

# ABEL – A New Language for Assumption-Based Evidential Reasoning under Uncertainty\*

**B. Anrig, R. Haenni and N. Lehmann**

Institute of Informatics  
University of Fribourg  
CH-1700 Fribourg  
Switzerland

**Phone:** +41 (26) 300 83 31

**Fax:** +41 (26) 300 97 26

**E-Mail:** rolf.haenni@unifr.ch

**WWW:** [www-iiuf.unifr.ch/dss/haenni](http://www-iiuf.unifr.ch/dss/haenni)

January 1997

## **Abstract**

Today, different formalisms exist to solve reasoning problems under uncertainty. For most of the known formalisms, corresponding computer implementations are available. The problem is that each of the existing systems has its own user interface and an individual language to model the knowledge and the queries.

This paper proposes ABEL, a new and general language to express uncertain knowledge and corresponding queries. Examples from different domains show that ABEL is powerful and general enough to be used as common modeling language for the existing software systems. A prototype of ABEL is implemented in Evidenzia, a system restricted to models based on propositional logic. A general ABEL solver is actually being implemented.

---

\*Research supported by grant No.2100-042927.95 of the Swiss National Foundation for Research.

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Language Description</b>	<b>5</b>
2.1	The Command “tell” . . . . .	5
2.1.1	Type Definitions and Pre-Defined Types . . . . .	6
2.1.2	Variables and Assumptions . . . . .	7
2.1.3	Expressions . . . . .	8
2.1.4	Constraints . . . . .	8
2.1.5	Statements . . . . .	9
2.1.6	Modules . . . . .	10
2.2	The Command “observe” . . . . .	10
2.3	The Command “ask” . . . . .	11
2.4	Other Facilities . . . . .	13
<b>3</b>	<b>Modeling Hints</b>	<b>14</b>
3.1	Empty Gas Tank . . . . .	14
3.2	Duration of Activities . . . . .	16
3.3	Soccer World Cup Qualifying . . . . .	18
<b>4</b>	<b>Constraint Satisfaction</b>	<b>20</b>
4.1	Age of Children . . . . .	20
4.2	Pythagoras . . . . .	21
4.3	Physical Process . . . . .	22
<b>5</b>	<b>Diagnostics in Technical Systems</b>	<b>25</b>
5.1	Digital Circuits . . . . .	25
5.2	Multiple Measures . . . . .	27
5.3	Arithmetical Network . . . . .	29
5.4	Means-End Models . . . . .	30
<b>6</b>	<b>Causal Networks</b>	<b>35</b>
6.1	Medical Diagnosis . . . . .	38
6.2	Burglary . . . . .	40

<i>CONTENTS</i>	3
<b>7 Failure Trees</b>	<b>43</b>
<b>A ABEL Grammar</b>	<b>46</b>
<b>References</b>	<b>47</b>

## 1 Introduction

Today, different formalisms exist to solve reasoning problems under uncertainty. The most popular approaches are the theory of **Bayesian networks** (Lauritzen & Spiegelhalter, 1988) and the Dempster-Shafer **theory of evidence** (Shafer, 1976). For most of the known formalisms, corresponding computer implementations are available. De Kleer's idea of **assumption-based truth maintenance systems** (ATMS) proposes a general architecture for problem solvers in the domain of uncertain reasoning (de Kleer, 1986). Another general architecture is given by Shenoy's concept of **valuation networks** (Lauritzen & Shenoy, 1995). Popular software systems are Belief 1.2 (Almond, 1990), Ideal (Srinivas & Breese, 1990), MacEvidence (Hsia & Shenoy, 1989), Pulcinella (Saffiotti & Umkehrer, 1991), TresBel (Xu & Kennes, 1994), Graphical-Belief 2.0 (Almond, 1995), and Hugin (Andersen *et al.*, 1990). The problem is that each of the existing systems has its own user interface and an individual language to model the knowledge and the queries.

This paper proposes ABEL<sup>1</sup>, a general language to express uncertain knowledge and corresponding queries. In ABEL, uncertainty is expressed by **assumptions**, a special type of variable. Assumptions represent unknown circumstances, risks, or interpretations. Often, it is possible to assign probabilities to the values of an assumption. Furthermore, assumptions are the basic elements to build arguments for hypotheses. The idea of using assumptions has been introduced in de Kleer's ATMS. ABEL uses de Kleer's idea in a more general context of **assumption-based systems** (Kohlas & Monney, 1993).

Examples from different problem domains show that ABEL is powerful and general enough to be used as common modeling language for the existing software systems. A prototype of ABEL is implemented in Evidenzia (Lehmann, 1994; Haenni, 1996). This system is restricted to models based on propositional logic. A general ABEL solver is actually being implemented.

The easiest way to get familiar with ABEL is to look at concrete examples. This paper is a collection of examples in the domain of uncertain reasoning. It will serve as basis for building the general solver. Section 2 introduces the main elements of ABEL; Section 3 discusses three examples of modeling and reasoning with **hints** (Kohlas & Monney, 1995); Section 4 introduces **constraint satisfaction** problems (Mackworth, 1987) and gives three concrete examples; Section 5 describes problems of **diagnostics in technical systems** (Reiter, 1987); Section 6 treats examples of **causal** or **Bayesian networks** (Lauritzen & Spiegelhalter, 1988); finally, Section 7 shows an example of so-called **failure trees** (Shafer & Logan, 1987).

---

<sup>1</sup>ABEL stands for Assumption-Based Evidential Language.

## 2 Language Description

This section introduces the main elements of ABEL. The language is based on three other computer languages: (1) from **Common Lisp** (Steele, 1990) it adopts **prefix notation** and therewith a mess of opening and closing parentheses; (2) from **Pulcinella** (Saffiotti & Umkehrer, 1991) it uses the idea of the commands **tell**, **ask**, and **empty**; and (3) from the existing ABEL prototype (Lehmann, 1994; Haenni, 1996) it inherits the concept of **modules** and the syntax of the queries. A precise language description is given in Appendix A.

Working with ABEL usually involves three sequential steps. First, the given information is expressed using the command **tell** (see Subsection 2.1). The resulting model is called **basic knowledge base**. It describes the part of the available information that is relatively constant and static in course of time such as rules, relations, or dependencies between different statements or entities. Second, actual facts or observations about the concrete, actual situation are specified using the command **observe** (see Subsection 2.2). Facts and observations may change in course of time. The basic knowledge base completed by actual observations is called **actual knowledge base**. Finally, queries about the actual knowledge base are expressed using the command **ask**. For a given hypothesis, different types of queries are possible (see Subsection 2.3).

### 2.1 The Command “tell”

The most important feature in ABEL is the command **tell**. It is used to build the basic knowledge base. It consists of one or several lines called **instructions**. The contents of an instruction can be (1) a **definition** of types, variables, assumptions, or modules, (2) a **statement**, i.e. a rule or another part of the basic knowledge base, or (3) an application of a **module**. The sequence of instructions is interpreted in parallel as a conjunction. The order in which the instructions appear within a **tell**-command is therefore not important. The syntax of a **tell**-command is the following:

```
(tell <instr-1>
    <instr-2>
    ...
    <instr-n>)
```

Every instruction can be seen as a piece of information. Therefore, the command **tell** is used to add new pieces of information to the existing basic knowledge base. Note that the instructions of a **tell**-command can be distributed among several **tell**-commands. The following sequence of **tell**-commands, for example, is equivalent to the command above:

```
(tell <instr-1>)
(tell <instr-2>)
...
(tell <instr-n>)
```

Different `tell`-commands are often used to group related pieces of information and to separate these groups from each other. ABEL supports this technique by allowing the user to specify keywords for different `tell`-commands. A keyword is either a name with a preceding colon (e.g. `:part1`, `:part2`, etc.) or an integer. A keyword is always the first argument of a `tell`-command:

```
(tell <key>
  <instr-1>
  <instr-2>
  ...
  <instr-n>)
```

Keywords can be used to change or delete different parts of the knowledge base independently (see Subsection 2.4). In cases of big knowledge bases, this technique turns out to be very useful. It also helps to visualize the structure of the given information.

### 2.1.1 Type Definitions and Pre-Defined Types

Reasoning under uncertainty is always concerned with open questions. A question is determined by the set of possible answers called **domain**. An important element of ABEL is therefore the possibility of declaring domains. Note that different questions may have the same domain; they are said to be of the same **type**. Therefore, ABEL provides a command `type` to define domains. Such a set may contain symbols or numbers. Examples:

```
(type test (passed failed))
(type colors (red green blue yellow))
(type month (1 2 3 4 5 6 7 8 9 10 11 12))
...
```

There are also a few pre-defined types in ABEL:

- **integer**: the set of (positive and negative) integers  $\mathbb{Z}$ ;
- **real**: the set of (positive and negative) real numbers  $\mathbb{R}$ ;
- **binary**: a special type to declare propositional symbols.

It is also possible to define subsets of pre-defined types such as intervals. Examples:

```
(type year integer)
(type month integer[1..12])
(type size real[0..250])
(type pos-integer integer[0..])
(type neg-real real[..0])
...
```

### 2.1.2 Variables and Assumptions

The second step consists in defining **variables** and **assumptions**. Variables represent questions. For each variable, the set of possible values (answers) has to be specified, i.e. the type of the variable must be declared. Note that different variables can be of the same type. In ABEL, variables are defined as follows:

```
(var <v-1> <v-2> ... <v-n> <type>)
```

An individual `var`-command is therefore necessary for each type of variables. The type specifier can either be a pre-defined type, a user-defined type, or a new type specification according to the previous subsection. Examples:

```
(var c1 c2 colors)
(var a b n integer)
(var x y z real)
(var language (french german spanish english))
(var s real[0..250])
(var pi 3.1416)
(var p q r binary)
...
```

Assumptions are defined in a similar way. The difference between assumptions and variables is that assumptions represent uncertain events, unknown circumstances, or risks, rather than precise open questions. Assumptions are used to build arguments for hypotheses. Additionally, it is often possible to impose probabilities on the values of an assumption. The probabilities of different assumptions are assumed to be independent. In ABEL, probabilities are optional. If no probabilities are declared, then assumptions are only used to generate symbolic arguments (explanations). Assumptions can be defined as follows:

```
(ass <a-1> <a-2> ... <a-n> <type> <probabilities>)
```

The probabilities are given as a list of values between 0 and 1 summing to 1. The order in which the probabilities appear in the list corresponds to the order in which the values for the given type are defined. Examples:

```
(ass test1 test2 test (0.8 0.2))
(ass weather (sun clouds rain) (0.5 1/3 1/6))
(ass k1 k2 k3 colors)
(ass ok? binary 0.75)
...
```

Note that assumptions have always finite domains, i.e. the only pre-defined type allowed for assumptions is `binary` (propositional symbols). In this case, only the probability  $p$  for the positive literal has to be specified. The probability for the negative literal is given implicitly by  $1 - p$ .

### 2.1.3 Expressions

Based on numerical variables (type `integer` or `real`), it is possible to build compound (algebraic) **expressions**. The syntax of ABEL expressions corresponds to LISP expressions. It uses prefix notation: the first element within a pair of opening and closing parentheses is the name of one of the pre-defined algebraic operators. The remaining elements are the operands, i.e. names of numerical variables:

```
(<operator> <op-1> <op-2> ... <op-n>)
```

ABEL provides the following algebraic operators: `+`, `-`, `*`, `/`, `sqr`, `sqrt`, `exp`, `expt`, `log`, `abs`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `mod`, `min`, and `max`. The semantic of these operators corresponds to Common LISP (Steele, 1990). Examples:

```
(+ x y z)
(/ (* x y) z)
(- (max x y z) (min x y z))
(abs (sin (sqrt x)))
...
```

Note that variables (e.g. `x`, `y`, `c1`, `language`, etc.), assumptions (e.g. `test1`, `test2`, `weather`, `ok?`, etc.), numbers (e.g. `17`, `1/3`, `-23.5`, etc.), symbols (e.g. `french`, `sun`, etc.), and sets (e.g. `(german spanish)`, `(1 2 4 5 10)`, etc.) are also considered as (atomic) expressions.

### 2.1.4 Constraints

ABEL expressions can be used to build so-called **constraints**. They restrict the possible values of the variables or assumptions involved to a subset of their Cartesian product. Constraints always compare two expressions. The syntax is therefore as follows:

```
(<operator> <exp-1> <exp-2>)
```

ABEL provides the following operators to build constraints:

- `=` restricts a variable (1st expression) to a certain value (2nd expression), or restricts the two expressions to be equal;
- `/=` complement of `=`;
- `in` restricts a variable (1st expression) to a subdomain (2nd expression);
- `<` restricts the first (algebraic) expression to be smaller than the second (algebraic) expression;
- `<=` restricts the first (algebraic) expression to be smaller or equal than the second (algebraic) expression;

- > restricts the first (algebraic) expression to be greater than the second (algebraic) expression;
- >= restricts the first (algebraic) expression to be greater or equal than the second (algebraic) expression.

Examples:

```
(= c1 blue)
(= n 17)
(/= (+ x y) z)
(in language (german spanish))
(< x 23.5)
(>= (/ (sin x) (cos x)) (tan x))
ok?
...
```

Note that variables or assumptions of type **binary** (e.g. `ok?`, `p`, `q`, etc.) are also considered as (atomic) constraints.

### 2.1.5 Statements

Constraints can be used to build logical expressions called **statements**. A constraint itself is considered as (atomic) statement. ABEL supports the following different types of compound statements:

- (`->` `<stmt-1>` `<stmt-2>`): material implication;
- (`<->` `<stmt-1>` `<stmt-2>`): equivalence;
- (`and` `<stmt-1>` `<stmt-2>` ... `<stmt-n>`): logical and;
- (`or` `<stmt-1>` `<stmt-2>` ... `<stmt-n>`): logical (inclusive) or;
- (`xor` `<stmt-1>` `<stmt-2>` ... `<stmt-n>`): logical exclusive or;
- (`not` `<stmt>`): negation.

Examples:

```
(and (= n 17) (= c1 blue))
(-> (= (+ x y) z) (in language (german spanish)))
(not (>= (/ (sin x) (cos x)) (tan x)))
ok?
...
```

Statements are used to build the basic knowledge base. Every single statement is a part of the given information. A sequence of statements is interpreted as conjunction. The following two statements are therefore equivalent to the first statement of the previous examples:

```
(= n 17)
(= c1 blue)
```

Note that there are two pre-defined constraints `tautology` and `contradiction`. The first one represents a statement that is always true, whereas the second one is a statement that is always false.

### 2.1.6 Modules

The concept of **modules** has already been used in the existing ABEL prototype (Lehmann, 1994; Haenni, 1996). The idea of modules is that similar parts of the given information are modeled only once. The definition of a module can be compared with the definition of a LISP function. Every module has a name and it consists of a set of parameters and a body. The parameters of an ABEL module are variables or assumptions. The body of a module is a sequence of ABEL instructions (see Subsection 2.1). Therefore, it is possible to define local types, local variables, local assumptions, and local modules within the body of a module. The syntax for a module definition is the following:

```
(module <name> (<par-1> <par-2> ... <par-n>)
  <instr-1>
  <instr-2>
  ...
  <instr-n>)
```

Note that the parameters of a module have to be declared either as variables or assumptions. Additionally, types have to be specified for the parameters. The parameter list is therefore a sequence of variable and assumption definitions (see Subsection 2.1.2). Examples of ABEL modules can be found in Section 5.

A module can be used to generate similar parts of the given information. An **instance** of a module is obtained by “calling” the module with actual parameters:

```
(<name> <par-1> <par-2> ... <par-n>)
```

Note that the types of the actual parameters are implicitly given by the parameter specification of the module definition. It is therefore not necessary to specify the types of the actual parameters outside the module.

## 2.2 The Command “observe”

Usually, in order to complete a model, **observations** or **facts** are added to the basic knowledge base. Observations describe the actual situation or the concrete circumstances of the problem. Note that observations may change in course of time. It is therefore important to separate observations from the basic knowledge base. ABEL provides a command `observe` to specify observations. It expects a sequence of ABEL statements. The sequence is interpreted as a conjunction.

```
(observe <stm-1>
      <stm-2>
      ...
      <stm-n>)
```

Examples:

```
(observe (= n 17)
        (= c1 blue))
(observe (in language (german spanish)))
(observe ok?)
...
```

Statements given by `observe-` and `tell-` commands are treated similarly. The difference is that `observe-` statements can be deleted or changed independently when new observations were made (see Subsection 2.4). To change an observation, the same statement with the new values has to be re-written. For example,

```
(observe (= n 18))
```

would change the actual value of `n` to 18.

It is also possible to keep different observations for the same variables. If, for example, a technical system is tested with different input values (see Subsection 5.2), then the corresponding output values are different observations for the same variables. The ABEL command `observe` can also be used to treat such cases. The first argument of an `observe-` command can be a keyword (see Subsection 2.1) that specifies the point in time of the succeeding observations. Examples:

```
(observe :first-measure
        (= n 17))
(observe :second-measure
        (= n 18))
...
```

### 2.3 The Command “ask”

Queries about the actual knowledge base are expressed using the command `ask`. Generally, there are two different types of queries: (1) it can be interesting to get the available information about certain variables; and (2) it can be interesting to get symbolic or numerical arguments in favor or against certain hypotheses. In both cases, several queries can be treated at once:

```
(ask <query-1>
     <query-2>
     ...
     <query-n>)
```

In the first case, a query is simply an ABEL expression in the sense of Subsection 2.1.3. This type of query is useful, for example, in constraint satisfaction problems (see Subsection 4). Examples:

```
(ask language)
(ask x y)
(ask (+ x y z))
...
```

The second way to state queries is important in problems of assumption-based reasoning. The idea is to find arguments in favor or against hypotheses. For a given hypothesis, different types of arguments may be of interest: **support**, **quasi-support**, **plausibility**, or **doubt** (Haenni, 1996). A hypothesis is an ABEL statement in the sense of Subsection 2.1.5:

```
(sp <stm>)
(qs <stm>)
(pl <stm>)
(db <stm>)
```

Examples:

```
(ask (sp (and (= n 17) (< x 5))))
(ask (qs (in language (german spanish))))
(sp (not ok?))
...
```

Arguments are conjunctions of normal or negated ABEL constraints over assumptions. The support of a hypothesis, for example, is the set of all such conjunctions, which allow to deduce the hypothesis from the given knowledge base. Examples of arguments:

```
(and (= test1 passed) ok?)
(and (= k1 red) (not (= k2 red)))
...
```

It is also possible to ask for numerical arguments like **degree of support**, **degree of quasi-support**, **degree of plausibility**, or **degree of doubt** (Haenni, 1996):

```
(dsp <stm>)
(dqs <stm>)
(dpl <stm>)
(ddb <stm>)
```

A numerical argument is obtained by computing the probability for the corresponding symbolic argument. This computation is based on a priori probabilities given by the definition of the assumptions.

## 2.4 Other Facilities

Sometimes, it is necessary to delete the entire knowledge base or parts of it. For that purpose, ABEL provides a command `empty`. If `empty` is called without arguments, then the entire model is deleted. If the keyword `observe` is supplied, then it deletes only the observations. Additionally, by supplying one or several keywords used within `tell-` or a `observe-`commands, it is possible to delete only the corresponding parts of the knowledge base. Examples:

```
(empty)
(empty observe)
(empty :part1 :part2)
(empty :first-measure)
(empty :second-measure)
...
```

To obtain more readable ABEL models, it is possible to introduce comments into the code. According to Common LISP, there are two different ways to write comments:

```
#! This is a comment over one
   or several lines |#
; This is another comment
...
```

### 3 Modeling Hints

The **mathematical theory of hints** (Kohlas & Monney, 1995) provides a formal method for building models of imprecise, uncertain, and contradictory information. The information contained in a hint always belongs to one or several **questions**. Let  $\Theta$  denote the set of all possible answers for a certain question.  $\Theta$  is assumed to be complete in the sense that the true (but unknown) answer is exactly one of its elements.  $\Theta$  is called **frame of discernment**.

To model the uncertainty contained in a hint, another (finite) set  $\Omega$  of possible **interpretations** (of the information) has to be considered. Again, exactly one of the elements of  $\Omega$  is the (unknown) correct interpretation of the given hint; all other interpretations are wrong.

Each interpretation  $\omega \in \Omega$  permits to restrict the possible answers to a subset  $\Gamma(\omega) \subseteq \Theta$ . This means that if  $\omega$  turns out to be the correct interpretation of the hint, then the true answer must be within  $\Gamma(\omega)$ . Subsets  $\Gamma(\omega)$  are called **focal sets** of the hint.

A triple  $\mathcal{H} = (\Omega, \Theta, \Gamma)$  is called **hint** relative to a frame of discernment  $\Theta$  (Kohlas & Monney, 1995). A subset  $H \subseteq \Theta$  represents the hypothesis that the true answer belongs to  $H$ . In order to judge a hypothesis  $H$  in the light of a hint  $\mathcal{H}$ , it is necessary to find arguments that allow to prove or to disprove the hypothesis.

Hints can easily be described using ABEL. The following subsections show some concrete examples. They illustrate the use of hints and the way how to describe hints in ABEL.

#### 3.1 Empty Gas Tank

Consider the question whether the gas tank of a car is empty or not. Let  $\Theta = \{empty, full\}$  be the frame of discernment. Suppose the gas gauge is indicating an empty tank. This information is a hint with two possible interpretations  $\Omega_1 = \{ok, abnormal\}$ . *ok* stands for a correct functioning of the gauge, whereas *abnormal* stands for an abnormal functioning. Clearly, the focal sets are  $\Gamma_1(ok) = \{empty\}$  and  $\Gamma_1(abnormal) = \{empty, full\} = \Theta$ .

The hint described in this example can be modeled using ABEL. First, the sets  $\Theta$  for the gas tank and  $\Omega_1$  for the gauge have to be declared. Suppose that in 99 out of 100 cases the gauge is functioning correctly. The corresponding probabilities are therefore 0.99 and 0.01, respectively.

```
(tell
  (var tank (empty full))
  (ass gauge (ok abnormal) (0.99 0.01)))
```

The second step consists in encoding the focal sets of the hint. The focal sets determine a multi-valued mapping from  $\Omega_1$  to  $\Theta$ . This mapping can be modeled by the following material implications:

```
(tell
```

```
(-> (= gauge ok) (= tank empty))
(-> (= gauge abnormal) (in tank (empty full))))
```

To make the example more interesting, consider a second hint coming from the mileage counter. Knowing the car's average consumption, it is possible to compute the car's maximal reach  $r$ . Furthermore, the driver usually resets the mileage counter to zero each time the tank is refueled. But sometimes, the driver forgets to reset the counter. Suppose that the mileage counter is currently indicating a distance close to  $r$ . This information is a second hint about  $\Theta$ . It allows two different interpretations  $\Omega_2 = \{reset, forgotten\}$ . *reset* means that the mileage counter is indicating the correct distance covered since the last refueling, whereas the second interpretation *forgotten* assumes that this is not the case. The corresponding focal sets are clearly  $\Gamma_2(reset) = \{empty\}$  and  $\Gamma_2(forgotten) = \{full\}$ . Figure 3.1 shows the focal sets of the two given hints:

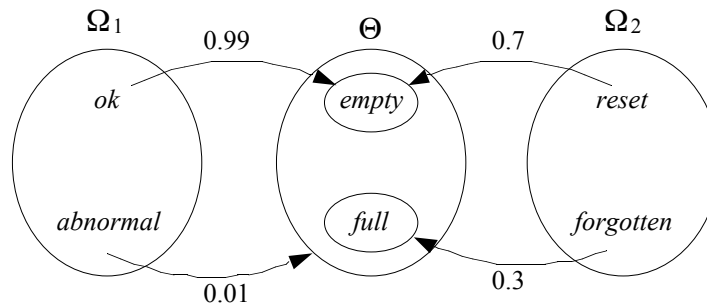


Figure 3.1: The focal sets for the given hints.

Suppose that in 7 cases out of 10 the mileage counter is reset. The probabilities are therefore 0.7 and 0.3, respectively. The following ABEL code describes the second hint and completes the model:

```
(tell
  (ass counter (reset forgotten) (0.7 0.3))

  (-> (= counter reset) (= tank empty))
  (-> (= counter forgotten) (= tank full)))
```

The model can now be used to judge hypotheses in the light of the given hints. In this example, the interesting question is whether the gas tank is empty or not. To obtain arguments in favor or against this question, the following ABEL instructions may be helpful:

```
(ask
  (sp (= tank empty))
  (sp (= tank full)))
```

The arguments obtained for the queries above are `(= counter reset)` for `(= tank empty)` and `(and (= gauge abnormal)(= counter forgotten))` for `(= tank full)`. To obtain corresponding numerical results, the following queries are necessary:

```
(ask
  (dsp (= tank empty))
  (dsp (= tank full)))
```

According to Dempster's rule of combination (Kohlas & Monney, 1995), this corresponds to a degree of support 0.996 for an empty gas tank, and 0.004 for a full tank.

### 3.2 Duration of Activities

Suppose the construction of a new house consists of three main tasks: (1) the construction of the foundation walls, (2) the roofing, (3a) the interior decoration, and (3b) the paintings. The time needed for task ( $i$ ) is denoted by  $d_i$ ,  $i = 1, 2, 3a, 3b$ . Assume that task (2) starts as soon as task (1) is finished, whereas task (3a) and task (3b) start in parallel as soon as task (2) is finished. This scenario is illustrated in Figure 3.2.

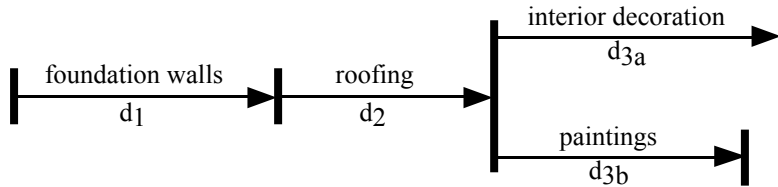


Figure 3.2: A house construction time schedule.

The total time  $d$  to construct the house is the sum  $d_1 + d_2 + d_3$  with  $d_3 = \max(d_{3a}, d_{3b})$ . The interesting question is the duration  $d$  of the project. This is a so-called **scheduling problem**. In this type of problems, the duration of the activities are traditionally supposed to be known. In reality, they can only be estimated in dependence of particular future conditions and circumstances.

In the case of house constructing, the duration of a project depends mainly on future meteorological conditions. For example, let  $\Omega_i = \{sun_i, clouds_i, rain_i\}$ ,  $i = 1, 2, 3$ , describe the possible weather conditions during phase ( $i$ ). If durations are expressed in days, then  $\Theta$  is the set of non-negative integers. Suppose that the durations are estimated as follows:

- for  $d_1$ :  $\Gamma_1(sun_1) = \{10, \dots, 14\}$ ,  $\Gamma_1(clouds_1) = \{12, \dots, 16\}$ , and  $\Gamma_1(rain_1) = \{20, \dots, 25\}$ ;
- for  $d_2$ :  $\Gamma_2(sun_2) = \Gamma_2(clouds_2) = \{8, 9, 10\}$ , and  $\Gamma_2(rain_2) = \{14, 15, 16\}$ ;
- for  $d_{3a}$ :  $\Gamma_{3a}(sun_3) = \Gamma_{3a}(clouds_3) = \Gamma_{3a}(rain_3) = \{10, \dots, 14\}$ ;
- for  $d_{3b}$ :  $\Gamma_{3b}(sun_3) = \{4, 5, 6\}$ ,  $\Gamma_{3b}(clouds_3) = \{7, 8\}$ , and  $\Gamma_{3b}(rain_3) = \{13, \dots, 17\}$ .

Obviously, the information contained in this example forms different hints, which can easily be transformed into corresponding ABEL model:

```

(tell
  (type weather (sun clouds rain))
  (ass w1 w2 w3 weather)
  (var d1 d2 d3a d3b integer[0..])

  (-> (= w1 sun)
    (and (>= d1 10) (<= d1 14)))
  (-> (= w1 clouds)
    (and (>= d1 12) (<= d1 16)))
  (-> (= w1 rain)
    (and (>= d1 20) (<= d1 25)))
  (-> (in w2 (sun clouds))
    (and (>= d2 8) (<= d2 10)))
  (-> (= w2 rain)
    (and (>= d2 14) (<= d2 16)))
  (-> (in w3 (sun clouds rain))
    (and (>= d3a 10) (<= d3a 14)))
  (-> (= w3 sun)
    (and (>= d3b 4) (<= d3b 6)))
  (-> (= w3 clouds)
    (and (>= d3b 7) (<= d3b 8)))
  (-> (= w3 rain)
    (and (>= d3b 13) (<= d3b 17))))

```

To complete the model, the relations between  $d$ ,  $d_1$ ,  $d_2$ ,  $d_3$ ,  $d_{3a}$ , and  $d_{3b}$  have to be added:

```

(tell
  (var d3 d integer[0..])

  (= d3 (max d3a d3b))
  (= d (+ d1 d2 d3)))

```

The model can now be used to judge hypotheses about the duration of the house construction. Examples of possible hypotheses are given in the following queries:

```

(ask
  (sp (> d 42))
  (sp (< d3b d3a))
  (pl (<= d 30)))

```

The results for these queries can be derived from the following table. It shows the minimal and maximal project durations for different weather conditions during the project phases.

Period 1	Period 2	Period 3	$d_{min}$	$d_{max}$
<i>sun</i>	<i>sun/clouds</i>	<i>sun/clouds</i>	28	38
<i>clouds</i>	<i>sun/clouds</i>	<i>sun/clouds</i>	30	40
<i>rain</i>	<i>sun/clouds</i>	<i>sun/clouds</i>	38	49
<i>sun</i>	<i>sun/clouds</i>	<i>rain</i>	31	41
<i>clouds</i>	<i>sun/clouds</i>	<i>rain</i>	33	43
<i>rain</i>	<i>sun/clouds</i>	<i>rain</i>	41	52
<i>sun</i>	<i>rain</i>	<i>sun/clouds</i>	34	44
<i>clouds</i>	<i>rain</i>	<i>sun/clouds</i>	36	46
<i>rain</i>	<i>rain</i>	<i>sun/clouds</i>	44	55
<i>sun</i>	<i>rain</i>	<i>rain</i>	37	47
<i>clouds</i>	<i>rain</i>	<i>rain</i>	39	49
<i>rain</i>	<i>rain</i>	<i>rain</i>	47	58

The support for ( $> d\ 42$ ) will therefore be ( $\text{and } (= w1\ \text{rain}) (= w2\ \text{rain})$ ), the support for ( $< d3b\ d3a$ ) will be ( $\text{or } (= w3\ \text{sun}) (= w3\ \text{clouds})$ ), and the plausibility for ( $\leq d\ 30$ ) will be ( $\text{and } (\text{not } (= w1\ \text{rain})) (\text{not } (= w2\ \text{rain})) (\text{not } (= w3\ \text{rain}))$ ).

### 3.3 Soccer World Cup Qualifying

Consider Europe's group 3 of the Soccer World Cup qualifying draw. Suppose that only two games (Switzerland vs. Azerbaijan and Finland vs. Hungary) remain to complete the qualification phase. The following table shows a possible situation in group 3 before the last two games were played:

1.	Norway	8	4	1	3	17:14	13
2.	Hungary	7	3	3	1	15:8	12
3.	Switzerland	7	3	2	2	11:9	11
4.	Finland	7	3	1	3	12:12	10
5.	Azerbaijan	7	1	1	5	4:16	4

Only the group winner is qualified for the 1998 World Cup to be held in France. Therefore, the main question – “who is going to be qualified?” – allows five different answers described by  $\Theta = \{Norway, Hungary, Switzerland, Finland, Azerbaijan\}$ . Other questions of interest are the final points for each team. Points are non-negative integers and therefore,  $\Theta_N = \Theta_H = \Theta_S = \Theta_F = \Theta_A = \mathbb{N}$  are the corresponding sets of possible answers.

Each of the two remaining games has three possible outcomes: (1) the home team wins, (2) a tie game, or (3) the guest team wins.  $\Omega_1 = \Omega_2 = \{home, tie, guest\}$  represent the two sets of possible results of the remaining games. A winning team obtains 3 and a losing team 0 points. In the case of a tie game, both teams obtain 1 point. This information forms a hint for each of the four teams involved in the two remaining games. Another (precise) hint is given by the fact that the final points for Norway are already known. In ABEL, these hints can be encoded as follows:

(tell

```

(type game (home tie guest))
(ass game1 game) ; Switzerland vs. Azerbaijan
(ass game2 game) ; Finland vs. Hungary
(var points-N points-H points-S points-F points-A integer[0..])
  #| the final points for N=Norway, H=Hungary, etc. |#
(= points-N 13))

(tell :game1
  (-> (= game1 home)
    (and (= points-S 14) (= points-A 4)))
  (-> (= game1 tie)
    (and (= points-S 12) (= points-A 5)))
  (-> (= game1 guest)
    (and (= points-S 11) (= points-A 7))))

(tell :game2
  (-> (= game2 home)
    (and (= points-F 13) (= points-H 12)))
  (-> (= game2 tie)
    (and (= points-F 11) (= points-H 13)))
  (-> (= game2 guest)
    (and (= points-F 10) (= points-H 15))))

```

To complete the model, the relation between the final points and the team to be qualified for the 1998 World Cup has to be encoded (the case where two teams have the same final amount of points is neglected):

```

(tell :qualified
  (var qualified (norway hungary switzerland finland azerbaijan))

  (-> (> points-N (max points-H points-S points-F points-A))
    (= qualified norway))
  (-> (> points-H (max points-N points-S points-F points-A))
    (= qualified hungary))
  (-> (> points-S (max points-N points-H points-F points-A))
    (= qualified switzerland))
  (-> (> points-F (max points-N points-H points-S points-A))
    (= qualified finland))
  (-> (> points-A (max points-N points-H points-S points-F))
    (= qualified azerbaijan)))

```

The model can now be used to treat queries like

```

(ask
  (sp (= qualified switzerland)))

```

where `(and (= game1 home) (not (= game2 guest)))` would be a possible answer.

## 4 Constraint Satisfaction

The formalism of constraint satisfaction problems has been successfully used in different areas of Artificial Intelligence. In recent years, a lot of research was done for developing new algorithms to solve constraint satisfaction problems. A good and short introduction to Constraint Satisfaction is given in (Mackworth, 1987).

Formally, a **constraint satisfaction problem** (CSP) involves a set of  $n$  **variables**  $X = \{X_1, \dots, X_n\}$ , each  $X_i$  associated with a **domain of values**  $D_i$ , and a set of **constraints**  $C = \{C_1, \dots, C_\ell\}$ . A constraint  $C_i(X_{i_1}, \dots, X_{i_j})$  is a subset of the Cartesian product  $D_{i_1} \times \dots \times D_{i_j}$ . It specifies the compatible values of the variables  $X_{i_1}, \dots, X_{i_j}$ . A **solution** is an assignment of values to the variables such that all constraints are satisfied. Solving a CSP means finding one, several or all solutions (if they exist).

If all domains  $D_i$  are finite, then the corresponding CSP is classified as a **finite CSP** (FCSP). A well-known FCSP is the  $n$ -queens problem. It consists in placing  $n$  queens on a  $n$ -by- $n$  chessboard such that no queen is in check with any other. In general, the main difficulty of solving a FCSP is that the set of potential solutions can be extremely large. For example, if each queen is put on a different line of the chessboard, there are  $n^n$  possible solutions to the  $n$ -queens problem.

Three constraint satisfaction problems are discussed in the following subsections. The example of Subsection 4.1 is a FCSP, i.e. the variables are restricted to finite domains. In contrast, the examples in Subsection 4.2 and 4.3 are general CSPs. Note that there is no uncertainty in traditional CSPs. The aim of a CSP is rather obtaining values for certain variables than getting arguments for hypotheses.

### 4.1 Age of Children

Consider the following example of a FCSP:

“Peter has three children. His friend Paul wants to know their ages and he asks Peter to tell him. Peter answers that the sum of their ages equals 13 and the product is exactly 56.”

This story can easily be modeled using ABEL. The model consists of three variables  $X = \{X_1, X_2, X_3\}$  with domains  $D = \{D_1, D_2, D_3\}$ ,  $D_i = \mathbb{N}$ . Each variable represents the age of one of the children. Two statements (constraints) are needed to model the information given by Peter, and one statement is necessary to express that  $X_1$  represents the age of the youngest child and  $X_3$  the age of the oldest. The following ABEL code models Peter’s FCSP:

```
(tell
  (var x1 x2 x3 integer[0..])

  (= (+ x1 x2 x3) 13)
  (= (* x1 x2 x3) 56)
```

```
(<= x1 x2)
(<= x2 x3))
```

In this example, it may be interesting to ask for the values of the variables  $X_1$ ,  $X_2$ , and  $X_3$ :

```
(ask x1 x2 x3)
```

Obviously, `(and (= x1 2)(= x2 4)(= x3 7))` would be a solution for this query. In fact, it is the only solution. If Peter had told Paul that the product of the ages were 36 (instead of 56), then two solutions would exist, namely `(and (= x1 2)(= x2 2)(= x3 9))` and `(and (= x1 1)(= x2 6)(= x3 6))`.

## 4.2 Pythagoras

As mentioned above, the main difficulty in solving a FCSP is the very large set of possible solutions. However, a FCSP has always a finite number of possible solutions. This is not the case for a general CSP where the variables are not restricted to finite domains. Solving a CSP can therefore be even harder. Nevertheless, for some cases there are algorithms to solve general CSP's very efficiently.

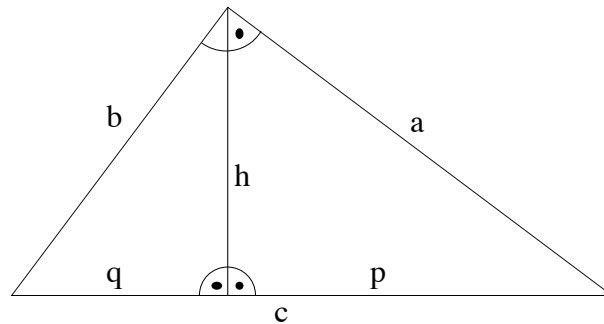


Figure 4.1: A right-angled triangle.

For example, consider the right-angled triangle shown in Figure 4.1. There is a set of variables  $X = \{a, b, c, q, p, h\}$  each of them with  $\mathbb{R}^+$  as its domain. The following equations describe the geometrical relations between the different variables:

$$a^2 + b^2 = c^2, \quad (4.1)$$

$$p^2 + h^2 = a^2, \quad (4.2)$$

$$q^2 + h^2 = b^2, \quad (4.3)$$

$$c * p = a^2, \quad (4.4)$$

$$c * q = b^2, \quad (4.5)$$

$$q * p = h^2. \quad (4.6)$$

Using ABEL, these equations can be written as follows:

```
(tell
  (var a b c q p h real[0..])

  (= (+ (sqr a) (sqr b)) (sqr c))
  (= (+ (sqr p) (sqr h)) (sqr a))
  (= (+ (sqr q) (sqr h)) (sqr b))
  (= (* c p) (sqr a))
  (= (* c q) (sqr b))
  (= (* q p) (sqr h)))
```

Now, suppose that the values  $a = 4$  and  $b = 3$  are observed. Given the values for two of the six variables, the remaining values are determined.

```
(observe
  (= a 4)
  (= b 3))

(ask c q p h)
```

In the framework of CSP a constraint is called **triggered** if the value of one of its variables is determined in function of the others. In the example above, the first constraint is triggered because the values of  $a$  and  $b$  are known. Therefore, the value of  $c$  is determined by  $c = \sqrt{a^2 + b^2}$ . Knowing the value of  $c$ , the constraints (4.4) and (4.5) are also triggered and the values of the variables  $p$  and  $q$  are determined by  $p = \frac{a^2}{c}$  and  $q = \frac{b^2}{c}$ . Finally, the value of  $h = \sqrt{q * p}$  can be obtained by the last constraint. The final solution is given by  $a = 4$ ,  $b = 3$ ,  $c = 5$ ,  $q = 1.8$ ,  $p = 3.2$ , and  $h = 2.4$ .

A framework called **Arithmetic Constraint Management System** (ACMS) has been proposed in (Donnet-Portenier, 1993). It can be used to solve CSP's consisting of arithmetic constraints. In some cases, the ACMS framework is not very helpful. If, for example, the values  $a = 4$  and  $q = 1.8$  are observed, then it is not possible to compute the values of the other variables, because no constraint is triggered in this case. Note that the solution is still the same:  $a = 4$ ,  $b = 3$ ,  $c = 5$ ,  $q = 1.8$ ,  $p = 3.2$ , and  $h = 2.4$ .

### 4.3 Physical Process

ABEL can also be useful to solve problems in the domain of physics. Consider the following simple situation:

A stone with a mass of  $m = 1\text{kg}$  is attached on a wire in a height of  $h = 10\text{m}$ . Calculate the following entities:

- the potential energy  $E_P$  of the stone (in Joule);
- the speed  $v_1$  of the stone after the wire is cut and the stone hits the ground (in m/s);
- the speed  $v_2$  as above but in km/h;
- the duration  $t$  until the stone reaches the ground (in seconds).

Several equations define the relationships between the physical variables. In the following,  $g$  stands for the gravity of the earth, i.e.  $g = 9.81\text{m/s}^2$ .

$$E_P = m * g * h, \quad (4.7)$$

$$E_K = 0.5 * m * v_1^2, \quad (4.8)$$

$$E_P = E_K, \quad (4.9)$$

$$v_1 = g * t, \quad (4.10)$$

$$v_2 = 3.6 * v_1. \quad (4.11)$$

Usually, these equations have to be rewritten as follows:

$$E_P = m * g * h, \quad (4.12)$$

$$v_1 = \sqrt{2 * g * h}, \quad (4.13)$$

$$v_2 = 3.6 * v_1, \quad (4.14)$$

$$t = \left(\frac{v_1}{g}\right). \quad (4.15)$$

Using the language ABEL or the ACMS framework, this last step is not necessary. In ABEL, this example can be encoded very easily: and natural:

```
(tell
  (var g      9.81)
  (var h m    real)
  (var E_p E_k real)
  (var v1 v2  real)
  (var time  real)

  (= E_p (* m g h))
  (= E_k (* 0.5 m (sqr v1)))
  (= E_p E_k)
  (= v2 (* 3.6 v1))
  (= v1 (* g time)))

(observe
  (= m 1)
  (= h 10))
```

The result of the following query will be  $E_P = 98.1\text{N}$ ,  $v_1 = 14.007\text{m/s}$ ,  $v_2 = 50.426\text{km/h}$ , and  $t = 1.428\text{s}$ .

```
(ask E_p v1 v2 time)
```

Now, suppose that values  $m = 0.2\text{kg}$ , and  $h = 50\text{m}$  are given for the same situation. In ABEL, only the observations have to be changed. The basic knowledge base remains the same:

```
(empty observe)
```

```
(observe  
  (= m 0.2)  
  (= h 50))  
  
(ask E_p v_1 v_2 time)
```

The results are now  $E_P = 98.1\text{N}$ ,  $v_1 = 31.321\text{m/s}$ ,  $v_2 = 112.755\text{km/h}$ , and  $t = 3.193\text{s}$ .

## 5 Diagnostics in Technical Systems

This section discusses four different problems of diagnostics in technical systems. Three of them consist of simple digital and mathematical networks, which can easily be generalized to more complex ones, whereas the fourth introduces flows into the diagnostic process. In fact, each example deals with some sort of interacting components. The messages sent between the components are either binary values, numerical values, or flows. In this type of problems, the aim is to find possible explanations for an incorrect system behavior or for failure states of some components.

### 5.1 Digital Circuits

Consider the digital circuit of a binary adder shown in Figure 5.1. The system  $S$  consists of five components, namely the logical gates  $and_1$ ,  $and_2$ ,  $xor_1$ ,  $xor_2$  and  $or$ . The in- and output values of the components are the logical values true and false (or interpreted as 1 and 0).

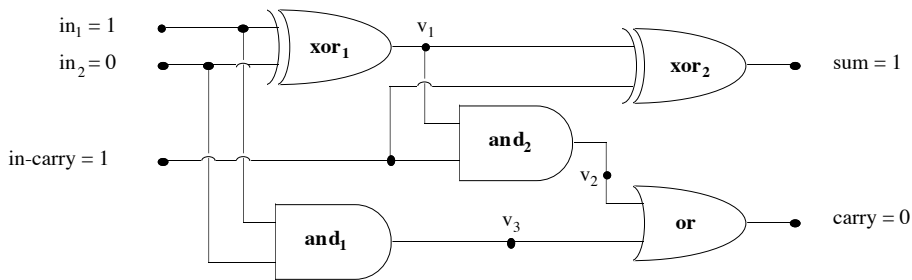


Figure 5.1: Digital circuit of a binary adder.

Suppose that each component has exactly two different operating modes, which is characterized by a predicate, for example  $ok\text{-}and_1$  for the component  $and_1$ . Figure 5.2 shows an  $and$ -gate with two inputs  $in_1$  and  $in_2$  and one output  $out$ .

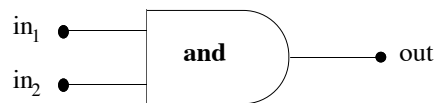


Figure 5.2: A simple  $and$ -gate.

Depending on the operation mode  $ok\text{-}and$ , the relations between these three input and output variables are as follows:

- $ok\text{-}and$  is true: this means that the  $and$ -gate is working correctly, i.e.  $out = in_1 \wedge in_2$ ;

- *ok-and* is false: this means that the and-gate is not working correctly, i.e.  $out = \sim(in_1 \wedge in_2)$ .

All other components of the system are treated similarly. Suppose that the probability for a correct functioning component is 0.99 for *and*-gates, 0.98 for *or*-gates, and 0.95 for *xor*-gates. For each type of component, a corresponding module is defined in ABEL:

```
(tell
  (module AND-GATE ((var in1 in2 out binary)
    (ass ok binary 0.99))
    (-> ok (<-> out (and in1 in2)))
    (-> (not ok) (not (<-> out (and in1 in2)))))

  (module OR-GATE ((var in1 in2 out binary)
    (ass ok binary 0.98))
    (-> ok (<-> out (or in1 in2)))
    (-> (not ok) (not (<-> out (or in1 in2)))))

  (module XOR-GATE ((var in1 in2 out binary)
    (ass ok binary 0.95))
    (-> ok (<-> out (xor in1 in2)))
    (-> (not ok) (not (<-> out (xor in1 in2)))))
```

To obtain the entire network, these modules can now be applied following the topology of Figure 5.1.

```
(tell
  (var ok-full-adder binary)

  (AND-GATE in1 in2 v3 ok-and-1)
  (AND-GATE in-carry v1 v2 ok-and-2)
  (XOR-GATE in1 in2 v1 ok-xor-1)
  (XOR-GATE in-carry v1 sum ok-xor-2)
  (OR-GATE v2 v3 carry ok-or)
  (<-> ok-full-adder (and ok-and-1 ok-and-2 ok-xor-1 ok-xor-2 ok-or)))
```

Note that it is not necessary to declare some of the variables used in the model explicitly. The types of the variables are determined according to their first occurrence in a module call. For example, *ok-and*<sub>1</sub> is a binary assumption with probability 0.99 as defined in the module *and-gate*.

Finally, the ABEL model is completed by the values of the in- and output variables of the system as shown in Figure 5.1:

```
(observe in1 (not in2) in-carry sum (not carry))
```

If every component is assumed to work correctly, then the actual observations are in conflict with the expected behavior of the system. Therefore, one or several components must be faulty, and the question is which ones.

An **explanation** for the system's behavior is a combination of faulty and correct components, such that the observed input and output values are not in contradiction with the knowledge base (Haenni, 1996). Possible explanations are produced by the following query:

```
(ask (sp tautology))
```

According to (Haenni, 1996), the following results are obtained:

$$\begin{aligned} &(ok\text{-}and_1 \wedge ok\text{-}and_2 \wedge \sim ok\text{-}xor_1 \wedge ok\text{-}xor_2 \wedge ok\text{-}or) \vee \\ &(ok\text{-}and_2 \wedge ok\text{-}xor_1 \wedge \sim ok\text{-}xor_2 \wedge \sim ok\text{-}or) \vee \\ &(ok\text{-}and_1 \wedge \sim ok\text{-}and_2 \wedge ok\text{-}xor_1 \wedge \sim ok\text{-}xor_2 \wedge ok\text{-}or) \vee \\ &(\sim ok\text{-}and_2 \wedge \sim ok\text{-}xor_1 \wedge ok\text{-}xor_2 \wedge \sim ok\text{-}or) \vee \\ &(\sim ok\text{-}and_1 \wedge \sim ok\text{-}xor_1 \wedge ok\text{-}xor_2 \wedge \sim ok\text{-}or) \vee \\ &(\sim ok\text{-}and_1 \wedge ok\text{-}xor_1 \wedge \sim ok\text{-}xor_2 \wedge \sim ok\text{-}or) \end{aligned}$$

There are six different explanations. The first explanation, for example, means that all components except  $xor_1$  are intact.

Sometimes, it may also be interesting to treat numerical queries. Examples:

```
(ask
  (dsp ok-xor-1)
  (dsp ok-and-1))
```

The corresponding results are 0.0298 and 0.9996, respectively (Haenni, 1996). Therefore,  $xor_1$  is probably the faulty component.

## 5.2 Multiple Measures

In the previous subsection, the input- and output values have only been measured once. Now, the case of several consecutive measures of the same variables is treated. The system is assumed to be stable in the following sense: if the variables are measured at time  $t = 1$  when some (unknown) components are not working correctly, then for any "later" measures, e.g. at time  $t = 2, 3, \dots$ , exactly the **same** components are still faulty. This idea of consistency of misbehavior over time (non-intermittency) was treated in the present context in (Raiman *et al.*, 1991).

Consider a network with two components, an *or*-gate and an *xor*-gate, connected as shown in Figure 5.3 (Raiman *et al.*, 1991). Suppose that each component has several operating modes. Its in- and outputs are measured twice. Therefore, there are two sets  $O_1$  and  $O_2$  of observations.

The components are modeled according to the previous subsection. The only difference is, that each component has four different operating modes, one correct mode `ok` and three failure modes `error1`, `error2`, and `error3`. A corresponding type called `mode-type` is introduced. Suppose the probabilities of 0.9, 0.05, 0.02, and 0.03, respectively, for the modes of both components.

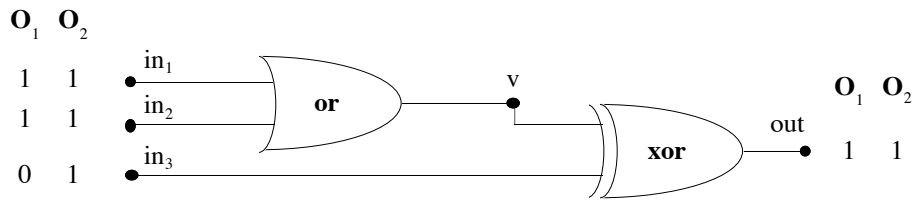


Figure 5.3: Multiple measures in the same network.

```
(tell
  (type mode-type (ok error1 error2 error3))

  (module OR-GATE ((var in1 in2 out binary)
    (ass mode mode-type (0.90 0.05 0.02 0.03)))
    (-> (= mode ok) (<-> out (or in1 in2)))
    (-> (= mode error1) out)
    (-> (= mode error2) (not out))
    (-> (= mode error3) (<-> out (not (or in1 in2)))))

  (module XOR-GATE ((var in1 in2 out binary)
    (ass mode mode-type (0.90 0.05 0.02 0.03)))
    (-> (= mode ok) (<-> out (xor in1 in2)))
    (-> (= mode error1) out)
    (-> (= mode error2) (not out))
    (-> (= mode error3) (<-> out (not (xor in1 in2)))))
```

Next, the modules are applied following the structure in Figure 5.3:

```
(tell
  (OR-GATE in1 in2 v mode-or)
  (XOR-GATE v in3 out mode-xor))
```

The two sets of observations  $O_1$  and  $O_2$  of Figure 5.3 can be modeled in ABEL using the construct `observe` and the keys `:measure-1` and `:measure-2`:

```
(observe :measure-1 in1 in2 (not in3) out)
(observe :measure-2 in1 in2 in3 out)
```

The following queries can be used to obtain explanations of the system's behavior as well as the degrees of support that the components are working properly:

```
(ask
  (sp tautology)
  (dsp (= ok mode-xor))
  (dsp (= ok mode-or))
```

If non-intermittency is assumed, then it can be concluded (Raiman *et al.*, 1991) that

$$p(\{\text{mode-xor} \neq \text{ok}\} | \{O_1, O_2\}) = 1.0,$$

$$p(\{\text{mode-or} \neq \text{ok}\} | \{O_1, O_2\}) \leq 0.5.$$

If non-intermittency is not assumed, then it can be concluded that

$$\begin{aligned} p(\{\text{mode-xor} \neq \text{ok}\}|\{O_1, O_2\}) &= 0.5, \\ p(\{\text{mode-or} \neq \text{ok}\}|\{O_1, O_2\}) &= 0.5. \end{aligned}$$

### 5.3 Arithmetical Network

In the previous subsections, only networks with binary variables have been considered. Now, the more general case of integer variables will be considered. The following example has been introduced in (Davis, 1984). Subsequently, it was used in several papers on model-based diagnostics (Reiter, 1987; de Kleer & Williams, 1987; Kohlas *et al.*, 1996).

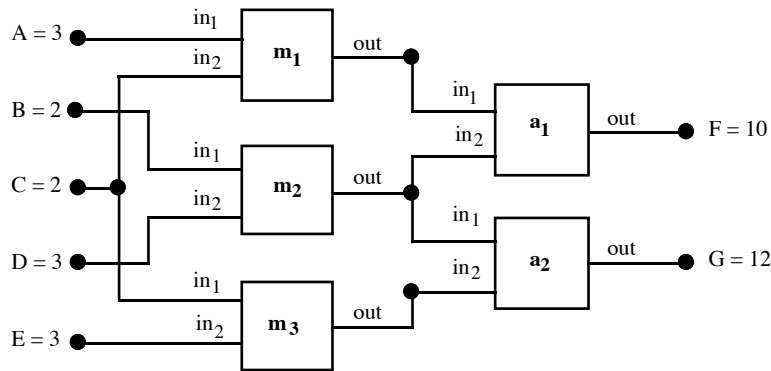


Figure 5.4: An arithmetical network.

The network consists of three multipliers  $m_1$ ,  $m_2$ ,  $m_3$ , and two adders  $a_1$ ,  $a_2$ . These components are connected as shown in Figure 5.4.

If only one failure mode is assumed, then a binary variable  $ok$  can be used to represent the two modes of the components. Figure 5.5 shows, for example, an adder component with inputs  $in_1$ ,  $in_2$  and one output  $out$ :

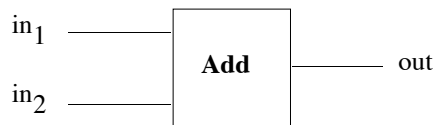


Figure 5.5: A simple adder.

The behavior of the component can be expressed by two material implications:

$$\begin{aligned} ok &\rightarrow (out = in_1 + in_2) \\ \sim ok &\rightarrow \sim(out = in_2 + in_2). \end{aligned} \tag{5.1}$$

In ABEL, a module is built for each type of component. The reliability of a component is modeled by an assumption `ok`. Assume the probability 0.95 for adders and 0.97 for multipliers.

```
(tell
  (module ADDER ((var in1 in2 out integer)
                 (ass ok binary 0.97))
    (-> ok (= out (+ in1 in2)))
    (-> (not ok) (/= out (+ in1 in2))))

  (module MULTIPLIER ((var in1 in2 out integer)
                      (ass ok binary 0.95))
    (-> ok (= out (* in1 in2)))
    (-> (not ok) (/= out (* in1 in2))))))
```

Next, the modules are applied following the topology of Figure 5.4.

```
(tell
  (var ok-network binary)

  (MULTIPLIER a c x ok-m1)
  (MULTIPLIER b d y ok-m2)
  (MULTIPLIER c e z ok-m3)
  (ADDER x y f ok-a1)
  (ADDER y z g ok-a2)
  (<-> ok-network (and ok-m1 ok-m2 ok-m3 ok-a1 ok-a2)))
```

The global variable `ok-network` reflects the state of the whole system. Observations are added as specified in Figure 5.4.

```
(observe
  (= a 3) (= b 2) (= c 2) (= d 3) (= e 3) (= f 10) (= g 12))
```

The observed values imply that some components must be faulty and the question is which ones. The following query can be helpful to obtain possible explanations:

```
(ask (sp tautology))
```

The result consists of four different configurations of faulty components (Kohlas & Monney, 1995):

$$\sim ok-m_1 \vee \sim ok-a_1 \vee (\sim ok-m_2 \wedge \sim ok-m_3) \vee (\sim ok-m_2 \wedge \sim ok-a_2).$$

## 5.4 Means-End Models

A **multi-level flow model** (MFM) based on explicit **means-end** is presented in (Larsson, 1994). This subsection discusses an example of this article and it shows how its methods can be implemented in ABEL.

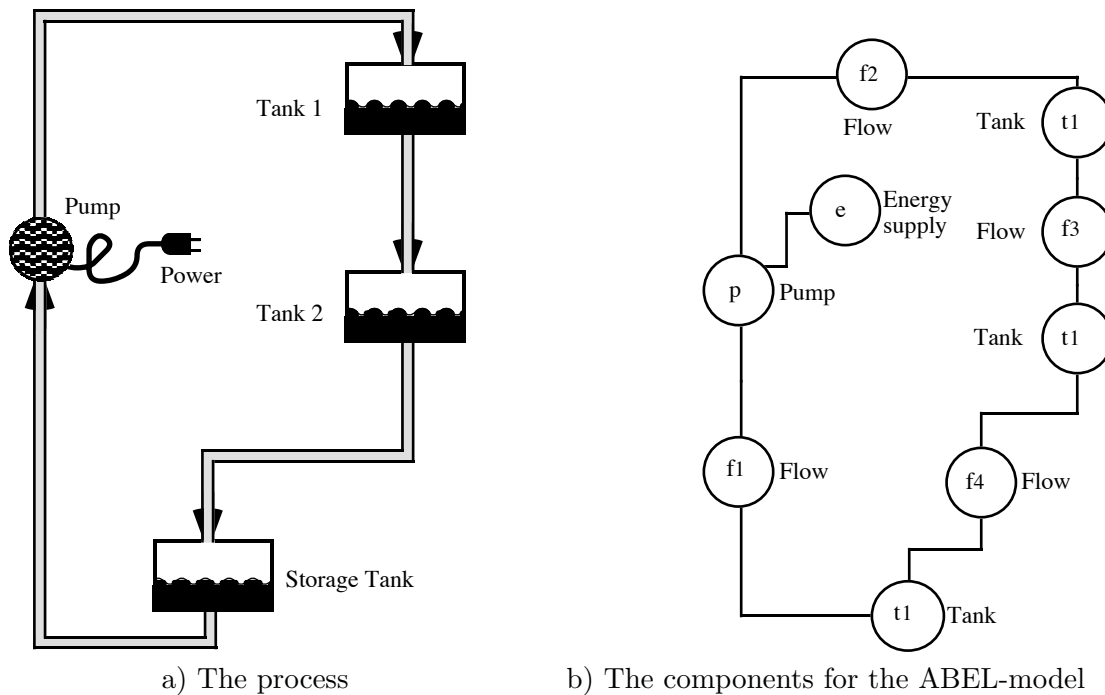


Figure 5.6: The tree tank example

The example consists of a tree tank model shown in Figure 5.6a. Water is pumped from a storage tank through the pump into tank 1. From tank 1 it flows down into tank 2 and then back into the storage tank. The main goal is to keep the level of tank 1 correct.

First, types and global constants are defined (flows are measured in  $10^{-6} \cdot \text{m}^3 \cdot \text{s}^{-1}$ ):

```
(tell
  (type flow-value real[0..1000])
  (type volume-value real[0..1000])
  (type change-value real[-1000..1000])
  (type derivation-value real)
  (var **highflow** 100)
  (var **lowflow** 0)
  (var **highvolume** 1000)
  (var **lowvolume** 0)
  (var **epsilon** 0.02)) ;tolerance (in %)
```

Figure 5.6b shows the structure of the example. Three kinds of modules are considered:

- flows  $f_1, \dots, f_4$ ;
- storage tanks  $t_1, \dots, t_3$  (for simplicity, consider only one kind of storage tank);
- pump and energy supply, which will be modeled in one component.

The flow component consists of two connectors, `in` and `out`, describing the flow of the water at the “beginning” of the flow and at its “end”. The variable `measured` represents the actual value of the flow (if available). Several variables deal with different states of the module, `ok`, `leak`, `highflow`, `lowflow`. Note that the error states are not exclusive. The global variable `**epsilon**` is used to ignore small measurement errors, i.e. the error `leak` occurs only if the inequality

$$\text{in} > \text{out} * (1 + \text{**epsilon**})$$

is true. The local variable `measure-ok` is true if the measured value (`measured`) is equal to the in- or outflow of the component. If `measure-ok` is false, then the measured value is not correct. Note that this variable is specified to be an assumption with probability 0.9.

```
(tell
  (module FLOW ((var input output measured real)
                (var ok binary)
                (ass leak highflow binary 0.01)
                (ass lowflow binary 0.05))
    (ass measure-ok binary 0.9)
    (<-> ok (and (not leak) (not highflow) (not lowflow)))
    (<-> leak (> input (* output (+ 1 **epsilon**))))
    (<-> highflow (or (> input **highflow**
                      (> output **highflow**)))
    (<-> lowflow (or (< input **lowflow**
                     (< output **lowflow**)))
    (-> measure-ok (or (= measured output)
                       (= measured input))))
```

The module of a storage tank has an in- and an out-flow (`in`, `out`), a variable indicating the measured volume (`measured-volume`), a variable `measured-derivation` indicating the derivation of the volume in time (if this is possible). The system states are represented by the variables `ok`, `lowvolume`, `highvolume`, `leak` and `fill` (c.f. previous module).

```
(tell
  (module TANK ((var input output flow-value)
                (var measured-volume volume-value)
                (var measured-derivation change-value)
                (var ok binary)
                (ass lowvolume leak binary 0.01)
                (ass highvolume binary 0.05)
                (ass fill binary 0.02))
    (var volume volume-value)
    (var derivation derivation-value)
    (ass volume-measure-ok binary 0.96)
    (ass derivation-measure-ok binary 0.85)
    (<-> ok (and (not lowvolume) (not highvolume)
                 (not leak) (not fill)))
    (<-> highvolume (> volume **highvolume**))
    (<-> lowvolume (< volume **lowvolume**))
    (<-> leak (< (* output (+ 1 **epsilon**)))
```

```

        (- input derivation)))
    (<-> fill (> (* output (+ 1 **epsilon**))
              (- input derivation)))
    (-> volume-measure-ok (= measured-volume volume))
    (-> derivation-measure-ok (= measured-derivation derivation)))

```

The last module deals with the pump and its energy supply:

```

(tell
  (module PUMP ((var input output flow-value)
                (var ok power leak binary)
                (ass switch-on binary 0.95)
                (ass power-supplied binary 0.99))
    (<-> ok (and power (not leak)))
    (<-> power (> output 0))
    (<-> leak (> input (* output (+ 1 **epsilon**))))
    (<-> power (and switch-on power-supplied))))

```

According to Figure 5.6b, the modules can now be applied:

```

(tell
  (FLOW in-f1 out-f1 measure-f1 ok-f1 leak-f1 highflow-f1 lowflow-f1)
  (FLOW in-f2 out-f2 measure-f2 ok-f2 leak-f2 highflow-f2 lowflow-f2)
  (FLOW in-f3 out-f3 measure-f3 ok-f3 leak-f3 highflow-f3 lowflow-f3)
  (FLOW in-f4 out-f4 measure-f4 ok-f4 leak-f4 highflow-f4 lowflow-f4)

  (TANK in-t1 out-t1 measured-volume-t1 measured-derivation-t1
        ok-t1 lowvolume-t1 highvolume-t1 leak-t1 fill-t1)
  (TANK in-t2 out-t2 measured-volume-t2 measured-derivation-t2
        ok-t2 lowvolume-t2 highvolume-t2 leak-t2 fill-t2)
  (TANK in-t3 out-t3 measured-volume-t3 measured-derivation-t3
        ok-t3 lowvolume-t3 highvolume-t3 leak-t3 fill-t3)

  (PUMP in-p out-p ok-p power-p leak-p switch-on-p power-supplied-p)

  (= out-f1 in-p)
  (= out-p in-f2)
  (= out-f2 in-t1)
  (= out-t1 in-f3)
  (= out-f3 in-t2)
  (= out-t2 in-f4)
  (= out-f4 in-t3)
  (= out-t3 in-f1))

```

Finally, the available flow measures and the observation that power is supplied are encoded as follows:

```

(observe
  (= measure-f1 100)
  (= measure-f3 200)
  power-supplied-p)

```

Based on the MFM, there are the following three methods for diagnostic reasoning (Larsson, 1994):

- **Measurement validation:** which measurement is correct and which not;
- **Alarm analysis:** which alarm is primary, which one is secondary;
- **Fault Diagnosis:** which components to check and/or replace (guidelines for the operator).

In ABEL, measurement validation is possible by introducing assumptions (e.g. `measure-ok` in the module `FLOW`) which are true if the measured values are correct. Second, alarm analysis is done by inspecting the symbolic supports of different alarms. Finally, fault diagnosis can be implemented using techniques known as “best-next-measurements”, but this is not (yet) possible in ABEL.

## 6 Causal Networks

Causal or inference networks are used in a number of areas to represent patterns of influence among variables. They consist of connected causal relations. A causal relation can be regarded as a rule of the form “if *cause*, then *effect*”. Examples of causal relations are: “if there is some rain tonight, then the grass will be wet tomorrow”, “if the next bingo-number is 7, then my wife wins 100 dollars”, or “if my father returns late at night, then my mother gets angry”. Thus, causality can be seen as any natural ordering in which knowledge of an event influences opinion concerning another event. This influence can be logical, physical, temporal, or simply conceptual (Lauritzen & Spiegelhalter, 1988).

Causes and effects can be modeled as variables. These variables are the nodes of the network. Causal relations between two variables are the edges. Figure 6.1 depicts a causal network consisting of only two nodes  $C$  and  $E$ . Usually, nodes are represented as circles and causal relations between them as arrows (Kohlas & Monney, 1995).

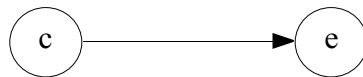


Figure 6.1: Causal relation between variables  $c$  and  $e$ .

The concept of directed acyclic graphs is an appropriate mathematical structure to describe causal networks. A pair  $\mathcal{G} = (\mathcal{N}, \mathcal{E})$  is called **directed graph** if  $\mathcal{N}$  is a set of nodes and  $E \in \mathcal{E}$  are **directed edges**, i.e. pairs  $(n_i, n_j)$  of nodes,  $n_i, n_j \in \mathcal{N}$ . Directed edges  $(n_i, n_j)$  are depicted as arrows from  $n_i$  to  $n_j$ .

In a directed graph  $\mathcal{G} = (\mathcal{N}, \mathcal{E})$  a **directed chain** from  $n_1$  to  $n_{q+1}$  is a sequence of edges  $E_1, \dots, E_q$  such that  $E_k = (n_k, n_{k+1})$  for  $k = 1, \dots, q$ .  $n_1$  is called the **initial endpoint** and  $n_{q+1}$  the **terminal endpoint** of the directed chain. A **directed cycle** is a directed chain in which no edge appears twice in the sequence of edges and in which the two endpoints are the same. If a directed graph  $\mathcal{G}$  has no cycles, then it is called **directed acyclic graph**. Figure 6.2 shows two directed graphs. The one on the left is acyclic, the one on the right is not.

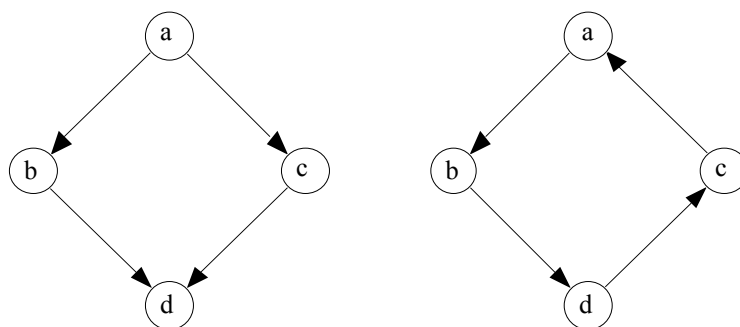


Figure 6.2: Two directed graphs.

Generally, a variable of a causal network can have any domain. In the following, only problems with binary variables are considered, i.e. nodes are variables with values 1 (true) and 0 (false). If the cause  $c$  of a causal relation is true, then the effect  $e$  of the relation is also true. This can be expressed as a logical implication  $c \rightarrow e$ . Binary variables of a causal network are therefore represented by propositional symbols.

Consider the directed acyclic graph on the left side of Figure 6.2. If it is interpreted as a causal network, then  $a$ , for example, is cause for  $b$  and  $c$ , whereas  $b$  and  $c$  are causes for  $d$ . Such multiple causes or multiple effects can be interpreted as conjunctions or as disjunctions. Thus, four different cases are possible:

- (1) If cause  $c$  is true, then all effects  $e_1$  to  $e_n$  are also true:

$$c \rightarrow e_1, c \rightarrow e_2, \dots, c \rightarrow e_n. \quad (6.1)$$

- (2) If cause  $c$  is true, then at least one effect  $e_1$  to  $e_n$  is also true:

$$c \rightarrow e_1 \vee e_2 \vee \dots \vee e_n. \quad (6.2)$$

- (3) If at least one of the causes  $c_1$  to  $c_n$  is true, then effect  $e$  is also true; besides  $c_1$  to  $c_n$  there are no other causes of  $e$ :

$$\begin{aligned} c_1 \rightarrow e, c_2 \rightarrow e, \dots, c_n \rightarrow e, \\ e \rightarrow c_1 \vee c_2 \vee \dots \vee c_n. \end{aligned} \quad (6.3)$$

- (4) If all causes  $c_1$  to  $c_n$  are true, then effect  $e$  is also true; this is the only way to cause  $e$ :

$$\begin{aligned} c_1 \wedge c_2 \wedge \dots \wedge c_n \rightarrow e, \\ e \rightarrow c_1, e \rightarrow c_2, \dots, e \rightarrow c_n. \end{aligned} \quad (6.4)$$

As shown in Figure 6.3, different graphical representations are used to distinguish these cases.

In (3) and in (4) it is assumed that besides  $c_1$  to  $c_n$  there are no other causes for  $e$ . This is called **accountability condition** (Pearl, 1988). It states that the model explicitly contains all causes that may produce  $e$ . Logically, this can be expressed through implications from  $e$  to  $c_i$ . Such implications pointing to the opposite direction are important especially for diagnostics analyses.

Some or all of the causal relations of a causal network may be uncertain. In such cases, effects are only caused under some circumstances. This type of uncertainty can be expressed by assumptions. A rule like “if cause  $c$  is true, then effect  $e$  is true under some circumstances” can be transformed into  $c \wedge a \rightarrow e$ .  $a$  represents the assumption that the necessary circumstances are present. If all causal relations of Figure 6.3 are assumed to be uncertain, then the following logical implications are produced:

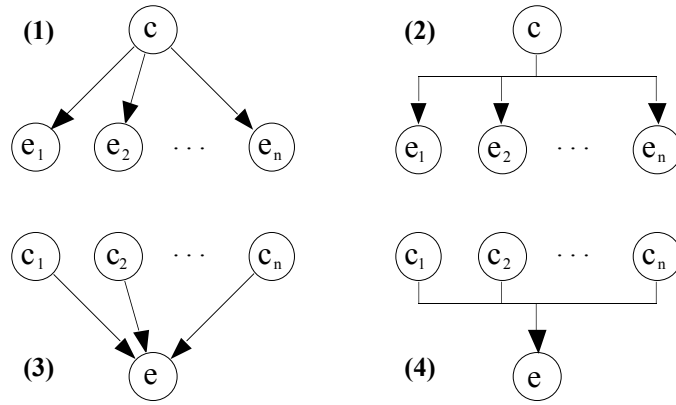


Figure 6.3: Multiple causes and multiple effects.

(1') If cause  $c$  is true, then – under some circumstances  $a_i$  – effect  $e_i$  is also true:

$$c \wedge a_1 \rightarrow e_1, c \wedge a_2 \rightarrow e_2, \dots, c \wedge a_n \rightarrow e_n. \quad (6.5)$$

(2') If cause  $c$  is true, then – under some circumstances  $a$  – at least one effect  $e_1$  to  $e_n$  is also true:

$$c \wedge a \rightarrow e_1 \vee e_2 \vee \dots \vee e_n. \quad (6.6)$$

(3') If cause  $c_i$  is true, then – under some circumstances  $a_i$  – effect  $e$  is also true; besides  $c_1$  to  $c_n$  there are no other causes of  $e$ :

$$\begin{aligned} c_1 \wedge a_1 \rightarrow e, c_2 \wedge a_2 \rightarrow e, \dots, c_n \wedge a_n \rightarrow e, \\ e \rightarrow (c_1 \wedge a_1) \vee (c_2 \wedge a_2) \vee \dots \vee (c_n \wedge a_n). \end{aligned} \quad (6.7)$$

(4') If all causes  $c_1$  to  $c_n$  are true, then – under some circumstances  $a$  – effect  $e$  is also true; this is the only way to cause  $e$ :

$$\begin{aligned} c_1 \wedge c_2 \wedge \dots \wedge c_n \wedge a \rightarrow e, \\ e \rightarrow c_1, e \rightarrow c_2, \dots, e \rightarrow c_n, e \rightarrow a. \end{aligned} \quad (6.8)$$

If in case (3') other (unknown) causes of  $e$  are possible, then an additional assumption  $a_{n+1}$  has to be introduced.  $a_{n+1}$  describes the uncertainty whether unknown causes of  $e$  are present.

(3'') If cause  $c_i$  is true, then – under some circumstances  $a_i$  – effect  $e$  is also true; besides  $c_1$  to  $c_n$  other causes of  $e$  are possible:

$$\begin{aligned} c_1 \wedge a_1 \rightarrow e, c_2 \wedge a_2 \rightarrow e, \dots, c_n \wedge a_n \rightarrow e, a_{n+1} \rightarrow e \\ e \rightarrow (c_1 \wedge a_1) \vee (c_2 \wedge a_2) \vee \dots \vee (c_n \wedge a_n) \vee a_{n+1}. \end{aligned} \quad (6.9)$$

According to this prescription, all relations of a causal network can be transformed into set of material implications. This system of logical formulas can be used to judge hypotheses about variables of the causal network. Arguments for such hypotheses are expressions consisting of assumptions about the uncertain circumstances of the causal relations. They can be used to make **predictions** of effects or to give **explanations** (diagnoses) of observed effects.

### 6.1 Medical Diagnosis

A doctor has to decide whether a patient, who complains about shortness-of-breath, suffers from bronchitis, lung cancer, or tuberculosis. This fictitious example was first mentioned in (Lauritzen & Spiegelhalter, 1988) in order to illustrate the use of Bayesian networks. Here, the same example is applied to the field of assumption-based reasoning.

The doctor's medical knowledge helps him to find the causes of the patient's symptoms. His knowledge is composed of what he learned at medical school and of what he knows from his practical experience. It can be summarized as follows (Lauritzen & Spiegelhalter, 1988):

“Shortness-of-breath (dyspnoea) may be due to tuberculosis, lung cancer, or bronchitis, or none of them, or more than one of them. A recent visit to an under-developed country increases the chances of tuberculosis, while smoking is known to be a risk factor for both lung cancer and tuberculosis. The results of a single chest X-ray do not discriminate between lung cancer and tuberculosis; nor does the presence or absence of dyspnoea.”

In Figure 6.4 this small piece of fictitious medical knowledge is shown as causal network. Direction of causality is from top to bottom. Note that some effects have more than one cause and that some causes produce more than one effect. Thus, multiple causes are interpreted as disjunction, whereas multiple effects are interpreted as conjunction.

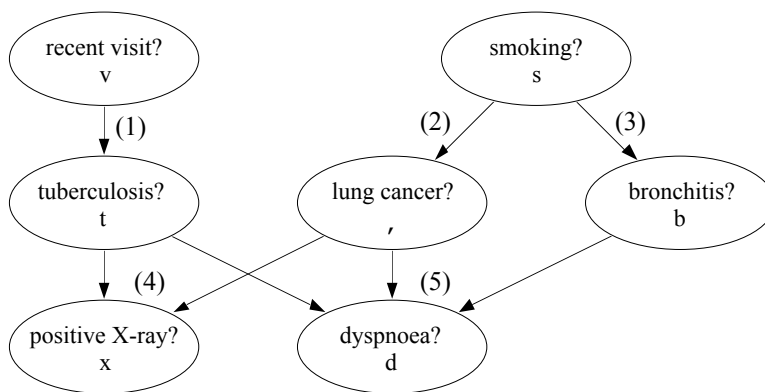


Figure 6.4: Causal network for the fictitious example.

All causal relations of Figure 6.4 are uncertain. Furthermore, it is assumed that for all effects there are other (unknown) causes. Thus, the strategy described in (3'') has to be used in order to obtain the following sets of logical implications:

$$(1) \quad v \wedge a_1 \rightarrow t, \quad a_2 \rightarrow t, \quad t \rightarrow (v \wedge a_1) \vee a_2; \quad (6.10)$$

$$(2) \quad s \wedge a_3 \rightarrow \ell, \quad a_4 \rightarrow \ell, \quad \ell \rightarrow (s \wedge a_3) \vee a_4; \quad (6.11)$$

$$(3) \quad s \wedge a_5 \rightarrow b, \quad a_6 \rightarrow b, \quad b \rightarrow (s \wedge a_5) \vee a_6; \quad (6.12)$$

$$(4) \quad t \wedge a_7 \rightarrow x, \quad \ell \wedge a_8 \rightarrow x, \quad a_9 \rightarrow x, \\ x \rightarrow (t \wedge a_7) \vee (\ell \wedge a_8) \vee a_9; \quad (6.13)$$

$$(5) \quad t \wedge a_{10} \rightarrow d, \quad \ell \wedge a_{11} \rightarrow d, \quad b \wedge a_{12} \rightarrow d, \quad a_{13} \rightarrow d \\ d \rightarrow (t \wedge a_{10}) \vee (\ell \wedge a_{11}) \vee (b \wedge a_{12}) \vee a_{13}. \quad (6.14)$$

If the probabilities

$$p(a_1) = 0.1, p(a_2) = 0.01, p(a_3) = 0.2, p(a_4) = 0.1, p(a_5) = 0.3, \\ p(a_6) = 0.1, p(a_7) = 0.9, p(a_8) = 0.8, p(a_9) = 0.1, p(a_{10}) = 0.9, \\ p(a_{11}) = 0.8, p(a_{12}) = 0.7, p(a_{13}) = 0.1.$$

are supposed, then the knowledge base can be written as follows:

```
(tell
  (var visit tuberculosis x-ray lung-cancer
    smoker bronchitis dyspnoea binary)
  (ass a1 a6 a9 a13 binary 0.1)
  (ass a2 a4 binary 0.01)
  (ass a3 binary 0.2)
  (ass a5 binary 0.3)
  (ass a7 a10 binary 0.9)
  (ass a8 a11 binary 0.8)
  (ass a12 binary 0.7)

  (-> (and visit a1) tuberculosis)
  (-> a2 tuberculosis)
  (-> tuberculosis (or (and visit a1) a2))

  (-> (and smoker a3) lung-cancer)
  (-> a4 lung-cancer)
  (-> lung-cancer (or (and smoker a3) a4))

  (-> (and smoker a5) bronchitis)
  (-> a6 bronchitis)
  (-> bronchitis (or (and smoker a5) a6))

  (-> (and lung-cancer a7) x-ray)
  (-> (and tuberculosis a8) x-ray)
  (-> a9 x-ray)
  (-> x-ray (or (and lung-cancer a7)
    (and tuberculosis a8) a9))
```

```
(-> (and tuberculosis a10) dyspnoea)
(-> (and lung-cancer a11) dyspnoea)
(-> (and bronchitis a12) dyspnoea)
(-> a13 dyspnoea)
(-> dyspnoea (or (and tuberculosis a10)
                  (and lung-cancer a11)
                  (and bronchitis a12) a13)))
```

If the doctor observes that the patient who suffers from dyspnoea is a smoker and that he has visited an under-developed country recently, then  $d$ ,  $s$ , and  $v$  can be added as observations to the basic knowledge base:

```
(observe dyspnoea smoker visit)
```

Now, symbolic or numerical arguments for hypotheses like “Does the patient suffer from tuberculosis?”, “Does the patient suffer from bronchitis?”, etc., may be of interest. For the hypotheses  $t$  (tuberculosis) and  $b$  (bronchitis) the system may report the following results (Haenni, 1996):

? (ask (sp tuberculosis))	? (ask (sp bronchitis))
(a1 and a10) or	(a5 and a12) or
(a1 and a13) or	(a6 and a12) or
(a2 and a10) or	(a5 and a13) or
(a2 and a13) or	(a6 and a13) or
(a1 and a3 and a11) or	(a1 and a5 and a10) or
(a1 and a4 and a11) or	(a1 and a6 and a10) or
(a1 and a5 and a12) or	(a2 and a5 and a10) or
(a1 and a6 and a12) or	(a2 and a6 and a10) or
(a2 and a3 and a11) or	(a3 and a5 and a11) or
(a2 and a4 and a11) or	(a3 and a6 and a11) or
(a2 and a5 and a12) or	(a4 and a5 and a11) or
(a2 and a6 and a12)	(a4 and a6 and a11)
? (ask (dsp tuberculosis))	? (ask (dsp bronchitis))
0.1924	0.5857

From symbolic support the doctor gets configurations of assumptions that allow him to explain or deduce the hypothesis. In contrast, numerical degrees of support help to judge and compare the results quantitatively. Here, bronchitis is almost three times as credible as tuberculosis.

## 6.2 Burglary

The example considered here is a small story around the alarm system of Mr. Holmes’ house (Pearl, 1988):

“A burglary in Mr. Holmes house generates an alarm if the alarm system is functioning. But the alarm may also be caused by an earthquake or by other

(unspecified) reasons. The neighbors of Mr. Holmes, Dr. Watson and Mrs. Gibbons, phone Mr. Holmes in the case of an alarm. Possibly, Dr. Watson may also phone Mr. Holmes as a joke. Mrs. Gibbons is hard of hearing, and she may possibly not be able to hear the alarm. Furthermore, if Mr. Holmes' daughter is at home, then she surely will phone too in the case of an alarm. Finally, if there is an earthquake and if the earthquake is registered, then there is a confirmation of it on the radio."

In Figure 6.5 the above story is shown as a causal network.

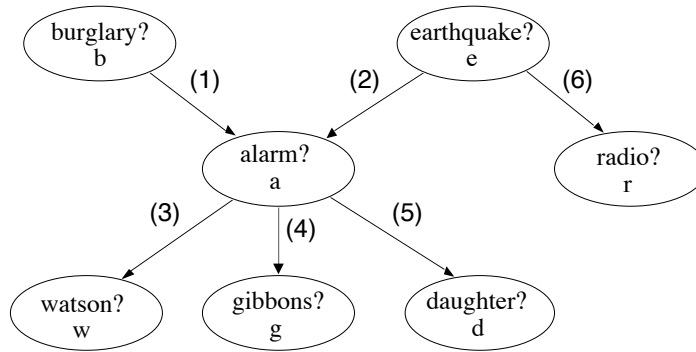


Figure 6.5: Causal network for the burglary example.

Using the strategy described in (3'') the following sets of logical implications are obtained:

$$(1) \quad b \wedge a_1 \rightarrow a, \quad e \wedge a_2 \rightarrow a, \quad a_3 \rightarrow a, \\ a \rightarrow (b \wedge a_1) \vee (e \wedge a_2) \vee a_3; \quad (6.15)$$

$$(2) \quad a \wedge a_4 \rightarrow w, \quad a_5 \rightarrow w, \quad w \rightarrow (a \wedge a_4) \vee a_5; \quad (6.16)$$

$$(3) \quad a \wedge a_6 \rightarrow g, \quad a_7 \rightarrow g, \quad g \rightarrow (a \wedge a_6) \vee a_7; \quad (6.17)$$

$$(4) \quad a \wedge a_8 \rightarrow d, \quad a_9 \rightarrow d, \quad d \rightarrow (a \wedge a_8) \vee a_9; \quad (6.18)$$

$$(5) \quad e \wedge a_{10} \rightarrow r, \quad a_{11} \rightarrow r, \quad r \rightarrow (e \wedge a_{10}) \vee a_{11}. \quad (6.19)$$

The probabilities for the assumptions  $a_1, \dots, a_{11}$  have to be supposed:

$$p(a_1) = 0.9, p(a_2) = 0.9, p(a_3) = 0.1, p(a_4) = 0.7, p(a_5) = 0.3, \\ p(a_6) = 0.1, p(a_7) = 0.1, p(a_8) = 0.7, p(a_9) = 0.3, p(a_{10}) = 0.9, p(a_{11}) = 0.1.$$

Using ABEL the burglary example can be modeled as follows:

```

(tell
  (var burglary alarm earthquake watson gibbons daughter radio binary)
  (ass a1 a2 a10 binary 0.9)
  (ass a4 a8 binary 0.7)
  (ass a3 a6 a7 a11 binary 0.1)
)

```

```

(ass a5 a9 binary 0.3)

(-> (and burglary a1) alarm)
(-> (and earthquake a2) alarm)
(-> a3 alarm)
(-> alarm (or (and burglary a1) (and earthquake a2) a3))

(-> (and alarm a4) watson)
(-> a5 watson)
(-> watson (or (and alarm a4) a5))

(-> (and alarm a6) gibbons)
(-> a7 gibbons)
(-> gibbons (or (and alarm a6) a7))

(-> (and alarm a8) daughter)
(-> a9 daughter)
(-> daughter (or (and alarm a8) a9))

(-> (and earthquake a10) radio)
(-> a11 radio)
(-> radio (or (and earthquake a10) a11))

```

Suppose now that Mr. Holmes receives a phone call from his neighbor Dr. Watson stating that he hears an alarm sound from the direction of Mr. Holmes's house. Further, there is neither an announcement of an earthquake on the radio, nor Mr. Holmes's daughter phones him.

```
(observe watson (not daughter) (not radio))
```

Now, symbolic or numerical arguments for hypotheses may be of interest. Here, Mr. Holmes will be most interested in whether there is a burglary or not.

```
(ask
  (sp burglary)
  (dsp burglary)
  (dpl burglary))
```

## **7 Failure Trees**

```

(var starting-system-state
  (ok defective-switch transmission-not-in-park
    some-other-fault-in-starting-system))
(var switch-state
  (ok defective-ignition-switch defective-starter-relay
    defect-in-some-other-switch))

(var car-starts binary))

```

Each leaf of the tree in Figure 7.1 has a corresponding assumption which implies the respective state of the variable.

```

(tell
  (ass ass-faulty-fuel-system binary 0.05)
  (-> ass-faulty-fuel-system
    (= car-state faulty-fuel-system))

  (ass ass-something-else binary 0.02)
  (-> ass-something-else
    (= car-state something-else))

  (ass ass-weak-or-faulty-battery binary 0.10)
  (-> ass-weak-or-faulty-battery
    (= battery-system-state weak-or-faulty-battery))

  (ass ass-faulty-battery-connections binary 0.05)
  (-> ass-faulty-battery-connections
    (= battery-system-state faulty-battery-connection))

  (ass ass-transmission-not-in-park binary 0.10)
  (-> ass-transmission-not-in-park
    (= starting-system-state transmission-not-in-park))

  (ass ass-some-other-fault-in-starting-system binary 0.10)
  (-> ass-some-other-fault-in-starting-system
    (= starting-system-state some-other-fault-in-starting-system))

  (ass ass-defective-ignition-switch binary 0.05)
  (-> ass-defective-ignition-switch
    (= switch-state defective-ignition-switch))

  (ass ass-defective-starter-relay binary 0.05)
  (-> ass-defective-starter-relay
    (= switch-state defective-starter-relay))

  (ass ass-defect-in-some-other-switch binary 0.10)
  (-> ass-defect-in-some-other-switch
    (= switch-state defect-in-some-other-switch)))

```

The following statements are necessary to connect the variables of the failure tree and to complete the basic knowledge base:

```
(tell
  (<-> car-starts (= car-state ok))
  (<-> (not (= battery-system-state ok))
        (= car-state faulty-battery-system))
  (<-> (not (= starting-system-state ok))
        (= car-state faulty-starting-system))
  (<-> (not (= switch-state ok))
        (= starting-system-state defective-switch)))
```

Suppose that the car does not start. If the starting system has already been checked, then the following observations are added:

```
(observe
  (not car-starts)
  (= starting-system-state ok))
```

The main interest in this kind of problems is to locate the cause of the failure. The following queries may be helpful to find the faulty component of the car:

```
(ask (dsp (= car-state faulty-battery-system))
     (dsp (= car-state faulty-fuel-system))
     (dsp (= switch-state defective-starter-relay)))
```

## A ABEL Grammar

$\langle \text{Abel} \rangle ::= \{ \langle \text{TellPart} \rangle \mid \langle \text{ObservePart} \rangle \mid \langle \text{AskPart} \rangle \mid \langle \text{EmptyPart} \rangle \}^+$   
 $\langle \text{TellPart} \rangle ::= (\text{tell} \ [ \langle \text{KeyWord} \rangle ] \{ \langle \text{Instruction} \rangle \}^+)$

$\langle \text{Instruction} \rangle ::= \langle \text{Definition} \rangle \mid \langle \text{Statement} \rangle \mid \langle \text{ModuleInst} \rangle$   
 $\langle \text{Definition} \rangle ::= \langle \text{TypeDef} \rangle \mid \langle \text{VarDef} \rangle \mid \langle \text{AssDef} \rangle \mid \langle \text{ModuleDef} \rangle$   
 $\langle \text{TypeDef} \rangle ::= (\text{type} \ \{ \langle \text{TypeName} \rangle \}^+ \ \langle \text{Type} \rangle)$   
 $\langle \text{VarDef} \rangle ::= (\text{var} \ \{ \langle \text{VarName} \rangle \}^+ \ \langle \text{Type} \rangle)$   
 $\langle \text{AssDef} \rangle ::= (\text{ass} \ \{ \langle \text{AssName} \rangle \}^+ \ \langle \text{Type} \rangle \ [ \langle \text{AssValue} \rangle ])$   
 $\langle \text{ModuleDef} \rangle ::= (\text{module} \ \langle \text{ModuleName} \rangle \ (\{ \langle \text{FParam} \rangle \}) \ \langle \text{ModuleBody} \rangle)$   
 $\langle \text{Type} \rangle ::= \langle \text{PrimType} \rangle [ \langle \text{Interval} \rangle ] \mid \langle \text{TypeName} \rangle \mid \langle \text{TypeSet} \rangle \mid \text{number}$   
 $\langle \text{PrimType} \rangle ::= \text{integer} \mid \text{real} \mid \text{binary}$   
 $\langle \text{Interval} \rangle ::= [ \langle \text{Lower} \rangle \dots \langle \text{Upper} \rangle ] \mid [ \langle \text{Lower} \rangle \dots ] \mid [ \dots \langle \text{Upper} \rangle ]$   
 $\langle \text{Lower} \rangle ::= \text{number} \mid \text{symbol}$   
 $\langle \text{Upper} \rangle ::= \text{number} \mid \text{symbol}$   
 $\langle \text{TypeSet} \rangle ::= (\{ \text{symbol} \mid \text{number} \}^+)$   
 $\langle \text{TypeName} \rangle ::= \text{symbol}$   
 $\langle \text{VarName} \rangle ::= \text{symbol}$   
 $\langle \text{AssName} \rangle ::= \text{symbol}$   
 $\langle \text{AssValue} \rangle ::= \text{number} \mid (\{ \text{number} \}^+)$   
 $\langle \text{ModuleName} \rangle ::= \text{symbol}$   
 $\langle \text{FParam} \rangle ::= \langle \text{VarDef} \rangle \mid \langle \text{AssDef} \rangle$   
 $\langle \text{ModuleBody} \rangle ::= \{ \langle \text{Instruction} \rangle \}^+$   
 $\langle \text{Statement} \rangle ::= \langle \text{1-Statement} \rangle \mid \langle \text{2-Statement} \rangle \mid \langle \text{n-Statement} \rangle \mid$   
 $\quad \langle \text{Constraint} \rangle \mid \text{contradiction} \mid \text{tautology}$   
 $\langle \text{1-Statement} \rangle ::= (\text{not} \ \langle \text{Statement} \rangle)$   
 $\langle \text{2-Statement} \rangle ::= ((\rightarrow \mid \leftarrow) \ \langle \text{Statement} \rangle \ \langle \text{Statement} \rangle)$   
 $\langle \text{n-Statement} \rangle ::= ((\text{and} \mid \text{or} \mid \text{xor}) \ \{ \langle \text{Statement} \rangle \}^+)$   
 $\langle \text{Constraint} \rangle ::= (\langle \text{BoolOp} \rangle \ \langle \text{Expression} \rangle \ \langle \text{Expression} \rangle) \mid \langle \text{BinVar} \rangle$   
 $\langle \text{BoolOp} \rangle ::= = \mid \neq \mid < \mid \leq \mid > \mid \geq \mid \text{in}$   
 $\langle \text{Expression} \rangle ::= (\langle \text{Operator} \rangle \ \{ \langle \text{Expression} \rangle \}^+) \mid \langle \text{AtomicExp} \rangle$   
 $\langle \text{AtomicExp} \rangle ::= \langle \text{VarName} \rangle \mid \langle \text{AssName} \rangle \mid \text{number} \mid \text{symbol} \mid \langle \text{TypeSet} \rangle$   
 $\langle \text{Operator} \rangle ::= \langle \text{ArithOp} \rangle \mid \langle \text{FuncOp} \rangle \mid \langle \text{TrigOp} \rangle$   
 $\langle \text{ArithOp} \rangle ::= + \mid - \mid * \mid /$   
 $\langle \text{FuncOp} \rangle ::= \text{sqr} \mid \text{sqrt} \mid \text{exp} \mid \text{expt} \mid \text{log} \mid \text{abs} \mid \text{min} \mid \text{max} \mid \text{mod}$   
 $\langle \text{TrigOp} \rangle ::= \text{sin} \mid \text{cos} \mid \text{tan} \mid \text{asin} \mid \text{acos} \mid \text{atan}$   
 $\langle \text{BinVar} \rangle ::= \text{symbol}$   
 $\langle \text{ModuleInst} \rangle ::= (\langle \text{ModuleName} \rangle \ \{ \langle \text{AParam} \rangle \})$   
 $\langle \text{AParam} \rangle ::= \text{number} \mid \text{symbol}$   
 $\langle \text{ObservePart} \rangle ::= (\text{observe} \ [ \langle \text{KeyWord} \rangle ] \{ \langle \text{Statement} \rangle \}^+)$

$\langle \text{KeyWord} \rangle ::= \text{:symbol} \mid \text{number}$   
 $\langle \text{AskPart} \rangle ::= (\text{ask} \ \langle \text{Query} \rangle)$

$\langle \text{Query} \rangle ::= \langle \text{Expression} \rangle \mid \langle \text{AssQuery} \rangle$   
 $\langle \text{AssQuery} \rangle ::= \langle \text{SymbolicAssQuery} \rangle \mid \langle \text{NumericAssQuery} \rangle$   
 $\langle \text{SymbolicAssQuery} \rangle ::= ((\text{sp} \mid \text{qs} \mid \text{pl} \mid \text{db}) \langle \text{Statement} \rangle)$   
 $\langle \text{NumericAssQuery} \rangle ::= ((\text{dsp} \mid \text{dqs} \mid \text{dpl} \mid \text{ddb}) \langle \text{Statement} \rangle)$   
 $\langle \text{EmptyPart} \rangle ::= (\text{empty} \mid [\text{tell} \mid \text{observe} \mid \langle \text{ObsKey} \rangle])$

## References

- Almond, R.G. 1990. *Fusion and Propagation of Graphical Belief Models: an Implementation and an Example*. Ph.D. thesis, Department of Statistics, Harvard University.
- Almond, R.G. 1995. *Graphical Belief Modeling*. Chapman and Hall.
- Andersen, S.K., Olesen, K.G., Jensen, F.V., & Jensen, F. 1990. HUGIN – a Shell for Building Bayesian Belief Universes for Expert Systems. *Pages 332–338 of: Shafer, G., & Pearl, J. (eds), Readings in Uncertain Reasoning*. Morgan Kaufmann.
- Davis, R. 1984. Diagnostic Reasoning based on Structure and Behaviour. *Artificial Intelligence*, **24**, 347–410.
- de Kleer, J. 1986. An Assumption-based TMS. *Artificial Intelligence*, **28**, 127–162.
- de Kleer, J., & Williams, B.C. 1987. Diagnosing Multiple Faults. *Artificial Intelligence*, **32**, 97–130.
- Donnet-Portenier, C. 1993. *Traitement de modeles quantitatifs au moyen du langage de contraintes ACMS (Arithmetic Constraint Modeling System)*. Ph.D. thesis, Institut für Informatik, Universität Freiburg.
- Haenni, R. 1996. *Propositional Argumentation Systems and Symbolic Evidence Theory*. Ph.D. thesis, Institut für Informatik, Universität Freiburg.
- Hsia, Y.T., & Shenoy, P.P. 1989. An Evidential Language for Expert Systems. *Pages 9–14 of: Ras, Z.W. (ed), Methodologies for Intelligent Systems*. North-Holland.
- Kohlas, J., & Monney, P.A. 1993. Probabilistic Assumption-Based Reasoning. *In: Heckerman, & Mamdani (eds), Proc. 9th Conf. on Uncertainty in Artificial Intelligence*. Kaufmann, Morgan Publ.
- Kohlas, J., & Monney, P.A. 1995. *A Mathematical Theory of Hints. An Approach to the Dempster-Shafer Theory of Evidence*. Lecture Notes in Economics and Mathematical Systems, vol. 425. Springer.
- Kohlas, J., Monney, P.A., Anrig, B., & Haenni, R. 1996. *Model-Based Diagnostics and Probabilistic Assumption-Based Reasoning*. Tech. Rep. 96–09. University of Fribourg, Institute of Informatics.

- Larsson, J.E. 1994. Diagnosis Based on Explicit Means-End Models. *Artificial Intelligence*, **80**, 29–93.
- Lauritzen, S.L., & Shenoy, P.P. 1995. *Computing Marginals Using Local Computation*. Working Paper 267. School of Business, University of Kansas.
- Lauritzen, S.L., & Spiegelhalter, D.J. 1988. Local Computations with Probabilities on Graphical Structures and their Application to Expert Systems. *Journal of Royal Statistical Society*, **50**(2), 157–224.
- Lehmann, N. 1994. *Entwurf und Implementation einer annahmenbasierten Sprache*. Diplomarbeit. Institute of Informatics, University of Fribourg.
- Mackworth, A.K. 1987. Constraint Satisfaction. *Pages 205–211 of: Shapiro, S.C. (ed), Encyclopedia of Artificial Intelligence*. J. Wiley, New York.
- Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann Publ. Inc.
- Raiman, O., de Kleer, J., Saraswat, V., & Shirley, M. 1991. Characterizing Non-Intermittent Faults. *National Conference on AI*, **9**, 849–854.
- Reiter, R. 1987. A Theory of Diagnosis From First Principles. *Artificial Intelligence*, **32**, 57–95.
- Saffiotti, A., & Umkehrer, E. 1991. *PULCINELLA: A General Tool for Propagating Uncertainty in Valuation Networks*. Tech. Rep. IRIDIA, Université de Bruxelles.
- Shafer, G. 1976. *The Mathematical Theory of Evidence*. Princeton University Press.
- Shafer, G., & Logan, R. 1987. Implementing Dempster’s Rule for Hierarchical Evidence. *Artificial Intelligence*, **33**, 271–298.
- Srinivas, S., & Breese, J. 1990. IDEAL: A Software Package for Analysis of Influence Diagrams. *In: Proceedings of the Sixth Uncertainty Conference in AI, Cambridge, MA*.
- Steele, G. L. 1990. *Common Lisp – the Language*. Digital Press.
- Xu, H., & Kennes, R. 1994. Steps Toward Efficient Implementation of Dempster-Shafer Theory. *Pages 153–174 of: Yager, R.R., Fedrizzi, M., & Kacprzyk, J. (eds), Advances in the Dempster-Shafer Theory of Evidence*. John Wiley and Sons.