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

Winter term 2013

November 5, 2013

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 of 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: $0.8 \cdot p_2$
- 0, otw.

The deadline for the first turn-in is **Friday, November 29, 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 13, 23:55**.

Up to **6 points** can be scored for **modelling** the components and the system (Section 1), another **9 points** are achievable for correctly solving the **diagnosis** part (Section 2 and 3). Thus the maximum score amounts to 15 points. In order **to pass** this project you are required to attain at least **8 points**. **(Please note that modelling only the components and the system (i.e. completing only Section 1) will NOT be sufficient for a positive grade!)** Detailed information on how to submit your project is given in Section 4.

For 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-2013w@kr.tuwien.ac.at.

0 General Problem Setting

In this exercise we are going to model a simple **emergency power system**, as might be used to protect a hospital from power outages or degraded performance of the external power supply (e.g. the public power grid). We start with describing the functionality of the system as an answer set program that will be run using the DLV system (see Section 1). Once the system is in place, we will be able to perform error diagnosis using the two techniques presented in the lecture: that being consistency-based diagnosis (Section 2) and abductive diagnosis (Section 3).

A schematic view of the system is shown in Figure 1. It consists of one substation `s`, which acts as the interface between the external power supply and the consumers of the power, one generator `g` (e.g. a diesel generator set) that is able to produce substantial amounts of power by itself, and a unit of batteries `b` that will ensure that power is available

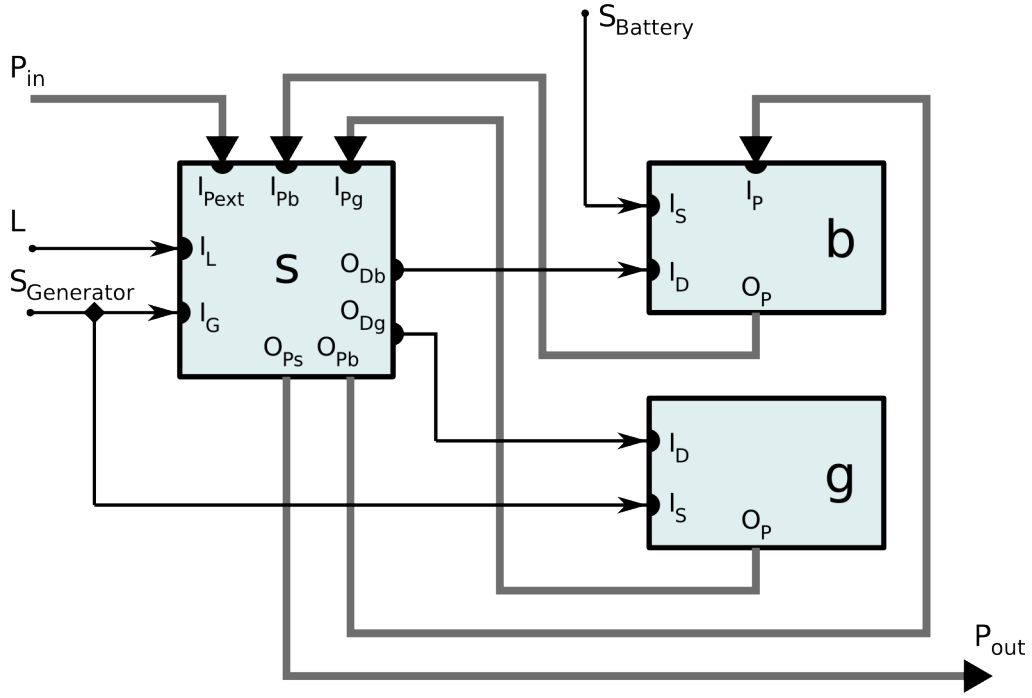

Figure 1: Model of an emergency power system.

when the generator is not running yet. As can be seen in the figure, the substation `s` also controls the amount of power produced by the generator and the battery and collects their output.

There are two types of connections between the different components: control lines and transmission lines. The former are displayed as thin, black lines while the latter are thick lines in gray. The arrow indicates the flow of information or power while the name on the ends of a line denotes the assigned input or output interface on that component. Each control line can be set to one of multiple predefined finite values and on transmission lines in our model, the amount of power can be measured in (integral amounts of) Kilowatt.

Note that, for the sake of simplicity, we do not model various aspects of such a system, such as differences in voltage, inverters et cetera.

The system has the following **input signals**:

- power from the external power supply to the substation $\texttt{P}_{\texttt{in}}$.
- the state $\texttt{S}_{\texttt{Battery}}$ of the battery `b`, which can be `full`, `half_full` or `empty`,
- the state $\texttt{S}_{\texttt{Generator}}$ of the generator `g`, which can be `not_running` or `running`,
- the current load on the system, i.e. the amount of power required by consumers, `L`.

The system has one **output signal** $\texttt{P}_{\texttt{out}}$.

The system's behaviour can be characterised as follows:

Depending on the amount of power from the external power supply $\texttt{P}_{\texttt{in}}$ and the current load on the system `L` additional power will be generated. To do so the system has two options: Using the generator component, amounts of power up to the generator's capacity $\texttt{C}_{\texttt{G}}$ can be produced. The generator only provides energy when it is running. In case of an outage, that is, when the system detects that additional power is needed, there will be a short span of

time until the generator is started up. Therefore, as a temporary source of energy, the battery component is used until the generator is running, provided that the battery is not empty. The battery also features a certain capacity $\texttt{C}_\texttt{B}$ with respect to the maximum amount of power it can provide at once. Moreover, when more external power is available than needed, the substation may redirect excess power to the battery in order to recharge it.

Note that we do not model changes of state in these components. Specifically, the running state of the generator $\texttt{S}_{\texttt{Generator}}$ and the state of charge of the battery $\texttt{S}_{\texttt{Battery}}$ are external inputs. However, the substation sends a control signal to the generator and the battery component with a value equal to the kW each should produce. The substation then accumulates the produced power along with any remaining amount of external power and forwards the power to the consumers as $\texttt{P}_{\texttt{out}}$.

The electrical current flow of the system is modelled with predicate $p(C, I, V)$, where C is the component, I is the input (or output) interface, and V is the power (in kW). For this exercise, the maximal power is 75 kW (so the values of V must be integers between 0 and 75 inclusively). For the control lines, use the predicate $c(C, I, V)$, where C is again the component, I is the input (or output) interface, and V is the corresponding value.

Note: None of the tests used for grading your submission will set the load `L` on the system to be higher than 25 kW.

The separate components of the system behave as follows (unless they are defective):

- The substation `s` receives external power $\texttt{P}_{\texttt{in}}$ on input `ipext`, the power from the generator component on input `ipg`, the power from the battery component on input `ipb`, the value of the current load `L` on input `il`, and the state of the generator $\texttt{S}_{\texttt{Generator}}$ on input `ig`. Depending on these values, the substation assigns the following values to its output interfaces:
  - Generally, when the external power supply (on interface `ipext`) provides less power than required (i.e, load `il`), then
    - if the generator's running ($\texttt{ig} = running$), the substation will request the difference in required power from the generator by signaling that difference on output `odg`,
    - if the generator isn't running, the substation will instead demand the respective amount of power from the battery via output `odb`.
  - In case the external power supply provides more power than required, the substation will redirect excess power to the battery via output `opb`.
  - In any other case the substation will not activate any of the outputs regarding the battery and the generator, i.e., $\texttt{odb} = \texttt{odg} = \texttt{opb} = 0$.
  - The output power going out of the system (`ops`) equals the amount of power received from all power inputs (`ipext`, `ipg`, `ipb`) minus the amount of power redirected to the battery (`opb`), but no more than 150% of the amount required (`il`), that is, $0 \leq \texttt{ops} \leq min(\texttt{ipext} + \texttt{ipg} + \texttt{ipb}, \lfloor \frac{3}{2}\texttt{il} \rfloor)$.[1]
- The generator `g` receives its state $\texttt{S}_{\texttt{Generator}}$ on input `is` and the value of the power demanded from the substation on input `id`. Depending on these two inputs the following values are assigned to the outputs:
  - If the generator is running ($\texttt{is} = running$) and the amount of power demanded (`id`) is within the generator's capacity ($\texttt{C}_\texttt{G}$), it will produce as much power as demanded, i.e., `op = id`.
  - If the generator is running, but the amount of power demanded is greater than the generator's capacity, then $\texttt{op} = \texttt{C}_\texttt{G}$.
  - Otherwise it produces no power, thus $\texttt{op} = 0$.
- The battery `b` receives its state $\texttt{S}_{\texttt{Battery}}$ on input `is`, the value of the power demanded from the substation on input `id`, as well as actual power from the substation on input `ip`. Depending on these inputs the following values are assigned to the outputs:

[1] There is no built-in predicate for integral division in DLV. If you're stuck modelling this, have a look at the built-ins DLV has to offer. If you run into problems with safety read up on `#int`.

- Depending on the battery's state of charge, the *internal amount of excess power* E is determined:
  - If the battery is full ($\texttt{is} = full$), then at least the amount of incoming power ($\texttt{ip} \geq 0$) will be redirected back to the substation, that is, excess power $E = \texttt{ip}$.
  - If, on the other hand, the battery is not full ($\texttt{is} \in \{\texttt{empty}, \texttt{half\_full}\}$), then the excess power $E = 0$.
- Unless the battery is empty ($\texttt{is} \neq \texttt{empty}$), it will produce at least part of the remaining amount of demanded power $R = \texttt{id} - \texttt{ip}$. Thus
  - If the remaining demanded amount of power R is less than or equal to the capacity of the battery ($\texttt{C}_\texttt{B}$), then output power $\texttt{op} = max(E, \texttt{id})$.
  - If the remaining demanded amount of power R is greater than the capacity, then $\texttt{op} = \texttt{ip} + \texttt{C}_\texttt{B}$.
- When the battery is empty, it will not produce any power on its own, but incoming power may be redirected to reach at least part of the demanded output power. In this case it will under no circumstances however output more power than demanded.

1 Modelling the System as an Answer-Set Program (6 pts.)

1.1 The components

In this part of the first task, you will model the behavior of each component as an answer-set program. Specifically, use DLV to encode the components, using the following predicates:

- $\texttt{c}(\texttt{C}, \texttt{I}, \texttt{V})$ represents control information, where
  - C is the name of the component,
  - I is an interface (input or output), and
  - V is a value.

  For example, $\texttt{c}(\texttt{g}, \texttt{id}, 5)$ represents the fact that on interface id of the component g, the value 5 is present.
- $\texttt{p}(\texttt{C}, \texttt{I}, \texttt{V})$ represents information of power transmission interfaces, where
  - C is the name of the component,
  - I is an interface (input or output), and
  - V is a value.

  For example, $\texttt{p}(\texttt{g}, \texttt{op}, 10)$ represents the fact that component g has a power output of 10 on the interface op.
- $\texttt{const}(\texttt{C}, \texttt{V})$, defines two constants, specifically:
  - $\texttt{const}(\texttt{cg}, \texttt{V})$, where V is capacity $\texttt{C}_\texttt{G}$, i.e., the peak power of the generator, and
  - $\texttt{const}(\texttt{cb}, \texttt{V})$, where V is capacity $\texttt{C}_\texttt{B}$, i.e., the peak power of the battery.
- $\texttt{substation}(\texttt{C})$, $\texttt{generator}(\texttt{C})$, and $\texttt{battery}(\texttt{C})$ define if component C is a substation, a generator, and a battery, respectively.

1.1.1 Writing tests for the components

Before you start encoding, design (at least) five test cases for each component, testing whether the implementation follows the specification of the components. For example

```
generator(g). const(cg, 20).
c(g, id, 5). c(g, is, running).
expect_c(g, id, 5). expect_c(g, is, running).
expect_p(g, op, 5).
```

This test case represents a generator with an incoming demand value of 5 on `id` and the state `running` on `is`, resulting in a power output of 5 at `op`.

Name these files X`.test`n`.dl`, where $X \in \{$`s`, `g`, `b`$\}$ (for substation, generator and battery), and n is a number ($1 \leq n \leq 5$).

1.1.2 Defining the components

Write three DLV programs, `s.dl`, `g.dl` and `b.dl`, which describe the behavior of the components.

Note:

- DO NOT use constants from the figures for the components (such as `s`, `g` and `b`) but the corresponding predicates (`substation(S)`, `generator(G)` and `battery(B)`).
- 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(C)`). 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 $[0, 75]$. Therefore, start DLV with the option $-$`N` $=$ `75`.

1.1.3 Testing the components

Use the test cases to evaluate your implementation. To do so, copy the following program and save it as `component_tester.dl`:

```
UNCOMPUTED_c(C, O, X) :- expect_c(C, O, X), not c(C, O, X).
UNCOMPUTED_p(C, O, X) :- expect_p(C, O, X), not p(C, O, X).
UNEXPECTED_c(C, O, X) :- c(C, O, X), not expect_c(C, O, X).
UNEXPECTED_p(C, O, X) :- p(C, O, X), not expect_p(C, O, X).
DUPLICATED_c(C, O, X, Y) :- c(C, O, X), c(C, O, Y), X < Y.
DUPLICATED_p(C, O, X, Y) :- p(C, O, X), p(C, O, Y), X < Y.
```

This program detects for each component `C` on output `O` (either control or transmission output) whether a value `X` was expected but not calculated. In that case, `UNCOMPUTED_c(C, O, X)` (respectively `UNCOMPUTED_p(C, O, X)`) holds. On the other hand, when a different value for `X` is calculated (but not expected), then `UNEXPECTED_c(C, O, X)` (resp. `UNEXPECTED_p(C, O, X)`) holds. In case that there are two different values `X`, `Y` on output `O`, the predicate `DUPLICATED_t(C, O, X, Y)` will indicate this.

For testing your model, use the following DLV call:

```
$ dlv -N=75 X.dl X.testn.dl component_tester.dl
```

where $X \in \{$`s`, `g`, `b`$\}$ and n 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=c,p,someHelperPredicate`.

1.2 Wiring up the system

In this second part of the task, we are now constructing the complete system from the separate components. To connect the system to its environment use the following predicates and constants:

- `p_in(pin, V)` defines the amount of power coming into the system from the external power supply,
- `c_in(load, V)` defines the load on the system, i.e. the currently required amount of power,
- `c_in(sgenerator, V)` defines the state of the generator,
- `c_in(sbattery, V)` defines the state of the battery,
- `p_out(pout, V)` defines the power leaving the system in Kilowatt,

Note: None of the tests used for grading your submission will set the load `L` on the system to be higher than 25 kW.

1.2.1 Writing test cases for the complete system

Similarly to what was done for the components, define (at least) nine test cases for the complete system which should cover as much functionality as possible. For example:

```
c_in(sgenerator, not_running). c_in(sbattery, full).
c_in(load, 20). p_in(pin, 15).
expect_p_out(pout, 20).
```

Name your files `system.test`n`.dl`, where n is an integer ($1 \leq n \leq 9$).

1.2.2 Modelling of the complete system

Now you can model the complete system. Therefore, define the different components by their names (use the constants `s`, `g` and `b`), connect the global input and output variables and link the input and output interfaces of the components with each other (according to the system specification).

Furthermore, define the following constant values for the capacity of the plants and the maximal charging capacity:

- `const(cg, 20)`, the capacity C_G of the generator component, and
- `const(cb, 5)`, the capacity C_B of the battery component.

Save these definitions in the file `connect.dl`.

1.2.3 Testing the system

Use your test cases to check the model. For this purpose, consider the following modification of the DLV program introduced above (save this program under the name `system_tester.dl`):

```
UNCOMPUTED_c(O, X) :- expect_c_out(O, X), not c_out(O, X).
UNCOMPUTED_p(O, X) :- expect_p_out(O, X), not p_out(O, X).
UNEXPECTED_c(O, X) :- c_out(O, X), not expect_c_out(O, X).
UNEXPECTED_p(O, X) :- p_out(O, X), not expect_p_out(O, X).
DUPLICATED_c(O, X, Y) :- c_out(O, X), c_out(O, Y), X < Y.
DUPLICATED_p(O, X, Y) :- p_out(O, X), p_out(O, Y), X < Y.
```

For testing your system, use the following DLV call:

```
$ dlv -N=75 s.dl g.dl b.dl connect.dl system.testn.dl system_tester.dl
```

where n is the number of the test case.

Note:

- In order to help you verify that you encoded the system correctly, especially with respect to predicate names, we provided a sample system test case along with the expected output. It is available in the TUWEL course.

2 Consistency-Based Diagnosis (5 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_c and DUPLICATED_p from the programs of Sections 1.1.3 and 1.2.3, transform them into constraints and store them in a new file named constraints.consistency_based.dl. The reasoning behind this is that, if the system works correctly, DUPLICATED_c and DUPLICATED_p should never be derived, which is modelled via these constraints.

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

Therefore, create appropriate hypotheses (each of the three components can be abnormal) and save them in file consistency_based.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=75 -FR s.dl g.dl b.dl connect.dl consistency_based.hyp \
     constraints.consistency_based.dl testcase.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.

2.1 An initial diagnosis about an observed fault

From now on we will consider concrete observations of faulty systems. Figure 2 represents observations of our system exhibiting incorrect behavior. Represent these observations in the file 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!

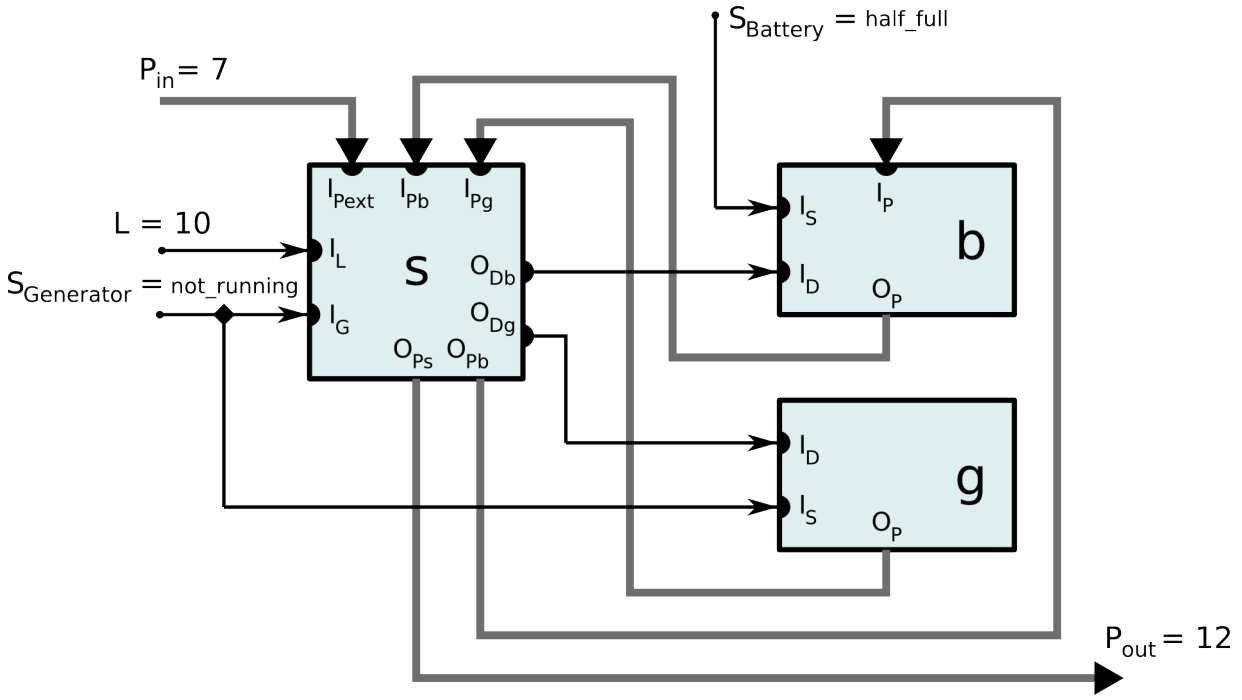

Figure 2: An observed fault.

2.2 Adding further measurements

Reconsider the situation described in Figure 2 and its minimal diagnoses. You have the reasonable suspicion that the generator g is not defective. Where would you do a measurement to confirm your suspicion?

Provided the generator actually is not defective, give an exemplary measurement which shows — when combined with the observations of Figure 2 — that g alone is not defective. Write the measurement to the file `fault.g_ok.obs`.

Now, find further measurements such that the only single (and thus also minimal) solution (in combination with the observations of Figure 2) is $\{\texttt{ab(b)}\}$ and store the observations in file `fault.b_ab.obs`.

Finally, provide measurements such that the only minimal diagnosis (in combination with the observations of Figure 2) is $\{\texttt{ab(b)}, \texttt{ab(g)}\}$ and save them in file `fault.bg_ab.obs`.

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

3 Abductive Diagnosis (4 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.

3.1 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 three such abnormalities:

- It is a known problem that the **substation** might exhibit **problems setting the demanded power for the battery component**. This is formalized as the ground atom `bad_battery_control(s)` and a change in behaviour of an affected substation. When present, the substation will set the power demand for the battery component twice as high as it normally would.
- The **generator component** is especially high-maintenance and once it has been unattended for too long it will exhibit poor performance. This is represented by `needs_maintenance(g)`; an affected generator's effective capacity will be reduced by 5kW.
- When the cells in a **battery** grow old the performance of the battery diminishes. In case `old_cells(b)` is applicable to a battery component, it will behave as if it were empty even when it is actually `half_full`.

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(C)` in the rules for some component `C`.

Therefore we will use `ok(s)`, `ok(g)` and `ok(b)` as hypotheses, where the presence of `ok(C)` means that component `C` is working correctly, that is, as specified in Section 0. Now create a new file `abductive.hyp` containing these six possible hypotheses.

Add another file named `constraints.abductive.dl` containing constraints that, for $\texttt{C} \in \{\texttt{s}, \texttt{g}, \texttt{b}\}$, prohibit the presence of both `ok(C)` 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 component programs, create copies of files `s.dl`, `g.dl` and `b.dl`, saved under the new names `s.abductive.dl`, `g.abductive.dl` and `b.abductive.dl`.

Note: Naturally you should test the adapted component programs, 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.

3.2 Diagnosing with the adapted system

As a final preparation step we have to adapt the observations from Figure 2 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 `fault.obs` into two sets: One consisting of external input to system and save it as file `fault.abductive.dl`, we will simply add these facts to the theory. Save the rest (i.e. the measured output) as file `fault.abductive.obs`.

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

```
$ dlv -N=75 -FD s.abductive.dl g.abductive.dl b.abductive.dl connect.dl \
    abductive.hyp constraints.abductive.dl fault.abductive.dl \
    fault.abductive.obs
```

You should already only get a single diagnose, that being $\{\texttt{ok(b)}, \texttt{ok(g)}, \texttt{bad\_battery\_control(s)}\}$. Consider why and how abductive diagnoses differ from those we found in consistency-based diagnosis.

Find measurements such that the only (and thus also minimal) solution — in combination with `fault.abductive.dl` but without necessarily `fault.abductive.obs` — is $\{\texttt{ok(s)}, \texttt{ok(g)}, \texttt{old\_cells(b)}\}$ and store the observations in file `fault.abductive.b_old.obs`.

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

4 Submission Information

Before you upload your solution, check your files with `sample_test.dl` (see TUWEL), and verify that you did not mistype predicate names or constant symbols. If you upload a file with typos, automatic tests will fail and you will get fewer points.

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:

- X`.test`n`.dl`, where $X \in \{$`s`, `g`, `b`$\}$ and n is an integer ($1 \leq n \leq 5$),
- `s.dl`, `g.dl` and `b.dl`,
- `system.test`n`.dl`, where n is an integer ($1 \leq n \leq 9$),
- `connect.dl`,
- `component_tester.dl`,
- `system_tester.dl`,
- `consistency_based.hyp`,
- `constraints.consistency_based.hyp`,
- `fault.obs`,
- `fault.g_ok.obs`,
- `fault.b_ab.obs`,
- `fault.bg_ab.obs`,
- `s.abductive.dl`, `g.abductive.dl` and `b.abductive.dl`,
- `abductive.hyp`,
- `constraints.abductive.hyp`,
- `fault.abductive.dl`,
- `fault.abductive.obs`,
- `fault.abductive.b_old.obs` and
- `fault.abductive.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.