Department of IT-Security

Master thesis

Application of an Assumption-based Truth Maintenance System for Model-based Diagnosis

submitted by

Konstantin Ruhmann

Flotowstr. 15

22083 Hamburg

supervised by

Prof. Dr. Sebastian Iwanowski

Prof. Dr. Gerd Beuster

September 2, 2016

Contents

List of Figures

Notation

Notation	Explanation
$\in$	Member of
$\cap$	Intersection
$\cup$	Union
$\setminus$	Difference
$\subset$	Proper subset
$\subseteq$	Subset
$\neg$	Logical connective: Not
$\wedge$	Logical connective: And
$\vee$	Logical connective: Or
$\rightarrow$	Logical connective: Implication
$\equiv$	Logical equivalent
$\top$	Tautology
$\bot$	Falsity
$\vdash$	Holds in /Implication that can be derived
$\models$	Model for

1

Introduction

1.1 Overview

The diagnosis of a system with a large number of components in general requires to deal with the dependencies of the components. Model-based diagnosis enables a systematic approach to diagnosis, where the system under diagnosis is separated to the diagnosis engine. The system is modeled in a structured way, which allows the detection of the differences between normal and faulty behavior of a system.

The model-based diagnosis in this thesis is based on a compounded architecture. An inference engine interacts with an assumptions-based truth maintenance system (ATMS) to determine conflicts in a logical model of a system and consequently provides diagnoses. The ATMS is a problem solver and allows hypothetical reasoning for a given set of assumptions. It ensures the consistency of contradicting inferences and is able to compute the minimal set of assumptions, which are necessary to belief a certain proposition.

1.2 Approach

The ATMS is a problem solver, which can be used for Horn formulas. It maintains a dependency network to trace assumptions. At first the dependency network and the label computation is introduced for the ATMS. Afterwards the focus-ATMS (FATMS), an extension of the ATMS, is presented. It enables to reduce the computational costs of labeling. In addition a prototypically ATMS/FATMS with a simple interface is implemented.

The diagnosis component uses the problem solver for conflict determination. Therefore the interaction to the ATMS/FATMS is explained by the necessary interface

calls. The determined conflicts are used with appropriate manner to generate diagnoses for the behavior of the system. In order to reduce the amount of diagnoses an algorithm is presented to generate only the most preferable diagnoses. The diagnosis engine is implemented and is used to diagnose a circuit.

1.3 Scope and delimitation

In a short introduction the notation for propositional logic is given. Furthermore the terms rule-based system and truth maintenance systems are introduced, as they are used later on. A TMS variant, the justification-based TMS, is mentioned for completeness, but is not explained in detail.

The section for the ATMS starts with an introduction for logician and subsequently the foundations of an ATMS are explained. The formal node definition is necessary for computing the labels within the ATMS. An algorithm for the label computation with example is described. Furhtermore the algorithm is extended for the variant focus-ATMS (FATMS). The ATMS and the label update algorithm refers to Johan de Kleer [FK93] and the FATMS refers to [TI94; Tăt97].

The model-based diagnosis is described by an example. Therefore an exemplary circuit is given. The diagnosis engine is able to interact with both variants of the ATMS and the FATMS. The process of diagnosis refers to [TI94; Iwa15a; Iwa15b].

Conclusively an overview of the prototypical implementation is given.

2

Foundations

2.1 Propositional logic

2.1.1 Syntax

An atomic formula is a proposition and can be either true or false. The syntax of propositional logic is defined by an inductive process to allow complex formulas:

1. All atomic formulas are formulas.
2. For every formula F, $\neg F$ is a formula.
3. For all formulas F and G, also $(F \vee G)$ and $(F \wedge G)$ are formulas.

$\neg$, $\wedge$ and $\vee$ are logical connectives. The negation ($\neg$) is an unary connective, whereas the conjunction ($\wedge$) and the disjunction ($\vee$) are binary connectives. The connective implication $(A \rightarrow B)$ is an abbreviation for $(\neg A \vee B)$. Equivalence $(A \leftrightarrow B)$ is an abbreviation for $((A \rightarrow B) \wedge (B \rightarrow A))$ (cf. [Sch08, pp.3f]).

2.1.2 Semantic

The meaning of a formula is defined by the semantic. The elements of the set $\{True, False\}$ are truth values. An assignment for a formula is a function A, which assigns a truth value to each atomic formula. The function A is extended by $\widehat{A}$ to apply any formulas. This is done by an inductive process.

1. For every atomic formula F: $\widehat{A}(F) = A(F)$
2. $\widehat{A}(F \wedge G) = \begin{cases} True, \text{ if } \widehat{A}(F) = True \text{ and } \widehat{A}(G) = True \\ False, \text{ otherwise} \end{cases}$

3. $\widehat{A}(F \vee G) = \begin{cases} True, \text{ if } \widehat{A}(F) = True \text{ or } \widehat{A}(G) = True \\ False, \text{ otherwise} \end{cases}$

4. $\widehat{A}(\neg F) = \begin{cases} True, \text{ if } \widehat{A}(F) = False \\ False, \text{ otherwise} \end{cases}$

The distinction between A and $\widehat{A}$ is used for the definition. In further applications the distinction is disregarded and the function is called A. An assignment A is suitable for a formula F if the function A is defined for every atomic formula in F. (cf. [Sch08, pp.5-6]).

2.1.3 Models

A formula F is satisfiable, if A is a suitable assignment and $A(F) = True$. Then A is a model for F.

$$A \models F$$

If A is an assignment, but A(F) = False, then the formula is invalid.

$$A \not\models F$$

If every suitable assignment for F is a model for F, then the formula is valid (tautology). This is indicated by $\models F$. In the other case the formula is invalid $\not\models F$. Truth tables can be used to determine the satisfiability of a formula, because the construction of a truth table take into account all possible assignments (cf. [Sch08, p.9]).

2.1.4 Logical Implication

A formula G is a logical implication of formula F it every model of F is also a model of G.

$$F \models G$$

The definition is expandable to sets of formulas. If F is a set of formulas $\{f_1, \ldots, f_n\}$ and G is a set of formulas $\{g_1, \ldots, g_n\}$ then F implies G if every model satisfying all formulas of F also satisfies all formulas of G.

$$f_1, \ldots, f_n \models g_1, \ldots, g_n$$

Two formulas are equivalent if $F \models G$ and $G \models F$. This is indicated by $F \equiv G$ (cf.[Sch08, pp.10,14]).

There exists some important equivalences. Some of them are shown in the following table (cf. [Ros12, p.27]).

Equivalence	Name
$p \wedge q \equiv q \wedge p$ $p \vee q \equiv q \vee p$	Commutative laws
$p \vee (p \wedge q) \equiv p$ $p \wedge (p \vee q) \equiv p$	Absorption laws

Table 2.1: Logical Equivalences

2.1.5 Syntactical Derivation

In semantic truth tables are used to determine (in-)satisfiability or validity of a formula. The disadvantage of truth tables is the algorithm expense. For a formula containing n atomic formulas the amount of combinations is 2^n, which each have to be evaluated (cf. [Sch08, p.12]).

Therefore in propositional logic exists several inference rules, which allow to derive formulas by a given set of formulas. These syntactical derivation are indicated with an $\vdash$. Some syntactical rules are shown in the following table (cf. [Ros12, p.72]).

Name	**Inference rule**	**Description**
Modus ponens	$p \wedge (p \rightarrow q) \vdash q$	If p then q and p are true. Therefore q is true.
Conjunction	$(p, q) \vdash (p \wedge q)$	p and q are true separately. Therefore they are true conjointly.

Table 2.2: Inference rules

2.1.6 Horn formulas

A formula F in conjunctive normal form, which is a conjunction of disjunctions of propositions, is a Horn formula if every disjunction in F contains at most one positive proposition. For example the following formula is a Horn formula. The formula consists of five conjunctions of disjunctions (cf. [Sch08, p.23]).

$$(A \vee \neg B) \wedge (\neg C \vee \neg A \vee D) \wedge (\neg A \vee \neg B) \wedge D \wedge \neg E$$

Horn formulas can be rewritten in a more convenient way namely implications. Therefore two additional formulas are introduced. $\top$ is an arbitrary tautology and

$\bot$ is an arbitrary unsatisfiable formula (cf. [Sch08, p.24]).

$$(B \to A) \land (C \land D \to D) \land (A \land B \to \bot) \land (\top \to D) \land (E \to \bot)$$

2.2 Rule-based systems

A rule is a formal conditional if-then sentence and logical systems use them to derive propositions. Another term for rule is *justification*, which is used in the following sections. For instance the rule

$$A \to B$$

is given. The rule means, if A is true, then B is true. Material implications are a common way to express rules, where on the left side of the implication is the premise and on the right side is the consequent. An exemplary knowledge base for a rule-based system is given with the following rules.

$$\begin{aligned} A \land B &\to C \\ B &\to D \end{aligned}$$

If the rule-based system has evidence, that the proposition B is $True$, then the system can derive by using the inference rule modus ponens that the proposition D is $True$. Furthermore a rule is directed. A rule is only applied if the premise is $True$. The system has no evidence, that the proposition A is $True$, therefore the premise for the first rule is not $True$ and the rule is not applied. (cf. [BK08, p.72-81]).

Horn formulas can be used to create a knowledge base for logical systems. They have the advantage that efficient algorithms are known to determine satisfiability or unsatisfiability. For instance the marking algorithm can be used, which is described in [Sch08, p.24]. But Horn formulas have the limitation, that negated literals can not be used. For example the rule $(A \land \neg B) \to C$ is not a valid Horn formula. Its transformation is $(\neg A \lor B \lor C)$, where more than one positive literal exists (cf. [PM10, p. 185]).

2.3 Truth Maintenance Systems

2.3.1 Hypothetical reasoning

Hypothetical reasoning uses assumptions to reason with unsure or incomplete data to progress solving a problem. Therefore in a knowledge base hypothetical scenarios are modeled, which inherently can lead to a inconsistent knowledge base (cf. [Tăt97, p.20]). For instance a small knowledge base for a circuit with two justifications is given(cf. [BK08, p.209]).

$$\begin{aligned} s &\rightarrow l \\ s \wedge \neg w &\rightarrow \neg l \end{aligned}$$

The proposition s means that the switch is closed and if the proposition is negated it is open. The proposition w means that the wire is ok and if negated the wire is broken. The proposition l indicates that the lamp is lit and if negated the lamp is dark.

Initially the system works as expected. If the switch is closed s the lamp lits l. This is derived by the first justification of the knowlegde base. In case of a failure, where in addition the observation $\neg w$ is added, the system derives the propositions l and $\neg l$. This is a contradiction, because a lamp can either be lit or be dark. A simple rule-based system is not able to handle contradictions. A conclusion is added to the KB and is used for further inferences. Furthermore conclusions cannot be revoked (cf. [BK08, p.209-210]).

A truth maintenance system (TMS) is able to handle contradictions and ensures the consistency in a knowledge base. A TMS provides the functionality of hypothetical reasoning and reason maintenance. It allows to assimilate inconsistent knowledge and is able to restore consistency. The TMS is able to answer what inferred data can be believed if a set of assumptions is believed. The TMS avoids to restart the reasoning from the beginning, when the set of assumptions changes (cf. [Tăt97, p. 20]).

2.3.2 Architecture

A reasoning system, which uses a TMS, is separated in the components inference engine (IE) and TMS. The components interact via a well-specified interface.

1. **Inference Engine**
   The inference engine has full knowledge of the domain and provides justifications

to the TMS. The IE depends on the specific problem.

2. **TMS**
   The TMS receives justifications by the IE. It proceeds the justifications and is able to answer questions to the IE. The TMS has a predefined interface and is independent of the problem. The TMS acts like a database for the inference engine.

The interaction between an IE and a TMS is shown in figure 2.1. The IE provides assumptions and justifications to the TMS. The TMS delivers believes and contradictions to the IE (cf. [Tät97, p.21], [FK93, p.158]).

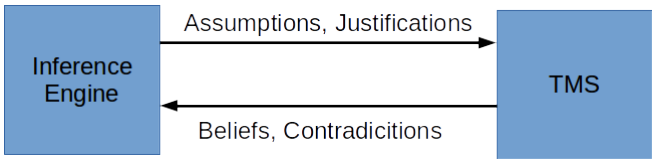

Figure 2.1: Interaction between an IE and a TMS (cf. [FK93, p.158])

Typically a TMS provides the following tasks. First it makes an entailment check, which decides what else can be believed if a set of assumptions is believed. The second task is a consistency check, which ensures that the set of beliefs is consistent and the third task is to compute minimal-support-sets. These are the minimal sets of assumptions, which have to be believed for a certain proposition. An TMS operates incrementally, which means that the addition of inferences (logical conclusions) leads to an update of the current beliefs. In most TMSs it is not possible to delete assumptions or inferences (cf. [Tät97, pp. 16,21-22]).

There are several kinds of TMSs. The most known variants of TMSs are the justification-based TMS(JTMS) and the assumptions-based TMS (ATMS). Both TMSs are limited to Horn formulas. The JTMS is able to accomplish only the first two tasks. It provides much more efficient algorithms for the first two tasks as the ATMS. The ATMS supports all three tasks, but the ATMS uses an in worst case exponential time algorithm to accomplish the third task, which then allows to solve the first two tasks (cf. [Tät97, p.16]).

A context is a set of assumptions. The JTMS is a single-context TMS, which means that the JTMS maintains the beliefs for one context at a time. The ATMS is a multiple-context TMS, which means that it can maintain beliefs for all contexts in parallel(cf. [Tät97, p.24]).

3

Assumption-based Truth Maintenance System

3.1 Introduction

3.1.1 Motivation

An assumption-based truth maintenance system (ATMS) maintains atomic propositions in a dependency network. For example the propositions A, B, C, p and the justifications $A \wedge B \rightarrow p$, $B \wedge C \rightarrow p$ and $A \wedge C \rightarrow \bot$ are given. The rule $A \wedge C \rightarrow \bot$ is a contradiction, because $\bot$ is derived. The dependency network is shown in figure 3.1.

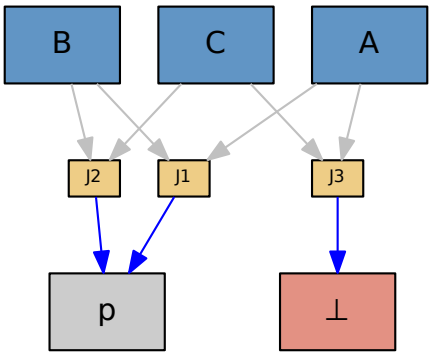

Figure 3.1: Dependency network

Logically expressed the ATMS can determine if a proposition holds when a set of

propositions and a set justifications are given (cf. [Kle86, p.143]). The propositions are assumptions and they can be conjunct with the logical connective $\wedge$. A set of assumptions is a context and the ATMS works in all contexts at once without explicitly computing them (cf. [Kle86, p.160]).

$$\left.\begin{array}{c} A \\ B \\ C \\ A \wedge B \\ A \wedge C \\ B \wedge C \\ A \wedge B \wedge C \end{array}\right\} \overset{?}{\vdash}_{Justifications} \left\{\begin{array}{l} A \\ B \\ C \\ p \\ \bot \end{array}\right.$$

For example the ATMS can determine if the proposition p holds in the context $A \wedge B$.

$$A \wedge B \vdash_{Justifications} p$$

The determination is based on the main task of the ATMS. The ATMS is able to determine immediately, which minimal conjunctions of assumables are necessary to allow the derivation of a proposition (cf. [Kle86, p.145]). In the given example the ATMS computes the conjunctions $A \wedge B$ and $B \wedge C$ for the propositions p.

$$\left.\begin{array}{c} A \wedge B \\ B \wedge C \end{array}\right\} \vdash_{Justifications} p$$

3.1.2 Foundations

An ATMS maintains atomic propositions and associates to each proposition a node. A node is a data structure to store additional information for a proposition. As usual a proposition can be either $True$ of $False$. Justifications describe how a proposition is derivable from other propositions.

$$x_1 \wedge x_2 \wedge \cdots \rightarrow c$$

The propositions $x_1 \wedge x_2 \wedge \ldots$ are the antecedents and the proposition c is the consequent. A justification is a material implication and is limited to Horn formulas. Therefore the ATMS does not allow negated literals in justifications (cf. [Kle86, p.143]). The formulas $\top$ and $\bot$ for Horn formulas are also covered by justifications. If a set of antecedents is empty, then the consequent is unconditionally $True$ and

this corresponds to the Horn formula $\top \rightarrow c$. The proposition c is called a *premise* (cf. [Kle86, p.146]). A justification with the consequent $\bot$ stands for a conjunction of antecedents being *False*. The node $\bot$ is called the *contradiction node*.

$$x_1 \wedge x_2 \wedge \cdots \rightarrow \bot$$

A proposition, which is believed to be true, is called an *assumption*. A set of assumptions is called an *environment* and logically an environment is a conjunction of assumptions (cf. [Kle86, p.142]).

$$A \wedge B \wedge \cdots = \{A, B, \dots\}$$

Two types of environments are distinguished in an ATMS. An environment is inconsistent, if its propositions allow the derivation of the contradiction node. Otherwise the environment is consistent. An abbreviation for an inconsistent environment is the term *nogood*.

The figure 3.2 presents an exemplary dependency network for the nodes A, B, C, p, $\bot$ and the justifications $A \wedge C \rightarrow p$, $B \rightarrow p$, $\top \rightarrow C$ and $A \wedge B \rightarrow \bot$.

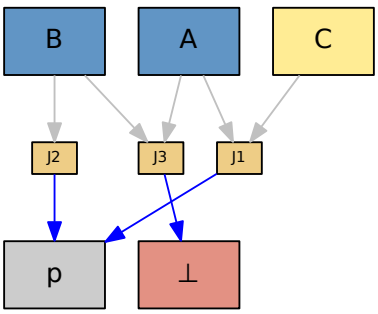

Figure 3.2: Justifications

A justification is presented by a box, which is labeled by an ascending numbering. The incoming arrows to a justification are the antecedents, while the outgoing arrow is the consequent. The premise C, justified by $\top \rightarrow C$ is indicated by the yellow color of the node. The blue color of A and B indicates that these nodes are assumptions. Assumptions and premises are indicated in this thesis for convenience with a capital letter.

3.1.3 Labels

A label is a set of environments and every node has an associated label. The structure of an exemplary label can be seen in figure 3.3. In the example the label consist of two environments and each environment consists of assumptions. An assumption can appear in several environments.

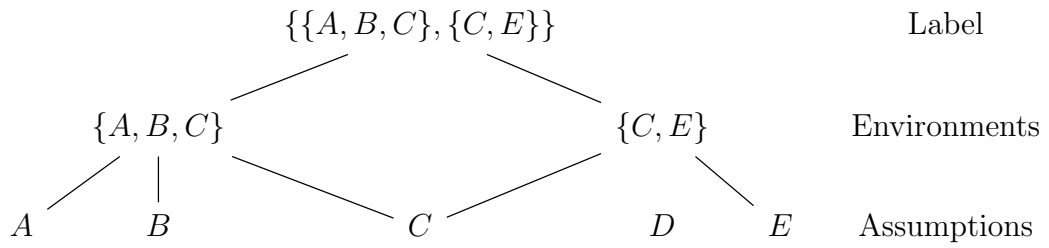

Figure 3.3: Structure of a label

A label is computed by justifications and describes ultimately how a node depends on assumptions. The labels of all nodes are precomputed by the ATMS and the ATMS ensures that each label of a node is consistent, sound, complete and minimal with respect to the current set of justifications (cf. [Kle86, p.144-145]).

1. **Consistent**
   A label is consistent if all environments of the label are consistent.

2. **Sound**
   A label is sound if the node is derivable from each environment of the label.

3. **Complete**
   A label of a node is complete if every consistent environment, which allows the derivation of the node, is a superset of some environment of the label.

4. **Minimal**
   A label is minimal if no environment of the label is a superset of any other environment of the label.

The label of the contradiction node maintains inconsistent environments. Therefore for this node the properties consistent and complete are slightly adapted. The label is "consistent" if all environments of the label are inconsistent. Related to the complete property, each inconsistent environment is a superset of some environments of the label.

The definition of a label ensures that a proposition is derivable by a consistent environment E in exactly those cases that E is a superset of any environment of n's

label. A node has an empty label if there is no environment justifying the node (cf. [Kle86, p.145]).

3.1.4 Interface

In order to complete the introduction an interface for the ATMS is developed for this thesis. In the background a label update algorithm exists. The algorithm ensures the properties consistency, soundness, completeness and minimality of a label.

1. **addJustification**(n, X)
   The method takes two arguments. The argument n is the consequent and the argument X is a set of antecedents. The method adds a justification to the ATMS. The label update algorithm is required to ensure the label definition.

2. **addAssumption**(n)
   The method indicates that the proposition n is assumed. Afterwards the label update algorithm is required to ensure the label definition.

3. **addPremise**(n)
   The method indicates that the proposition n is a premise. The method addJustification with n as consequent and the empty set as antecedent is called.

4. **addNogood**(X)
   The method indicates that the set of propositions X is a contradiction. The method addJustification with $\bot$ as the consequent and the set X as antecedent is called.

5. **getEnvironments**(n)
   The parameter n is a proposition. The function returns the environments of n's label.

6. **getConflicts**()
   The function returns the nogoods of the contradiction node's label. Therefore the function getEnvironments with $\bot$ is called.

7. **isValidIn**(n, X)
   This function returns a boolean flag. The method returns True, if the node n holds in context X. Therefore the environment X has to be consistent and a superset of any environment of n's label.

3.2 Node definition

A node contains additional information to handle the dependencies between the nodes in an ATMS (cf. [Ano02]). A node consists of a tuple and is referred by an identifier. The identifier is the proposition's name. The elements of the tuple are sets.

$$N \quad := \quad (C, J, L) \tag{3.1}$$

C **Consequents**
The set C consists of identifiers, which are affected, if the node's label changes.

J **Justifications**
The elements of the set J are sets of antecedents, whose consequent is the node.[1] A set of antecedents consists of identifiers, which enable to refer to the corresponding nodes.

L **Label**
The elements of the set L are environments. An environment is a set and consists of the assumptions' identifiers.

Given a node by its identifier p it is possible to reference an element of the tuple by an index, e.g. C_p, J_p and L_p. An exemplary application is shown for the proposition p.

$$p := (\{\}, \{\{A, B\}\}, \{\{A, B\}\})$$

The list of consequents is empty. Hence no proposition is affected, if the label of p changes. An exemplary justification $A \wedge B \rightarrow p$ is stored in set J. The set L consists of an exemplary environment $\{A, B\}$.

The elements of the set C are added by the following procedure. Having a justification with c as consequent and X as a set of antecedents

$$X \rightarrow c,$$

then for each node $x \in X$ the consequent c is a member of C_x. For instance the justification $A \wedge B \rightarrow p$ is given. The element p is added to C_A and C_B.

The node definition is presented for the following example. The example consists of the assumptions A,B,C and the justifications $A \wedge B \rightarrow p$, $B \wedge C \rightarrow p$ and $A \wedge C$

[1] Therefore in J it is only required to store the sets of antecedents and not the consequent. For example if $A \rightarrow p$ is given, then $J_p = \{\{\text{A}\}\}$.

$\rightarrow \bot$. The dependency network is shown in figure 3.4, where below the node names the labels are presented. The label computation is described in the following section.

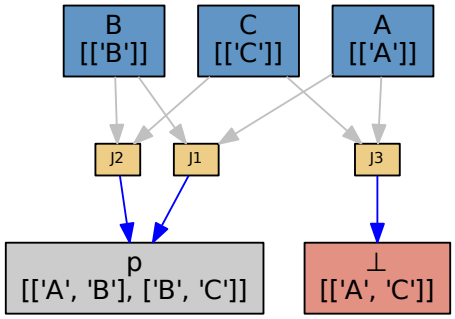

Figure 3.4: Dependency network with labels

A full application of the node definition for the example is:

$$
\begin{aligned}
A &:= (\{\bot, p\}, \{\}, \{\{A\}\}) \\
B &:= (\{p\}, \{\}, \{\{B\}\}) \\
C &:= (\{\bot, p\}, \{\}, \{\{C\}\}) \\
p &:= (\{\}, \{\{A, B\}, \{B, C\}\}, \{\{A, B\}, \{B, C\}\}) \\
\bot &:= (\{\}, \{\{A, C\}\}, \{\{A, C\}\})
\end{aligned}
$$

The set C for each node is required if a new environment is discovered for the node's label. For the given example the consequent relation is presented by a blue arrow in figure 3.5, where below the node name C is given.

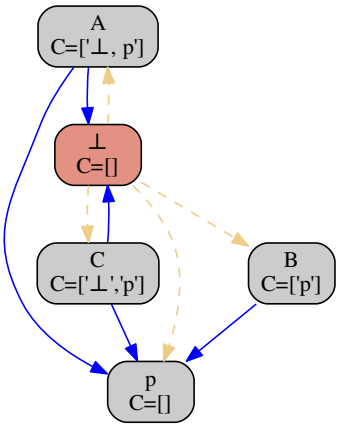

Figure 3.5: Consequent relation

The figure 3.5 also illustrates the essential role of the contradiction node. All nodes are a consequent of the node $\bot$. This relation is indicated by a yellow arrow. The reason is, if a new inconsistent environment is discovered then the ATMS has to ensure that all other labels are still consistent. The consequents of node $\bot$ are not stated in $C_\bot$, because of its solely reducing property.

3.3 Label computation

3.3.1 Overview

The computation of a node's label is done in two steps. At first environments, determined by an inductive process, are added to the label and afterwards supersets and nogoods are removed. The first step achieves completeness and soundness, whereas in the second step the labels become minimal and consistent.

In the following presented algorithm is explained how a label of a node can be exemplary computed (cf. [Ano02]). In the next section 3.4 an algorithm for the label computation is presented, where only the incremental label changes are propagated in the dependency network.

3.3.2 Generating environments

In the first part of the algorithm an inductive process is used to generate environments for the label of a node. If the inductive process is applied for node p, then the labels

of the antecedents are used to compute the label L_p. The antecedents with the consequent p are stored in the set J_p. A set of antecedents consists of identifiers, therefore the labels of the antecedents are easily determined by referring their nodes. Then these labels are combined to generate the environments for the label of node p. This is stated in the first part of the inductive process in formula 3.2. The second part of the inductive process is valid if the proposition is an assumption. Then an environment, consisting only of the proposition's identifier, is added to the node's label (formula 3.3).

Inductive process for generating environments

1. **For all nodes N**

$$\begin{aligned} L_N \quad := \quad \{ & \\ & E_1 \cup E_2 \cup \cdots \cup E_i \mid \\ & \exists X \in J_N : X = \{x_1, x_2, \ldots, x_i\} \wedge \\ & E_1 \in L_{x_1} \wedge E_2 \in L_{x_2} \wedge \cdots \wedge E_i \in L_{x_i} \\ \} & \end{aligned} \tag{3.2}$$

2. **If N is an assumption**

$$\{N\} \in L_N \tag{3.3}$$

The label computation is shown for an example.The example consists of the assumptions A, B, C, D, E and the justifications $A \wedge B \rightarrow p$, $B \wedge C \wedge D \rightarrow p$, $A \wedge C \rightarrow q$, $D \wedge E \rightarrow q$ and $A \wedge B \wedge E \rightarrow \bot$. The dependency network is shown in the following figure 3.6.

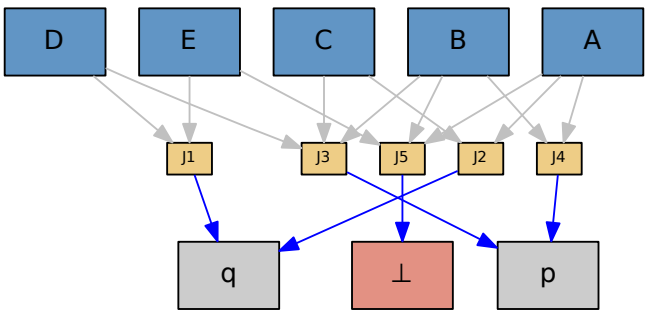

Figure 3.6: Example for the label computation

For each assumption A, B, C, D, E, the inductive process is applied. The assumptions have no justifications, thereore no environments are generated by the first part of the inductive process. In the second part of the process the environment of itself are added to their labels. These environments are called *self environments* in this thesis. The result can be seen in figure 3.7.

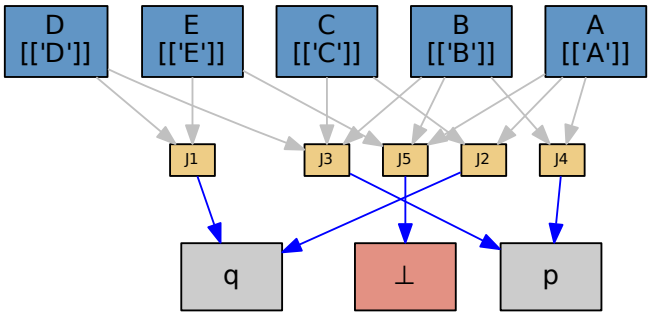

Figure 3.7: Self environments are added

The labels of the nodes p, q and $\bot$ are computed by the first part of the inductive process. The label computation is exemplary shown for node p. The label computation for node q and $\bot$ is similar. The extract of the dependency network for node p is shown in figure 3.8. The set J_p consists of the sets of antecedents $\{A, B\}$ and $\{B, C, D\}$.

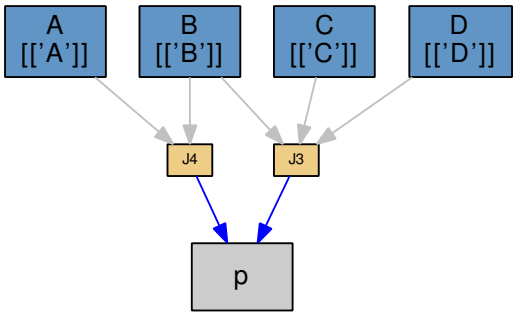

Figure 3.8: Computing the label L_q

For each element in J_p the *Cartesian union* is computed. The term denotes in this thesis that the environments of the antecedents' labels are combined to generate the environments for the consequent's label. An exemplary application of the Cartesian

union is shown at first for the labels of the set of antecedents $\{A, B\}$.

$$\begin{aligned} & \{E_1 \cup E_2 \mid E_1 \in L_A \wedge E_2 \in L_B\} \\ = \; & \{E_1 \cup E_2 \mid E_1 \in \{\{A\}\} \wedge E_2 \in \{\{B\}\}\} \\ = \; & \{\{A\} \cup \{B\}\} \\ = \; & \{\{A, B\}\} \end{aligned}$$

The application of the Cartesian union for the second set of antecedents $\{B, C, D\}$ is.

$$\begin{aligned} & \{E_1 \cup E_2 \cup E_3 \mid E_1 \in L_B \wedge E_2 \in L_C \wedge E_3 \in L_D\} \\ = \; & \{E_1 \cup E_2 \cup E_3 \mid E_1 \in \{\{B\}\} \wedge E_2 \in \{\{C\}\} \wedge E_3 \in \{\{D\}\}\} \\ = \; & \{\{B\} \cup \{C\} \cup \{D\}\} \\ = \; & \{\{B, C, D\}\} \end{aligned}$$

The label of p consist of the Cartesian union for each set of antecedents and therefore the label L_p is

$$L_p = \{\{A, B\}, \{B, C, D\}\}.$$

The result for the label computation for the nodes p, q and $\bot$ is shown in figure 3.9.

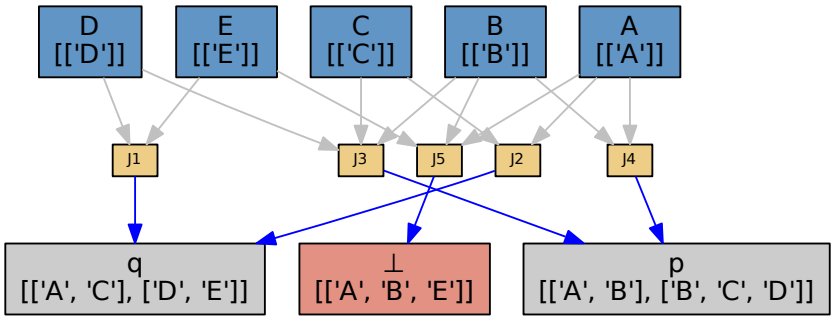

Figure 3.9: Result of the label computation for L_p, L_q and $\bot$

The Cartesian union is required because the environments of the antecedents' labels are disjunct and the Cartesian union produces all combinations for the consequent's node. This is presented for an additional justification $p \wedge q \rightarrow r$. The labels of the nodes p and q are shown in figure 3.9.

The application of the formula 3.2 is equal to the preceding computations, but the application considers the Cartesian union with more than one environment in a

label.

$$
\begin{aligned}
& \{E_p \cup E_q \mid E_p \in L_p \wedge E_q \in L_q\} \\
= \; & \{E_p \cup E_q \mid E_p \in \{\{A,B\},\{B,C,D\}\} \wedge E_q \in \{\{A,C\},\{D,E\}\}\} \\
= \; & \{ \\
& \quad \{A,B\} \cup \{A,C\}, \{A,B\} \cup \{D,E\}, \\
& \quad \{B,C,D\} \cup \{A,C\}, \{B,C,D\} \cup \{D,E\} \\
& \} \\
= \; & \{\{A,B,C\},\{A,B,D,E\},\{A,B,C,D\},\{B,C,D,E\}\}
\end{aligned}
$$

The result is shown in figure 3.10.

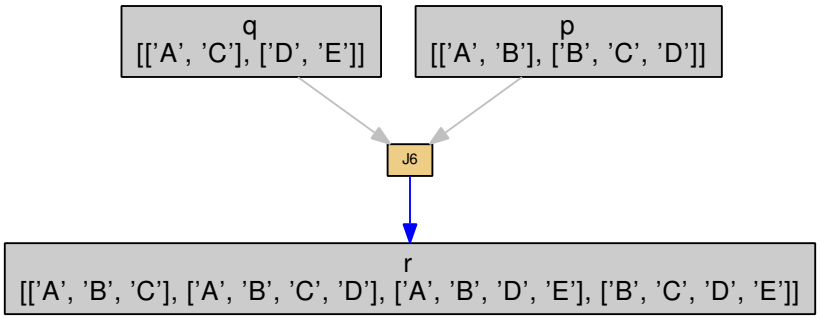

Figure 3.10: Computing the label L_r

The label generation for a premise is also covered by the inductive process. If a proposition p is considered as a premise, then an empty set of antecedents is added to the set J_p. The application of the formula 3.2 produces at least an empty environment for L_p. The empty environment is a subset of any environment and therefore the node holds in every context.

3.3.3 Reducing environments

In the second part of the algorithm the label is adjusted for supersets and contradicted environments. In the preceding section only completeness and soundness of the label is achieved. In this section non-minimal and inconsistent environments are removed.

1. **Removing supersets**
   In classic set theory the subset definition is as follows:

$$M \subseteq N \iff \forall x \in M \rightarrow \forall x \in N$$

   This means that $M \subseteq N$ is a subset if and only if every element of M is also

an element of N (cf. [Ros12, p. 119]). In general speaking the subset M would be seen as the redundant set, because all elements are also in the superset N. Environments have a property that change this point of view, as supersets are removed.

The elements of an environment are conjunct and therefore the conclusion is not valid. For example we consider the environments $E_1 = \{A, B\}$ and $E_2 = \{A, B, C\}$. If $E_1 \subseteq E_2$ and all propositions of E_1 are true, we cannot imply that E_2 is true, because we have no information about the proposition C. But in the other way the conclusion is feasible. If all elements of E_2 are true, then E_1 is also true.

If both environments are member of a label, which means that both allow the derivation of the label's node. Then we can remove E_2 as it doesn't care if the additional proposition C is true. The node is still derivable by the subset. Therefore we can remove all supersets in L_N .

$$\{x \in L_N \mid \exists y \in L_N : y \subset x\} \notin L_N \tag{3.4}$$

2. **Removing nogoods**
   If an environment is a superset of a nogood then the environment is also a nogood.

   For example we have the environments $E_1 = \{A, B\}$ and $E_2 = \{A, B, C\}$. If E_1 is a nogood, then the propositions A and B allow the derivation of the node $\bot$. As these both proposition are also in E_2, the environment E_2 is also a nogood. Therefore all supersets of any nogood are removed.

$$\{x \in L_N \mid \exists y \in L_\bot : y \subseteq x\} \notin L_N \tag{3.5}$$

   This second step differs for the label of the contradiction node, because its label is not adjusted for contradicted environments. If new environments are discovered for the label of the contradiction node all other labels are adjusted for contradicted environments.

The example of figure 3.10 is continued. The label of node r contains non-minimal and contradicted environments.

$$L_r := \{\{A, B, C\}, \{A, B, D, E\}, \{A, B, C, D\}, \{B, C, D, E\}\}$$

The environment $\{A, B, C, D\}$ is a superset of $\{A, B, C\}$ and therefore removed. The environment $\{A, B, D, E\}$ is a superset of the nogood $\{A, B, E\}$ and therefore also removed.

$$L_r := \{\{A, B, C\}, \{B, C, D, E\}\}$$

Conclusively the label node r is complete, consistent, sound and minimal. The removal process is shown in figure 3.11.

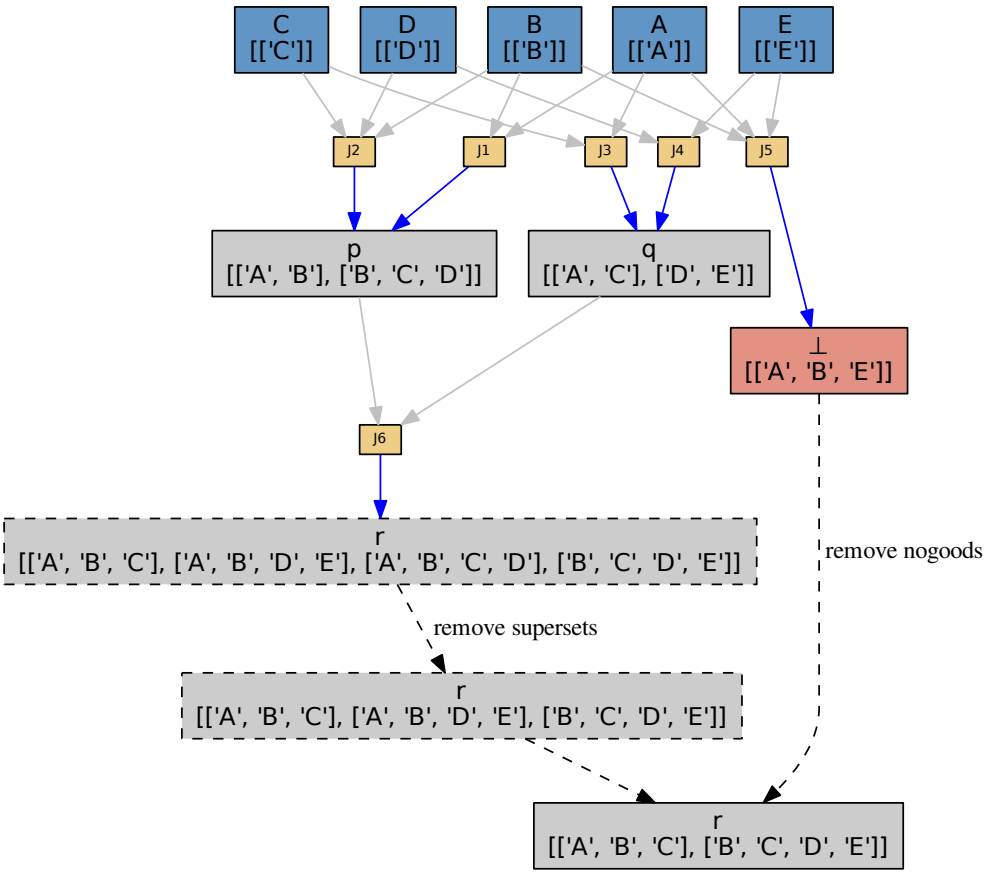

Figure 3.11: Reducing environments

3.3.4 Preserving the truth

The label is the truth layer of the ATMS. It enables to determine if a propositions holds in a given context. Therefore all label operations have to be covered by inference rules or substitutions of semantic equivalences.

1. The Cartesian union computes the mininmal sets of assumptions, which are

necessary to derive a proposition. The inference rule modus ponens uses these sets of assumptions and the justifications to derive a proposition.

Having a minimal set with one assumption $\{A\}$ for proposition B and the justification $A \rightarrow B$ then the proposition B can be derived.

$$A \wedge (A \rightarrow B) \vdash B$$

2. Removing nogoods and supersets of a label is covered by the logically equivalence absorption.

   For example if we have $A \wedge C \rightarrow N$ and $A \rightarrow N$. Then we can write:

$$\begin{array}{rcl} & A \vee (A \wedge C) & \rightarrow N \\ \equiv & A & \rightarrow N \end{array}$$

   Therefore $A \wedge C$ can be removed.

3.4 Label update algorithms

3.4.1 Overview

The preceding simple computation is inefficient, because the justifications for a node are full computed to generate its label [FK93, p. 433]. An algorithm, which only propagates the incremental changes of the labels, is described by Johan de Kleer and is adapted to the definitions. In this thesis this algorithm is called the basic algorithm. [FK93, p. 434f]

The focus algorithm extends the basic algorithm to delay environments, which are not a subset of any focus. This reduces the label computation cost, if not all contexts are required. An application for an FATMS is shown in the model-based diagnosis, when it is used for conflict determination. This means, that the inference engine (IE) is interested for the environments of the contradiction node. The ATMS computes all conflicts for all contexts at once, whereas the FATMS only detects conflicts for the contexts, which are in focus.

3.4.2 Basic algorithm

3.4.2.1 Methods

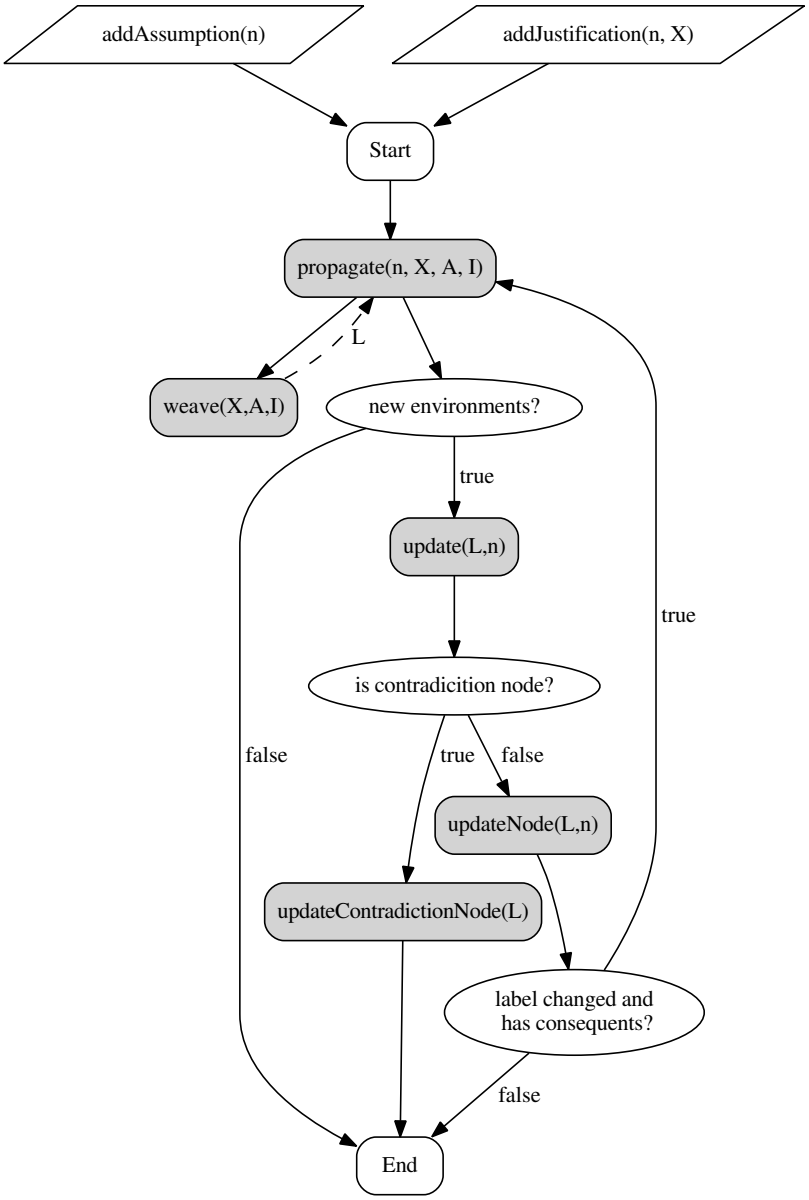

Figure 3.12: Control flow basic algorithm

The basic algorithm is initialized by the interface methods addAssumptions(n) and addJustifications(n,X). The method **propagate** is called for node n and invokes the functions **weave** in order to compute new environments. If new environments are computed, then the method **update** is called, which solely distinguishes if the current node n is the contradiction node or not.

If the current node is the contradiction node, then the method **updateContradicitionNode** is called and afterwards the control flow ends.

If the current node is not the contradiction node, then the method **updateNode** is called and the node's label is updated. If the node's label change and the node has consequents, then the method **propagate** is called recursively. Thereby the *incremental change*, which is a set of environments added to the node's label, are passed as argument. [FK93, p. 434f]

The methods are now described in detail. Afterwards an example is given.

propagate(n, X, A, I)

1. Parameters

   n: A node, whose label is going to change.

   X: Set of antecedents, whose consequent is n.

   A: A set for the preceding node. The node is wrapped in a set to enable to be empty.

   I: Set of environments, which is the incremental change.

2. Return value: void

The procedure propagate initializes the label update algorithm. If the interface method addJustification(n,X) is called, then their parameters are the consequent n and a set of antecedents X. The consequent's node and the nodes of the antecedents are passed to the method propagate as n and X. The argument A is empty, because the method is not called recursively and there is no predecessor. The argument I is a set with an empty environment, which has a neutral behavior in the following processing. An exemplary call for the justification $A \wedge B \rightarrow p$ is:

`propagate(`p`, {`A, B`}, ∅, {∅} )`

If the interface method addAssumption(n) is called, then its parameter n is a proposition. This proposition's node is passed to the method propagate as n. The parameters X and A are empty. The parameter I consists of the self environment of the node. An exemplary call for the assumption A is:

`propagate(`A`, ∅, ∅, {{`A`}})`

The argument A is always empty, if the label update algorithm is initialized. If the method propagate is called recursively, then A consists of the preceding (also

called antecedent) node and the set I consists of the environments, which are added to the antecedent node.

The method propagate calls the method **weave**(X,A,I) and receives a set environments L. If the set L is not empty the method **update**(L,n) is called.

weave(X, A, I)

1. Parameters

   X: Set of antecedents

   A: A set for the preceding node. The node is wrapped in a set to enable to be empty.

   I: Set of environments, which is the incremental change.

2. Return value: Set of environments

For each node in $X \setminus A$ the labels are gathered. The node in set A is disregarded, because only a subset of its label, the incremental change, is used. The Cartesian union is computed for the gathered labels and the set I. Conclusively in L all supersets and nogoods are removed and L is returned.

update(L,n)

1. Parameters

   L: Set of environments

   n: A node, whose label is going to change.

2. Return value: void

The method update receives a set of environments L and a node n. The sole task of update is to determine if n is a normal node or the contradiction node. If n is a normal node **updateNode**(L,n) is called. Otherwise **updateContradictionNode**(L) is called.

updateNode(L,n)

1. Parameters

   L: Set of environments

   n: A node, whose label is going to change.

2. Return value: void

This method updates the label L_n. At first all supersets in L of any environment L_n are removed. The set L is now the *incremental change*.

If L is not empty, new environments are discovered for L_n. Therefore the environments in L are added to L_n. Any supersets in L_n are removed. Then the incremental change is propagated to the consequents of n.

For each consequent $c \in C_n$ the sets of antecedents $X \in J_C$ are regarded. If n is a member of X, then the method **propagate**(n, X, A, I) is called with this set of antecedents. The arguments for the recursive call are

`propagate(`c`,` X`,` $\{n\}$`,` L`),`

where n is now the antecedent node and is wrapped in a set. The set L is the incremental change.

In circling dependency networks it is possible that L decrease during the recursively calls for each justification. Therefore if an environment of L is removed in L_n, then it is also removed in L. If L is empty an early termination is done.

updateContradictionNode(L)

1. Parameters

   L: Set of environments

2. Return value: void

The method receives a set of environments L. These environments are added to the label of the contradiction node. Subsequently in each node all supersets of the discovered nogoods are removed.

3.4.2.2 Example

The example of the preceding section is reused, but the order of justifications is changed to demonstrate the propagation procedure. The assumptions A, B, C ,D, E and the justifications $A \wedge B \rightarrow p$, $B \wedge C \wedge D \rightarrow p$, $D \wedge E \rightarrow q$, $p \wedge q \rightarrow r$ and the contradiction $A \wedge B \wedge E \rightarrow \bot$ are given. The dependency network can be seen in figure 3.13. [Ano02]

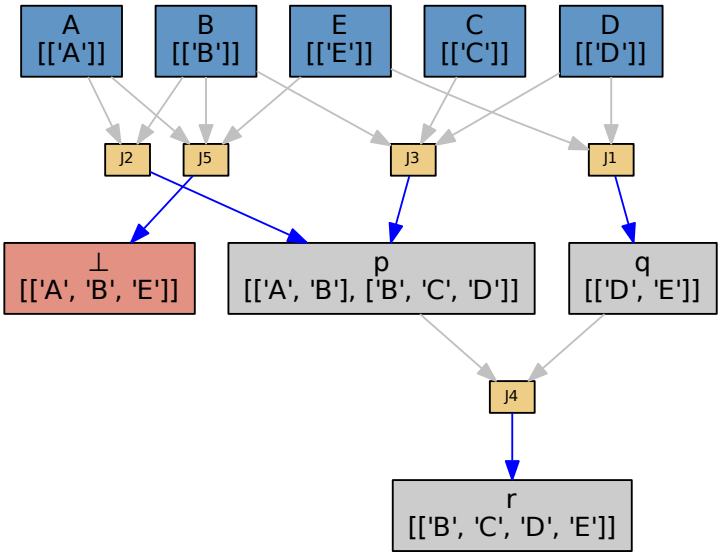

Figure 3.13: Initial situation for the basic algorithm example

In order to demonstrate the propagation procedure the justification $A \wedge C \rightarrow q$ is added. The algorithm is initialized by the interface method

```
addJustification( q, ['A','C'] ) .
```

Before the algorithm starts, in order to apply the node definition, the set of antecedents $\{A, C\}$ is added to the set J_p and q is added to the sets C_A and C_C. Then the method **propagate**(n,X,A,I) is called with

$$propagate(q, \{A, C\}, \{\}, \{\emptyset\}).$$

Subsequently the function **weave**(X,A,I) is called to generate L.

$$L = weave(\{A, C\}, \{\}, \{\emptyset\}).$$

In function weave at first $X \setminus A$ is computed to gather the nodes for Cartesian union. As the set A is empty, all antecedents of X are considered.

$$\{A, C\} \setminus \{\} = \{A, C\}$$

The Cartesian union is computed for the labels of antecedents and the set I. The

result is stored in L.

$$\begin{aligned} L &= \{E_A \cup E_C \cup E_I \mid E_A \in L_A \wedge E_C \in L_C \wedge E_I \in I\} \\ &= \{E_A \cup E_C \cup E_I \mid E_A \in \{\{A\}\} \wedge E_C \in \{\{C\}\} \wedge E_I \in \{\emptyset\}\} \\ &= \{\{A\} \cup \{C\} \cup \emptyset\} \\ &= \{\{A, C\}\} \end{aligned}$$

After checking L for supersets and nogoods, the label L is returned to the method **propagate**.

As L is not empty the method **update(L,n)**

$$update(\{\{A, C\}\}, q)$$

is called. And because q is not the contradiction node **updateNode(L,n)** is called.

$$updateNode(\{\{A, C\}\}, q)$$

In method updateNode the incremental change is computed. The set L is already the incremental change, because in $L = \{\{A, C\}\}$ is no superset of any environment of $L_q = \{\{D, E\}\}$. The environments of L are added to L_q.

$$L_q = \{\{A, C\}, \{D, E\}\}$$

The label computation for node q is finished, but the incremental change L has be propagated to all consequents of q (fig. 3.14). In this case, the consequent of q is node r.

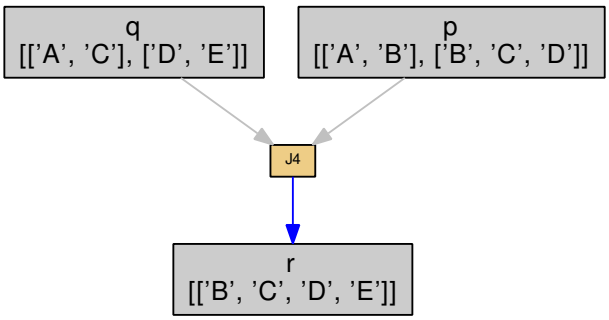

Figure 3.14: Consequent of node q is r

For each justification in $J_r = \{\{p, q\}\}$, where q is in the list of antecedents the method **propagate(n,X,A,I)** is called. Therefore propagate is called for the element $\{p, q\}$. For propagate the first parameter is the node r, which is the node going to

change, and the second parameter is the set of antecedents $\{p, q\}$. Furthermore the parameter A is the node q, which is wrapped in a set, and the parameter I is the incremental change L.

$$propagate(r, \{p, q\}, \{q\}, \{\{A, C\}\})$$

Thus **weave(X,A,I)** is called

$$L = weave(\{p, q\}, \{q\}, \{\{A, C\}\}).$$

In function *weave*, at first $X \setminus A$ is computed to gather the nodes for the Cartesian union, where node q is omitted.

$$\{p, q\} \setminus \{q\} = \{p\}$$

The Cartesian union is computed for the label of p and the set I. The result is stored in L.

$$\begin{aligned} L &= \{E_p \cup E_I \mid E_p \in L_p \wedge E_I \in I\} \\ &= \{E_p \cup E_I \mid E_p \in \{\{A, B\}, \{B, C, D\}\} \wedge E_I \in \{\{A, C\}\}\} \\ &= \{\{A, B\} \cup \{A, C\}, \{B, C, D\} \cup \{A, C\}\} \\ &= \{\{A, B, C\}, \{A, B, C, D\}\} \end{aligned}$$

In L the element $\{A, B, C, D\}$ is a superset and is removed. In L aren't any nogoods and therefore function weave returns $L = \{\{A, B, C\}\}$.

As L is not empty **update(L,n)**

$$update(\{\{A, B, C\}\}, r)$$

is called. Because r is not the contradiction node **updateNode(L,n)**

$$updateNode(\{\{A, B, C\}\}, r)$$

is called.

In method updateNode the incremental change is computed. The set L is already the incremental change, because in $L = \{\{A, B, C\}\}$ is no superset of any environment of $L_r = \{\{B, C, D, E\}\}$. The environments of L are added to L_r.

$$L_r = \{\{A, B, C\}, \{B, C, D, E\}\}.$$

The label computation for node r is finished. As r has no consequents no recursive propagation is done.

The calling method *updateNode* checks unnecessary if L is reduced, since all consequents are processed. Afterwards the control flow terminates.

3.4.3 Focus algorithm

3.4.3.1 Overview

The focus-ATMS (FATMS) is an extension of the ATMS, which delays environments to reduce the computational cost of labeling. In comparison to an ATMS, which works in all contexts in parallel, the FATMS is able to define several contexts. The contexts are defined by a *focus*, which is a set of environments. The environments of the focus and their subsets are the contexts for the FATMS. Therefore the FATMS in this thesis avoids the enumeration of all contexts (cf. [Tät97, p. 38-39]).

The FATMS processes and propagates environments only, if they are *in-focus*. An environment is in-focus, if it is a subset of any environment of the focus. An environment is *out-of-focus*, if it is not in-focus (cf. [Tät97, p. 38-39]). For example the focus F consists of the environments $\{A.B\}$ and $\{C\}$.

$$F = \{\{A.B\}, \{C\}\}$$

Then the contexts $\{\}$, $\{A\}$, $\{B\}$, $\{A, B\}$ and $\{C\}$ are in-focus. The contexts are $\{A, C\}$, $\{B, C\}$and $\{A, B, C\}$ out-of-focus.

In order to block environments, the node definition 3.1 for the FATMS is extended.

$$N \quad := \quad (C, J, L, bL) \tag{3.6}$$

The sets C, J and L are identical to the preceding definition. The set bL consists of environments and is called the *blocked label*. A set of environments in bL consists of the assumptions' identifiers.

Environments of the set bL depend on the current focus. Environments, computed by the label update algorithm, which don't agree to the current focus are added to the set bL. Their addition to the node's label L and their propagation in the dependency network is delayed. As it is possible to change the focus, it can not be stated that L consists only of in-focus environments and bL consists only of out-of-focus environments. If a focus changes, then the environments in the blocked label of a node, which become in-focus, are processed. But environments in L, which

become out-of-focus, remain in the label, because they are already processed and propagated (cf. [Tăt97, p.39]).

The preceding example with the assumptions A, B, C, D, E and the justifications $A \wedge B \rightarrow p$, $B \wedge C \wedge D \rightarrow p$, $A \wedge C \rightarrow q$, $D \wedge E \rightarrow q$, $p \wedge q \rightarrow r$ and $A \wedge B \wedge E \rightarrow \bot$ are reused to demonstrate the delaying of environments. In figure 3.15 the dependency network is shown, where below the node name the label L and the blocked-label bL are presented.

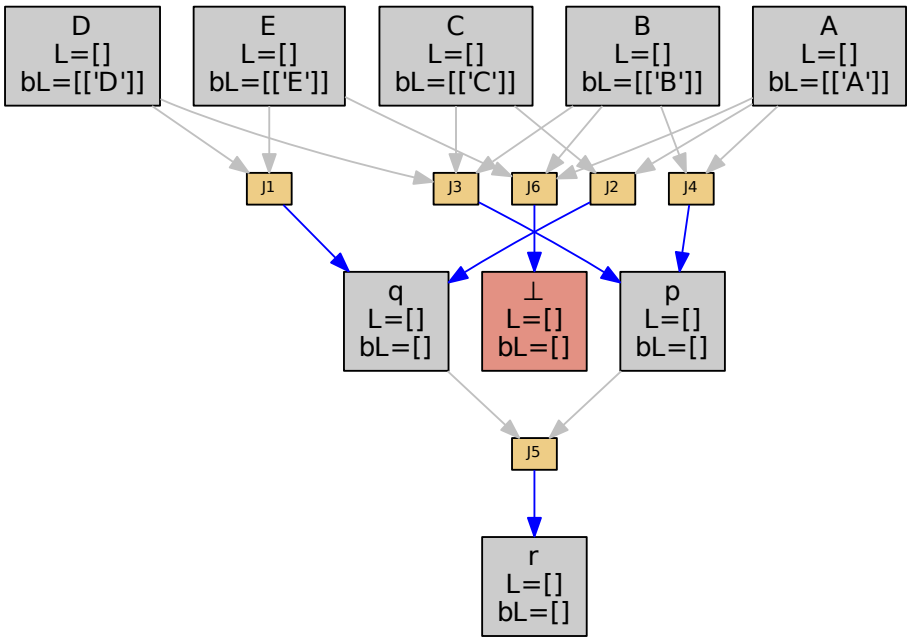

Figure 3.15: Initial situation, where the focus is empty

The focus of the FATMS is empty, therefore the self environments of the assumptions are blocked. This can be seen in fig. 3.15, where the self-environment is a member of the set bL. Hence, no environments are propagated, all labels are empty. When the environment $\{A, B, C\}$ is added to the focus, then all environments with the property in-focus are processed and propagated. The result can be seen in figure 3.16.

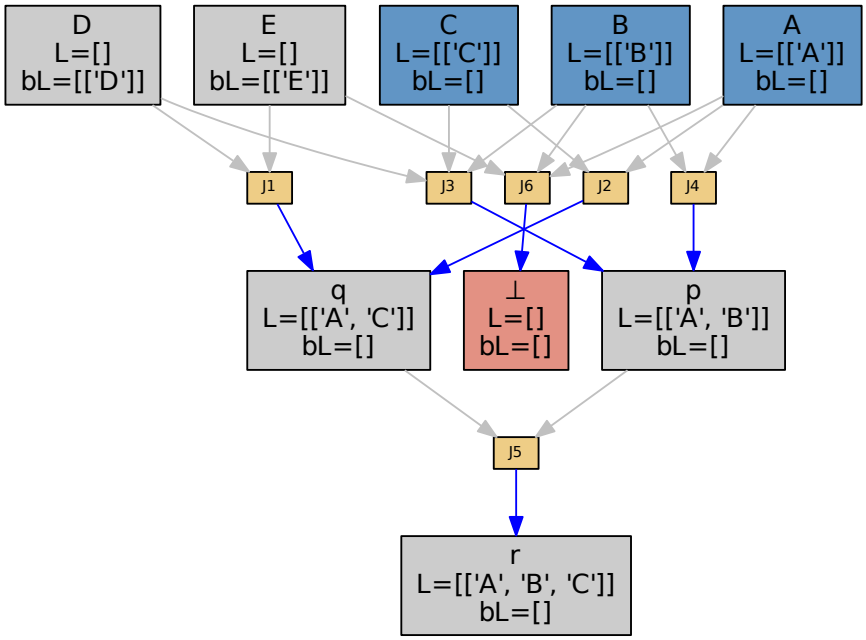

Figure 3.16: Environment $\{A, B, C\}$ is added to the focus

3.4.3.2 Interface

The interface of the ATMS is extended for the FATMS:

1. **extendFocus(X)** An environment is added to the focus of the FATMS. The label update algorithm is required for all nodes having in-focus environments in bL.

2. **changeFocus(sX)** The current focus of the FATMS is replaced by a given focus sX. The label update algorithm is required for all nodes having in-focus environments in bL.

3. **getFocus()**
   Returns the environments of the focus.

The properties completeness and consistency of a label are weaker than for the ATMS, as they are only relative ensured to the current focus (cf. [Tăt97, p.39], [TI94]). Therefore it is necessary to adapt the interface method isValidIn(n,X) for the FATMS.

4. **isValidIn(n, X)**
   This function returns a boolean flag. The method returns True, if the node n holds in context X. Therefore the environment X has to be consistent and a superset of any environment of n's label. In addition the environment X has to be in-focus.

3.4.3.3 Methods

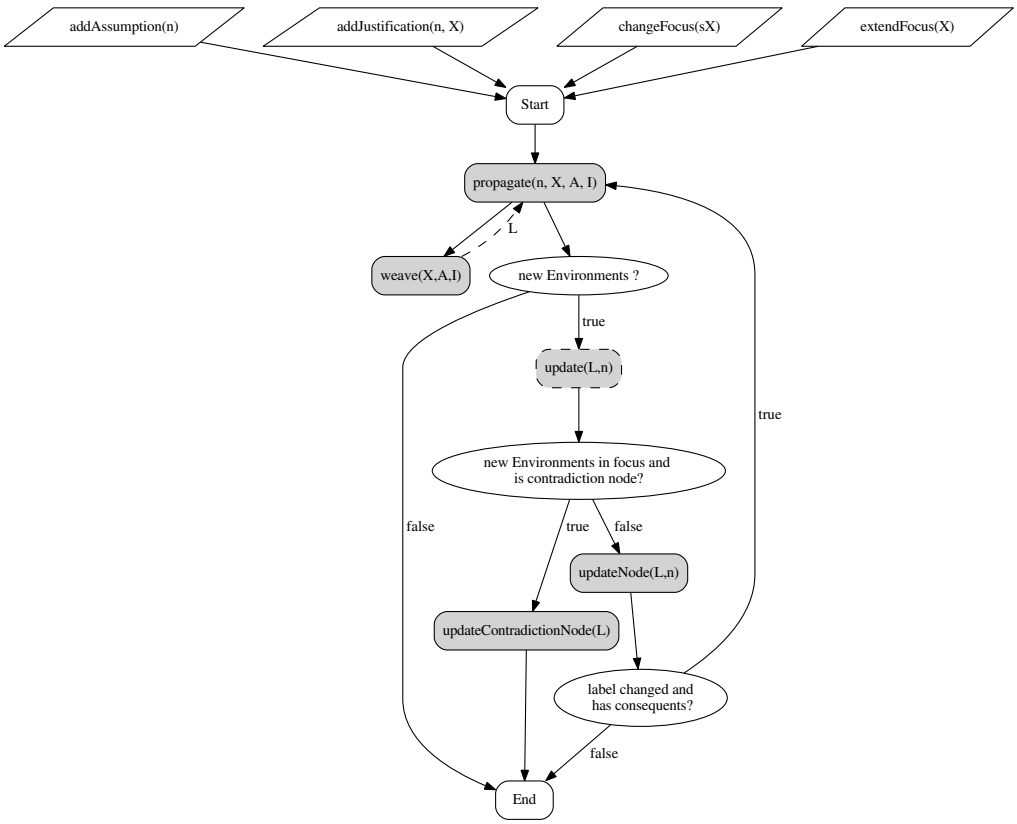

Figure 3.17: Contol flow focus algorithm

The focus algorithm extends the basic algorithm. The methods **propagate**, **updateNode**, **updateContradictionNode** and the function **weave** are reused. The method **update** is overridden in order to add the FATMS capabilities.

update(L,n)

1. Parameters

   L: Set of environments

   n: A node, whose label is going to change.

2. Return value: void

The method update of the basic algorithm is overridden. The function update receives the set of environments L and a node n. All environments of L, which are out-of-focus are added to the set bL_n and removed in L. Subsequently in bL_n all supersets are removed.

If in L are environments and n is the contradiction node, then the procedure **updateContradictionNode(L)** is called. Subsequently in all blocked labels all supersets of the environments in L are removed. If in L are environment and n is not the contradiction node, then the procedure **updateNode(L,n)** is called. Subsequently in bL_n all supersets of the environments in L are removed.

3.4.3.4 Example

The preceding example, where the environment $\{A, B, C\}$ is in the focus, is now continued (fig. 3.16).

By adding the environment $\{D, E\}$ into the focus

```
extendFocus(['D', 'E'])
```

the label update is initialized for the nodes D and E, as their blocked environments are now in focus. The justification $D \wedge E \rightarrow q$ computes the environment $\{D, E\}$. The environment is in-focus and therefore added to the label of q. The justification $B \wedge C \wedge D \rightarrow p$ computes the environment $\{B, C, D\}$. The environment is out-of-focus and therefore the environment is added to bL_p. The same applies to the environment $\{A, B, E\}$, which is computed by the justification $A \wedge B \wedge E \rightarrow \bot$. The environment is a member of $bL_\bot$. The result can be seen in figure 3.18.

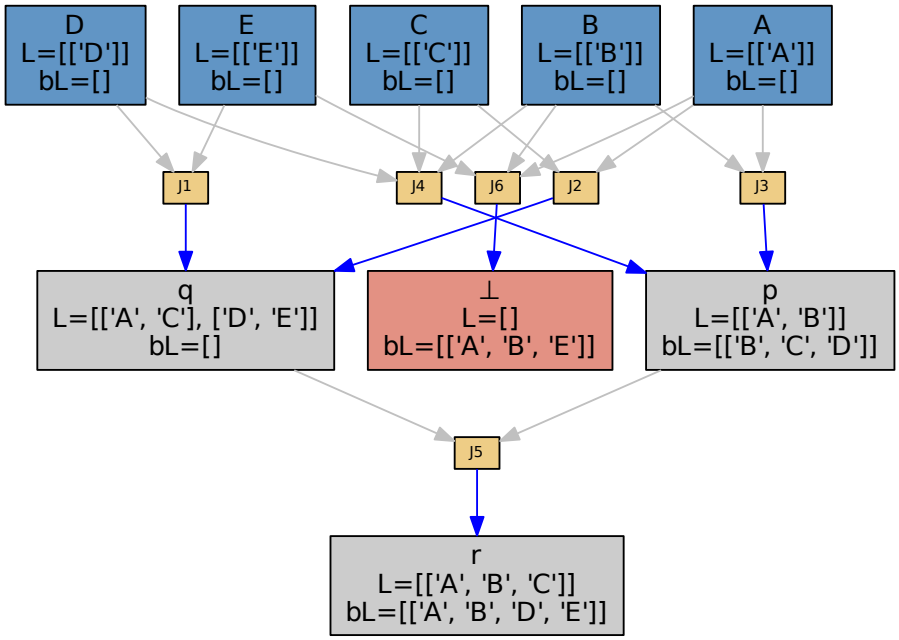

Figure 3.18: Environment $\{D, E\}$ is added to the focus

3.4.4 Circling dependency networks

Circular justifications are supported by the label update algorithm. The following example with the justifications $p \to q$ and $q \to p$ is considered. The dependency network is shown below.

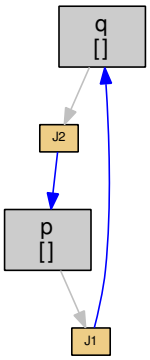

Figure 3.19: Circling example

The circulation has no effect, because both labels are empty. By adding assumption A and the justification $A \to p$, the environment $\{A\}$ is added to p and

propagated to q and from there back to p. There the algorithm terminates, because the environment $\{A\}$ is already a member of p's label (fig. 3.20).

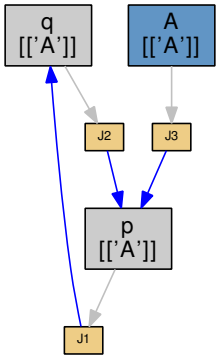

Figure 3.20: Circling example extended

Even if the loop back from q take into account a few more nodes, any environment from q would be a superset of the environment $\{A\}$ and therefore removed, so that the termination occur due the superset removal. This is shown in figure 3.21 (cf. [Kle86, p. 155]).

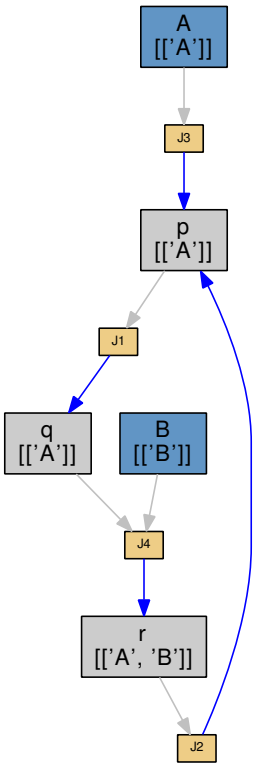

Figure 3.21: Circling example customized

3.5 Diagnosis example

The application of an ATMS is shown for a small example (cf. [PM10, p.209]). A simplified bath situation (fig. 3.22) is given, where a washbasin tap and a shower tap exist. If a tap is open, water flows in the washbasin or in the shower. Unless the plug of the washbasin or shower is not in water flows in the drain, otherwise the bath floor becomes wet. The ATMS is used to compute the minimal set of assumptions why water is on the floor. Therefore the example is modeled in Horn formulas.

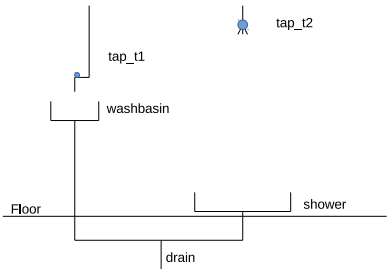

Figure 3.22: Bath diagnosis

The situation is modeled by the following rules.

$$
\begin{aligned}
washbasin\_wet &\leftarrow t_1\_open \\
water\_in\_drain &\leftarrow washbasin\_wet \wedge washbasin\_unplugged \\
water\_on\_floor &\leftarrow washbasin\_wet \wedge washbasin\_plugged \\
shower\_wet &\leftarrow t_2\_open \\
water\_in\_drain &\leftarrow shower\_wet \wedge shower\_unplugged \\
water\_on\_floor &\leftarrow shower\_wet \wedge shower\_plugged
\end{aligned}
$$

In order to apply the example with an ATMS, each rule is added by the interface method addJustification(n,X), for example

```
addJustification( 'washbasin_wet' , ['t1_open']).
```

Assumptions correspond to observable propositions. The assumptions for this example are

$$
\begin{aligned}
&t_1\_open \\
&t_2\_open \\
&washbasin\_unplugged \\
&washbasin\_plugged \\
&shower\_unplugged \\
&shower\_plugged
\end{aligned}
$$

They are added by the interface method addAssumption(n), for example

```
addAssumption( 't1_open' ).
```

As it is not possible, that the plug of the washbasin or the shower is in or not in at the same time, this is added by a contradiction. It is an integrity constraint, which ensures that conflicting propositions cannot be true at the same time (cf. [PM10, p.185]).

$$\bot \leftarrow washbasin\_unplugged \wedge washbasin\_plugged$$
$$\bot \leftarrow shower\_unplugged \wedge shower\_plugged$$

They are added by the interface method addNogood(X), for example

```
addNogood( ['washbasin_unplugged', 'washbasin_plugged'] ).
```

This kind of modeling is used, because the ATMS is limited to Horn formulas, which prevent negated literals in justifications (cf.[Kle86, p.143]). A rule like $(A \wedge \neg B) \rightarrow C$ is not a valid Horn formula. If the rule is transformed, it has two positive literals $(\neg A \vee B \vee C)$.

The resulting dependeny network is shown in the following figure 3.23.

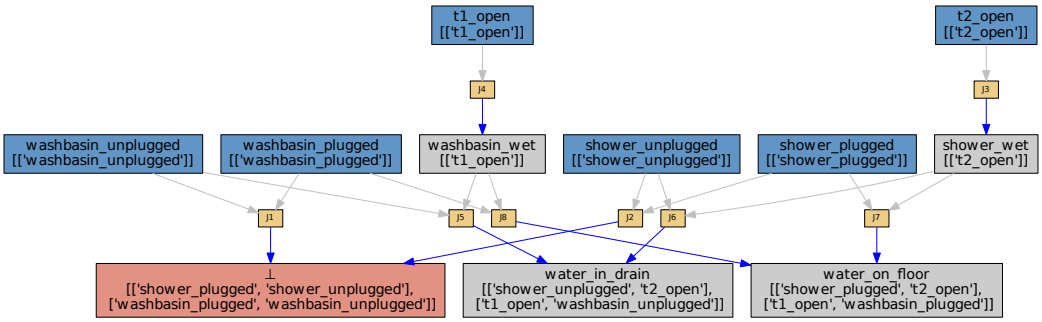

Figure 3.23: Dependency network for the bath diagnosis

The interface method `getEnvironments(n)` can be used to obtain the environments for a node. If the set of environments is empty, then no environment allows the derivation of that node. The proposition $water\_on\_floor$ has two minimal environments, which allow its derivation.

$$\left.\begin{array}{c} t_1\_open \wedge washbasin\_plugged \\ t_2\_open \wedge shower\_plugged \end{array}\right\} \vdash_{Justifications} \text{water\_on\_floor}$$

The interface method `isValidIn(n, X)` can be used to obtain a boolean flag if a node is valid for a given environment. A node holds in the provided environment

if the environment is a superset of any environment of the node's label and the environment is consistent. The environment is consistent, if it is not a superset of any nogood. An exemplary usage of the method isValidIn for the proposition *water_on_floor* would be true, if the assumptions {*t2_open*, *shower_plugged*} are given.

```
isValidIn('water_on_floor', ['t2_open', 'washbasin_plugged'] )
```

The interface method `getConflicts()` returns the set of environments, which allow the derivation of the contradiction node. Although Horn formulas do not allow disjunctions and negated literals to be input, they can be derived (cf. [PM10, p.185]).

$$\left.\begin{array}{c} washbasin\_plugged \wedge washbasin\_unplugged \\ shower\_plugged \wedge shower\_unplugged \end{array}\right\} \vdash_{Justifications} \bot$$

A nogood corresponds to a disjunction of negated literals. For example:

$$\neg shower\_plugged \vee \neg shower\_unplugged$$

This means, that at least one of the assumptions has to be $False$.

This small diagnosis example is solved with an ATMS, because the ATMS can immediately determine the minimal sets of assumptions for a proposition. Solving this example with an easy implemented inference engine, which uses the marking algorithm, which is an efficient algorithm to check satisfiability for Horn formulas (cf. [Sch08, p. 28]), requires the explicit computation of all contexts. In this example these are $2^6 = 64$ contexts, where 6 is the amount of assumptions. Thereby contradicted and minimal environments have to determined. The ATMS computes immediately the minimal consistent environments for all propositions and considers all contexts without explicit computing them all.

4

Model-based Diagnosis

4.1 Introduction

4.1.1 Overview

Model-based diagnosis provides a general domain independent approach to diagnosis. The knowledge about the *process of diagnosis* is separated from the knowledge about the *system under diagnosis*. Therefore the diagnosis engine can be reused for several applications. The diagnostic task is to determine why a system under diagnosis is not functioning as it was indented.(cf. [Tăt97, p. 6])

The system under diagnosis describes how components interact with each other. It is divided into components to reuse the components for different systems (cf. [Tăt97, p. 17], [Iwa15a]).

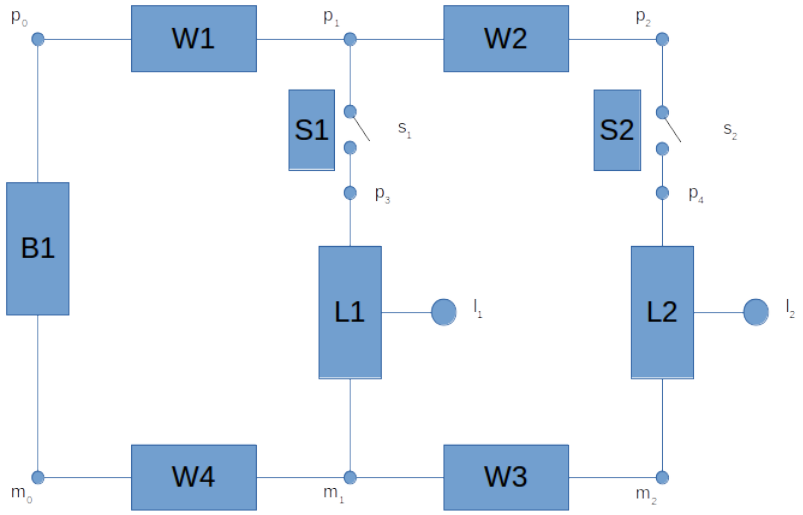

Figure 4.1: System model

The process of diagnosis is shown for the system in figure 4.1. The example consists of four types of components: a battery (B_1), wires (W_x), switches (S_x) and lamps (L_x). Each component has ports to interact with other components. For example wire W_1 is connected to port p_0 and p_1. In this scenario port p_1 is connected to three components W_1, W_2 and S_1. An additional junction for three components is not considered in this scenario to reduce the amount of components. The ports p_0, p_1, p_2, p_3, p_4 and m_0, m_1, m_2 correspond to links between components. The values for them can be either supply power p or ground g. The ports s_1, s_2, l_1 and l_2 enable to introduce observations to the diagnosis process. In case of a switch the position can be up or down. The position *down* indicates that power flows through the switch and *up* that no power flows through the switch. A lamp can either be *lit* or be *dark*.

The process of diagnosis uses conflicts triggered by observations to deduce components, which may be faulty (cf. [TI94, p.4].[Iwa15a]). For example if the switch position s_1 is down and the lamp l_1 is not lit then at least one component is faulty. Therefore for each component behavioral modes are defined, which describe how a component behaves in correct and faulty situations. In the sub process *conflict detection* the modes of the components and the observations are used to obtain conflicts (cf. [Iwa15b]). A conflict describes a discrepancy between a logical model of a system and the physical device. Subsequently these conflicts are used in the process *candidate elaboration*. A candidate is an assignment of a behavior mode to each component (cf. [Iwa15b]). The goal is to determine which behavior modes of the components can explain the given observations. If l_1 is not lit an explanation might be that the component $L1$ is broken, but another explanation could be that the battery is uncharged. These explanations are generated within the diagnose process and can be ordered by probability.

4.1.2 Behavioral modes

For each component behavioral modes are defined. At least one correct behavior mode is given, which is the first mode. Further modes are faulty modes, which are ordered by preference and probability (cf. [Iwa15b]). The behavioral modes of the types in the given example are shown in the following table 4.1. The table consist of the type name, the mode name, values for the variables of the type, rules for each mode, a probability for the mode and a description as necessary.

For example the type battery has two variables a_1 and a_2, which are assigned in this example to p_0 and m_0. Allowed values are supply voltage p and ground g. These variables are used to define rules for the modes. The rules are considered as,

if the component is given mode then the rules for that mode are valid. For example if $a_1 = p_0$ and $a_2 = m_0$ and the battery is in mode 1, then because of the empty list of antecedents $p_0 = p$ and $m_0 = g$ are valid. If the battery is in mode 2, then only $m_0 = g$ is valid. The rules for the battery have an empty list of antecedents, but consequentially the mode is also part of the list of antecedents, which is further shown.

Type	**Mode**	**Values**	**Rules**	**P**	**Description**
Battery	1	$a_1, a_2 \in \{p, g\}$	$() \rightarrow$ "$a_1 = p$" $() \rightarrow$ "$a_2 = g$"	0.8	Battery works as expected
	2		$() \rightarrow$ "$a_2 = g$"	0.2	Battery error
Wire	1	$a_1, a_2 \in \{p, g\}$	("$a_1 = p$") $\rightarrow$ "$a_2 = p$" ("$a_1 = g$") $\rightarrow$ "$a_2 = g$" ("$a_2 = p$") $\rightarrow$ "$a_1 = p$" ("$a_2 = g$") $\rightarrow$ "$a_1 = g$"	0.999	Power flows through the wire in both directions
	2			0.001	Wire error
Switch	1	$a_1, a_2 \in \{p, g\}$ $a_3 \in \{up, down\}$	("$a_1 = p$" $\wedge$ "$a_3 = down$") $\rightarrow$ "$a_2 = p$" ("$a_1 = g$" $\wedge$ "$a_3 = down$") $\rightarrow$ "$a_2 = g$" ("$a_2 = p$" $\wedge$ "$a_3 = down$") $\rightarrow$ "$a_1 = p$" ("$a_2 = g$" $\wedge$ "$a_3 = down$") $\rightarrow$ "$a_1 = g$"	0.98	Power flows through the switch in both directions, if the switch is down.
	2		("$a_1 = p$") $\rightarrow$ "$a_2 = p$" ("$a_1 = g$") $\rightarrow$ "$a_2 = g$" ("$a_2 = p$") $\rightarrow$ "$a_1 = p$" ("$a_2 = g$") $\rightarrow$ "$a_1 = g$"	0.01	Switch position down
	3			0.01	Switch error
Lamp	1	$a_1, a_2 \in \{p, g\}$ $a_3 \in \{lit, dark\}$	("$a_1 = p$" $\wedge$ "$a2 = g$") $\rightarrow$ "$a_3 = lit$" ("$a_1 = p$" $\wedge$ "$a_2 = p$") $\rightarrow$ "$a_3 = dark$" ("$a_1 = g$" $\wedge$ "$a_2 = g$") $\rightarrow$ "$a_3 = dark$" ("$a_1 = g$" $\wedge$ "$a_2 = sv$") $\rightarrow$ "$a_3 = lit$" ("$a_1 = p$" $\wedge$ "$a_3 = lit$") $\rightarrow$ "$a_2 = g$" ("$a_1 = p$" $\wedge$ "$a_3 = dark$") $\rightarrow$ "$a_2 = p$" ("$a_1 = g$" $\wedge$ "$a_3 = dark$") $\rightarrow$ "$a_2 = g$" ("$a_1 = g$" $\wedge$ "$a_3 = lit$") $\rightarrow$ "$a_2 = p$" ("$a_2 = p$" $\wedge$ "$a_3 = lit$") $\rightarrow$ "$a_1 = g$" ("$a_2 = p$" $\wedge$ "$a_3 = dark$") $\rightarrow$ "$a_1 = p$" ("$a_2 = g$" $\wedge$ "$a_3 = dark$") $\rightarrow$ "$a_1 = g$" ("$a_2 = g$" $\wedge$ "$a_3 = lit$") $\rightarrow$ "$a_1 = p$"	0.9	Lamp is lit, if on one port is power and on the other port is ground
	2		$() \rightarrow$ "$a_3 = dark$"	0.1	Lamp error

Table 4.1: Types and behavior modes

In order to apply a type to a component a unique component's name is necessary. Then a mode can be indicated by the component's name and the mode number. For example the mode names for the battery B1 are B1-1 and B1-2. The delimiter is an "-".

The modeling of the local behavior of a component cannot be automated so far. A model for a component should consider the balance between complexity and completeness. A model becomes more reliable the more accurate the model

is. Otherwise reasoning becomes harder with these models. The models of the components can be combined to generate several systems for the diagnosis (cf. [Tăt97, p. 18]).

4.1.3 Notations

The representation of the state of a system can be done in two notations. In the first notation the mode names are presented in a set. For example if all components are in mode one then the representation is

{ B1-1, W1-1, W2-1, S1-1, S2-1, L1-1, L2-1, W3-1, W4-1 }.

In the second notation the modes are ordered by their component (cf. [Iwa15b]). In the given example the static order is B1, W1, W2, S1, S2, L1, L2, W3, W4. If all components are in mode 1 then the representation is:

(111111111)

If there is no proposition made about a component in the first notation the component is omitted and in the second representation indicated by a zero. For example if L_1 is in mode 2 and it is the only proposition, then the representation is {L1-2} or (000002000). The first notation is the native structure of an ATMS, whereas the second notation is common in model-based diagnosis and is mostly preferred because of its shortness.

4.2 Conflict detection

4.2.1 Compute conflicts

A conflict is a set of behavioral modes, which form a contradiction. The conflict detection is the task of the ATMS. The system under diagnosis is added to the ATMS by applying interface methods.

To determine explanations for the observations the modes of the components are added to the ATMS as assumptions. For example for the component W1 the interface calls are:

```
addAssumption("W1-1")

addAssumption("W1-2")
```

The rules are transfered to justifications. In table 4.1 the rules are defined with variables. By adding a justification to the ATMS the variables are instantiated and the mode is added to the list of antecedents. For example the rule (”$a_1 = p$”) → ”$a_2 = p$” of mode W1-1 and the variables $a_1 = p_0$ and $a_2 = p_1$ is the justification (W1-1 ∧ ”$p_0 = p$”) → ”$p_1 = p$”. The interface call is:

```
addJustification("p1=p", [ "W1-1", "p0=p" ])
```

Further rules for mode W1-1 are added on the same way. In order to obtain conflicts contradictions are defined. For component W_1 the contradictions (p0=sv ∧ p0=g) → ⊥ and (p1=sv ∧ p1=g) → ⊥ are required. They indicate that at most one value exists at the ports p0 and p1. The interface calls for the contradictions are:

```
addNogood([ "p0=p", "p0=g" ])

addNogood([ "p1=p", "p1=g" ])
```

The resulting dependency network is shown in figure 4.2.

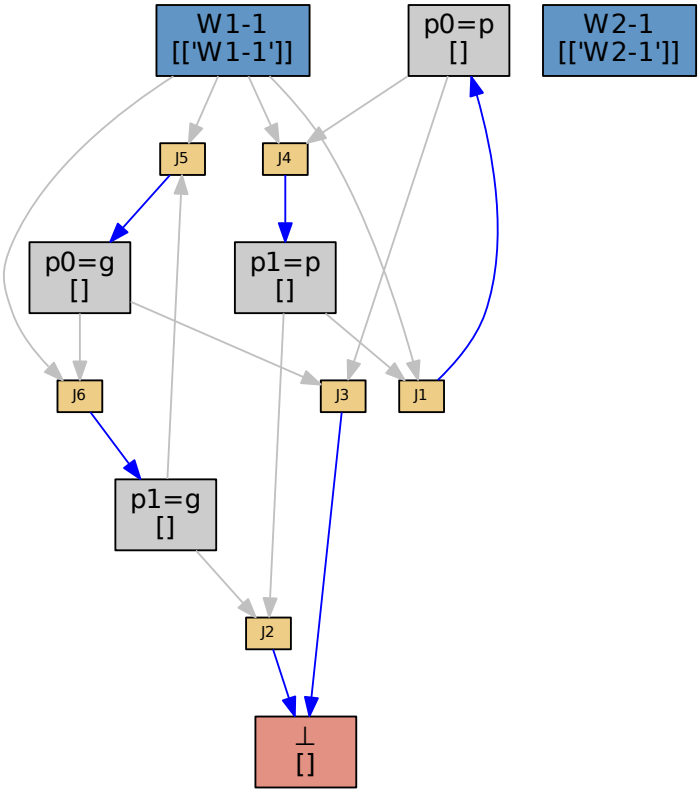

Figure 4.2: Dependency network for wire W1 with two modes

If the component B1 is also added, the dependency network increases, which is shown in figure 4.3. The contradiction for port m_0 is also added.

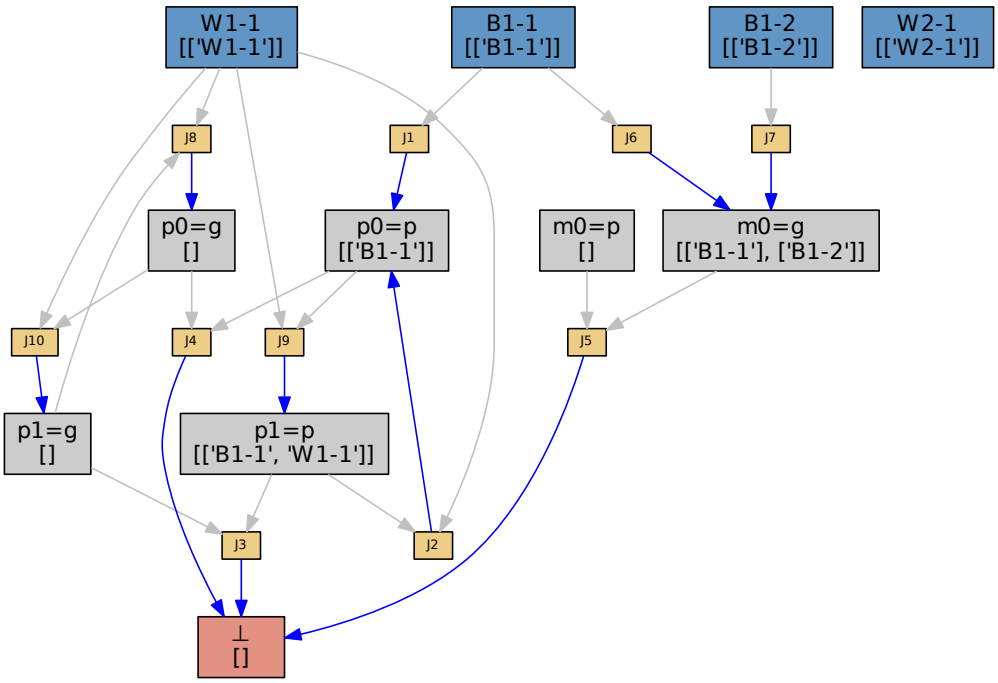

Figure 4.3: Dependency network is extended by component B1

Adding all components to the ATMS results in a large dependency network. The full list of contradiction is given below. All entries are added by the interface method `addNogood(X)`.

$$
\begin{aligned}
\text{p0=p} \wedge \text{p0=g} &\rightarrow \bot \\
\text{p1=p} \wedge \text{p1=g} &\rightarrow \bot \\
\text{p2=p} \wedge \text{p2=g} &\rightarrow \bot \\
\text{p3=p} \wedge \text{p3=g} &\rightarrow \bot \\
\text{p4=p} \wedge \text{p4=g} &\rightarrow \bot \\
\text{m0=p} \wedge \text{m0=g} &\rightarrow \bot \\
\text{m1=p} \wedge \text{m1=g} &\rightarrow \bot \\
\text{m2=p} \wedge \text{m2=g} &\rightarrow \bot \\
\text{s1=up} \wedge \text{s1=down} &\rightarrow \bot \\
\text{s2=up} \wedge \text{s2=down} &\rightarrow \bot \\
\text{l1=lit} \wedge \text{l1=dark} &\rightarrow \bot \\
\text{l2=lit} \wedge \text{l2=dark} &\rightarrow \bot
\end{aligned}
$$

Conflicts occur by adding observations to the ATMS. This is shown for a small example. In figure 4.4 the dependency network is shown for mode L1-2. The integrity constraint that port l_1 can either be dark or be lit at the same time is also added.

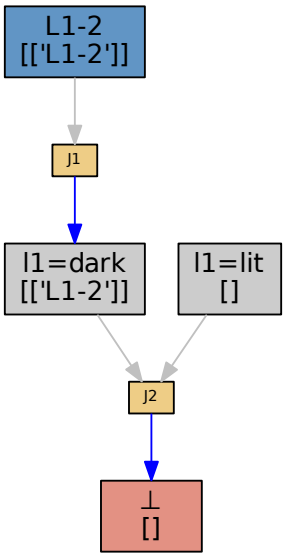

Figure 4.4: Dependency network for mode L1-2

Adding the observation that l_1 is lit results in a conflict. This is shown in figure 4.5. Introducing the observation $l_1 = lit$ is archived by the interface call:

```
addPremise("l1=lit")
```

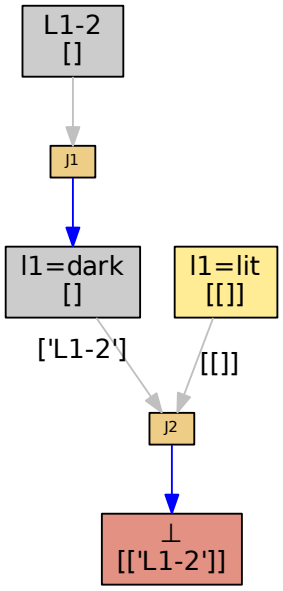

Figure 4.5: Conflict generated by introducing an observation

Subsequently the label update algorithm is applied. The labels of "l1=dark" and "l1=lit" are used to generate the label of the node ⊥, whereby the environment {L1-2} is added. As a new environment is added to the label of ⊥, all labels are adjusted for the contradicted environment. Therefore in the nodes "L1-2" and "l1=dark" the environment {L1-2} is removed. This means with respect to the observations, that the component L1 cannot be in mode 2, as this mode produces a conflict. The representation of the conflict in the second notation is (000002000).

The environments of the contradiction node are the conflicts for the model-based diagnosis and can be gathered by the interface method `getConflicts()`.

4.2.2 Mode conflicts

The ATMS can compute conflicts because it works in all contexts at once. A context is a combination of modes. As a component can at most be in one mode, contexts with more than one mode for a component do not need to be regarded. For example the ATMS does not need to regard the following context.

$$\{\text{B1-1}, \text{B1-2}, \text{W1-1}\}$$

The removal of such contexts is done by contradictions. Therefore the following contradictions are additionally added by the interface method `addNogood(X)`. As the second notation cannot present such mode conflicts they are indicated by "MCnf".

$$
\begin{aligned}
\text{B1-1} \wedge \text{B1-2} &\rightarrow \bot \\
\text{W1-1} \wedge \text{W1-2} &\rightarrow \bot \\
\text{W2-1} \wedge \text{W2-2} &\rightarrow \bot \\
\text{S1-1} \wedge \text{S1-2} &\rightarrow \bot \\
\text{S2-1} \wedge \text{S2-2} &\rightarrow \bot \\
\text{L1-1} \wedge \text{L1-2} &\rightarrow \bot \\
\text{L2-1} \wedge \text{L2-2} &\rightarrow \bot \\
\text{W3-1} \wedge \text{W3-2} &\rightarrow \bot \\
\text{W4-1} \wedge \text{W4-2} &\rightarrow \bot
\end{aligned}
$$

4.3 Candidate elaboration

4.3.1 Foundations

A candidate is an assignment of exactly one behavioral mode to each component of the system. The diagnosis process starts with the candidate (111111111), which indicates that all components are in behavioral mode one. If all components are not faulty no conflicts arise and the system is working properly. If in the example both switches are down and only one lamp is lit, then the system is not working properly and some of the modes are not in behavioral mode 1. The candidate generation process generates new candidates which explain these symptoms. A symptom is the discrepancy between the model and the physical system. A consistent candidate is also called a diagnosis.

The process of diagnosis is triggered by observations, whereby conflicts are generated [Iwa15a]. In the example the switch position and the lamplights can be observed. Possible values are:

$$
\begin{aligned}
s1 &\in \{up, down\} \\
s2 &\in \{up, down\} \\
l1 &\in \{lit, dark\} \\
l2 &\in \{lit, dark\}
\end{aligned}
$$

The interface of the diagnosis engine is as follows:

1. **makeDiag(A)**

The argument A is a dictionary, which contains an assignment of observation point and value. An exemplary call for makeDiag is

$$\text{makeDiag}(\{\ \text{s1: down, s2: down}\ \})\ .$$

The method adds the observations to the ATMS/FAMTS and computes the preferred candidates.

2. **getCandidates()**
   The method returns a list of pairs. The first element of the tuple is the candidate. The second element is the probability of the candidate. The probabilities for the modes of different components are indepedent to each other. Therefore the probability is the result of the multiplication of the probabilities for each mode (cf. [Iwa94, p.6]).

4.3.2 Preferred candidates

If a candidate becomes inconsistent by a conflict it is replaced by its successors. For example if we have an arbitrary system with four components C1, C2, C3, C4 and a conflict (1101) then the initial candidate (1111) becomes inconsistent. The detection that (1111) is inconsistent is done by a superset test:

$$\{\ \text{C1-1}\ ,\ \text{C2-1}\ ,\ \text{C4-1}\ \} \subseteq \{\ \text{C1-1}\ ,\ \text{C2-1}\ ,\ \text{C3-1}\ ,\ \text{C4-1}\ \}$$

The set of successors for (1111) are the candidates (2111), (1211), (1112). In each successor one behavior mode is increased. The successor (1121) is still inconsistent for the conflict and therefore not created. The process is shown in figure 4.6. Consistent candidates are indicated by the color green.

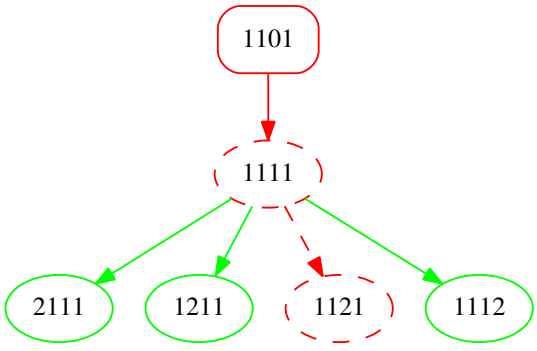

Figure 4.6: Successors for the start candidate

By continuously applying this candidate generation scheme the amount of candidates massively increases. To reduce the amount of relevant candidates an algorithm is used where candidates preferred by other candidates are not created. This is possible because the modes for a component are ordered by preference and probability (cf. [Tăt97, p.21]).

A candidate A is preferred to candidate B, if each mode of A is at most the number of the mode of candidate B (cf. [Iwa15b]). For example

(1211) is preferred to (2211).

If the preceding example is continued and the new conflict (2100) is discovered, then the candidate (2111) becomes inconsistent and is replaced by the successors (3111) and (2211). The successors (2121) and (2112) are inconsistent and therefore not created. This is shown in figure 4.7.

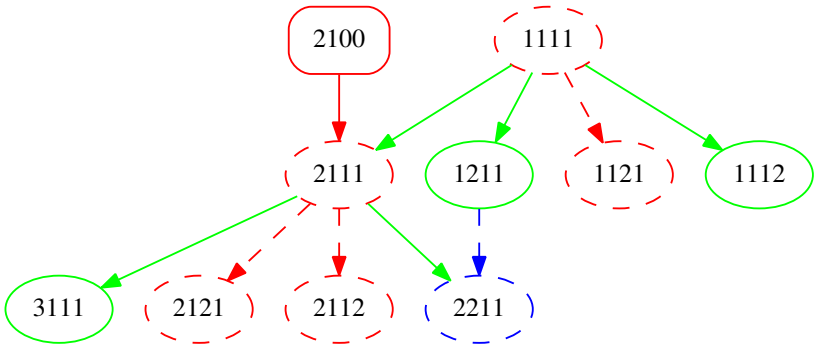

Figure 4.7: Elimination of preferred candidates

The successors are verified that they are not preferred by any existing consistent candidates. The candidate (3111) is not preferred by any existing candidate and therefore it is a new consistent candidate. The successor (2211) is preferred by the existing candidate (1211) (blue link) and therefore the candidate (2211) is not created.

The algorithm for generating candidates receives all existing conflicts, the current consistent candidates and a new conflict. The algorithm returns a set of consistent candidates:

genCandidates(cnfs, candidates, new_cnf)

1. Verify that the current candidates are consistent for the new conflict and remove inconsistent candidates from the set of consistent candidates.

2. Generate recursively all successors of the inconsistent candidates. Successors, which are inconsistent for the same conflict as its predecessor are not created.

3. If a successor has no preferred candidate, the successor is added to the set of consistent candidates.

4. Return consistent candidates

The application of the algorithm is shown by continuing the preceding example, where the conflict (0002) is added. The algorithm is called with the existing conflicts, the consistent candidates and the new conflict (0002).

$$\begin{aligned}\texttt{genCandidates(}\ & \{(1101),(2100)\},\\ & \{(1211),(1112),(3111)\},\\ & (0002)\\ \texttt{)}\ &\end{aligned}$$

The algorithm discovers, that candidate (1112) is inconsistent and removes it from the set of consistent candidates. Afterwards the successors are generated for this candidate. As in fig 4.8 is shown, the only consistent successor is (1113). Further candidates are inconsistent because of the same conflict as its predecessor. The candidate (1113) is not preferred by any other candidate and is added to the set of consistent candidates. In the end the set of consistent candidates $\{(1211),(1113),(3111)\}$ is returned.

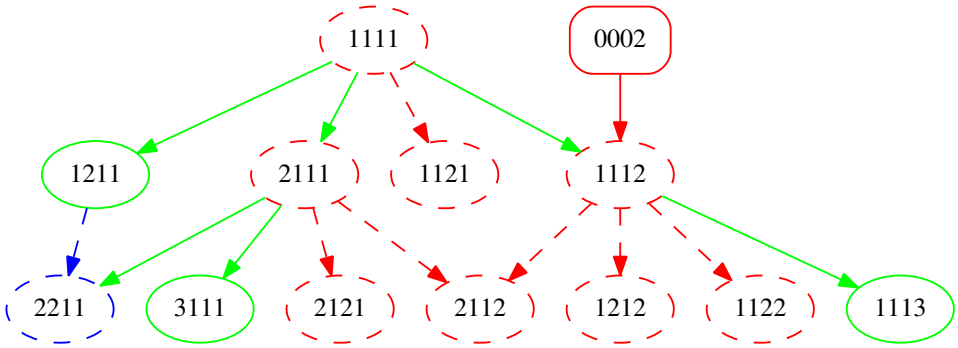

Figure 4.8: Applying the candidate generation algorithm

The algorithm is described in [Iwa15b]. The reference also gives an idea for the preference lattice of the canidates.

4.3.3 Basic diagnosis

The basic diagnosis uses the ATMS. If an observation is given by the interface method makeDiag(A), the observation is added to the ATMS and the conflicts are determined. For each new conflict the candidate generator algorithm is called. The process of the basic diagnosis is:

1. Add observation.
2. Detect conflicts with the help of the ATMS.
3. Generate candidates for each new discovered conflict.

Consider the following example. The observations that lamp 1 is dark and switch 1 is down are done. These observations are added to the diagnosis engine by calling

```
makeDiag('s1': 'down', 'l1': 'dark') .
```

These observations are added to the ATMS and conflicts are computed. The method makeDiag calls the inferface method of the ATMS `addPremise(n)` for each observation. This is indicated in the over-dimensioned figure 4.11, where the observations have the color yellow and the conflicts are in the contradiction node. The mode conflicts "MCnf" are hidden in this figure. They are necessary for the conflict generation, but irrelevant for the candidate generation. The discovered conflicts are $\{(110201010), (111211111)\}$. They are obtained by the interface method of the ATMS `getConflicts()`.

For each conflict the candidate generation algorithm is called. The set of initial conflicts is empty, the set of initial candidates is $\{(111111111)\}$ and the first processed conflict is (110201010).

```
genCandidates({}, {(111111111)}, (110201010))
```

The initial candidate does not become inconsistent therefore the set of consistent candidates is returned unchanged (fig. 4.9).

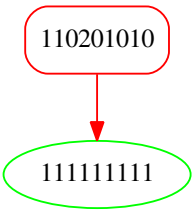

Figure 4.9: Processing of conflict (110201010)

By processing the second conflict the arguments for the algorithm are

`genCandidates({(110201010)}, {(111111111)}, (110101010)) .`

The process of the algorithm is illustrated in figure 4.10. The successor (111211111) is inconsistent because of the first conflict (110201010) and therefore a recursion is done.

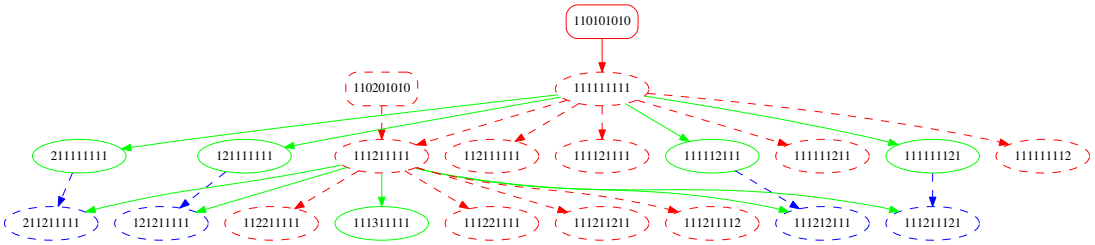

Figure 4.10: Processing of conflict (110101010)

The consistent candidates are gathered by the interface method `getCandidates()`. The consistent candidates, ordered by probability, are:

	Candidate	Probability
1	(211111111)	15.5 %
2	(111112111)	6.9 %
3	(111311111)	0.63 %
4	(111111121)	0.06 %
5	(121111111)	0.06 %

Table 4.2: List of candidates with probabilities

The highest probability is, that the battery is not working, followed by probability that the lamp 1 is broken. The third candidate indicates that the switch is broken and the last two indicate wire errors.

In order to demonstrate the capabilities of the dependency network the following observations are added additionally.

`makeDiag('s2': 'down', 'l2': 'lit')`

Then the first three candidates change, because now it is not probable anymore that only the battery is uncharged. Additional to the battery another component has to be in a faulty mode.

	Candidate	Probability
1	(111112111)	6.9 %
2	(111311111)	0.63 %
3	(211131111)	0.16 %

Table 4.3: Extract of the first three candidates with probabilities

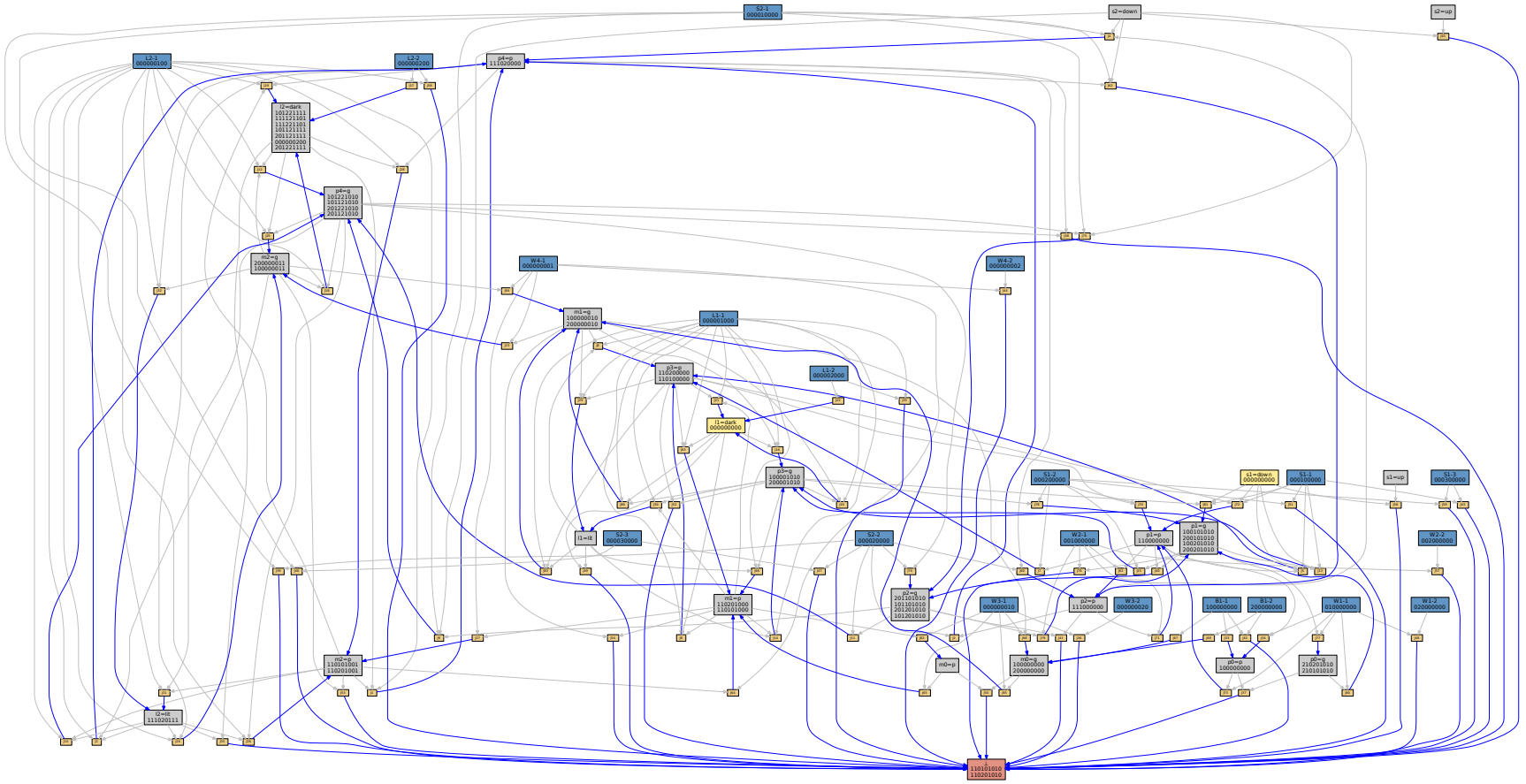

Figure 4.11: Dependency network with all components and introduced observations for the basic diagnosis

4.3.4 Focus diagnosis

The Focus diagnosis bases on the FATMS. In large systems it is not feasible to consider all possible mode combinations at once, therefore the contexts, looking for conflicts, are highly restricted (cf. [Tät97, p.21]).

The focus diagnosis starts with the initial candidate (111111111) and this is the only context the FATMS is searching for conflicts. For clarification all assumptions in the FATMS unequal to the first mode are not propagated. This is shown in figure 4.14, where some assumptions, usually displayed by a blue box, are gray, because their self environment is delayed.

The diagnosis process for the focus diagnosis differs to the basic diagnosis as the consistent candidates, computed by the candidate generation algorithm, replace the current focus of the FATMS. Subsequently the conflict detection is done. This process is repeated until no new consistent candidates are discovered. The full diagnosis process is as follows:

1. Add observation.
2. Do, while new candidates are found.
   (a) Replace current focus with the set of consistent candidates.
   (b) Detect conflicts with the help of the FATMS.
   (c) Generate candidates for each new discovered conflict.

The preceding example is reused for the focus diagnosis. The same algorithm is used to generate the candidates, therefore the resulting candidates are the same. The example shows how the process differs to the basic diagnosis, because the focus diagnosis has to change the focus in order to obtain new conflicts. By calling

```
makeDiag('s1': 'down', 'l1': 'dark')
```

the diagnosis process is started. The observations are added to the FATMS by the interface method `addPremise(n)` and the conflicts are obtained by the interface method `getConflicts()`. The first discovered conflict is (110101010). In comparison to the basic diagnosis both conflicts were discovered at once. In the next step the consistent candidates are computed.

```
genCandidates({}, {(111111111)}, (110101010))
```

The result can be seen in the following figure 4.12.

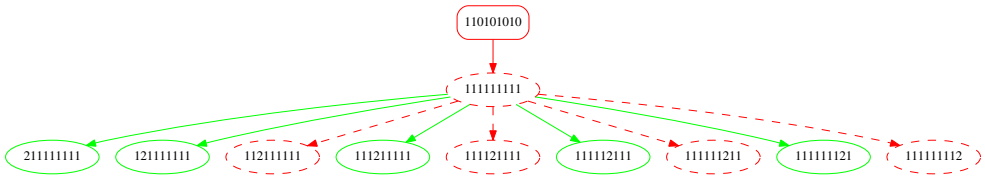

Figure 4.12: Processing of conflict (110101010)

As new candidates are discovered, the process is repeated. The consistent candidates replace the current focus of the FATMS to detect new conflicts. This is done by the interface method of the FATMS `changeFocus(sX)`, where sX is the set of consistent candidates. Subsequently the conflicts are obtained by the interface method `getConflicts(⊥)`.

Thereby the conflict (110201010) is discovered. Applying the candidate generator with the old conflicts, the consistent candidates and the conflict (110201010) are shown in the following figure 4.13.

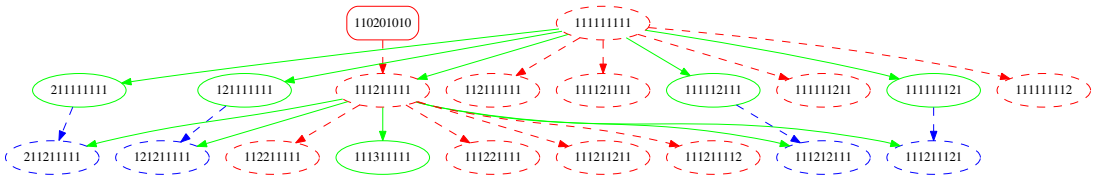

Figure 4.13: Processing of conflict (110201010)

By subsequently adding the consistent candidates to the focus, no new conflicts are discovered and therefore no new candidates are found and the process ends. The dependency network for the diagnosis process is shown in the over-dimensioned figure 4.14.

The added mode conflicts to the FATMS are not necessary for the focus diagnosis, because as long as the candidate generator does not create not allowed candidates, these mode conflicts are not in any focus. As they are never propagated, they remain in the delayed label of the contradiction node.

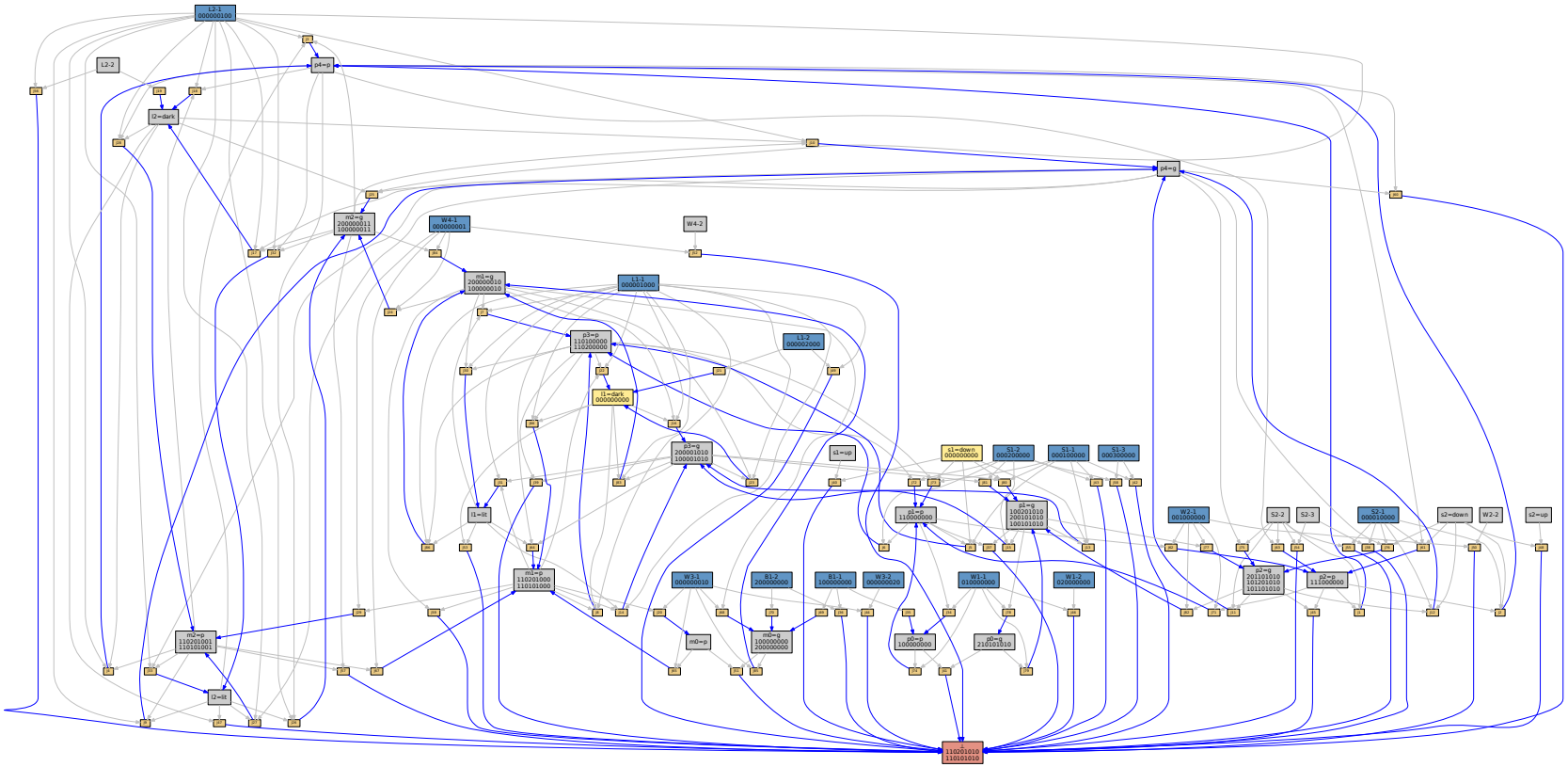

Figure 4.14: Dependency network with all components and introduced observations for the focus diagnosis. In comparison to the basic diagnosis some assumptions, for example L2-2 (top left), are not propagated.

4.4 Comparison of the diagnosis engines

The basic diagnosis and the focus diagnosis use the same candidate generation algorithm. Therefore this is not the subject of the comparison. The basic diagnosis uses the general ATMS and the focus diagnosis uses the FATMS. The aim of the FATMS is to reduce the label computation by delaying environments. Therefore the total amount of different environments in an ATMS/FATMS (diff envs) and the average label length of labels with environments (avg label length) are compared for both diagnosis engines.

	Observation	Basic diagnosis		Focus diagnosis	
		diff envs	avg label length	diff envs	avg label length
1	Initial situation	43	1.5	0	
2	s1=down, l1=dark	72	1.9	39	1.4
3	s2=down, l2=lit	121	3.3	109	2.9

Table 4.4: Comparison of the diagnosis engines

The comparison in table 4.4 shows the amount of different environments and the average label length in the order the observations are added to the diagnosis engines. The initial situation describes the preparation of the ATMS and FATMS. The ATMS cannot delay environments and therefore 43 environments are created during the initializing of the dependency network. The average length of a label is about 1.5 environments per label. In the FATMS no focus is set, therefore no environments are propagated.

By adding the first observation $s1 = down$ and $l1 = dark$ to both diagnosis engines, the FATMS is able to reduce the label computation. The amount of different environments is significantly reduced and the label length is in average shorter than for the ATMS. By adding the second observation $s2 = down$ and $l2 = lit$ to both diagnosis engines the advantage of the FATMS decreases, but the amount of different environments and the average label length is still lower than for the ATMS. Therefore the label computation is reduced for both observations. The reference [TI94] can be used for further reading.

5

Prototypical implementation

5.1 Overview

The prototypical implementation consists of two parts. The first one is the ATMS and the second one is the diagnosis engine, which depends on the first one.

The prototypes are implemented in Python 2.7. The main functions can be used without additional modules. In order to draw dependency networks the tool and module *graphviz* is required. Further the module *nose* is a test framwork and is used for the ATMS/FATMS.

5.2 ATMS

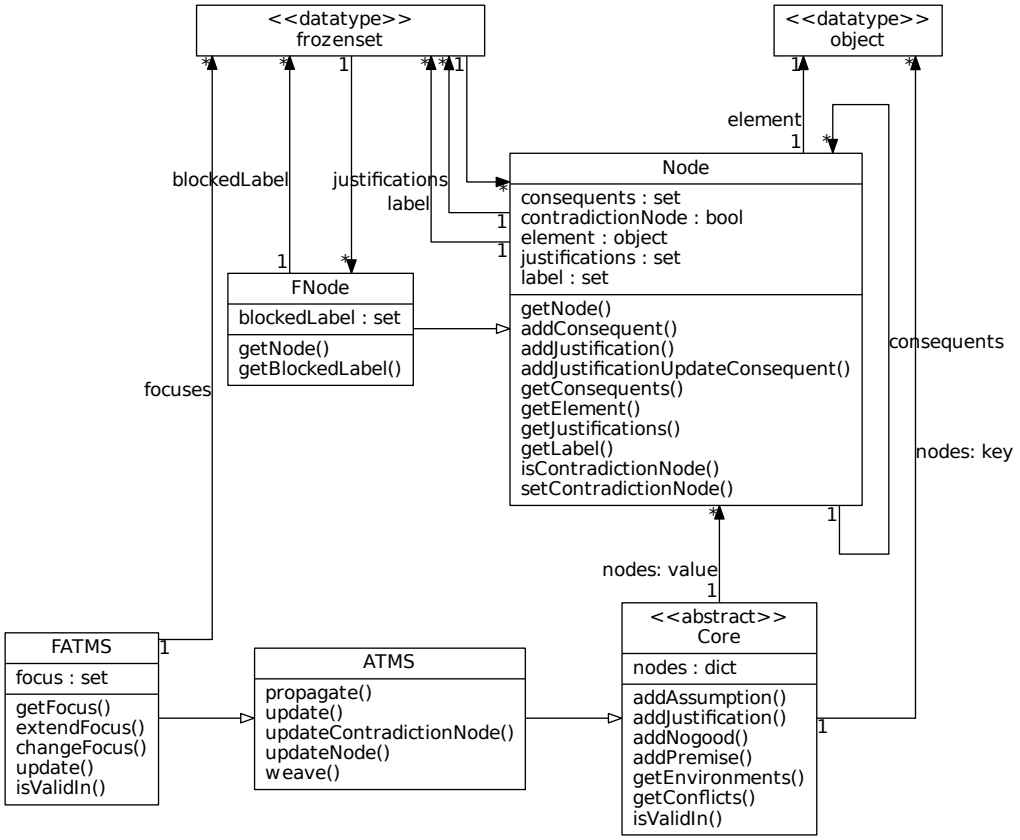

Figure 5.1: UML-Diagram ATMS

The class Core is an abstract class and maintains the structure of the ATMS. Each proposition, which is a distinct hashable object, is associated to a node (cf. [Ano]). Therefore the variable *nodes* is a dictionary, where to each proposition a Node-object is mapped.

The classes ATMS and FATMS can be instantiated to provide the different ATMS variants. The class ATMS inherits from the class Core and provides the label update algorithm. The FATMS inherits from the class ATMS and extends the label update algorithm to provide the FATMS capabilities.

The class Node represents the definition of a node. The set *consequents* contains recursively Node-objects. The datatype frozenset is a built-in data type to represent sets in the build-in data type set (cf. [Ano]). This construct is used for the set *label* and the set *justifications*, where sets of Node-objects are stored. As the ATMS

works only with references, by calling the interface method getEnvironments() the references are replaced with the actual objects and therefore the variable *element* is necessary. The class FNode inherits from the class Node and extends the Node-object with the set *blockedLabel*.

An exemplary usage of the ATMS/FATMS can be seen in the appendix B.1, B.2.

5.3 Diagnosis

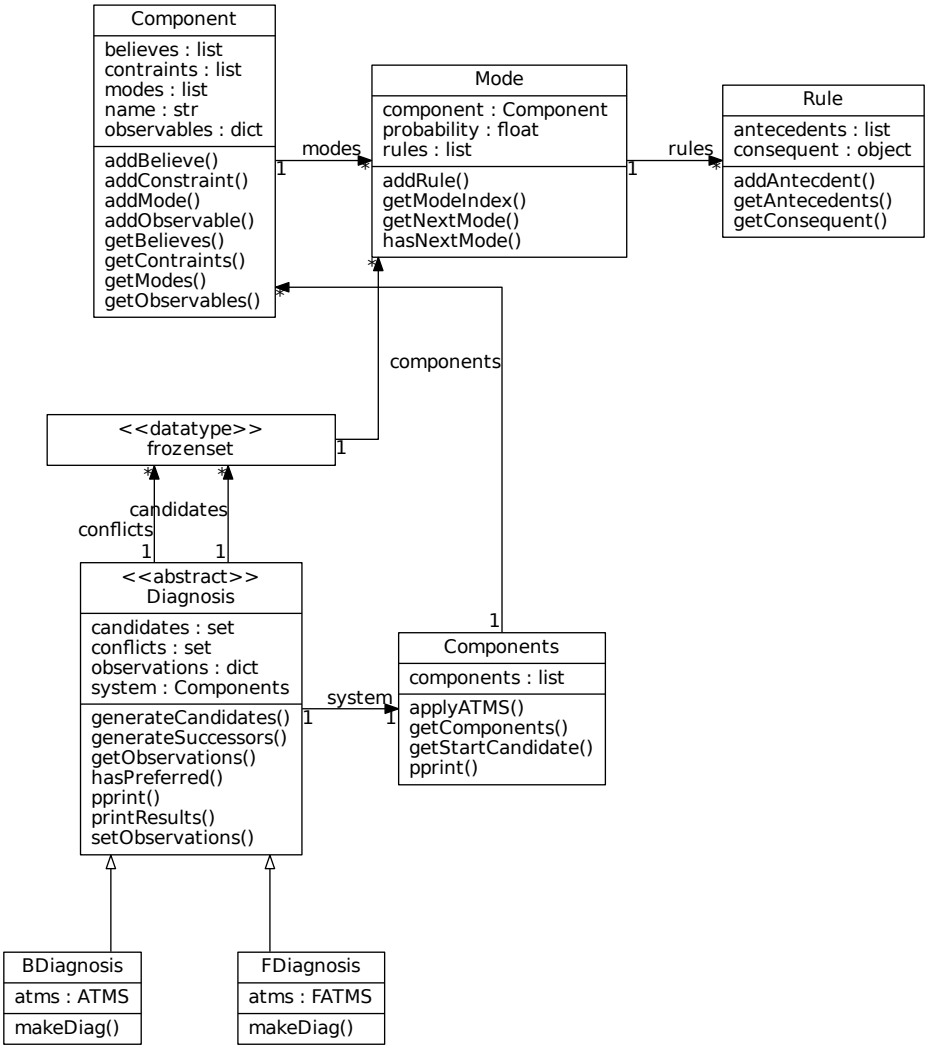

Figure 5.2: UML-Diagram Diagnosis

A component consists of modes and each mode has associated rules. This relation is mapped to the classes Component, Mode and Rule. The class Components provides the structure of a system.

The class Diagnosis provides the candidate generation methods for the diagnosis process. The subclasses BDiagnosis and FDiagnosis can be instantiated. Therefore a system has to be passed. The class BDiagnosis uses the classic ATMS, whereas the FDiagnosis uses the FATMS.

Each Mode-object is an assumption. Therefore *conflicts* in the class Diagnosis, computed by the AMTS/FATMS, consist of sets of Mode-objects. This is indicated by the built-in datatype frozenset. The same applies to the set of candidates, which are used additionally as focuses for the FATMS.

An exemplary usage of the diagnosis engine can be seen in the appendix B.3.

6

Conclusion

This thesis describes an assumption-based truth maintenance system (ATMS), which can be used for hypothetical reasoning and reason maintenance. Therefore the ATMS is introduced and its operation described. The focus-ATMS (FATMS) is an extension of the ATMS and attempts to reduce the computational cost. The ATMS/FATMS is applied for model-based diagnosis, which is a systematic approach to diagnosis. The diagnosis engine processes conflicts, computed by the ATMS/FATMS, to generate candidates for the system under diagnosis. In order to improve the comprehensibility the ATMS/FATMS and the diagnosis engine are prototypically implemented.

The introduction of the ATMS begins with a motivation and is followed by the foundations, where the basic terms are described. Each proposition has an associated label, which is maintained by the ATMS. A definition of the label and its importance for the derivation of the label's node is presented. Then an interface is given for the ATMS, which presents how the ATMS can be used. The label computation is presented for an example. Therefore a node definition and an inductive process for generating environments is presented. Succesivley non-minimal and inconsistent environments are removed. Furthermore a label update algorithm is described, which propagates only the incremental change. The FATMS is an extension of the ATMS. Therefore the node definition and the label update algorithm is extended to delay environments, which are out-of-focus. The label update algorithms are described by an example. Conclusively a diagnosis example for the application of the ATMS is given.

The model-based diagnosis is presented for an exemplary circuit. In the beginning an overview of the diagnosis process is given and behavior modes for the types of the circuit's components are defined. These types are used to define a system under diagnosis. A diagnosis engine uses the ATMS/FATMS to determine conflicts in

a system under diagnosis. The interaction between the diagnosis engine and the ATMS/FATMS is presented by the interface calls of the ATMS/FATMS. Subsequently by providing observations the ATMS/FATMS computes conflicts for the candidate elaboration. In order to reduce the amount of candidates an algorithm is applied to compute only the most preferable candidates. The diagnosis engine is distinguished by two types. The basic diagnosis uses the ATMS and the focus diagnosis uses the FATMS. The process of diagnosis is compared for both types. Conclusively a comparison of the label computation of the ATMS and FATMS for the same diagnosis tasks is presented.

The thesis accomplishes the targeted aims. The ATMS/FATMS is described and prototypically implemented. It can be used by an interface for hypothetical reasoning. A simple example is given in this thesis, which can be adapted. In order to analyze an ATMS/FATMS, the dependency network can be visualized. This is an helpful feature to create a knowledge base. The model-based diagnosis is described for an example and the prototypical implementation can be used to diagnose further simple circuits. The interaction between the inference engine and the ATMS/FATMS is exemplary presented by the interface calls of the ATMS/FATMS, which leads to an improved understanding of the compounded architecture. The comparison of the delaying of environments has shown that the FATMS is able to reduce the amount of environments and therefore the label computation.

Bibliography

[Ano] Anonymous. *Built-in Types*. URL: https://docs.python.org/2/library/stdtypes.html#set (visited on 05/26/2016).

[Ano02] Anonymous. *Assumption-Based Truth Maintenance*. 2002. URL: https://www.cse.unsw.edu.au/~billw/cs9414/notes/kr/atms/atms.html (visited on 05/26/2016).

[BK08] C. Beierle and G. Kern-Isberner. *Methoden wissensbasierter Systeme*. 4th ed. Wiesbaden: Viewweg + Teubner, 2008.

[FK93] K. Forbus and J. de Kleer. *Building Problem Solvers*. Reading, Massachusetts: MIT Press, 1993.

[Iwa15a] S. Iwanowski. *Model-Based Reasoning*. Wedel, 2015. URL: http://www.fh-wedel.de/fileadmin/mitarbeiter/iw/Lehrveranstaltungen/2015WS/AAI/AAI4.4.pdf (visited on 08/23/2016).

[Iwa15b] S. Iwanowski. *Model-Based Reasoning, Details*. Wedel, 2015. URL: http://www.fh-wedel.de/fileadmin/mitarbeiter/iw/Lehrveranstaltungen/2015WS/AAI/AAI4.4Details.pdf (visited on 08/23/2016).

[Iwa94] S. Iwanowski. *An Algorithm for Model-Based Diagnosis that Considers Time, Annals of Mathematics and Artificial Intelligence 11*. Baltzer, 1994.

[Kle86] J. de Kleer. *An Assumption-based TMS*. North-Holland: Elsevier Science Publishers, 1986.

[PM10] D. Poole and A. Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge: Cambridge University Press, 2010. URL: http://artint.info/ (visited on 06/13/2016).

[Ros12] K. Rosen. *Discrete Mathematics and Its Applications*. New York: McGraw-Hill, 2012.

[Sch08] U. Schöning. *Logic for Computer Scientists*. Boston: Birkhäuser, 2008.

[Tăt97] M. Tătar. *Dependent defects and aspects of efficiency in model-based diagnosis*. Hamburg, 1997. URL: https://www.researchgate.net/publication/35729370 (visited on 07/14/2016).

[TI94] M. Tătar and S. Iwanowski. *Aspects of Efficient Focusing, 5th International Workshop on Principles of Diagnosis (DX)*. New Paltz (USA), 1994.

Appendices

A

Content of the CD

1. Master thesis
2. Implementation
   (a) ATMS/FATMS
   (b) Diagnosis Engine

B

Program examples

B.1 ATMS

```
from ATMS import ATMS

atms = new ATMS()

atms.addAssumption('A')
atms.addAssumption('B')
atms.addAssumption('C')
atms.addAssumption('D')
atms.addAssumption('E')

atms.addJustification('p', ['A', 'B'])
atms.addJustification('p', ['B', 'C', 'D'])
atms.addJustification('q', ['A', 'C'])
atms.addJustification('q', ['D', 'E'])

atms.addJustification('r', ['p', 'q'])

atms.addNogood(['A', 'B', 'E'])

print atms                  #print full atms
atms._draw('FILENAME') #draw dependency network
```

B.2 FATMS

```
from FATMS import FATMS

atms = new FATMS()
atms.addAssumption('A')
atms.addAssumption('B')
atms.addAssumption('C')
atms.addAssumption('D')
atms.addAssumption('E')

atms.addJustification('p', ['A', 'B'])
atms.addJustification('p', ['B', 'C', 'D'])
atms.addJustification('q', ['A', 'C'])
atms.addJustification('q', ['D', 'E'])

atms.addJustification('r', ['p', 'q'])

atms.addNogood(['A', 'B', 'E'])

atms.extendFocus(['A', 'B', 'C'])

print atms                  #print full atms
atms._draw('FILENAME') #draw dependency network
```

B.3 Diagnosis

The components can be created by the static method loadComponent. The usage of the method is explained in the *readme* of the diagnosis engine. The arguments for the method are the component name, an assignment of the variables and a structure of the component. In the structure variables exists, which are replaced by the given assignment.

```
from Components import Components, loadComponent
from Diagnosis import BDiagnosis, FDiagnosis
```

```
from CTypes.power import power
from CTypes.wire import wire_with_errormode as wire
from CTypes.lamp import lamp

S = Components ( [
    loadComponent(
        name='B1',
        a={ "a1": "p0", "a2": "m0"},
        ctype= "power"),
    loadComponent(
        name='W1',
        a={ "a1": "p0", "a2": "p1"},
        ctype= "wire"),
    loadComponent(
        name='L1',
        a={ "a1": "p1", "a2": "m1", "a3": "l1" },
        ctype= "lamp"),
    loadComponent(
        name='W2',
        a={ "a1": "m0", "a2": "m1"},
        ctype= "wire")
    ])

d = BDiagnosis(S)
#d = FDiagnosis(S)
d.makeDiag({'l1': 'dark'})
print d.getCandidates()
```

Index

Affidavit

I hereby declare that this master thesis has been written only by the undersigned and without any assistance from third parties.
Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

Place, Date	Konstantin Ruhmann