Abductive reasoning in Prolog and CHR

A short introduction for the KIIS course, autumn 2005

Henning Christiansen
Roskilde University, Computer Science Dept.
Denmark

1 Deduction, abduction, and induction in logic programming

The philosopher C.S.Peirce (1839–1914) is considered a pioneer in the understanding of human reasoning, especially in the specific context of scientific discovery. His work is often cited in computer science literature but probably only few computer scientists have read Peirce's original work. We recommend [7] as overview of Peirce's influence seen from the perspective of computer science.

Peirce postulated three principles as *the* fundamental ones:

- **Deduction**, reasoning within the knowledge we have already, i.e., from those facts we know and those rules and regularities of the world that we are familiar with. E.g., reasoning from causes to effects:
  "If you make a fire here, you will to burn down the house."
- **Induction**, finding general rules from the regularities that we have experienced in the facts that we know; these rules can be used later for prediction:
  "Every time I made a fire in my living room, the house burnt down, aha, ... the next time I make a fire in my living room, the house will burn down too".
- **Abduction**, reasoning from observed results to the basic facts from which they follow, quite often it means from an observed effect to produce a qualified guess for a possible cause:
  "The house burnt down, perhaps my cousin has made a fire in the living room again."

In fact, Peirce had alternative theories and definitions of abduction and induction; we have adopted the so-called syllogistic version, cf. [7]. We can replicate the three in logic programming terms:

- A Prolog system is a purely deductive engine. It takes a program of rules and facts, and it can calculate or check the logical consequences of that program.
- Induction is difficult; methods for so-called inductive logic programming (ILP) have been developed, and by means of a lot of statistics and other complicated machinery, it synthesizes rules from collections of “facts” and “observations”. We refer to [2][1] for an overview of different applications. Inductive logic programming has been successfully applied for molecular biology concerned with protein molecule shapes and human genealogy. See [10] for an in-depth treatment of ILP methods.
- Abductive logic programming; roughly means from a claim of goal that is required to be true (i.e., being a consequence of the program), to extend to program with facts so that the goal becomes true. See [9] for an overview. Abduction has many applications; we may mention planning (e.g., goal is “successful project ended” and the facts to be derived are the detailed steps of a plan to achieve that goal), diagnosis (goal is observed symptoms, the facts to be derived comprise the diagnosis, i.e., which specific components of the organism or technical system that malfunction). An important area for abduction is language processing, especially discourse analysis (the discourse represents the observations, the facts to be derived constitute an interpretation of that discourse). We will look into some of these in more detail below and give references.

However, we should be aware that while deduction is a logically sound way of reasoning, this is generally not the case for abduction and induction. We will make a simple analysis for abduction. Assume a logical knowledge base $\{a \to c, b \to c\}$ where the arrow means logical implication. If we know c, an abductive argument may propose that a is the case, however, this is not necessarily true as it might that b is the case and not a, or it could even be the case that none of a and b are the case, and that there is another and unknown explanation for c. Abduction is often described as reasoning to the best explanation. i.e., best with respect to the knowledge we have available.

2 A definition of abductive logic programming

(This section is adapted from [3]). An abductive logic program [9] is usually specified as a triplet $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$ where $\mathcal{P}$ is a logic program, $\mathcal{A}$ a set of *abducible* predicates that do not occur in the head of any clause of $\mathcal{P}$, and $\mathcal{IC}$

[1]A bit old; if you are interested, you should search for more recent overview papers and consult proceedings of the recent ILP conferences; see http://www.informatik.uni-trier.de/~ley/db/conf/ilp/index.html.

a set of integrity constraints assumed to be consistent. Assume additionally that $\mathcal{P}$ and $\mathcal{IC}$ can refer to a set of *built-in* predicates that have a fixed meaning identified as a theory $\mathcal{B}$; a predicate in $\mathcal{P}$ that is neither abducible nor built-in is called *defined.* A typical built-in predicate is the `dif/2` predicate of SICStus Prolog [11]; `dif(X,Y)` means that `X` and `Y` must be different. We assume for simplicity in the following that $\mathcal{IC}$ refers to abducible and built-in predicates only.

Given an abductive logic program $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$, we define for pairs of sets of abducibles and built-in atoms $\langle A, B \rangle$, a *consistent ground instance* to be a common ground instance[2] $\langle A', B' \rangle$ of $\langle A, B \rangle$ so that[3]

- $\mathcal{B} \models B'$ (the instance of built-ins is satisfied)
- $\mathcal{B} \cup A' \models \mathcal{IC}$ (the instance of abducibles respects the integrity constraints)

For simplicity and without loss of generality, we consider only ground queries;[4] an *abductive answer* to a query Q is a pair of finite sets of abducible and of built-in atoms $\langle A, B \rangle$ such that

- $\langle A, B \rangle$ has at least one consistent ground instance $\langle A', B' \rangle$,
- for any such $\langle A', B' \rangle$, we have $\mathcal{P} \cup A' \models Q$.

In other words, the abductive answer should be consistent and, if the abducible atoms were added to the programs as if they were ordinary Prolog facts, the query would succeed.

Minimality and Compaction

It is often required that an abductive answer be minimal measured in the number of abduced literals (or, alternatively, in a subset relation or subsumption ordering). If, for example, the query "`?- happy(me).`" has the set

$$A_1 = \{\texttt{rich(me)}, \texttt{has(me,nice\_students)}, \texttt{moon(full)}\}$$

is an abductive answer, and that

$$A_2 = \{\texttt{rich(me)}, \texttt{has(me,nice\_students)}\}$$

also is an abductive answer, then somehow, the fact `moon(full)` does not seem very interesting, as the smaller answer A_2 can explain the query. The

[2]We recall that an expression is ground if it contains no variable. A common ground instance means that the same grounding substitution is applied to both A and B.

[3]Recall that $\models$ can be read as "entails" or " has the logical consequence".

[4]For example, "`?- happy(me).`" is a ground query but "`?- happy(X).`" is not. Considering only ground queries means that we can ignore the traditional answer substitution; this can easily be added.

answer A_2 is minimal provided that none of its subsets are abductive answers, i.e., neither {`rich(me)`} nor {`has(me,nice_students)`} can explain the query.

Most published abduction algorithms try to unify a new abducible with one already produced (as to produce answers of a minimum number of literals), and tries out different alternatives under backtracking. This does not guarantee minimality in cases when, say, one branch of executions produces abducibles a and b but another may produce only a. Minimal answers can be selected by post-processing all answers found in this way. However, we argue that this principle which we call *compaction* is not always obvious or desirable, and we suggest it be optionally specified for selected abducible predicates.

If, for example, someone's car was stolen in Paris and his wallet in New York, it seems over-constrained to assume by default that the thieves are the same one. In other words, if both

$$B_1 = \{\texttt{steel(X,wallet)},\ \texttt{steel(Y,car)},\ \texttt{thief(X)},\ \texttt{thief(Y)}\}$$

and

$$B_2 = \{\texttt{steel(X,wallet)},\ \texttt{steel(X,car)},\ \texttt{thief(X)}\}$$

are abductive answers to a given query, we claim that it is not obvious that B_2 should be considered the best. In fact, when you write B_2 as $B_1 \cup \{\texttt{X=Y}\}$ seems that B_1 is smallest in some sense.

3 Abduction implemented in Prolog with a little help from CHR

(This section adapted from [6, 3]; see those papers for references to earlier work).

Constraint Handling Rules [8], CHR, is a declarative, rule-based language for writing constraint solvers and is now included as an extension of several versions of Prolog. Operationally and implementation-wise, CHR extends Prolog with a constraint store, and the rules of a CHR program serve as rewriting rules over constraint stores. CHR is declarative in the sense that its rules can be understood as logical formulas. Constraint predicates must be declared as such and can then be called from a Prolog program; see [8] for details. The following example declares a constraint predicate `a` and defines a so-called propagation rule.

```
constraints a/1.
a(1), a(2) ==> fail.
```

This rule identifies a state as illegal if it contains the two indicated constraints. As first noticed by [1], there is a clear analogy between abducibles plus integrity constraints and CHR's constraints plus rules.

The implementation of abduction in Prolog with CHR is simple. Abducibles are viewed as constraints in the sense of CHR: the logic program is executed by the Prolog system; whenever an abducible is called it is added automatically by CHR to the constraint store and CHR will activate integrity constraints whenever relevant. The complete hand-coded implementation of an abducible predicate `a/1` is provided by the following three lines.

```
:- use_module(library(chr)).
handler abduction.
constraints a/1.
```

Compaction for `a/1` is implemented by a single CHR rule; the following provides a correct implementation.

```
a(X), a(Y) ==> true | (X=Y ; dif(X,Y)).
```

The implementation used in the HYPROLOG system [3] applies a slightly optimized version of this rule using low-level facilities of CHR.

In addition, the integrity constraints mentioned above can be written directly as CHR rules.

In a comparison with other abductive logic programming systems we may emphasize the following; see [3] for more detailed comments.

- The indicated method is limited with respect to negation. There are important applications (see, e.g., [9]) that requires the creation of new abducibles from within negated calls to predicates. Our method can handle a limited form of so-called explicit negation which is hard-wired into the HYPROLOG system [3], but which cannot handle the indicated applications.

- However, for the applications that this methodology can handle, it is considerably faster that other, known systems for abduction. The reason for this is that we utilize the underlying technology in an optimal and direct way, as no intermediate level of interpretation is involved.

- Important applications such as many cases of diagnosis and natural language analysis can be modeled nicely without negation.

4 A first example of abduction in Prolog+CHR

This is extends a standard example from the literature, used by [9] and others. Consider the following Prolog program:

```
grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.
```

It has two rules and no facts. Obviously the following query fails when executed in Prolog:

```
?- grass_is_wet.
no
```

On the other hand, an abductive system should not give up that easily. It should do what it can to enforce that the query succeeds, and the only way it can do so is by suggesting which facts to add to the program.

A typical abductive interpreter do not actually modify the program source text but produces an abductive answer (as defined above) consisting of those facts, that if they were added to the program would make the query succeed. With an abductive interpreter, and provided that predicates `rained_last_night` and `sprinkler_was_on` are declared as abducibles, we may experience a dialogue as follows:

```
?- grass_is_wet.
rained_last_night ? ;
sprinkler_was_on ? ;
no
```

We can illustrate the meaning of these two answers by stating that the query `?- grass_is_wet.` would succeed when executed by a Prolog system given one of the following two Prolog programs (no CHR or abduction involved this time).

```
grass_is_wet:- rained_last_night.    grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.     grass_is_wet:- sprinkler_was_on.
rained_last_night.                   sprinkler_was_on.
```

Basically, an abductive interpreter works in the way that when it enters an abducible goal that otherwise would fail in Prolog, it simply notes that goal as part of the abductive answer. As we indicated above, CHR's constraint store can serve as a container for the abductive answer being built up gradually as the execution goes on.

Consider the following program that combines Prolog and CHR as we indicated above; two predicates are declared as constraints so that they will be treated as abducibles:

```
handler garden_humidity1.

constraints rained_last_night/0, sprinkler_was_on/0.

grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.
```

When this program is executed, the constraints entered are added to the constraint store. The declaration `handler garden_humidity.` is not important, but CHR's syntax requires such a declaration.

The resulting constraint store is printed as part of the answer and we get exactly the following when asking a query for `grass_is_wet`.

```
?- grass_is_wet.
rained_last_night ? ;
sprinkler_was_on ? ;
no
```

Integrity constraints can be seen as a way to rule out "weird" abductive answers, and these integrity constraints can be understood intuitively and in their precise semantics as integrity constraints for databases. Let us extend the program above with one more abducible predicate and a straightforward CHR rule that serves as an integrity constraint.

```
handler garden_humidity2.

constraints rained_last_night/0, sprinkler_was_on/0,
            full_moon_last_night/0.

rained_last_night, full_moon_last_night ==> fail.

grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.
```

The integrity constraint reads: If the constraint store contains the two constraints indicated by its lefthand side, then its body (following the arrow) is executed and here producing a failure. The following query to the program shows that only one answer is produced, as the potential second one triggers the integrity constraint.

```
?- full_moon_last_night, grass_is_wet.
full_moon_last_night,
sprinkler_was_on ? ;
no
```

Integrity constraints can contain any executable expression a its body, in particular another abducible. Consider the following.

```
handler garden_humidity3.

constraints rained_last_night/0, sprinkler_was_on/0,
            full_moon_last_night/0, mixed_weather_last_night/0.

rained_last_night, full_moon_last_night ==> mixed_weather_last_night.

grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.
```

This results in the following:

```
?- full_moon_last_night, grass_is_wet.
rained_last_night, mixed_weather_last_night ? ;
sprinkler_was_on ? ;
no
```

Notice that the additional abducible `mixed_weather_last_night` is only produced by for the first answer, as the integrity constraint is not triggered for the second one.

5 Database view update considered as abduction

The following program written in Prolog plus CHR defines a little database with integrity constraints, that express so-called key constraints on the `father` and `mother` relations, i.e., "you can only have one father and only one mother". The `grandfather` predicate corresponds to a view definition in a traditional database.

```
:- use_module(library(chr)).
handler view_update.

constraints father/2, mother/2.

father(X,Y), father(Z,Y) ==> Z=X.
mother(X,Y), mother(Z,Y) ==> Z=X.

grandfather(X,Z):- father(X,Y), father(Y,Z).
grandfather(X,Z):- father(X,Y), mother(Y,Z).

current_db:-
    father(peter, paul),
    father(paul,jens),
    mother(marie,jens).
```

It may seem a bit confusing that the current database is not defined by means of facts as usual, but this strange way above is needed in order to provide an interaction between new and old database facts.

In order to perform a view update to a given database, we can give it as a query, here using the auxiliary predicate `current_db`. The query below reads informally "In which ways can some `X` be a grandfather of `jens`. Notice that we get the the whole updated database as answer.

```
?- current_db, grandfather(X,jens).
X = peter,
father(peter,paul),
father(paul,jens),
mother(marie,jens),
father(peter,paul),
father(paul,jens) ? ;

father(peter,paul),
father(paul,jens),
mother(marie,jens),
father(X,marie),
mother(marie,jens) ? ;
no
```

The first answer suggests that `peter` could be grandfather of `jens`, provided the new fact `father(paul,jens)` is added to the database. The second answer does not indicate a specific value for `X` but indicates that any value v for `X` will do provided that `father(`v`,marie)` is added to the database. (For technical reasons that we shall not comment on here, some database facts becomes duplicated; this is easy to avoid by simple techniques in CHR.)

6 Simple diagnosis problems formulated as abduction

In a case of medical diagnosis, the patient is explaining various symptoms, "I have pain here, and here, but not here ....", and the doctor's job is to give the diagnosis which may the identification of a particular decease that can explain the observed symptoms. We can consider this as a case of abduction. The doctor's knowledge includes a large collection of rules about which deceases and conditions that may cause which symptoms. In the particular case, he should be able to figure out what are the specific deceases and conditions that can explain this patient's symptoms. Maybe a combination of deceases must be suggested in order to explain symptoms.

It is interesting to notice that the doctor uses his *knowledge* in order to provide statements about the patient's internal state without actually opening the patient. In other words, he is making predictions about hidden information that cannot be immediately checked in reality, and what is available to make the judgment are externally observed symptoms. This little discussion also indicates the potential unsound nature of abduction, in that the doctor may have chosen the wrong out of alternative diagnoses. This may then be corrected by new observations, say, that the suggested medical treatment has no effect, by further questioning of the patient, or from surgical investigations.

We use the notion of diagnosis in a more wide sense for finding explanations for systems that shows certain wrong behaviour. By system, we mean here a structure of interconnected primitive components, and where possible malfunctioning is caused by the malfunctioning of one or more of the primitive components. We will formulate this task as an abductive problem and show how it can be solved with abductive logic programming.

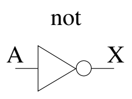
Example: Logical circuits

(Copied with a few changes from [5] in order to make this document self-contained). We indicate here how logical circuits can be simulated by Prolog programs, and later we extend the example with abduction for diagnosis.

Logical circuits represent a graphical formalism that serves as an abstract model of a class of electronic circuits composed by conductors and gates that can be thought of as performing logical operations. A physical component corresponding to the "not" gate below behaves in the following way: If a potential of about five volts is imposed on the input connector to the left in the diagram below, a potential of about zero volts can be observed at the output connector to the right (and the other way round for an input of about zero volts). The actual technology may be based on another pair of voltages than roughly-five/roughly-zero; the only interesting property is that the gates behave in a consistent way with respect to the logical behaviour. See, e.g., [12], chap. 3, for background and more information.

We represent the value corresponding to logical "truth" by the constant symbol `1` and logical "falsity" by `0`. The fact that these symbols in some context may serve as numbers in Prolog is of no interest here; we could in principle have used any other pair of two distinct constant symbols.

A given gate (or circuit) can be defined as a predicate whose arguments represent the gate's (or circuit's) input and output connectors. A "not" gate, for example, can be specified by the following table.

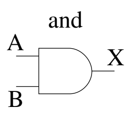


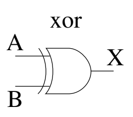
A	X
0	1
1	0

The corresponding definition in Prolog is the following sequence of facts, one for each row in the table.

```
not(0, 1).
not(1, 0).
```

"And" and "exclusive-or" gates are specified in similar ways, and so on for gates corresponding to other logical functions.





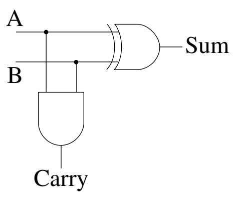
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

The corresponding predicates `and(A, B, X)` and `xor(A, B, X)` are defined as follows.

```
and(0, 0, 0).         xor(0, 0, 0).
and(0, 1, 0).         xor(0, 1, 1).
and(1, 0, 0).         xor(1, 0, 1).
and(1, 1, 1).         xor(1, 1, 0).
```

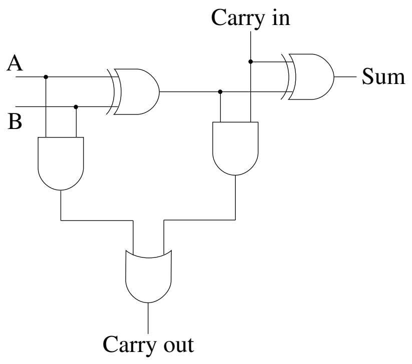
In the graphical language, gates are put together by connecting the components by means of conductors. In Prolog, we can do very much the same thing, except that we use variables instead of conductors. The following picture shows a so-called half-adder circuit.



It can be described by a predicate defined as follows.

```
halfadder(A, B, Carry, Sum):-
    and(A, B, Carry),
    xor(A, B, Sum).
```

The following more complicated circuit is known as a full adder.



It involves some internal conductors that are not connected to the circuit's external connectors. In the Prolog version, these conductors are replaced by variables that recur in the subgoals of the body but do not occur in the head, here X, Y, and Z.

```
fulladder(Carryin, A, B, Carryout, Sum):-
   xor(A, B, X),
   and(A, B, Y),
   and(X, Carryin, Z),
   xor(Carryin, X, Sum),
   or(Y, Z, Carryout).
```

The program explained in this section is a model of a physical system and we can use the program to predict properties of this system. We may, for example, pose a query that for a given set of input values (abstract potentials) calculates the output values.

```
?- fulladder(1, 0, 1, C, S).
C = 1
S = 0 ? ;
no
```

This shows that the circuit for adding a 0 and a 1 given a previous carry of 1 produces sum 0 with new carry 1, and it appears that this is the only solution.

6.1 Diagnosis based on the assumption of periodic faults

"Periodic fault" is a technical term which is a bit misleading, as it refers a fault that occurs now and then with irregular and unpredictable intervals. Thus such periodic faults are very difficult to find, and as we will see, also computationally very complex.

We describe the principles for the example of logical circuits introduced above. First of all, we need to assign an identifier to each occurrence of a gate in the circuit in order to point out (in the abductive answers), which components are defect. We may extend the clauses that define given circuit in the following way.

```
halfadder(A, B, Carry, Sum):-
        and(A, B, Carry, and0),
        xor(A, B, Sum, xor0).

fulladder(Carryin, A, B, Carryout, Sum):-
        xor(A, B, X, xor1),
        and(A, B, Y, and1),
        and(X, Carryin, Z, and2),
        xor(Carryin, X, Sum, xor2),
        or(Y, Z, Carryout, or1).
```

Notice that we have used predicates for the individual gates which take the extra argument for the identifier. We should then define these predicates so that they take into account the possible malfunctioning of the gate. We give the definition for "`and`" and explain it in the following.

```
and(A,B,X,ComponentId):-
   and(A,B,X),
   perfect(ComponentId).

and(A,B,X,ComponentId):-
   and(A,B,Z), disturbe(Z,X),
   defect(ComponentId).

disturbe(0,1).
disturbe(1,0).
```

The `disturbe` predicate is applied for the other types of gates as well in a similar way. The point of using `disturbe` is that we only notice that something is wrong with a given component if it produces the wrong result for given input; `and/3` refers to the previous definition of logical-and.

Before discussing the details of predicates `perfect` and `defect`, let us consider how observations are given. Observations represent observed tests assumed to be made using some actual device, whose structure is modeled by the logic programs shown. More precisely, observations and recorded samples of input-output pairs. For example, if a given half-adder device when given input `A=1`, `B=1` produces `Carry=1` and `Sum=1`, this indicates a wrong result and that something must be wrong inside the circuit.

The purpose of the extended predicates is, then, that we can present them with queries of observed input-and-output in the following ways, for having them analyzed. For example:

```
?- halfadder(1,1,1,1).
```

The output should, then be abducible answers consisting of `perfect` and `defect` facts. Referring to the method for implementing abduction in Prolog plus CHR above, we can make the two to work as abducible predicates by the following declarations.[5]

```
:- use_module(library(chr)).
handler diagnosis.
constraints perfect/1, defect/1.
```

[5]Notice that we could have implemented this example with only the `defect` predicate, with the default assumption being perfect behaviour. However, having the two is practical for the modification of the method that we will do in the following subsection.

We define the following integrity constraints that govern the interaction among the `perfect` and `defect` constraints.

```
defect(X)  \ defect(X) <=> true.
perfect(X) \ perfect(X) <=> true.
defect(X)  \ perfect(X) <=> true.
```

The first two rules simply remove duplicates, but the third one is interesting. It is a simpagation rule that reads intuitively: if a gate once have shown to be defect, it will be remembered even though the gate in some cases has produced the correct result. In other words, a given gate must expose wrong behaviour at least once in order to be registered as defect. This materializes the periodic fault assumption. Let us test a few queries.

```
?- halfadder(1,1,1,1).
perfect(and0),
defect(xor0) ? ;
no
```

This shows that exclusive-or gate that determines the `Sum` output argument needs to be defect in order to produce this behaviour, and also that there is no reason, from the given observation, to assume the and-gate to be defect.

The problem hight complexity given by the periodic fault assumption is apparent when we give more observations to a more complex circuit. Consider the following query that represents three wrong results.

```
?- fulladder(1,0,1,1,1), fulladder(1,1,1,0,0), fulladder(0,0,0,0,1).
```

It produces the total of 512 possible solutions, including duplicate solutions produced in different ways. This high number arises from a number of reasons.

- The method proposes a lot of explanations that include defects that compensate for each other.
- Combining this with the periodic fault assumption which says that a defect gate may sometimes do the right and sometimes the wrong, the set of possibilities explodes.
- Finally, the method is not able to use observations about correct behaviour in any way.[6]

The last point is critical. This corresponds to a doctor who is not able to take into account the observation that "... otherwise the patient is strong and healthy."

[6]In fact, even for queries representing entirely correct behaviour, the method proposes a lot of possible defects that mutually compensate for each other.

6.2 Diagnosis based on the assumption of consistent faults

Consistent faults means that a given primitive component always exposes the same output for the same input. In the example of logical circuits, it may be that case that a particular occurrence of a gate produces a 0 when anding two 1es. This means that it is not sufficient to record just that the gate is defect, but we must record for which inputs it gives right results and for which inputs it gives wrong results. For this purpose, we extend the abducible predicates with additional arguments so that, say, `defect(and3,1,1)` means that the indicated gate `and3` gives wrong results when given the input of two 1es. We can implement this in the following way, with integrity constraints that materialize the assumption of consistent faults.

```
:- use_module(library(chr)).
handler diagnosis.
constraints perfect/3, defect/3.

defect(X,In1,In2)  \ defect(X,In1,In2)  <=> true.
perfect(X,In1,In2) \ perfect(X,In1,In2) <=> true.
defect(X,In1,In2)  , perfect(X,In1,In2) <=> fail.
```

As in the previous version, the first two rules remove duplicates, and the last one is important. It indicates that it is not possible to create an abductive explanation that claims two different behaviours for the same component and given input.

The predicate definitions for the entire circuits remain the same, and those for the individual gates are adapted as follows.

```
and(A,B,X,ComponentId):-
   and(A,B,X),
   perfect(ComponentId,A,B).

and(A,B,X,ComponentId):-
   and(A,B,Z), disturbe(Z,X),
   defect(ComponentId,A,B).
```

The following is an example of a query that consists of one wrong and one correct observation.

```
?- halfadder(1,1,0,1), halfadder(1,0,0,1).
defect(and0,1,1),
defect(xor0,1,1),
perfect(and0,1,0),
perfect(xor0,1,0) ? ;
no
```

Only one answer is given. It is interesting to see that the answer also gives information about how throughly the circuit has been tested. It can be seen that the `and0` gate's behaviour on input `0-0` has not been tested, so if the purpose is a thorough test of the circuit, it might be a good idea to try more samples so that `defect(and0,0,0)` or `perfect(and0,0,0)` may show up.

Another interesting query is this one.

```
?- halfadder(1,1,1,0), halfadder(1,1,1,1), halfadder(0,0,1,0).
no
```

How can we interpret this results? Well, it indicates that there is no diagnosis for these observations under the consistent fault assumption. Two things can be wrong, then. Either the actual device does not obey the consistent fault assumptions (in which case we must go back to periodic fault assumption), or that the observations are wrong so that we should redo the tests using the device once again.

To see the difference between periodic and consistent, we recall that the following query produced 512 solution with perodic, and under consistent fault assumption, it produces “only” 144 solutions.

```
?- fulladder(1,0,1,1,1), fulladder(1,1,1,0,0), fulladder(0,0,0,0,1).
```

This is still quite many solutions, but we may improve by adding samples of correct input-out behaviour for our circuit (which, of course, only can be defended if we have performed test using the given device, that actually produced these results).

```
?- fulladder(1,1,0,1,0), fulladder(0,1,0,0,1), fulladder(0,0,1,0,1),
   fulladder(1,0,1,1,1), fulladder(1,1,1,0,0), fulladder(0,0,0,0,1).
```

This limits to now 48 solutions, and a careful study of those indicates that many of them describe combinations of mutually compensating errors which makes the circuit produce the correct samples in wrong ways, so to speak.

To compensate for this, we introduce a principle that we may call the **correct-results-produced-in-correct-way assumption**, which may not always be realistic but may help to bring down complexity. By this principle, we indicate that those inputs that are run through the different gates for the correct observation are assumed always to be correct, i.e., for such combinations we should freeze suitable `perfect` abducibles. This is very easy to implement in the model presented so far. First of all, notice that the clauses for the `and/4` predicate are ordered so that the solution consisting of `perfect` abducibles is tried before any other which includes `defect` abducibles. In other words, the first solution found for observations of correct behaviour will consists of `perfect` abducibles only. (Any other possible solutions with mutually compensating errors are produced afterwards on backtracking.)

Here Prolog's cut can be used effectively as indicated in the following schema for giving the queries.

`?-` ⟨*correct-samples*⟩ `, !,` ⟨*incorrect-samples*⟩`.`

This means that backtracking can only occur inside the analysis of the incorrect samples, and the abducibles from the correct ones stay fixed as `perfect`.

We may apply this principle to the qeury above.

```
?- fulladder(1,1,0,1,0), fulladder(0,1,0,0,1), fulladder(0,0,1,0,1),
   !,
   fulladder(1,0,1,1,1), fulladder(1,1,1,0,0), fulladder(0,0,0,0,1).
```

In fact this gives a disappointing `no` which may be a consequence of the correct-results-produced-in-correct-way assumption not being applicable here. Or rather, it indicates that these data have been produced artificially. A more convincing example is the following which without the cut produced 72 solutions, but with the cut only the one solution indicated.

```
?- fulladder(0,1,1,1,0), fulladder(0,1,0,0,1), fulladder(0,0,1,0,1),
   !,
   fulladder(1,0,1,1,1), fulladder(1,1,1,0,0), fulladder(0,0,0,0,1).

defect(xor2,1,1),
defect(and2,0,1),
defect(xor2,1,0),
defect(or1,1,1),
defect(xor1,0,0) ? ;
no
```

7 Discourse analysis as abduction

This section is not expanded in the present version; we refer to the articles [4, 3] for more information; perhaps you may need also to consult an introduction to Definite Clause Grammars which can be found in most textbooks on Prolog.

References

[1] Slim Abdennadher and Henning Christiansen. An experimental CLP platform for integrity constraints and abduction. In *Proceedings of FQAS2000, Flexible Query Answering Systems: Advances in Soft Computing series*, pages 141–152. Physica-Verlag (Springer), 2000.

[2] Ivan Bratko and Stephen Muggleton. Applications of inductive logic programming. *Commun. ACM*, 38(11):65–70, 1995.

[3] H. Christiansen and V. Dahl. HYPROLOG: a new approach to logic programming with assumptions and abduction. In Maurizio Gabbrielli and Gopal Gupta, editors, *Proceedings of Twenty First International Conference on Logic Programming (ICLP 2005)*, Lecture Notes in Computer Science, 2005. To appear.

[4] H. Christiansen and V. Dahl. Meaning in Context. In Anind Dey, Boicho Kokinov, David Leake, and Roy Turner, editors, *Proceedings of Fifth International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-05)*, volume 3554 of *Lecture Notes in Artificial Intelligence*, pages 97–111, 2005.

[5] Henning Christiansen. Introduction to Prolog as a database language. A course note, 2003. `http://www.ruc.dk/~henning/KIIS05/DatabaseProlog.pdf`

[6] Henning Christiansen and Veronica Dahl. Assumptions and abduction in Prolog. In Elvira Albert, Michael Hanus, Petra Hofstedt, and Peter Van Roy, editors, *3rd International Workshop on Multiparadigm Constraint Programming Languages, MultiCPL'04; At the 20th International Conference on Logic Programming, ICLP'04 Saint-Malo, France, 6-10 September, 2004*, pages 87–101, 2004.

[7] Peter A. Flach and Antonis C. Kakas, editors. *Abduction and Induction: Essays on their relation and integration.* Kluwer Academic Publishers, April 2000.

[8] Thom Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.

[9] A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming,* vol. 5, Gabbay, D.M, Hogger, C.J., Robinson, J.A., (eds.), Oxford University Press, pages 235–324, 1998.

[10] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf, editors. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Computer Science.* Springer, 1997.

[11] Swedish Institute of Computer Science. SICStus Prolog user's manual, Version 3.12. Most recent version available at `http://www.sics.se/isl`, 2004.

[12] Andrew S. Tanenbaum. *Structured Computer Organization.* Addison-Wesley, 4th edition, 1999.