# Wizards of Oz
# Description of the 2009 DXC Entry

**Alban Grastien** [*,**] **Priscilla Kan-John** [*,**]

[*] *NICTA, Canberra Research Lab, Canberra, Australia*
[**] *The Research School of Information Sciences and Engineering (RSISE), The Australian National University, Canberra, Australia*

## 1. INTRODUCTION

This paper describes `Wizards of Oz`, our entry to Tiers 1 and 2 of the industrial track of DX Competition. We present the algorithm along with the hypotheses made, the known limitations, and the possible improvements.

We first describe the general principle. The modeling is then presented. The diagnosis algorithm is described in more details. We then present technical points. Finally, we discuss the results of the competition and draw conclusions on the results.

## 2. GENERAL PRINCIPLE

`Wizards of Oz` is based on incremental solving of static diagnosis problems. We judged that the dynamic part of the behaviour can be abstracted for the following reasons:

- The observations are state-based and not event-based; there is actually no event (see Limitation 1).
- The observations of the system can be entirely defined by the current state of the system (fault variables and control variables), with the exception of the temperature sensors where the *derivation* is defined by the state of the system (see 5.3).
- The state at a time does not depend on the state at previous time (except that the faults are considered permanent, see Hypothesis 1). In particular, circuit breakers do not trip because of overcurrent.

*Limitation 1.* Note that the algorithm does not automatically detect when the value of a sensor drops, for instance if the sensor IT140 decreases from 20 to 18 A.

The diagnoser monitors the control actions. Whenever the algorithm considers that the state of the system is stable, a diagnosis procedure is launched. A diagnosis candidate is a set of faulty components together with their faults (all other components considered normal).

*Hypothesis 1.* The faults are permanent.

Due to Hypothesis 1, a diagnosis candidate can only grow. Provided that the set of candidates at every time is complete, updating this set consists in adding faults to the current candidates. The diagnoser starts from the set of candidates at the previous time step, and runs Algorithm 1. This procedure is very similar to Reiter's `diagnose` (Reiter [1987]).

The generation of the diagnosis candidates is illustrated Figure 1. Starting from a single empty diagnosis candidate

**Algorithm 1** Basic diagnosis procedure of `Wizards of Oz`

```
for each diagnosis candidate do
  if the simulation of the system is consistent with the
  observations then
    The candidate is confirmed
  else
    A conflict is extracted.
    New diagnosis candidates are generated from the
    current candidate and the conflict.
  end if
end for
```

inconsistent with the observations, a conflict with three faults is found: $\{f_1, f_2, f_3\}$. This conflict is expanded generating the three candidates in the second layer. The first two candidates are consistent with the observations; the last candidate generates the conflict $\{f_4, f_5\}$. Finally, four candidates are selected.

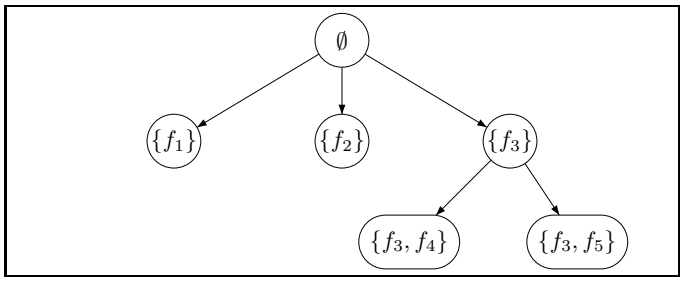


Fig. 1. Generation of the diagnosis candidates at time $t$

When new observations are available at time $t' > t$, the diagnosis is updated starting from the previous set of candidates. This is illustrated Figure 2. The diagnosis candidates $\{f_1\}$ and $\{f_2\}$ are no longer consistent with the observations. The first candidate can be extended with fault $f_6$ or $f_7$. However, the second candidate cannot be extended. The diagnosis ends with four candidates, each of which contains two faults.

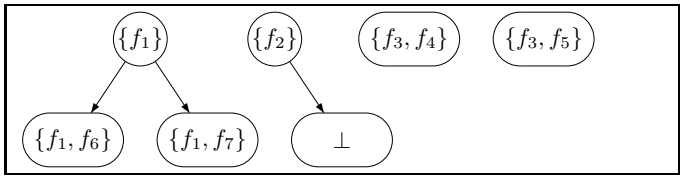


Fig. 2. Generation of the diagnosis candidates at time $t'$

## 3. MODELING

The model of the system is complete, which means that its behaviour is completely specified even in case of faults. This implies that we considered that all possible behaviours were represented in the data samples provided. In particular, we are not sure how the fans behave in case of over- or underspeed.

*Limitation 2.* `Wizards of Oz` requires a complete model that includes the faulty behaviour.

*Limitation 3.* The model may be inaccurate regarding a number of faulty behaviours.

The system is modeled by a set of variables and a set of rules. We distinguish four types of variables:

- control variables: their value is perfectly known.
- fault variables: they model the faults.
- intermediate variables: they model internal values like voltage and intensity (amperage).
- observable variables: they model the output value of the sensors.

The diagnosis engine also include a fifth type of variables: the so-called *undeterministic* variables, which are not necessary for the Adapt system. The purpose of these variables is to allow the modeling of non-deterministic behaviour: depending on the value of these variables, the system will generate different behaviours. The intermediate variables are not essential as the observable variables can be directly assigned from the control and fault variables, but they are useful for compact modeling.

The model is represented by a set of *rules* that indicates the value of the variables depending on the values of other variables. The dependency between the value of the variables is represented Figure 3.

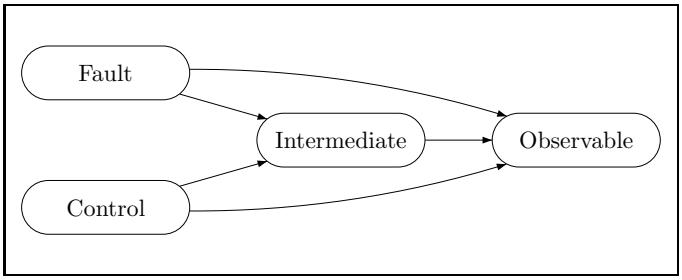


Fig. 3. Dependencies between variables

It can be tricky to model electrical circuits because there are default rules. Typically, when the circuit contains cycles and the electricity can come from any part, the voltage of a line is null iff no path can be found from a source to this line. However, in view of the answers from the DXC, we decided to formulate the following hypothesis.

*Hypothesis 2.* In any line, the current can flow in only one direction. [1]

The current basically flows from the left to the right on the graphical representation of the system provided by the DXC committee.

[1] The correctness of this hypothesis was confirmed by the DXC committee.

An example of rules is given Table 1 for the relay Figure 4. The figure is based on the graphical representation provided by the DXC committee. Two wires 1 and 2 are represented. Each wire $k$ is modeled by two intermediate variables $u_k$ (voltage) and $i_k$ (intensity of the current). The relay itself is modeled by

- one control variable $c$ that represents what the state of the relay should be if there is no fault,
- one fault variable $f$ that indicates whether the relay is working properly or is stuck, and
- one intermediate variable $r$ that represents the actual state of the relay (open or closed).

Finally, the sensor on the relay is represented by the fault variable $s$ indicating whether the sensor is working properly or is stuck and the observable variable $o$ that represents what observation the sensor generates. The rules are easy to understand from the description of the variables.

Note that these rules generate a DAG of assignments. This DAG is dynamically built; for instance the value of $u_2$ (rules 4 to 6) may depend on the value of $r$, the value of $u_1$ or both. The DAG is later used to generate conflicts (see Section 4). It is preferable to have a smaller DAG whenever possible; typically, if Rules 4 and 5 can be applied, it is preferable to use Rule 4 as $u_1$ may depend on a big number of variables.

*Improvement 1.* The rules should be given a preference ordering so that preferable rules can be applied when they overlap.

In practice, real-valued variables are not associated a unique value but an interval (often reduced to a single value) which allows to some extend to model non-deterministic behaviours.

| precondition | $\rightarrow$ | effect |
|---|---|---|
| $f = StuckOpen$ | $\rightarrow$ | $r := Open$ |
| $f = StuckClose$ | $\rightarrow$ | $r := Close$ |
| $f = Normal$ | $\rightarrow$ | $r := c$ |
| $r = Open$ | $\rightarrow$ | $u_2 := 0$ |
| $u_1 = 0$ | $\rightarrow$ | $u_2 := 0$ |
| $r = Close$ | $\rightarrow$ | $u_2 := u_1$ |
| $r = Open$ | $\rightarrow$ | $i_1 := 0$ |
| $r = Close$ | $\rightarrow$ | $i_2 := i_1$ |
| $s = StuckOpen$ | $\rightarrow$ | $o := Open$ |
| $s = StuckClose$ | $\rightarrow$ | $o := Close$ |
| $s = Normal$ | $\rightarrow$ | $o := r$ |

Table 1. Set of rules for the relay Figure 4

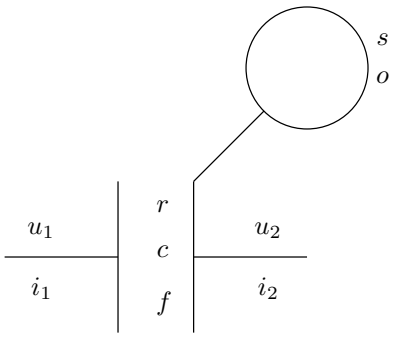


Fig. 4. Example of a relay

The model relies on an additional simplifying hypothesis.

*Hypothesis 3.* A line cannot be fed by two lines.

Because of time constraints, we did not write the model in an external file. Figure 6 (at the end of the paper) gives an example of the Java code that was written to represent the model. In this example, the method `Wire.createWire(?model,?name)` generates two variables `u_?name` and `i_?name` that model the tension and intensity of the electricity through the wire. The method `createRelay(?w1,?w2,?rname,?sname)` defines a relay and its sensor between the two wires `w1` and `w2` as explained previously; in practice, calling this method generates five variables and eleven rules that are stored in the model. The Figure 6 therefore shows the topology (list of components and connections) of the system. This topology is transformed in a set of variables and rules that are explicitely enumerated before being sent to the diagnosis engine.

## 4. DIAGNOSIS ALGORITHM

The higher level of the diagnosis algorithm monitors the control commands and observations before calling the lower level of the procedure on a snapshot of observations. As indicated before, the lower level is performed only on *stable* observations. The definition of stability is tricky, especially because of noise in the real-valued sensors. Wrongly considering a set of observations as unstable is not worrying as long as it does not happen too often. We chose the following compromise:

- Whenever a command is issued, wait for six seconds.
- Whenever the observations are significantly different from the previous observations, wait for two and half seconds.

*Limitation 4.* This does not scale-up well to very large systems where there might always be significant differences somewhere.

Significant difference between real values is defined as follows: $a$ is significantly greater than $b$ iff $0.9 \times a > b + 0.2$. Special cases are defined for some particular components. For instance, the voltage sensors $E265$ and $E267$ sometimes produce values around $0.3V$ even when there is no voltage.

*Improvement 2.* The definition of significant difference should be refined for each component.

When the lower procedure is run, the system is simulated as presented in Algorithm 1. If an observed value and its corresponding simulated value significantly differ, then a conflict is produced: the conflict is defined as the set of variables that induced the value of the simulated observable according to the DAG presented in the previous section. If several conflicts are found, the one with minimal cardinality is used to generate new diagnosis candidates.

Figure 5 presents an example of a DAG representing how the variables were assigned. For instance, in this figure the intermediate variable $i_3$ was assigned to its current value because of the values of $c_1$ and $f_3$. Consider that the simulated value of $o_2$ is different from its observed value.

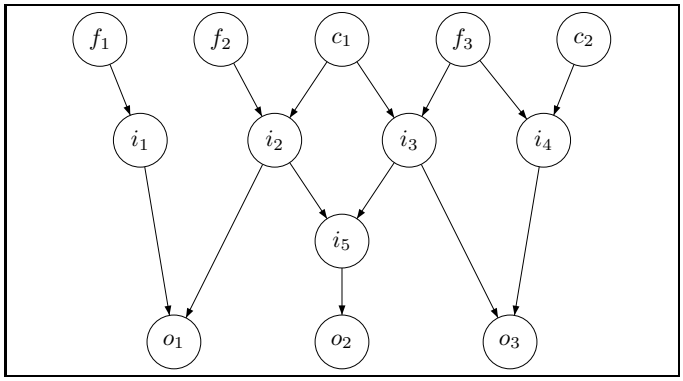


Fig. 5. Example of propagation

Then, since the value of $o_2$ depends only on $f_2$, $c_1$ and $f_3$, one of these three variables must be incorrectly set in the simulation. Since the control value $c_1$ is perfectly known, the incorrect variable must be one of the two others: the conflict is $\{f_2, f_3\}$.

In order to reduce the number of candidates, the maximum size of a diagnosis candidate is arbitrary set to the size of the minimum cardinality diagnosis plus two. So, if a diagnosis with two faults is found, the diagnoses with strictly more than four faults are disregarded.

Note that if a diagnosis candidate matches the observations, its supersets are discarded.

## 5. MISCELLANEOUS

We now present a number of details.

### *5.1 Probability Estimation*

When several diagnosis candidates are found, a weight is computed for each candidate. A weight $w_f$ is associated with each fault $f$ (usually $w_f = 1$). The candidate weight of $\Delta$ is then computed by the following formula:

$$\frac{1}{(0.1 + \Sigma_{f \in \Delta} w_f)^4}.$$

This formula was chosen in order to give a large weight for the diagnosis candidates that have few faults or faults with low weight. Note that a large fault weight generates a small diagnosis weight. The values are then normalised so that the sum of the diagnosis weight is one.

For instance, consider a diagnosis of three candidates $\{\{f_1\}, \{f_2\}, \{f_3, f_4\}\}$ with the following weights: $w_{f_1} = 0.5$; $w_{f_2} = 1$; $w_{f_3} = 1$; $w_{f_4} = 1$. The weight of the candidates are: $w_{\Delta_1} = 7.716$; $w_{\Delta_2} = 0.683$; $w_{\Delta_3} = 0.051$. When normalised, the values are: $w_{\Delta_1} = 0.91$; $w_{\Delta_2} = 0.08$; $w_{\Delta_3} \simeq 0.01$. In this case, the most likely diagnosis is $\Delta_1$ since it has the largest weight.

*Improvement 3.* To remove the confusion that associates a high weight fault with a low weight diagnosis, the probability estimation scheme should be modified.

### *5.2 Stuck and Offset Variables*

The variables representing offset and stuck values for real-valued variables are treated in a slightly different manner.

These variables are associated with an observable variable so that their value can be easily determined when a conflict is found. For instance, if a stuck variable $s$ is associated with the observable variable $o$, if the simulated value of $o$ is 2 and its observed value is 0, then the diagnosis candidate including a stuck fault on the sensor will associate the value 0 to $s$.

Since the diagnosis is tested at every snapshot, a diagnosis candidate incorrectly stating that a sensor is stuck is quickly dispelled (actually, the hypotheses containing a stuck sensor are tested even when the diagnosis procedure is not run).

When a diagnosis candidate containing a stuck sensor is consistent with the observations, there is also often a diagnosis candidate containing an offset sensor instead of the stuck sensor. A stuck sensor can be diagnosed as follows: if the sensor returns the same real value a number of times, it is probably stuck and the offset hypothesis can be disregarded. To ensure completeness, we keep both hypotheses. Now, if the stuck fault hypothesis is consistent with the observations, it is the most likely hypothesis. Therefore, we give a very small weight to the stuck fault so that the weight of its diagnosis is large. Conversely, an offset fault is often easy to identify because there are very few explanations apart from the offset; for instance, if a voltage sensor is inconsistent with all other sensors on the same line (downstream and upstream), the only single fault diagnosis is an offset fault. Thus, the correct diagnosis candidate will compete with candidates with several faults, and we can safely give a large weight to offset faults (here 2).

To summarize:

- If a sensor is stuck, then several diagnosis candidates will be generated: one stating that the sensor is stuck (with high probability), one stating that the sensor is offset (with low probability), plus others with several faults (therefore even lower probability).
- If a sensor is offset, then several diagnosis candidates will be generated: one stating that the sensor is stuck which will soon be dispelled, one stating that the sensor is offset (with low probability), plus others with several faults (therefore even lower probability). Since all other diagnoses have lower probability, the correct diagnosis will remain as the most likely.

We see one limit and one improvement that can be made to this approach.

*Limitation 5.* When the sensor is stuck but returns values that are consistent with the observations, the fault is not detected.

We do not consider it is a real limitation in the sense that, when the stuck sensor returns values within acceptable range, there is no reason to suspect a fault.

*Improvement 4.* The weight of the stuck fault could be modified dynamically according to the number of consecutive time steps with the same value.

This improvement would avoid giving a high probability to an incorrect diagnosis candidate just before dispelling the candidate.

### *5.3 Evolution Variables*

For some variables, what is relevant is not their current values but how their values evolve. Typically, the value of a temperature sensor on a bulb increases when the light is on, and decreases otherwise. Furthermore, the value of the sensor is bounded, which means that at some point, the temperature stops increasing.

These variables are not directly modeled as real values. Instead, they take value within this discrete set:

`TOP_OR_UP` the temperature is going up or is already at the maximum;
`BOT_OR_DOWN` the temperature is going down or is already at the minimum;
`STUCK` the value did not change;
`UNKNOWN` the value is slightly going up (but it is impossible to choose between `TOP_OR_UP` and `BOT_OR_DOWN`);
`OFFSET` the value changed dramatically.

The evolution is computed on three time steps ($1.5s$) to remove the noise. The simulation returns a value within this set, excluding `UNKNOWN`. The consistency between an observed value and a simulated value is defined by the Table 2.

| $s\downarrow$ $o\rightarrow$ | T | B | S | U | O |
|---|---|---|---|---|---|
| T | $\top$ | $\bot$ | $\bot$ | $\top$ | $\bot$ |
| B | $\bot$ | $\top$ | $\top$ | $\top$ | $\bot$ |
| S | $\bot$ | $\bot$ | $\top$ | $\bot$ | $\bot$ |
| O | $\top$ | $\top$ | $\top$ | $\top$ | $\top$ |

Table 2. Equivalence between simulation (s) and observation (o)

For instance, if the model simulates a stuck value, then the observed value of the sensor should be stuck. If the model simulates that the temperature should go down, then it may also be in an unknown state or appears stuck.

*Limitation 6.* With this modeling, a diagnosis candidate including an offset of the sensor no longer considers this sensor (line of $\top$ in Table 2).

We consider that this limitation is not too important, since if a fault different from an offset fault occurred, then other sensors will probably see this fault. Therefore, an (incorrect) diagnosis candidate including an offset on a temperature sensor will also contain other faults and therefore have a small weight.

## 6. DX COMPETITION RESULTS

The results of `Wizards of Oz` during the DX Competition are uneven. Since it was the first edition of the competition, it was difficult to know precisely which part of the solver required improvement.

We now examine the weaknesses of `Wizards of Oz` identified during the competition.

### *6.1 Detection time*

In `Wizards of Oz`, the detection of a fault is notified when the first diagnosis is performed, which is a rather long time since we wait for stable observations. In a future version, `Wizards of Oz` should either:

- announce a fault detection as soon as the empty diagnosis is no longer consistent with the observations – the isolation can be performed later (This requires experiments to make sure this method is tolerant to noise); or
- include a dedicated fault detection algorithm.

We believe this improvement could generate easily much better results.

### *6.2 Identification time*

The high value for the identification time is a direct consequence of the stability required by the diagnosis algorithm. In order to improve this dimension, it will be required to include temporal information to the model. A possibility is to model some variables as evolution variable but this framework is not ideal for monitoring some variables (intensity for instance).

Instead, we currently propose to introduce a notion of duration in the values of variable. With each variable will be associated a time tag indicating for how long this variable has had this value. This framework will enable to determine eg. that a pump is being stopped, from which it is possible to estimate accurately its output and the input intensity. It will no longer be required to wait until the observations are stable.

When a diagnosis is simulated, it is the continuation of a diagnosis that was consistent with the previous observations (and possibly no longer consistent with the current diagnosis). The values of the variables in a similation can be compared to the values of the same variables in the previous simulation in order to determine when the value of the variable changed. This is how the time tag is updated.

The concrete implementation would require more formal work, in particular to detect side-effects. Other possible light introduction of temporality may be developped.

### *6.3 False negative*

The implications of these bad results are unknown and require more analyses.

## 7. CONCLUSION

We presented `Wizards of Oz`, a diagnosis algorithm that is based on a static approach. The algorithm waits for the observations to be stabilized before running a diagnosis procedure based on Reiter's approach.

We identified a number of limitations of this approach. We also proposed several modifications that could improve the performances of this algorithm.

## 8. ACKNOWLEDGEMENT

This research was supported by NICTA. NICTA is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian National Research Council.

## REFERENCES

R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence Journal (AIJ)*, 32(1):57–95, 1987.

```
// Wire 235
Wire w235 = Wire.createControlledWire(_result, "E235", 24);
_result.observe(w235.u, "OBS_E235", "Offset_E235", "Stuck_E235",1);

// Wire 240
Wire w240 = Wire.createWire(_result, "W240");
_result.observe(w240.u, "OBS_E240", "Offset_E240", "Stuck_E240",1);
_result.observe(w240.i, "OBS_IT240", "Offset_IT240", "Stuck_IT240");
adapt.createCB(w235, w240, "CB236", "ISH236");

// wire WI 242
Wire w242 = Wire.createWire(_result, "W242");
_result.observe(w242.u, "OBS_E242", "Offset_E242", "Stuck_E242");
adapt.createRelay(w240, w242, "EY244", "ESH244A");

// wire WI 261
Wire w261 = Wire.createWire(_result, "W261");
_result.observe(w261.u, "OBS_E261", "Offset_E261", "Stuck_E261");
_result.observe(w261.i, "OBS_IT261", "Offset_IT261", "Stuck_IT261");
adapt.createRelay(w242, w261, "EY260", "ESH260A");

// wire WI 263
Wire w263 = Wire.createWire(_result, "W263");
adapt.createCB(w261, w263, "CB262", "ISH262");

// wire WI 265
Wire w265 = Wire.createWire(_result, "W265");
_result.observe(w265.u, "OBS_E265", "Offset_E265", "Stuck_E265");
adapt.createFrequencyTransmitter(w265, "ST265");
adapt.createInverter(w263, w265, "INV2");

// wire WI 267
Wire w267 = Wire.createWire(_result, "W267");
_result.observe(w267.u, "OBS_E267", "Offset_E267", "Stuck_E267");
_result.observe(w267.i, "OBS_IT267", "Offset_IT267", "Stuck_IT267");
adapt.createCB(w265, w267, "CB266", "ISH266");

// wire WI 276
Wire w276 = Wire.createWire(_result, "W276");
adapt.createRelay(w267, w276, "EY275", "ESH275");

adapt.createFan(w276, "FAN416", "ST516", 120, 900);

adapt.createTemperatureSensor("TE228", 69, 75);
adapt.createTemperatureSensor("TE229", 69, 75);
```

Fig. 6. Example of Java code for the model