` are instructed to act warily, passing real money if both of their sources provide real money, and otherwise passing fake money.

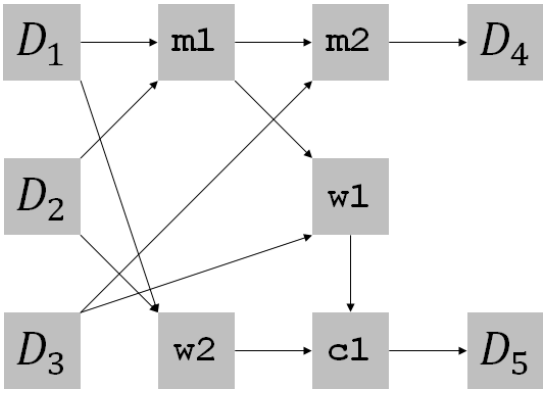

Figure 3: Your counterfeit money network.

Strawmen `m1` and `m2` are instructed to act mistrustfully, suspecting a conspiracy and thus passing fake money if both of their sources provide the same, and real money otherwise.

2.2 Problem Encoding (2.5 pts.)

In this subtask, you will model the network as an answer-set program. Specifically, use DLV to encode the system, using the following predicates:

- `m(S)`, `w(S)` and `c(S)` are representing whether a strawman is mistrustful, wary, or credulous, where `S` is the name of the strawman.
- `s1(S,M)` and `s2(S,M)` are representing the two sources of a strawman `S`, where `M` is the money he receives (either `0` or `1`).
- `t(S,M)` represents the target of a strawman `S`, where `M` is the money he passes (either `0` or `1`).

To connect the network to its source deposits, use the following predicates:

- `d1(M)` represents the source deposit D_1,
- `d2(M)` represents the source deposit D_2, and
- `d3(M)` represents the source deposit D_3.

For connecting the network with the target deposits, use the following predicates:

- `d4(M)` represents the target deposit D_4, and
- `d5(M)` represents the target deposit D_5.

2.2.1 Writing tests

Before you start encoding, design the test cases for the system, testing whether the implementation follows the specification of the components. Since the network has three source deposits and each can be either `0` or `1`, we can create $2^3 = 8$ distinct test cases. For example

```
d1(1). d2(0). d3(0).
expect_d4(1). expect_d5(0).
```

This test case represents a network with source deposits $D_1 = 1$, $D_2 = 0$, and $D_3 = 0$, expecting target deposits $D_4 = 1$ and $D_5 = 0$.

Name these files `money_test`n`.dl`, where n is a number ($1 \leq n \leq 8$).

Important

Creating and submitting test cases is mandatory, i.e. if you do not submit your test files you will get no points for Section 2.

2.2.2 Defining the network

Write a DLV program, `money.dl`, which describes the behavior of the network.

Note

Use the unary predicate `ab` (for abnormal) for the consistency-based diagnosis in DLV to specify your hypotheses. Furthermore, `ab` must only occur in combination with a default negation (that is, like `not ab(S)`). Ensure that no other predicate than `ab` occurs (default) negated.

Hint

Use the *built-in* predicates of DLV (e.g.: $X = Y + Z$). For correct usage of these arithmetic built-in predicates, it is mandatory to define an upper bound for integers. For this exercise, it is sufficient to use a range of $[0, 2]$. Therefore, start DLV with the option $-\texttt{N} = 2$.

2.2.3 Testing the network

Use the test cases to evaluate your implementation. To do so, copy the following program and store it in a file called `money_test_network.dl`.

```
UNCOMPUTED_d4(M) :- expect_d4(M), not d4(M).
UNCOMPUTED_d5(M) :- expect_d5(M), not d5(M).
UNEXPECTED_d4(M) :- d4(M), not expect_d4(M).
UNEXPECTED_d5(M) :- d5(M), not expect_d5(M).
DUPLICATED_d4(X,Y) :- d4(X), d4(Y), X < Y.
DUPLICATED_d5(X,Y) :- d5(X), d5(Y), X < Y.
```

This program detects whether a value `M` was expected but not calculated. In that case, `UNCOMPUTED_d4(M)` (respectively `UNCOMPUTED_d5(M)`) holds. On the other hand, when a different value for `M` is calculated (but not expected), then `UNEXPECTED_d4(M)` (resp. `UNEXPECTED_d5(M)`) holds. In case that there are two different values `X, Y` in target deposit D_4 (resp. D_5), the predicate `DUPLICATED_d4(X, Y)` (resp. `DUPLICATED_d5(X, Y)`) will indicate this.

For testing your model, use the following DLV call:

```
$ dlv -N=2 money.dl money_testk.dl money_test_network.dl
```

where k is the number of the test case.

Hint

- You can get rid of DLV's superfluous output by adding the `-silent` option.
- Furthermore, if at some point you are only interested in predicates of a certain kind, you can filter by adding another option like this: `-filter=d4,d5,someHelperPredicate`.

2.3 Consistency-Based Diagnosis (3 pts.)

In the second task of this exercise you will use your model and perform consistency-based diagnosis with it.

First we have to define some constraints that are required due to the nature of consistency-based diagnosis: Take the rules for `DUPLICATED_d4` and `DUPLICATED_d5` from the programs of Section 2.2.3, transform them into constraints

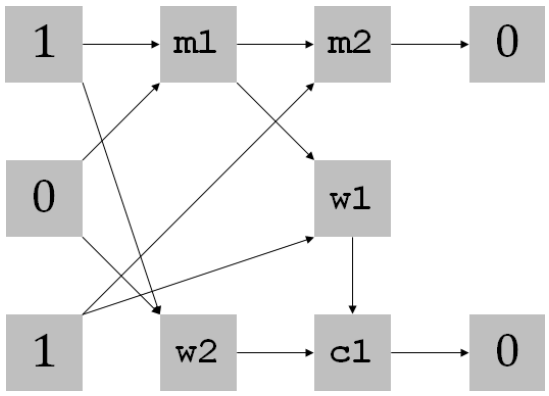

Figure 4: An observed fault.

and store them in a new file named `cbd_cstr.dl`. The reasoning behind this is that, if the network works correctly, `DUPLICATED_d4` and `DUPLICATED_d5` should never be derived, which is modelled via these constraints.

When making a consistency-based diagnosis, we check which strawmen `S` seem to perform their subtask incorrectly, represented by `ab(S)`. We select a subset of all possible facts of form `ab(S)`, such that our model and the given observations are consistent.

Therefore, create appropriate hypotheses (each of the five strawmen can be abnormal) and save them in file `cbd.hyp`.

Exercise: make a diagnosis for your test cases

Use your test cases for the complete network as observations in a diagnosis problem. Copy the files, replace the suffix `.dl` by `.obs`, and remove the prefix `expect_` from your predicate names.

Then, perform consistency-based diagnosis computing all diagnoses (DLV option `-FR`), single-fault diagnoses (DLV option `-FRsingle`), and subset-minimal diagnoses (DLV option `-FRmin`). Interpret the obtained results!

An exemplary DLV call:

```
$ dlv -N=2 -FR money.dl cbd.hyp cbd_cstr.dl money.obs
```

> **Note**
>
> This subtask does not influence the grading of your submission. Nevertheless, we recommend that you do it for a better understanding of the modelled system and the diagnosis part.

2.3.1 An initial diagnosis about an observed fault

From now on we will consider concrete observations of faulty systems. Figure 4 represents observations of our network exhibiting incorrect behavior. Represent these observations in the file `cbd_fault.obs`.

Again, perform consistency-based diagnosis computing all diagnoses (DLV option `-FR`), single-fault diagnoses (`-FRsingle`), and subset-minimal diagnoses (`-FRmin`). Interpret the obtained results!

2.3.2 Adding further measurements

Reconsider the situation described in Figure 4 and its minimal diagnoses. You have the reasonable suspicion that strawman `m1` is doing nothing wrong. Where would you do a measurement to confirm your suspicion?

Provided that `m1` actually is doing nothing wrong, give an exemplary measurement which shows — when combined with the observations of Figure 4 — that `m1` alone is not doing nothing wrong. Write the measurement to the file `cbd_fault_m1_ok.obs`.

Now, find further measurements such that the only single (and thus also minimal) solution (in combination with the observations of Figure 4) is $\{\texttt{ab(c1)}\}$ and store the observations in file `cbd_fault_c1_ab.obs`.

Finally, provide measurements such that the only minimal diagnosis (in combination with the observations of Figure 4) is $\{\texttt{ab(m1)}, \texttt{ab(m2)}\}$ and save them in file `cbd_fault_m1m2_ab.obs`.

> **Note**
>
> For all these subtasks it is not necessary to copy the measurement values of `cbd_fault.obs` to the new observation-files. Instead read both relevant observation-files when running the diagnoses with DLV.

2.4 Abductive Diagnosis (3 pts.)

For the last task we will be taking a look at another mode of diagnosing, namely abductive diagnosis.

When performing abductive diagnosis the set of hypotheses does not only consist of atoms of predicate `ab`, but of any sort of ground atom. Diagnoses are then those subsets of the set of hypotheses, such that the theory (in our case the model of the network) along with the respective subset of hypotheses entails all of the observations.

Therefore this sort of diagnosis is useful when additional domain knowledge is available. Extra information about how the system works and what might influence its behaviour can lend deeper insight into why errors occur — using abductive diagnoses we gain such insight in the form of diagnoses.

2.4.1 Additional domain knowledge

We will now add additional domain knowledge to our model. Specifically, the additional knowledge is about why strawmen might behave unexpectedly under certain circumstances. Here we consider one such abnormality:

- When strawmen are passing and receiving money, they could talk to each other. If strawmen instructed to act credulously or mistrustfully experience that over time they keep less real money than those instructed to act warily, they could start to act warily as well. This is represented by `wary(S)`, where `S` is a strawmen instructed to act credulously or mistrustfully.

As mentioned, in abductive diagnosis we are not limited to the `ab` predicate and can use arbitrary ground atoms as hypotheses. In general, however, we can then not use those ground atoms in their (default) negated form, which conflicts with our current usage of `not ab(S)` in the rules for some strawman `S`.

Therefore we will use `ok(S)` as hypotheses, where the presence of `ok(S)` means that strawman `S` is acting as instructed. Now create a new file `abd.hyp` containing the various hypotheses.

Add another file named `abd_cstr.dl` containing constraints that, for each strawman `S`, prohibit the presence of both `ok(S)` and the respective hypothesis indicating a defect.

As a next step, incorporate these changes in the modelling of the strawman. Make sure that strawmen only act as instructed when one of the two hypotheses relevant to that strawman is present. Instead of implementing all these changes in the original program, create a copy of `money.dl` and save it under the new name `abd_money.dl`.

> **Note**
>
> Naturally you should test the adapted program, there is however no requirement to submit such test cases for grading. Remember that when testing you will have to add a respective hypothesis as a ground atom.

2.4.2 Diagnosing with the adapted system

As a final preparation step we have to adapt the observations from Figure 4 to be compatible with an abductive diagnosis. Since in an abductive diagnosis every observation must follow from the theory in conjunction with some hypotheses, we will be unable to diagnose anything when observations such as external inputs are present.

Therefore split the contents of file `cbd_fault.obs` into two sets: One consisting of external input to the network (i.e. the content of the source deposits) and save it as file `abd_fault.dl`, we will simply add these facts to the theory. Save the rest (i.e. the measured content of the target deposits) as file `abd_fault.obs`.

Now we are finally ready to make an abductive diagnosis. Run DLV with the adapted observed fault:

```
$ dlv -N=2 -FD abd_money.dl \
    abd.hyp abd_cstr.dl abd_fault.dl \
    abd_fault.obs
```

You should only get three diagnoses, them being $\{$`ok(m1)`, `ok(m2)`, `ok(w1)`, `ok(w2)`, `wary(c1)`$\}$, $\{$`ok(w1)`, `ok(w2)`, `wary(m1)`, `wary(m2)`, `wary(c1)`$\}$ and $\{$`ok(w1)`, `ok(w2)`, `ok(c1)`, `wary(m1)`, `wary(m2)`$\}$. Consider why and how abductive diagnoses differ from those we found in consistency-based diagnosis.

Find measurements such that the only solutions — in combination with `abd_fault.dl` but without necessarily `abd_fault.obs` — are $\{$`ok(m2)`, `ok(w1)`, `ok(w2)`, `wary(m1)`, `wary(c1)`$\}$ and $\{$`ok(m2)`, `ok(w1)`, `ok(w2)`, `ok(c1)`, `wary(m1)`$\}$. Store the observations in file `abd_fault_m1_wary.obs`.

Find measurements such that not even a single diagnose can be found — in combination with `abd_fault.dl` but without necessarily `abd_fault.obs` — and store them in file `abd_fault_empty.obs`. Consider why this can happen.

3 Submission Information

Submit your solution by uploading a ZIP-file with the files shown below to the TUWEL system. Name your file `XXXXXXX_project.zip`, where `XXXXXXX` is replaced by your immatriculation number.

Make sure that the ZIP-file contains the following files:

- `course_test`n`.dl`, where n is an integer ($1 \leq n \leq 5$)
- `course_guess.dl`
- `course_check.dl`

- `money_test`n`.dl`, where n is an integer ($1 \leq n \leq 8$)
- `money_test_network.dl`
- `money.dl`

- `cbd.hyp`
- `cbd_cstr.dl`
- `cbd_fault.obs`
- `cbd_fault_m1_ok.obs`
- `cbd_fault_c1_ab.obs`
- `cbd_fault_m1m2_ab.obs`

- `abd.hyp`
- `abd_cstr.dl`
- `abd_money.dl`
- `abd_fault.dl`
- `abd_fault.obs`
- `abd_fault_m1_wary.obs`
- `abd_fault_empty.obs`

Make sure that your ZIP-file does not contain subdirectories, and do not forget to add files, otherwise the automatic tests will fail, and you cannot get the full score.