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

Winter term 2015

November 11, 2015

Important: This task description is valid only in the semester specified above! In other semesters, do not assume that the task description remains the same, unless the sheet is redistributed with an updated semester specification!

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 27, 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 11, 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, each but the first subpart (**2.5 points**) is worth **3 points**. 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 have 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-2015w@kr.tuwien.ac.at .

1 ASP Modelling (8.5 pts.)

1.1 Problem Specification of the Magical Matching Problem

The *Magical Matching Problem* is defined as follows. You are given a permutation T (the text) of length n and an other permutation P (the pattern) of length m with $m \leq n$. Recall that the permutation T resp. P can be viewed as a (bijective) function $\{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ resp. $\{1, 2, \ldots, m\} \to \{1, 2, \ldots, m\}$. For both bijections T and P we use its inverses denoted by T^{-1} and P^{-1}.

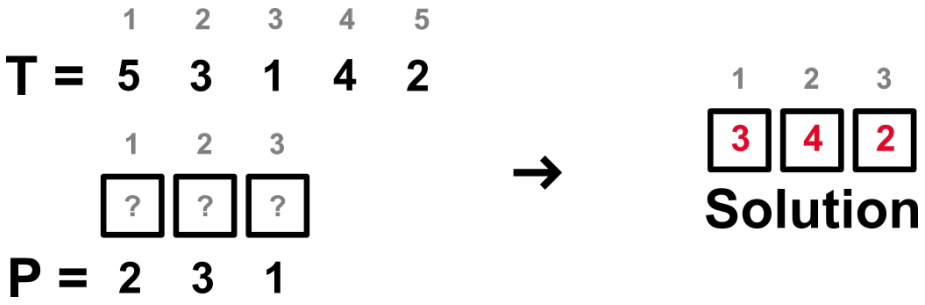

Figure 1: Example with only one solution.

Now the goal is to find a matching of P into T, where a matching is a

(1) *subsequence of* T having

(2) the *same relative order as* P.

Formally, such a matching is an injective function $\phi : \{1, 2, \ldots, m\} \to \{1, 2, \ldots, n\}$ with the following properties:

(1) *subsequence property*: $T^{-1}(\phi(i)) > T^{-1}(\phi(j))$ for i, j with $i > j$

(2) *same relative order*: $\phi(P^{-1}(i)) < \phi(P^{-1}(i+1))$ for $1 \leq i \leq m-1$

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 magical matching problem (the text, the pattern and its length) as a set of logic program facts. The answer sets of your program must describe all possible matchings fulfilling the subsequence property (using T) and having the same relative order as P according to the problem description above. 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:

`t(i,x)` represents that at position `i` of the text T the number `x` is set.
Ex.: `t(1,3)`.

`p(j,y)` designates y to be placed at position j of the pattern P.
Ex.: `p(2,1)`.

`patternlength(l)` designates `l` is equal to m, i.e. the length of P.
Ex.: `patternlength(2)`.

Use the binary predicate `solution` as output of your encoding; `solution(i,x)` then indicates that at the position $1 \leq$ `i` $\leq m$ the matching maps to number $1 \leq$ `x` $\leq n$.

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 `matching_test`k`.dl`, where k is a number ($1 \leq k \leq 5$).

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

```
t(1,5). t(2,3). t(3,1). t(4,4). t(5,2).
p(1,2). p(2,3). p(3,1).
patternlength(3).
```

Now you are ready to write the actual answers-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 — `matching_guess.dl` and `matching_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 could generate all $5^3 = 125$ *potential solutions* (you can try LATER to reduce the search space and make it even more sophisticated!):

1. `{solution(1,5), solution(2,5), solution(3,5)}`
2. `{solution(1,5), solution(2,5), solution(3,3)}`
3. `{solution(1,5), solution(2,5), solution(3,1)}`
4. ···
5. `{solution(1,5), solution(2,3), solution(3,1)}`
6. `{solution(1,5), solution(2,3), solution(3,4)}`
7. `{solution(1,5), solution(2,3), solution(3,2)}`
8. ···
9. `{solution(1,3), solution(2,4), solution(3,2)}`
10. ···

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 `solution` predicate you might use `-pfilter=solution` 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 `matching_test1.dl`, we may call

```
$ dlv matching_test1.dl matching_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.

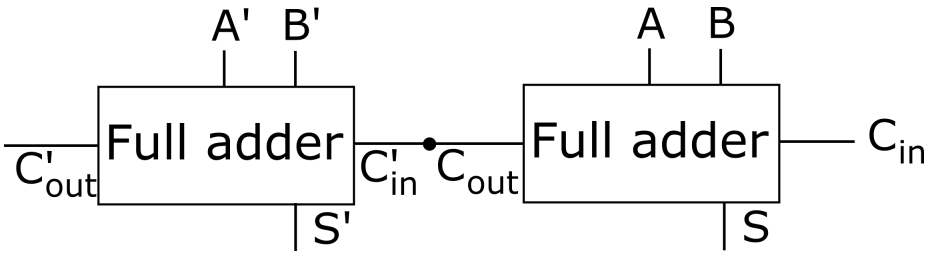

Figure 2: A 2-Bit Ripple-carry adder.

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 are subsequences of T. For instance, the first candidate cannot be a solution since one number (5 in this case) occurs more than once (contradicting constraints, both (1) and (2)). On the other hand, the last candidate is indeed a solution and it corresponds to the solution shown in Figure 1.

Thus, the computation of inadmissible solutions has to be avoided, this is what the *checking* part is for. The file `matching_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 fillings will be computed. This can be done with

```
$ dlv matching_test1.dl matching_guess.dl matching_check.dl
```

which should produce the approriate answer set(s). The only one for this example is,

```
{solution(1,3), solution(2,4), solution(3,2)} .
```

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. A simple example would be the text being the neutral (identity) permutation and P being defined as $P(i) := m + 1 - i$ for $1 \leq i \leq m$. Can you imagine how the instance 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 given the schematic structure of a *2-Bit Ripple-carry adder* seen in Figure 2. It consists of two *full adders*, which can be seen in Figure 3. Your task is to model this whole circuit (ripple-carry adder) as an answer-set program by implementing the functionality of a fulladder only once (and not two times). Such a circuit has three inputs (`A`, `B`, $\texttt{C}_{\texttt{in}}$) and two outputs (`S`, $\texttt{C}_{\texttt{out}}$), all of them can either be `0` or `1`.

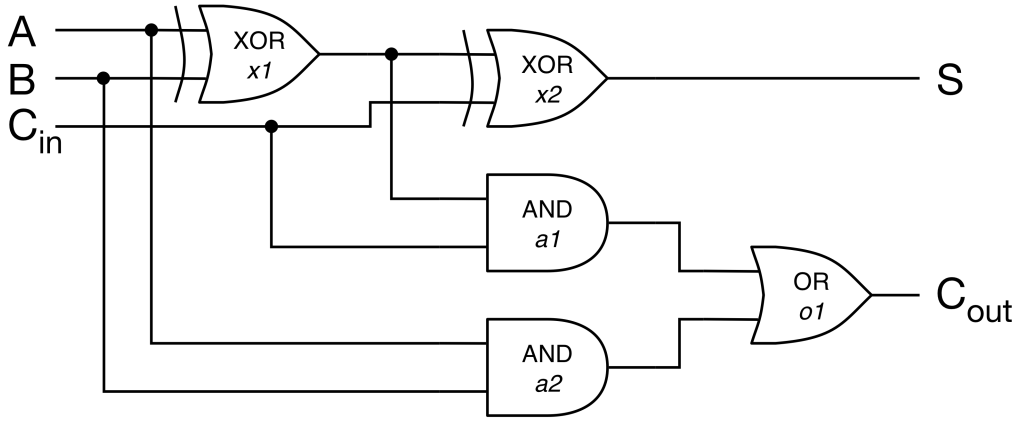

Figure 3: A full adder.

2.2 Problem Encoding (2.5 pts.)

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

- `adder(A1)` represents a full adder where `A1` is its name
- `succ(A1, A2)` represents the fact that adder `A2` succeeds adder `A1`, i.e. the carry bit C_{out} of adder `A1` is directly connected to C_{in} of adder `A2`
- `xor(G)`, `and(G)` and `or(G)` are representing the corresponding logic gates, where `G` is the name of the gate.
- `in1(A1, G, V)` and `in2(A1, G, V)` are representing the two input signals of a gate `G` for adder `A1`, where `V` is the assigned value (either `0` or `1`).
- `out(A1, G, V)` represents the output signal of a gate `G` for adder `A1`, where `V` is the assigned value (either `0` or `1`).

To connect the circuit to its input signals, use the following predicates:

- `a_in(A1, V)` represents the input bit A,
- `b_in(A1, V)` represents the input bit B, and
- `c_in(A1, V)` represents the carry bit C_{in} for adder `A1`.

For connecting the system with the output signals, use the following predicates:

- `s_out(A1, V)` represents the sum S, and
- `c_out(A1, V)` represents carry bit C_{out} for adder `A1`.

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 circuit has two input signals for each adder plus one additional carry bit C_{in} for the first adder and each can be either `0` or `1`, we can create $2^{(2*2+1)} = 32$ distinct test cases. You are required to hand in 8 of them. For example

```
a_in(ad1,1). b_in(ad1,0). c_in(ad1,0).
a_in(ad2,0). b_in(ad2,0).
expect_s_out(ad1,1). expect_c_out(ad1,0).
expect_s_out(ad2,0). expect_c_out(ad2,0).
```

This test case represents an adder with input digits $A = 1$, $B = 0$ and carry $C = 0$, expecting a sum of 1 and a carry of 0.

Name these files `adder_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 circuit

Write a DLV program, `adder.dl`, which describes the behavior of the full adder.

Note

Use the binary 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(A,G)`). 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 $-$`N` $=$ `2`.

2.2.3 Testing the system

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

```
UNCOMPUTED_s_out(A,V) :- expect_s_out(A,V), not s_out(A,V).
UNCOMPUTED_c_out(A,V) :- expect_c_out(A,V), not c_out(A,V).
UNEXPECTED_s_out(A,V) :- s_out(A,V), not expect_s_out(A,V).
UNEXPECTED_c_out(A,V) :- c_out(A,V), not expect_c_out(A,V).
DUPLICATED_s_out(A,X,Y) :- s_out(A,X), s_out(A,Y), X < Y.
DUPLICATED_c_out(A,X,Y) :- c_out(A,X), c_out(A,Y), X < Y.
```

This program detects whether a value `V` was expected for adder `A` but not calculated. In that case, `UNCOMPUTED_s_out(A,V)` (respectively `UNCOMPUTED_c_out(A,V)`) holds. On the other hand, when a different value for `V` for adder `A` is calculated (but not expected), then `UNEXPECTED_s_out(A,V)` (resp. `UNEXPECTED_c_out(A,V)`) holds. In case that there are two different values `X,Y` on output S (resp. C) for adder `A`, the predicate `DUPLICATED_s_out(A,X,Y)` (resp. `DUPLICATED_c_out(A,X,Y)`) will indicate this.

For testing your model, use the following DLV call:

```
$ dlv -N=2 adder.dl adder_testk.dl adder_test_system.dl
```

where k is the number of the test case.

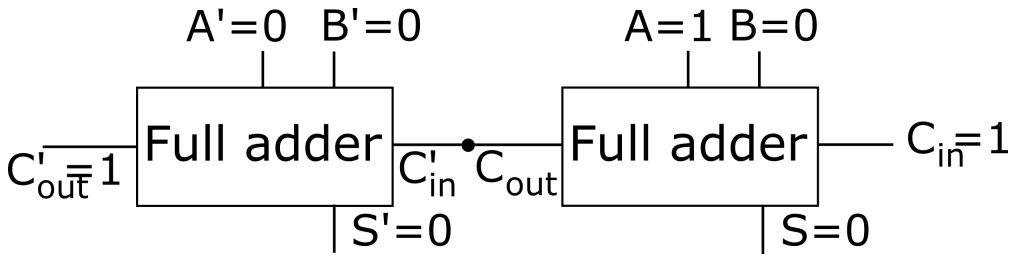

Figure 4: An observed fault.

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=s_out,c_out,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_s_out` and `DUPLICATED_c_out` from the programs of Section 2.2.3, transform them into constraints and store them in a new file named `cbd_cstr.dl`. The reasoning behind this is that, if the system works correctly, `DUPLICATED_s_out` and `DUPLICATED_c_out` should never be derived, which is modelled via these constraints.

When making a consistency-based diagnosis, we check which gates `G` of adder `A` seem to work incorrectly, represented by $\texttt{ab}(\texttt{A},\texttt{G})$. We select a subset of all possible facts of form $\texttt{ab}(\texttt{A},\texttt{G})$, such that our model and the given observations are consistent.

Therefore, create appropriate hypotheses (each of the five gates 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 system 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 adder.dl cbd.hyp cbd_cstr.dl adder_test1.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 system 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 the gate `x1` of adder `ad2` is not defective. Where would you do a measurement to confirm your suspicion?

Provided that `x1` actually is not defective, give an exemplary measurement which shows — when combined with the observations of Figure 4 — that `x1` of adder `ad2` alone is not defective. Write the measurement to the file `cbd_fault_ad2_x1_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 $\{\texttt{ab}(\texttt{ad1}, \texttt{o1})\}$ and store the observations in file `cbd_fault_ad1_o1_ab.obs`.

Finally, provide measurements such that the only minimal diagnosis (in combination with the observations of Figure 4) is $\{\texttt{ab}(\texttt{ad2}, \texttt{x2}), \texttt{ab}(\texttt{ad2}, \texttt{o1})\}$ and save them in file `cbd_fault_ad2_x2o1_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 system) 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 components might behave unexpectedly under certain circumstances. Here we consider one such abnormality:

- The components used for *AND* gates are very sensitive to electromagnetic radiation. Once one of them is exposed to electrostatic discharge or a gamma ray burst, they start to work like *NAND* gates (i.e., negated *AND* gates). This is represented by $\texttt{broken}(\texttt{A}, \texttt{G})$, where `G` is an *AND* gate (of adder `A`).

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 $\texttt{not ab}(\texttt{A}, \texttt{G})$ in the rules for some gate `G` and adder `A`.

Therefore we will use $\texttt{ok}(\texttt{A}, \texttt{G})$ as hypotheses, where the presence of $\texttt{ok}(\texttt{A}, \texttt{G})$ means that gate `G` of adder `A` is working correctly. Now create a new file `abd.hyp` containing the various hypotheses.

Add another file named `abd_cstr.dl` containing constraints that, for each gate `G` of each adder `A`, prohibit the presence of both $\texttt{ok}(\texttt{A}, \texttt{G})$ and the respective hypothesis indicating a defect.

As a next step, incorporate these changes in the modelling of the components. Make sure that components only function when one of the two hypotheses relevant to that component is present. Instead of implementing all these changes in the original program, create a copy of `adder.dl` and save it under the new name `abd_adder.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 system and save it as file `abd_fault.dl`, we will simply add these facts to the theory. Save the rest (i.e. the measured output) 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_adder.dl \
    abd.hyp abd_cstr.dl abd_fault.dl \
    abd_fault.obs
```

You should only get three diagnoses with `a1` of adder `ad1` always being broken. In addition there are

- gate `a2` of adder `ad2` OR
- gate `a1` of adder `ad2` OR
- both gates `a1` and `a2` of adder `ad2`

broken (and the remaining gates are not). Consider why and how abductive diagnoses differ from those we found in consistency-based diagnosis.

Find measurements such that the only solution — in combination with `abd_fault.dl` but without necessarily `abd_fault.obs` — is $\{$`ok(ad1,x1)`, `ok(ad1,x2)`, `ok(ad1,o1)`, `ok(ad1,a2)`, `broken(ad1,a1)`, `broken(ad2,a2)`, `ok(ad2,x1)`, `ok(ad2,x2)`, `ok(ad2,o1)`, `ok(ad2,a1)`$\}$. Use file `abd_fault_ad1_a1_ad2_a2_broken.obs` to store your observations.

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:

- `matching_test`k`.dl`, where k is an integer ($1 \leq k \leq 5$)
- `matching_guess.dl`
- `matching_check.dl`

- `adder_test`k`.dl`, where k is an integer ($1 \leq k \leq 8$)
- `adder_test_system.dl`
- `adder.dl`

- `cbd.hyp`
- `cbd_cstr.dl`
- `cbd_fault.obs`
- `cbd_fault_ad2_x1_ok.obs`
- `cbd_fault_ad1_o1_ab.obs`
- `cbd_fault_ad2_x2o1_ab.`

- `abd.hyp`
- `abd_cstr.dl`
- `abd_adder.dl`
- `abd_fault.dl`
- `abd_fault.obs`
- `abd_fault_ad1_a1_ad2_a2_broken.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.