Dependent Defects and Aspects of Efficiency in Model-Based Diagnosis

Dissertation
zur Erlangung des Doktorgrades
am Fachbereich Informatik der Universität Hamburg

vorgelegt von
Mugur Marius Tătar
aus Cluj-Napoca, Rumänien

Hamburg 1997

Genehmigt vom Fachbereich Informatik der Universität Hamburg
auf Antrag von

Prof. Dr. Bernd Neumann (Uni. Hamburg)

Prof. Dr. Wolfgang Menzel (Uni. Hamburg)

Prof. Dr. Peter Struß (TU München)

Hamburg, den 01.12.1997

Prof. Dr. Horst Oberquelle
Der Sprecher des Fachbereichs

Diese Arbeit wurde mit Unterstützung der Daimler-Benz AG,
Forschung Systemtechnik, Berlin, angefertigt

Abstract

Diagnosis is the task of finding the abnormal parts of a malfunctioning system. Model-based diagnosis uses a model of the system to generate, test and discriminate among hypotheses about the correctness of different parts of the system. This thesis addresses the following two questions:

1. How can we make the process of generating and testing hypotheses more efficient?
2. How to extend the theory of model-based diagnosis such as to diagnose conveniently a certain class of dynamic systems, including the ones with dependent defects?

With respect to the first question we confine our attention to RMS-based diagnostic engines. The RMS (Reason Maintenance Systems) are tools that support efficient hypothetical reasoning, used in many of the nowadays model-based diagnostic engines. In this context we address two sub-problems:

1.a. How can the RMS-tasks be supported more efficiently?
1.b. How to generate efficiently the candidate diagnostic hypotheses?

In answering the first sub-problem, we distinguish between several tasks which are useful in diagnosis and which are usually accomplished by an RMS: *(a)* consistency check, *(b)* entailment check, and *(c)* the computation of minimal supporting sets of assumptions. Starting from the observation that these services require to solve problems with different degrees of complexity, and that the current ATMS-like RMSs use the expensive service *(c)* to provide the first two less complex ones, we propose a new type of RMS, which: *(i)* separates the algorithms used to solve the task *(c)* from the ones used to solve the "easier" ones; *(ii)* supports the expensive generation of the minimal supporting sets of assumptions in an incremental way and at request only; and *(iii)* integrates the algorithms, such as to use the results of the "easier" tasks to control the computation of the minimal supporting sets of assumptions. The 2vRMS integrates features present at the focusing ATMS, at the lazy ATMS, and at the JTMS.

Regarding the problem of efficient candidate generation we discuss algorithms that compute in an incremental way some of the most plausible consistent candidates. The plausibility is defined using *preferences* among the individual behavioral modes of each component, and an additional *priority* relation that goes beyond the expressiveness limitations of the preference.

We further analyze the formal properties of the combined algorithms for reason maintenance and candidate generation with respect to the framework of propositional logic. The combined algorithms are proved to perform (heuristically guided) satisfiability checking and multiple model construction for propositional theories.

The second main topic of this thesis addresses the problem of dependent defects, a problem that was, until now, ignored in the field of model-based diagnosis. We introduce an extension of the current theories for diagnosis, which regards each component and the whole system as discrete-event systems (finite-state machines). The representation allows to model and to reason about: *(a)* the causality of the mode changes, *(b)* dynamic components with memory, and *(c)* the consequences of user's repair and state-changing actions. From a technical point of view, we aimed to reuse the RMS-based architecture for diagnosis discussed in the first part of this thesis, and to avoid, as much as possible, to increase the complexity of reasoning due to the more general framework.

Contents

Chapter 1

Introduction

Understanding the world around us seems to be a never ending task. One of the most powerful tools that humans have developed through history is the "model", an abstract reflection of the part of the world one is interested in. The process of understanding the world has reduced to the process of developing appropriate models and reasoning with these models. Whenever the models fail to explain a certain phenomenon, the models have to be debugged, extended, abandoned and exchanged with fundamentally different ones, or kept but used under more restrictive presuppositions. In trying to prove or disprove the appropriateness of a model to explain some phenomena, we have to rely on experiments whose observable aspects have to be explained by the models.

Model-based diagnosis reflects, in a certain extent, the above approach. Diagnosis can be seen as an attempt at understanding the world, while the model-based approach explicitly deals with the generation and selection of appropriate models.[1] The part of the world one wants to "understand" during a diagnostic session is called *system under diagnosis*. Traditionally the system under diagnosis is decomposed into smaller granularity interacting sub-systems down to the level of components and connections. The approach advocates an iterative process of hypotheses formation, hypotheses testing and hypotheses discrimination, where the hypotheses mainly refer to the correctness of specific parts of the system.

[1]However, the "generation of models" is understood in a limited sense, i.e. there is usually only a predefined range over which the considered models can vary. As a matter of fact, it happens that this is one of the current limitations of the approach.

1.1 What is model-based diagnosis ?

Model-based diagnosis provides a systematic approach to diagnosis. The knowledge about *the process of diagnosis* is separated from the knowledge about *the system under diagnosis*. This allows the same diagnostic engine to be (re)used for several diagnostic applications by simply loading different system descriptions. Furthermore, the knowledge about the system under diagnosis is provided in a structured way which facilitates reusability. This knowledge basically provides the description of the:

structure of the system in terms of interacting sub-systems, down to the level of basic components and connections; and of the

behavior (correct and faulty) of the basic components and connections.

The fine granularity at which the knowledge about the behavior is provided allows to reuse the same component descriptions in different application problems.

The components are regarded as sub-systems that interact with each other and with the surrounding environment via a set of interface parameters, usually grouped in (communication) ports. The connections are between such component ports. Usually, the connections enforce the equality of the parameters of the connected ports. The behavior description of each type of component is structured according to the behavior modes exhibited by that component. For each behavior mode a component specifies a constraint among the port-parameters (and eventually the component's internal parameters).

The diagnostic algorithms start from this basic knowledge in order to reason about the behavior of the global system in different situations and in order to elaborate hypotheses, explain symptoms and suggest experiments. Informally, the reasoning tasks that are performed can be described as follows:

repeat

(1) *candidate elaboration:* elaboration of diagnostic hypotheses that can explain the observations about the system;

(2) *candidate discrimination:* proposal of further information-providing experiments that could help to refute/confirm the current competing hypotheses;

until the final diagnosis is decided.

Usually each candidate hypothesis assumes a unique assignment of modes of behavior (correct or faulty) to each the component. Thus, each candidate can be regarded as a collection of assumptions about the modes of the individual components (and connections).

The *candidate elaboration* task is again an iterative process:

repeat

(1) *candidate generation:* generation of plausible candidates;

(2) *candidate testing:* testing of each candidate using the model of the system and the current set of observations about the system;

(3) *conflict detection:* retraction of the assumptions that proved to be wrong;

until "enough" competing candidates are gathered.[2]

The task of *candidate testing* can be further refined as:

(1) *prediction:* infer the consequences of adopting the candidate assumptions using the model and the observations;

(2) *contradiction detection:* if there are significant discrepancies between the predicted behavior and the observed one than fail the candidate;

Depending on the formalism used for knowledge representation, the prediction task could be realized, for instance, by equation solving, constraint propagation or logical entailment.

During the *conflict detection* those hypotheses that failed to explain[3] the observations are retracted. From reasons of efficiency the discovered contradictions are examined in more detail and one tries to learn as much as possible from the failure cases. Namely, in many cases only a subset of the components are responsible for predicting a certain value, and thus for leading to a certain contradiction. The *conflict detection* step identifies the minimal sets of components responsible for the contradictions. The modes assumed by such a set form a *minimal conflict*. Current strategies for *candidate generation* use the past minimal conflicts in order to avoid the generation of hypotheses that later will turn out to be wrong. In order to help identify the minimal conflicts among the assumptions of a candidate, in many cases, during the *prediction* step also the strictly necessary assumptions used on

[2]Usually only a subset of the most plausible candidates need to be elaborated in order to continue the *candidate discrimination* of the top-level diagnostic process.

[3]By "explanation" usually it is understood either logical entailment - in abductive diagnosis, or logical consistency – in consistency-based diagnosis.

a certain path of inference are recorded, usually using a truth maintenance system.

Without going into details, one could characterize the *candidate discrimination* task of the top-level cycle as a search for situations in which the different competing hypotheses (candidates) would entail distinct observable effects. This task also requires (among others) the ability to perform the above *prediction* and *contradiction detection* tasks. Different from the contradiction detection performed during candidate testing, during *candidate discrimination*, the discrepancies searched are among the predictions of the current competing hypotheses. All the competing candidates are consistent with the current set of observations, but there may exist parts of the model where the competing candidates entail distinct predictions. Observing the real value of the model parameters in those model parts brings more relevant information for the discrimination process.

1.2 Arguments for a model-based approach to diagnosis

Diagnosis continues to be a challenging domain in the area of artificial intelligence (see also [Str92]). Although there exist many knowledge-based programs that address the topic of diagnosis, one could classify the approaches into two large categories:

- the symptom-based (also called heuristic-based) approach, and
- the model-based approach.

The knowledge used by the symptom-based approach encodes, in some form, associations between *typical symptoms* and *possible defects* that could have produced the symptoms. For instance, such kind of knowledge would be:

> "if the bulb #12 is not lit when the switch #4 is ON then the fuse #5 may be broken or the battery is discharged"

Typical examples for this approach are most of the expert systems of the first generation.

Model-based diagnosis starts from a "deeper" knowledge (cf. [Byl90]). The knowledge about structure and behavior can be used to *derive* the

symptom-failure associations used by the first approach, but not vice-versa. The experience with the first-generation expert systems has shown that there are several limitations that prevent a wider application of these systems in practice. It is believed that many of them could be overcome by using deeper knowledge (cf. [HP89], [PL88], [Str91a], [Tat92]). Among the main drawbacks of relying on heuristics supplied by experts one notes the following:

- The main problem is *the acquisition of the heuristic knowledge*. This is a difficult and an expensive process; it needs domain experts, knowledge engineers, and, much worst, much time. Due to the extreme dynamics of the technological changes and due to the flexibility required to permanently adapt to the needs of the customers, the life-cycle of the products tends to become continuously shorter. Most of the relatively complex technical systems need a diagnostic and repair specification (e.g. handbooks) for the support and maintenance services before they can be released to the market. The time required to develop a symptom-based system is simply too long in these conditions. Also, there is no, or not much, time to accumulate experience with these systems.
- The knowledge cannot be reused when a new application must be dealt with, or when small changes in the systems under diagnosis are performed. The heuristics represent compiled knowledge and it is not evident what pieces of knowledge have to be updated when a structural change in the device is done. In most of the cases everything must be changed. This leads to extremely high costs for the maintenance of such systems.
- There are problems with guaranteeing the completeness and the correctness of the knowledge acquired from the experts;
- The explanations provided by such systems cannot go deeper than the encoded heuristics (cf. [HP88]).

Because of the strong mathematical background and due to the systematic and the modular form of the knowledge, the correctness of the knowledge base can be better addressed within the model-based approach. The same features make it easier to see which kind of errors are covered by a certain algorithm and a certain modeling paradigm.

The main argument supporting the model-based approach is that it dramatically reduces the costs of the knowledge acquisition. Most of the knowledge required can be either automatically acquired or reused. Figure 1.1

shows the different kinds and sources of the knowledge relevant for model-based diagnosis:

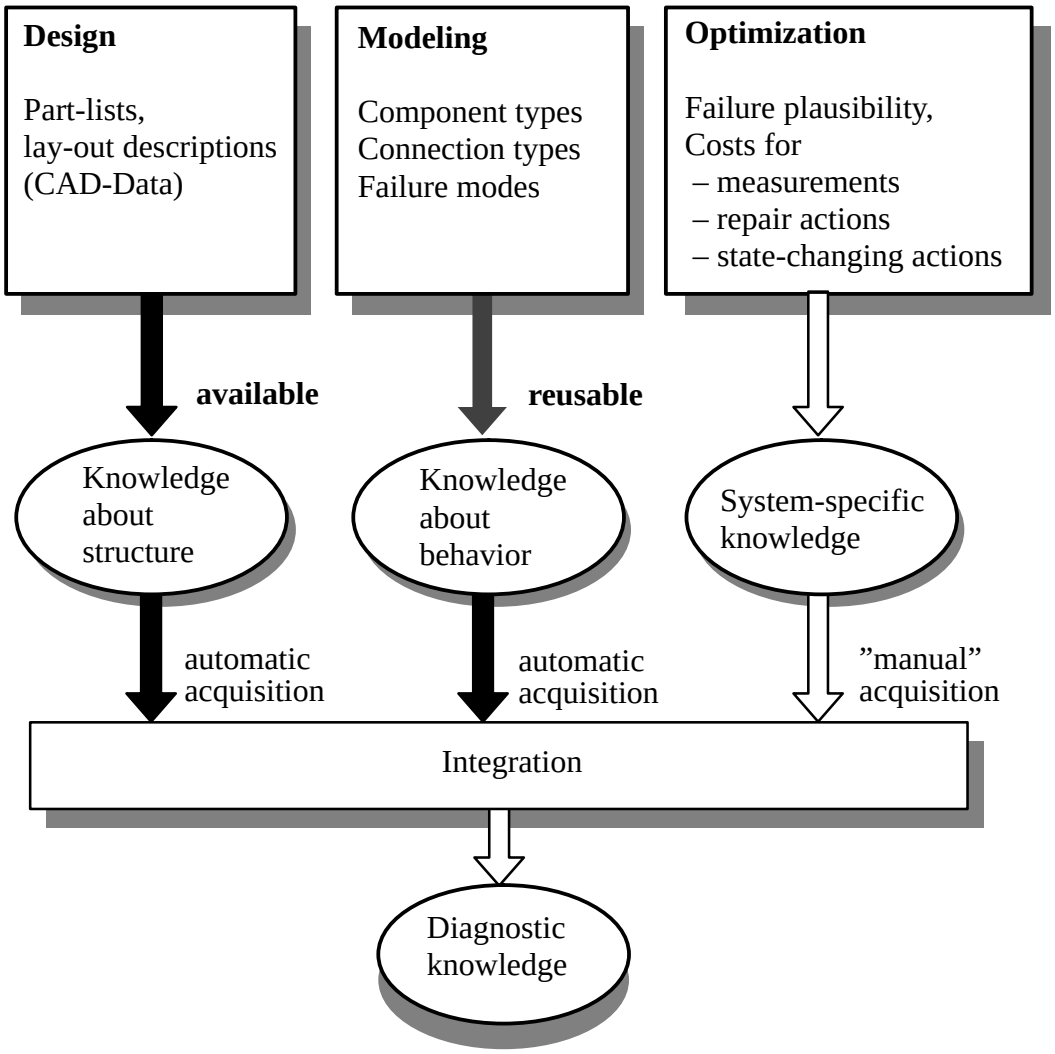

Figure 1.1: Knowledge Sources in Model-Based Diagnosis.

- The knowledge about the structure of a (technical) system can be obtained automatically using the CAD data already available;
- The knowledge about the behavior of the component types and of the connection types must be encoded by hand in most of the cases. How-

ever, this knowledge is *reusable*, i.e. a component/connection *type* must be described only once and then stored in a library for later usage. This requires that the modeling of the components is done in terms of aspects that are independent of the system in which the component is embedded - known as the "no function in structure" principle;

- The third category of knowledge is system-dependent and cannot be in general automatically generated or reused (e.g. knowledge about frequent failures, costs of performing repair, measurement and state-changing actions, etc). The knowledge of this category is, however, necessary mostly for *optimization* purposes. This means that the *correctness* of the diagnoses is not critically affected by the accuracy of this knowledge - only the cost (e.g. time) to reach to the correct solution depends on it. One could start with a raw formulation of this knowledge that could be later improved as more experience with a specific system is gained.

1.3 Applying model-based theory to practice: some problems

Although relatively young, the field of model-based diagnosis has reached a stage where real applications could be handled by this technique. However, there still are a series of problems that the approach encounters when faced with the complexity and the diversity of the current technical products and with the constraints of practice. These problems are multifarious and range from the inherent complexity of the algorithms and the difficulties to model components exhibiting complex behavior, to the insufficient support for diagnosis from the early design stages of the products and the lack of standard knowledge interchange formats.

1.3.1 The complexity of model-based diagnosis

It is known that the problems addressed by model-based diagnosis are NP-hard (cf. [Byl89]). Sources of the intractability are: the exponential space of possible diagnoses, the complexity of the models and the impact of time, the need to find the "best", i.e. the most plausible / most critical diagnoses, to name just a few of them. So it is no wonder that the algorithms used

are inherently exponential in the worst case. However, it is not clear how relevant is the "worst-case" measure to practice and it is perfectly legitimate that different algorithms that are exponential in the worst case, have a better or a worse behavior for some classes of problems relevant for practice.

Improving the efficiency of model-based diagnosis still remains an important research issue. Current techniques that aim to improve the efficiency are: the use of focusing strategies (e.g. [dK91, DS94]), the use of abstraction, simplification and hierarchies in modeling (e.g [Str91b], [Ham91b], [Moz91]), the compilation of the deep knowledge (e.g. [FGN90a], [Moz90]), the seek for better solutions for the impact of time on diagnosis (e.g. [Ham91b], [Lac91], [Dre94], [DJD+91]), the relaxation of some requirements of the problem (e.g. [FGN90b], [Moz92], [BB92]).

Our work described in Chapter 4 specifically addresses the efficiency of model-based diagnosis. We discuss there the efficiency of the reason maintenance component, and of the component responsible for proposing the diagnostic candidates to be checked next.

Efficiency was also an important concern in the diagnostic framework that we developed in Chapter 5, where we address the diagnosis of a certain class of dynamic systems. We avoid, in a certain extent, the usual inefficiency of performing temporal reasoning on top of a reason maintenance system by reusing the predictions across time as well as across belief contexts.

1.3.2 Restrictive presuppositions about the nature of faults

Model-based diagnosis dispensed with some of the presuppositions that other approaches to diagnosis usually depend on, e.g. the assumption that there are no multiple-faults, or that the modeling is complete. However, the current theory, still makes some presuppositions that fail to be satisfied in all application domains.

Commonly made assumptions are that the observations are correct, that probing has equal cost and that there are no unforeseen interactions among the components of the system - like, for instance, the bridge-faults in electronics (see also [Cla95, Dav84]).

There is still no satisfying way to deal with intermittent faults, although some progress in this respect has been made by [FL91, Lac91].

A commonly made assumption concerns the independence of the faults in

case of multiple failures. This assumption fails to reflect correctly the cases of cascading defects, e.g. when a fault occurring somewhere in a system causes others to occur in certain situations. In some domains such caused defects are quite frequent, much more frequent than multiple independent defects. This fact motivated our work described in Chapter 5.

1.3.3 Practically relevant aspects of model-based diagnosis that are not covered by the theory

In this category falls the need to propose test patterns during diagnosis. Testing was a problem that received much attention, but was mainly regarded in isolation of diagnosis. The tasks should be, however, interleaved. Repair is another task that was mainly regarded as an independent one. A current topic of research is the integration of diagnosis, testing and repair (cf. [Str94], [SW93], [McI94b], [FGN92]). The support offered by the diagnostic systems to the technicians should also consider the costs of the proposed diagnostic actions (e.g. measurements, state-changing actions and repair actions).

Although we do not address in this thesis specifically the problem of proposing testing and repair plans during diagnosis we had in mind the integration of these tasks within the framework proposed in Chapter 5. Repair actions can easily be modeled in our approach as transitions from defect modes to correct ones, controlled by "repair" inputs. Also, since in our approach the input assignments and the observations are implicitly represented as *assumptions*, it is easy to make hypothetical reasoning about possible future actions and their outcomes, a reasoning ability that, in our opinion, is required in order to elaborate testing and repair plans.

1.3.4 Modeling is "the hard part"

Modeling is the part of the diagnostic approach that could not, so far, be automated. Although, due to the reusability of the knowledge, modeling is much less expensive than in the symptom-based approaches, it still requires considerable efforts. Moreover, modeling still remains a kind of an art.

During modeling one has to consider the balance between complexity and completeness. The more accurate the models are, the more reliable the results of diagnosis, but the harder becomes the reasoning with the models. For instance, many model-based diagnostic engines use a reason maintenance

system (RMS) to cache the results of prediction for later reuse. The effectiveness of this optimization is considerably affected by the granularity at which modeling and reasoning is done. The degree of reusability tends to be much smaller when the models are numerical or very accurate. The usage of different modeling perspectives and of several abstraction levels is one direction of research (cf. [Str91b]).

While a qualitative or more abstract modeling promises to reduce the complexity of diagnosis, it sometimes increases the costs of the knowledge acquisition. There is no difficulty in computing automatically abstractions of different theories, the problem is that it is very difficult to find automatically which elements in the space of abstractions provide a good balance between completeness and complexity. Moreover, once a good abstraction for the behavior of a component (group of components) is found for an application, there is no guarantee that the same characterization would still prove useful in a different application. Thus, there appears to exist another balance: between the complexity of reasoning and the knowledge reusability. The less details one brings into the models, the less expensive is the reasoning, but the probability that the models can be reused decreases.

1.4 Aims of the thesis

The thesis intends to:

- Improve the efficiency of model-based diagnosis in RMS-based engines. We deal mainly with the efficiency of the RMS tasks and with the problem of efficient candidate generation;
- Extend the current model-based approaches in order to make possible to model and to diagnose systems with dynamic behavior and dependent defects.

1.5 Overview of the thesis

Chapters 2 and 3 provide background knowledge for the problematic discussed in this thesis. Chapter 2 gives an introduction to the reason maintenance systems. Chapter 3 gives a survey of the main work that formalizes the tasks of model-based diagnosis. Except the presentation of the JTMSset

from Chapter 2, a minor contribution of this thesis, Chapters 2 and 3 do not describe original research work.

Chapters 4 and 5 report the main original work in thesis. Chapter 4 considers two aspects of efficiency in an RMS-based diagnostic engine, namely, we analyze the efficiency of the tasks supported by the RMS and the efficiency of the component that proposes the "best" diagnostic candidates. The two sub-problems are related; both the RMS and the candidate generator module provide services at the propositional logic level. We argue that their algorithms play a complementary role. Most of the RMSs are logically complete only for propositional Horn theories. The integration with the candidate generator, basically a search tool, can overcome this incompleteness.

Chapter 5 addresses the problem of non-independent defects in diagnosis. We suggest here an approach in which the components and the whole system are regarded as finite-state machines. The approach allows to capture, at least partially, the causality of mode changes. As a side effect, it allows to easily integrate repair actions into the models and to represent components with dynamic behavior.

The rest of this section provides an abstract of the main results reported in Chapters 4 and 5.

The 2vRMS

The truth maintenance systems (TMS) or reason maintenance systems (RMS) are instruments used to record dependencies among data. There are several kinds of RMS systems, e.g. the ATMS (cf. [dK86a]), the JTMS (cf. [FdK93]), the LTMS (cf. [McA80]), which differ in the expressiveness of the allowed dependencies and in the amount of services provided.

Many of the nowadays model-based diagnostic engines use an ATMS-like RMS to record the dependencies among the inferred data. Among other reasons, the preference for the ATMS is due to the power of computing the *minimal* conflicts among the diagnostic assumptions, in case a diagnostic candidate fails to explain the symptoms.

Despite recent improvements in controlling the ATMS, like the focusing ATMS (cf. [DF90, FdK88]) and the lazy ATMS (cf. [KvdG93]), for complex problems the RMS remains a major resource consumer in diagnosis. This motivated our interest in further improving the efficiency of the reason maintenance tasks.

We start from a classification of the tasks usually accomplished by an RMS:

1. Entailment check: decide what else can be believed if a set of assumptions is believed;
2. Consistency check: decide if the set of beliefs is consistent;
3. Computation of the minimal-support-set: what are the minimal sets of assumptions that have to be believed in order to believe a certain data.

The ATMS family can accomplish all of the above tasks, while other RMSs, like the JTMS and the LTMS , can accomplish only the first two.[4] However, the JTMS and the LTMS use much more efficient algorithms to solve the first two tasks than the ATMS family does; in Horn theories the entailment check and the consistency check can be provided in linear time, but the ATMS use the expensive minimal-support-set labeling, exponential in the worst case, to solve the first two tasks as well.

The improved RMS that we propose in this thesis (2vRMS, i.e two-view RMS) is able to accomplish all of the above tasks. The first two ones are solved with an efficiency comparable to that of the JTMS / LTMS. The third one is supported by request only. Moreover, the computation of the minimal supporting environments is tightly controlled. In this respect, the 2vRMS integrates two views on data.

1. The focusing view: This view corresponds to a JTMSset (cf. Chapter 2), i.e. a multiple-context RMS that integrates a set of monotonic JTMSs.
2. The detailed view: This view corresponds to a kind of focusing ATMS with a lazy label computation.

A prototype 2vRMS was implemented and tested in conjunction with an elaborate diagnostic engine developed at the Daimler-Benz Research Center in Berlin. Significant reductions of the space and time required for diagnosis were obtained due to the usage of the 2vRMS (see Section 4.2.4).

Candidate generation

Usually in diagnosis there are very many possible candidates that are consistent with the observations, especially in the early diagnostic steps when

[4] The LTMS and the JTMS support, however, a relaxed version of the third task, namely, without guaranteeing the minimality and completeness of the supporting environments.

there is not enough evidence about the system. It is not feasible to consider all of the possible candidates in parallel. Current techniques for candidate generation focus only on a few of the most *plausible / critical* candidates from the ones that are possible.

We use a diagnostic framework where each component can be characterized with several modes of behavior. Each candidate chooses exactly one mode of behavior for each component. The plausibility / criticality information that we use to control the selection of the focus candidates is encoded by two relations: *preference* and *priority*.

The preference relation is as defined by Dressler and Struss (cf. [DS92]). A preference order among the modes of each component is used to induce a preference order among candidates. The partial order defined by the candidate preference imposes a lattice structure on the candidate space. We show how to take advantage of this structure in order to optimize the search.

The preference alone can only encode knowledge about the relative plausibility of the modes of a single component, e.g. it can encode knowledge saying that: *"a wire is most probably correct, but if it is defect then it is more likely to be broken than shorted to ground"*, but it can not encode knowledge saying that, e.g. *"the bulbs break more often than the switches, which, in turn, break more often than the wires"*. The focus selection heuristic based on the preference alone is in many cases not sharp enough: for large systems the set of preferred candidates is very large. In order to gain more control over the selection of the focusing candidates we define an additional *priority* relation that removes the expressiveness limitations of the preference.

The proposed algorithms can generate in an incremental way a few (say k) of the most preferred candidates having the highest priority.

We further analyze the properties of the candidate generation algorithms with respect to the framework of propositional logic. The 2vRMS (as well as the ATMS family and the JTMS) can perform satisfiability checking and model construction for Horn clauses. The candidate generator can perform satisfiability checking and model construction for purely positive and purely negative clauses. The combination 2vRMS (ATMS or JTMS) and the candidate generator can be used as a satisfiability checker and (preferred) model builder for *arbitrary* propositional clause sets, thus removing the expressiveness limitations of the RMS and of the candidate generator alone. In comparison with a "conventional" propositional theorem prover, the RMS - candidate generator architecture offers the incremental way of operation and, more importantly, the extra-logical control means encoded in the priority and

preference relations.

A further optimization which we suggest limits the degree of parallelism in investigating the search space. We introduce, so called, *secondary* choice sets, whose elements are assumed to be equally preferred. The search in the secondary choice sub-spaces can use techniques similar with those used in constraint satisfaction problems, where usually the construction of a single solution is attempted.

Diagnosis of dynamic systems and systems with cascading defects

The current approaches to model-based diagnosis are mostly adequate for the diagnosis of static systems with independent defects, either due to expressiveness, or complexity limitations. In some domains, however, dependent defects appear relatively frequent, anyway more frequent that multiple independent defects. We argue that systems with cascading defects cannot be properly diagnosed by the current approaches that start from the fault-independence assumption and neglect the dynamic of the mode changes.

We suggest an approach in which the components are modeled as tiny automata. The knowledge about the whole system is generated, as usually in model-based diagnosis, by composing the component behaviors using the structure information. The mode of behavior of each component is regarded as a state variable, i.e. is part of each component's "memory". The representation allows to model in a uniform way and to reason about: *(a)* the causality of the mode-changes; *(b)* dynamic components with memory; *(c)* the consequences of user's control and repair actions. The diagnoses generalize in our approach to transition paths consistent with the system description and the observations. The preference among the non-deterministic choices and the priority relations used by the candidate generator from Chapter 4 can be applied to define preference relations among transition paths.[5]

We were careful to avoid, in some extent, the well-known deficiency of reasoning across time using an RMS. The logical description of the system includes only temporal independent aspects. We show how to use such system descriptions in order to reuse the predictions across belief contexts, as in any RMS-based prediction system, as well as *across time*.

[5] If the modeled system has a deterministic behavior, and if one assumes complete information about the inputs, such non-deterministic choices appear only with respect to the initial state of the transition paths. Otherwise, such choices appear with respect to the state transitions as well. Only the deterministic case is discussed in this thesis.

Chapter 2

Reason Maintenance Systems

2.1 Introduction

As we have seen in Chapter 1, model-based diagnosis has at its basis the generation, testing and discrimination of hypotheses. Reason Maintenance (or truth maintenance) deals with the problem of correctly and efficiently performing hypothetical reasoning. Different kinds of reason maintenance systems (RMSs) underlay many of the model-based diagnostic engines in use today.

The goal of this chapter is to provide the background knowledge and the context for our discussion of the improved RMS that we are going to present in Chapter 4.

Readers familiar with the terminology and problematic of reason maintenance may skip this chapter.

2.1.1 The contents of this chapter

Section 2.2 introduces the basic terminology, the architecture, and the notions that are valid for the entire spectrum of RMSs that are discussed in this chapter.

Section 2.3 presents a survey of several RMSs: the JTMS, the LTMS, the JTMSset (a minor contribution of this thesis), the basic, focusing and the lazy ATMS, and the CMS. Further details about the algorithms, as well as further insights into the non-monotonic RMSs, are given in Appendix A.

Section 2.4 provides a comparative analysis of the services provided by the different kinds of monotonic RMSs from a logical point of view.

2.1.2 Hypothetical reasoning and reason maintenance

Hypothetical reasoning involves reasoning with unsure or incomplete data or knowledge. Without making assumptions about the missing or the uncertain data, in many cases one can make no progress with solving a certain problem. A sensible strategy for such cases is to try to estimate the plausibility of the possible variants and to continue reasoning by assuming that the current case is one of those most plausible ones. For instance, we cannot know if it will rain or not in the next 8 hours, but we can estimate the plausibility somehow and decide if we take an umbrella with us or not. If the assumptions were wrong this can manifest sooner or later in discovering an inconsistency in the knowledge.

Hypothetical reasoning inherently leads to reasoning with inconsistent knowledge. When an inconsistency is detected two problems must be solved, namely, (a) the assimilation of the inconsistency without inferring its consequences,[1] and (b) the restoration of consistency. In order to restore consistency one notes that, when inconsistency is detected, at least one of the assumptions used to infer it must be false. Thus, a basic task in order to achieve this is to analyze the patterns of reasoning and to detect which assumptions contributed to the inconsistency.

Reason Maintenance Systems are tools that provide the basic functionality that allows the assimilation of the inconsistencies and the restoration of the consistency. They maintain the sets of currently enabled (believed) assumptions and the dependencies between the inferred data. They can answer what inferred data can be believed if a set of assumptions are believed. They also detect when inconsistent inferences are made. In such cases they can report what set of assumptions is responsible for the inconsistency.

An RMS supports efficient hypothetical reasoning in that it prevents to restart reasoning from the beginning when the set of beliefs changes. Due to the dependencies between the inferred data that they maintain, they can restore the old inferences that can be safely believed when the set of believed assumptions change. They act like a database for the inferences made by a problem-solver.

[1] This is not obvious. It is known that in the first-order predicate logic, if an inconsistency is derived then anything else can be derived as well.

2.2 Basic concepts

There are several kinds of systems known as truth maintenance systems (TMS) or reason maintenance systems (RMS): the non-monotonic justification based truth maintenance system (nmJTMS, cf. [Doy79]), the monotonic variant of it (JTMS, cf. [FdK93]), the logic-based truth maintenance system (LTMS, cf. [McA80, McA82, McA90]), the assumption-based truth maintenance system (ATMS, cf. [dK86a]), and many descendants and variants of these. However, they all share some basic concepts and functionality. This section introduces the aspects that are common to all of these RMS systems. In [FdK93] one can find a good and systematic introduction to the different kinds of monotonic RMSs and to the way of efficiently using them in problem-solving. Another good introduction, covering also the nmJTMS, is in [Rei89]. In [Mar90] one can find an indexed literature of the reason maintenance systems. In [McD91] a unifying formalism for the JTMS, LTMS and the ATMS is proposed.

The Problem-Solver - RMS architecture

A reasoning system using an RMS separates in its architecture the Problem-Solving (PS) component and the RMS component. The problem-solver and the RMS interact via a well-specified interface:

- the PS communicates to the RMS:
  - currently enabled assumptions (beliefs);
  - inferences: what pieces of knowledge and what inference method was used to derive each new piece of knowledge;
  - questions about the belief in some pieces of knowledge;
- the RMS communicates to the PS:
  - answers to PS' queries;
  - inconsistencies: what subset of the currently enabled assumptions is responsible for the derivation of an inconsistency.

The decision what assumption(s) to retract when an inconsistency is discovered and eventually what new assumptions to enable, is sometimes left to the problem-solver, sometimes to the RMS. In most of the RMSs the PS can attach so called *consumers*, to the inferred data. The consumers are

procedures that the RMS should activate when the belief in the associated data changes in some way. This way, the PS can accord its inference with the belief status of data.

An RMS operates incrementally. The queries about the beliefs are interleaved with the addition of inferences and assumptions. An RMS usually takes advantage on the fact that before the addition of a new inference or before changing the set of enabled assumptions the knowledge was labeled with the correct beliefs and updates the beliefs in an incremental way.

In most of the RMSs the PS can only add inferences and assumptions and not delete them.

Although the knowledge may have a richer structure and semantics for the PS, the RMS systems that we discuss here regard the knowledge supplied by the PS as purely propositional.

Basic terminology

Some of the concepts have different names in different papers. The terminology we adopt here is closest to the ATMS (cf. [dK86a, dK86b, dK86c]).

assumption: piece of data designated by the PS whose influence on the inferred data must be traced by the RMS. Usually, the assumptions represent uncertain data that the PS may choose to believe or disbelieve at certain times.[2]

RMS **node:** data structure associated by the RMS with a proposition (datum) inferred by the PS. A node stores, besides the associated datum, dependency information. The dependency information contains references to the other nodes whose belief status influences (is influenced by) the belief in the current node (see bellow *belief constraint*). The dependencies connect the RMS nodes into a network. Usually, there are four categories of nodes: assumption nodes, premise nodes, derived nodes and contradiction nodes, described bellow.

node label: a slot of the RMS-node data structure that caches the current belief in the node. The main RMS task is to maintain the node labels. The

[2] However, the meaning of the assumptions to the PS is not known in the RMS. The RMS will just trace the dependence of the inferred data on the designated assumptions. It could be that the PS encodes something as an assumption just for indexation reasons, and not because it represents uncertain data.

way the nodes are labeled and the contents of the label depends on the particular kind of RMS.

assumption node: node associated with an assumption. When inconsistencies are detected the dependencies are traced back to the assumptions responsible for the derivation of the inconsistency.

premise node: node associated with a sure proposition. The belief in the premises does not depend on any assumptions, i.e. the premises are always believed.[3]

derived node: node associated with an inferred proposition. The belief in the associated proposition depends on the belief in other data, and ultimately depends on the belief of some assumptions.

contradiction node: node whose associated proposition represents the falsity. A contradiction node must never be believed. An inconsistency is detected whenever such a node receives a label indicating the belief.

When an inconsistency is detected this is signaled to the PS and some of the currently enabled assumptions must be retracted in order to restore consistency. The RMS can detect the minimal sets of currently enabled assumptions that support the contradiction.[4]

belief constraint: data structure representing a constraint between the beliefs of some nodes. The form of a belief constraint depends on the particular type of RMS, e.g. Horn clauses (also known as *justifications*) in the ATMS, and in the JTMS; non-monotonic justifications in the nm-JTMS and nm-ATMS; propositional clauses in the LTMS and in the CMS (see Section 2.3).

nogood: set of assumptions whose belief leads to the derivation of a contradiction. Some RMSs automatically store the discovered nogoods (e.g. the ATMS), others do not record the nogoods automatically (i.e. the PS has to do it).

environment: set of assumptions.

context: the set of propositions that are believed if a certain set of assumptions is believed (enabled). Each environment defines a context. The

[3]This does not mean that the premises have to be positive literals.

[4]The minimality is not always guaranteed. Also, usually, only one such set is searched (the ATMS is an exception).

contexts that contain a contradiction are inconsistent. Sometimes the contexts are also called "hypothetical worlds".

single / multiple context RMS: if a certain RMS maintains the beliefs for one context at a time it is called single-context (examples are the LTMS and the (nm-)JTMS). If it can maintain the beliefs (labels) in several contexts in parallel it is called multiple-context (e.g. the ATMS).

focusing environment / context: A context in which the PS is interested at a certain moment and in which the RMS labels the beliefs is called a focusing context. The set of assumptions defining a focusing context is a *focusing environment*. We say that a node *holds in focus* if there exists a focusing context where the node is believed.

contradiction handling: strategy of restoring the consistency. It basically must specify what assumptions to retract and enable after the discovery of each inconsistency. Some RMSs have an incorporated contradiction handling mechanism (e.g. the non-monotonic RMSs that use the defaults to specify what to believe when; the basic ATMS does the parallel search in all the possible consistent contexts at once and does not need external contradiction-handling). When the contradiction handling is not incorporated in the RMS (e.g. the JTMS) the RMS must offer hooks to the PS in order to connect it.

monotonic / non-monotonic RMS: in a monotonic RMS the belief in a proposition is based on the presence of belief in other propositions (e.g. "if a is believed and b is believed than believe c"). In a non-monotonic RMS the belief in a certain proposition can be also based on the *lack* of belief in other propositions (e.g. "if a is believed and $\neg b$ is not believed then believe b"). In monotonic RMSs the size of a context always increases when enabling more assumptions, while in a non-monotonic RMS the set of believed propositions may increase or decrease when more assumptions are enabled.

The ATMS (cf. [dK86a]), the JTMS (cf. [FdK93]) and the LTMS (cf. [McA80, McA82, FdK93]) are monotonic. The nmJTMS (cf. [Doy79]) and the nmATMS (cf. [Dre88, Dre90]) are non-monotonic.

consumer: procedure attached by the PS to an RMS node. This is a way of connecting the PS inference with the RMS operations. The RMS is used to *trigger* the PS inferences. Namely, the RMS must activate a consumer when a certain condition is satisfied by the node to which it was attached,

Problem-Solver	RMS
▷ create premise p, q	
▷ create assumptions A, B, C, D	
▷ create derived m, n	
▷ create contradiction $\perp$	
▷ add clause $p \wedge q \rightarrow m$	
▷ add clause $A \wedge B \wedge m \rightarrow n$	
▷ attach consumer $\mathcal{P}$ to node n	
▷ query belief of n	▷ False
▷ enable assumptions $\{A, B, C, D\}$	▷ activate consumer $\mathcal{P}$
▷ query belief of n	▷ True
▷ create derived g, h	
▷ add clause $g \wedge h \rightarrow \perp$	
▷ add clause $C \wedge n \rightarrow h$	
▷ query belief of h	▷ True
▷ add clause $B \wedge C \rightarrow g$	▷ Notice Nogood: $\{A, B, C\}$
▷ disable assumptions $\{A\}$	
▷ query belief of h	▷ False
...	...

Figure 2.1: A typical RMS - PS interaction

typically when the belief in the node changes. After a consumer is activated it is also removed from the list of pending consumers of the node. The activated consumers are usually placed into an agenda. The PS can later pick up the active consumers and execute them according to some scheduling strategy. At execution time the consumer receives as argument the RMS node to which it was attached. Typically, the consumers implement PS inference rules. During execution more nodes may be created, more belief constraints may be added, and new consumers can be attached.

A generic RMS-PS interaction based on the consumer mechanism is presented in more detail in Appendix A.1. When used properly, the consumer architecture guarantees that no inference is performed twice when switching between contexts and that no inference in which the problem-solver expresses interest is left out.

	Single Context	Multiple Context	
		all contexts	some contexts
Monotonic	JTMS LTMS	ATMS CMS LazyATMS	focusing ATMS JTMSset
Non-monotonic	nmJTMS	-	nmATMS

Table 2.1: Families of RMSs

2.3 Families of RMSs

The LTMS and the CMS are the only RMSs from those listed in Table 2.1 that accept as belief constraints arbitrary clauses and that can directly represent negation. The rest of the monotonic RMSs from above work with Horn clauses (called justifications) that can be represented by material implications, like $p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow q$. The justifications accepted by the nmJTMS and the nmATMS are like the ones of the JTMS / ATMS, except that one can require that some of the antecedents should be *out*, i.e. should not be believed.

The rest of this section presents the main features of the JTMS, JTMSset, LTMS, basic, focusing and lazy ATMS, and the CMS. Further details about the algorithms and about the non-monotonic RMSs, i.e. the nmJTMS and the nmATMS, are given in Appendix A.

2.3.1 The JTMS

The monotonic JTMS (cf. [FdK93]) is the simplest kind of RMS, but also a commonly used one. The JTMS works in one context at a time. The context is defined by the *set of currently enabled assumptions*. The PS has to explicitly enable / retract the assumptions.

The belief constraints accepted are *justifications*: $p_1 \wedge p_2 \wedge \ldots p_n \rightarrow q$, where $p_1, p_2, \ldots p_n$ are called the *antecedents* of the justification and q is called the *consequent* of the justification. In logical terms, the justifications are propositional Horn clauses.

The JTMS node label consists of one symbol indicating whether the node holds in the context defined by the currently enabled assumptions. In logical terms, this means that $\mathcal{A} \cup \mathcal{J} \models n$, i.e. the node follows from the union of the propositions representing the enabled assumptions ($\mathcal{A}$) and the set of clauses representing the justifications ($\mathcal{J}$).

Answering a query about the belief in some node means to simply return the label of that node; for instance `IN` can indicate the belief and `OUT` the *lack* of belief. The belief constraints defined by the justifications in terms of allowed node labels say simply that:

1. if all of the antecedents of a justification are `IN` then the consequent must be `IN`;
2. if all of the justifications in which a node n is the consequent have all the antecedents `OUT` then n is `OUT`.

In order to detect the inconsistency a contradiction node is used. Whenever the label of a contradiction node is `IN` this indicates that $\mathcal{A} \cup \mathcal{J}$ is not satisfiable. The consistency check is realized as a particular case of the entailment check.

In order to detect a subset of $\mathcal{A}$ responsible for an inconsistency or for the derivation of some node, the *well-founded support* of a node can be used.

Well-Founded Support

The notion of well-founded support is important in an JTMS for *(a)* providing explanations, *(b)* examining the causes of the contradictions, and *(c)* the incremental label update after retracting an assumption. A well-founded support for a node n is a *non-cyclic and non-redundant proof* for n, i.e. it is a sequence of justifications $J_1, \ldots, J_k$ such that:

1. J_k justifies node n;
2. all the antecedents of J_i are either enabled assumptions or are justified earlier in the sequence;[5]
3. no node has more than one justification in the sequence.

Note that there can be several well-founded supports for a node. The JTMS maintains only one for each node. Namely, the node data structure includes a slot (say $wfSupport$) that stores the justification currently providing the well-founded support for that node (if any). In case of inconsistency the well-founded support of the contradiction node can be traced back to detect the enabled assumptions involved. This simple mechanism is not enough to guarantee the minimality of the nogoods, however.

[5] The premise nodes can be seen as derived nodes justified with an empty set of antecedents, e.g. $\rightarrow p$

The labeling after the assumption enabling / disabling, the node creation and the justification addition, is performed incrementally, i.e. it is assumed that before these operations the labels were correct, and the label update inspects as few nodes as necessary. Except the assumption disabling, the (re)labeling algorithms performed after such operations are fairly trivial. In order to update correctly the labels after assumption retraction and to avoid to introduce unfounded belief, one has to proceed in two steps:

1. Label `OUT` all of the nodes whose well-founded support depended on the retracted assumption;
2. Inspect all of the nodes labeled `OUT` in the previous step to check if some of them cannot be labeled `IN`.

A more detailed description of the JTMS data structures and of the internal operation can be found in Appendix A.2. The logical properties of the JTMS are discussed at the end of this chapter.

2.3.2 The LTMS

The LTMS (Logic-based TMS, cf. [McA80, McA82, McA90, FdK93]) is a single-context, monotonic RMS. The belief constraints accepted by the LTMS are propositional *clauses*, but not necessarily Horn-clauses. The LTMS is more expressive than the JTMS because it can directly represent *negation*. Each propositional symbol is associated to an LTMS node. The node labels can take three values, namely:

- `true` meaning that the associated proposition is believed;
- `false` meaning that the negation of the proposition is believed;
- `unknown` indicating the *lack* of belief in either the proposition or its negation.

As usual, the nodes contain references to the constraints in which they are involved, i.e. to the clauses in which they are mentioned. There is no need for an explicit contradiction node to express inconsistency in the LTMS, inconsistency is simply detected by clause violation. The procedure that labels the nodes relies on *boolean constraint propagation* (BCP), an efficient inference method (i.e. P-complete, in the worst-case linear in the number of literals in clauses), but, obviously,[6] an incomplete one, equivalent to the *unit* resolution.

[6] Satisfiability in propositional logic is a well-known NP-complete problem.

Boolean Constraint Propagation

A clause is a disjunction of literals, where each literal is a proposition, or a negation of an atomic proposition. The literals that contain negation are called negative, otherwise they are positive. Given a labeling with `true`, `false` and `unknown` of the propositions, a clause can be:

1. Satisfied. At least one literal from the disjunction evaluates to true. For instance the clause $\neg x \vee y \vee \neg z$ is satisfied in the labeling: $\{(x, \texttt{true}), (y, \texttt{unknown}), (z, \texttt{false})\}$.
2. Violated. All the literals evaluate to false. The above clause is violated in the labeling $\{(x, \texttt{true}), (y, \texttt{false}), (z, \texttt{true})\}$.
3. Unit open. All the literals except exactly one evaluate to false. The remaining literal refers to a proposition that is labeled `unknown`. The assignment $\{(x, \texttt{true}), (y, \texttt{unknown}), (z, \texttt{true})\}$ makes the above clause unit open in y.
4. Non-unit open. The clause is in none of the above categories, i.e. no literal evaluates to true and more than one are unknown.

BCP starts with a certain labeling and tries to satisfy all the clauses by changing the labels from `unknown` to `true` or `false`. It maintains a stack *checkStack* of clauses that have to be examined (initially all clauses have to be examined) and a stack *violatedStack* of clauses that are violated. BCP sequentially pops a clause from the stack *checkStack* and pushes it to *violatedStack* if it is violated or, if the clause is unit open, satisfies that clause by changing the label of one proposition.

Analog to the well-founded support defined for the JTMS, in the LTMS exists the notion of well-founded explanation. When the BCP algorithm deduces the belief in some node the LTMS notes the clause that enforced that label. Like the JTMS, the LTMS maintains only one well-founded explanation per node. This can be used to find one (not necessarily minimal) subset of the enabled assumptions that support the belief of some node.

A more detailed description of the data structures and of the algorithms can be found in Appendix A.3. The logical properties of the LTMS are discussed at the end of this chapter.

2.3.3 The JTMS set

The JTMSset is a simple multiple-context monotonic RMS which integrates a set of JTMSs. The JTMSset is, however, more efficient than a set of independent JTMSs and represents a minor contribution of this thesis.

The JTMSset is able to work in several contexts in parallel. The set of contexts in which the JTMSset works at a certain moment are called the *focusing contexts*. The focusing contexts must be characterized using the sets of enabled assumptions per context, i.e. using the *focusing environments*.

The belief constraints are justifications like in the JTMS. The label of a node is no longer a simple symbol `IN` or `OUT` as in the JTMS, rather it is a *set* whose elements are the identifiers of the focusing contexts in which the node is `IN`. For instance, if at a certain moment the focusing environments are $\{1 : \{A, B, C\}, 2 : \{B, D, E\}, 3 : \{E, F, G\}\}$ and a node n holds in the contexts defined by the focusing environments 1 and 3, then the label of n would be $\{1, 3\}$.

The JTMS belief constraints require that whenever all the antecedents of a justification are `IN` the consequent of the justification must be also `IN`. In terms of the JTMSset representation this means that:

$$n.label = \bigcup_{J \in n.justifications} J.label;$$

$$J.label = \bigcap_{n \in J.antecedents} n.label;$$

where n is a node and J is a justification. Apart from these, the premises are labeled with the total set, the derived nodes with no justifications are labeled with the empty set while the assumptions are labeled with the set:[7]

$$label(a) = \{i \mid \ a \text{ is member of the } i\text{'th focusing environment}\}.$$

Figure 2.2 contains a small dependency network with the JTMSset labels.

The notion of well-founded support has the same meaning and importance as in the JTMS. In the JTMSset a node has different well-founded supports in different focusing contexts. The JTMSset maintains one well-founded support for each focusing context for each believed node. The JTMSset is more efficient than a set of independent JTMSs because:

[7] If the assumptions can be justified then also the propagated set must be added to the above set.

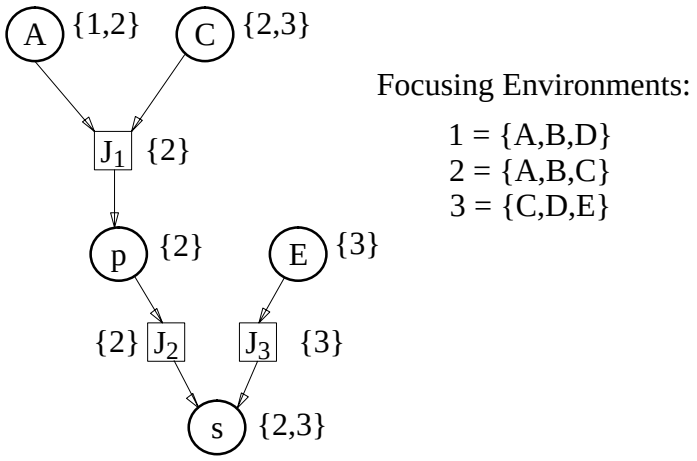

Figure 2.2: Dependency net in the JTMSset. A, B, C, D, E are assumptions; p, s are derived nodes; J_1, J_2, J_3 are justifications, e.g. $J_1 = (A, C \rightarrow p)$.

1. the JTMSset stores the nodes and the justifications only once;
2. the labeling algorithms touch the nodes that hold in the intersection of several focusing contexts only once in the JTMSset, while these node would have been several times touched if the JTMSs were independent. Thus, for instance, if a certain operation affects the set of nodes N_1, N_2 where the nodes form N_i hold in the focusing context i, then a set of independent JTMSs would touch a number of $|N_1| + |N_2|$ nodes, while the JTMSset touches only $|N_1| + |N_2| - |N_1 \cap N_2|$ nodes.

During reasoning the PS may add and remove environments to the focus of the JTMSset. The label update after focus changes is performed in an incremental way. When the PS removes some environment from the focus the JTMSset just marks this environment as "obsolete" (i.e. it adds its identifier to the set $tms.obsoleteFocus$), but does not immediately reprocess the labels. Of course, as soon as an environment is added to the obsolete focus, the JTMSset will stop to activate the consumers in that context. The reprocessing of the labels is delayed until the PS adds a new environment to the focus, but at this time the reprocessing can be performed incrementally, as follows:

> Suppose E_{old} is an old obsolete focusing environment having the index i in the JTMSset, and that E_{new} is a new focusing environment that will replace E_{old}. Then:

1. The assumptions from $E_{old} - E_{new}$ must be disabled in the context i;
2. The assumptions from $E_{new} - E_{old}$ must be enabled in the context i, but
3. No update of the labels has to be done at the nodes that hold in the context defined by the environment $E_{old} \cap E_{new}$;

If the focusing environments do not differ much in between, which is usually the case, at least in diagnosis, then most of the old labels do not have to be inspected and only a few incremental changes have to be performed.

Internally, the JTMSset maintains three data structures that define the actual focus:

1. $tms.focusEnvironments$: an ordered collection of environments. The index of a focusing environment in this ordered collection is used as the identifier of that focusing environment. The sets of such identifiers can be efficiently represented as bit sets.
2. $tms.obsoleteFocus$: the (bit) set specifying the set of elements from $tms.focusEnvironments$ that the PS removed from the focus;
3. $tms.activeFocus$: the (bit) set specifying the elements from $tms.focusEnvironments$ that are not in $tms.obsoleteFocus$ and have not been discovered inconsistent.

The focusing environments from $tms.focusEnvironments$ that are neither obsolete, nor active, are inconsistent. The consumers are activated only for the contexts defined by the active focus.

A more detailed description of the data structures and of the algorithms is given in Appendix A.4.

2.3.4 The ATMS

The ATMS (cf. [dK86a]) is a monotonic, multiple-context RMS which, like the JTMS, accepts as belief constraints *justifications*, i.e. Horn clauses. The ATMS was the underlying RMS in the original GDE (General Diagnostic Engine, cf. [dKW87]) and has received much attention in the model-based diagnostic community. The ATMS that we present in this section is sometimes known

as the basic ATMS due to the many extensions and improvements that built upon the original one.

In contrast with the RMSs presented until now, the basic ATMS works in *all consistent contexts* in parallel. When a certain context is discovered inconsistent the ATMS automatically stops working in that context and thus, it does not need the contradiction-handling component which was required by the JTMS, LTMS and the JTMSset. The ATMS labels each node with *all minimal and consistent sets of assumptions* which, together with the justification set, logically entail the proposition of that node.[8]

The ATMS maintains a concise characterization of the inconsistent contexts. This is provided by the minimal sets of assumptions that lead to the derivation of an inconsistency for a given set of justifications. These are called *minimal nogoods* and are stored in the ATMS nogood database. The inconsistency is detected, as in the JTMS, using contradiction nodes. The environments propagated to the label of a contradictory node are (minimal) nogoods.

Given the current set of justifications $\mathcal{J}$, the ATMS labels each node n with a set of environments $n.label$ having the properties:

1. *Completeness:* For any set of assumptions $\mathcal{A}$ such that $\mathcal{A} \cup \mathcal{J}$ is consistent and $\mathcal{A} \cup \mathcal{J} \models n$ there exists an environment $e \in n.label$ such that $\mathcal{A} \supseteq e$;
2. *Consistency:* For any $e \in n.label$ we have: $e \cup \mathcal{J}$ is consistent;
3. *Soundness:* For any $e \in n.label$ we have: $e \cup \mathcal{J} \models n$;
4. *Minimality:* The elements of the label are minimal with respect to set inclusion, i.e. $\forall e, e' \in n.label : e \subseteq e' \Rightarrow e = e'$.

The constraints that the labeling algorithms must maintain can be expressed as:

1. $e \in n.label \Leftrightarrow e$ is minimal in $n.label \wedge e$ is not a superset of a minimal nogood $\wedge (\exists J \in n.justifications$ s.t. $e \in propagSet(J))$;
2. $propagSet(n_1, n_2, \ldots, n_k \rightarrow n) = \{e_1 \cup \ldots \cup e_k | e_i \in n_i.label\}$.

Moreover, the label of the premises contains the empty environment, i.e. $n.label = \{\{\}\}$, the label of derived nodes without justifications is empty,

[8]The ATMS was characterized (cf. [RdK87]) to compute certain prime implicates of a set of propositional clauses. We will come later to this characterization.

i.e. $n.label = \{\}$, the assumptions mention themselves in the label,[9] i.e. $a.label = \{\{a\}\}$. The environments propagated to the contradictory nodes are minimal nogoods and are removed, together with their supersets, from all node labels as soon as they are discovered. Figure 2.3 depicts a small dependency network, with the labels attached by the ATMS.

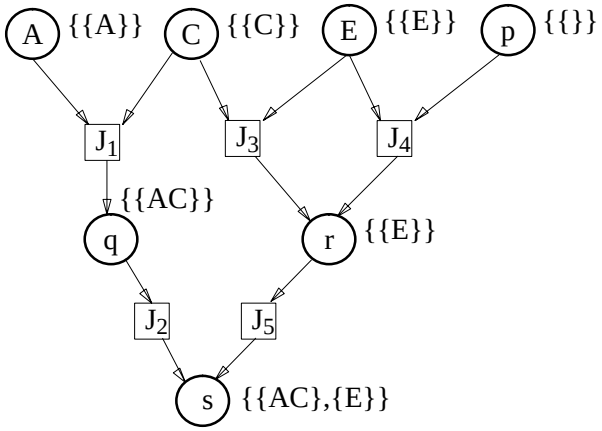

Figure 2.3: Small dependency network in the ATMS. A, C, E are assumptions; p is a premise; q, r, s are derived nodes; $J_1, \ldots, J_5$ are justifications.

The ATMS takes in account all the possible well-founded supports for each node. This information is not reflected in a special node slot, but in the node labels. Each environment from a node label corresponds to (at least) one well-founded support.

The entailment checking and the consistency checking are realized using the more powerful computation of the minimal supporting environments in the ATMS. However, the algorithms underlying the ATMS are known to be exponential in the worst case.[10]

More details about the data structures and algorithms can be found in Appendix A.5.

[9] Justified assumptions receive additionally the propagated label.

[10] Recall that for propositional Horn clauses the consistency check and the entailment check can be detected using polynomial algorithms, like the BCP.

2.3.5 The CMS

The CMS (Clause Management System, cf. [RdK87]) is a generalization of the ATMS that works with arbitrary clauses, like the LTMS does. In [RdK87] the ATMS was characterized to compute some prime implicates of a set of propositional Horn clauses. The CMS is also based on the computation of prime implicates, but for arbitrary clauses. Since a CMS would have much higher computational costs than the basic ATMS, and even for the basic ATMS they are in most of the cases prohibitive, we do not think that the CMS has much practical relevance. We list, however, the main theoretical aspects underlying the CMS (they provide also a characterization for the ATMS).

A *literal* is a propositional symbol or the negation of a propositional symbol. A *clause* is a finite disjunction of literals.

Definition 2.3.1 *Let Σ be a set of propositional clauses and M be a propositional formula. A clause S is a minimal support clause of M with respect to Σ, i.e. $S \in MinSupp(M, \Sigma)$ if and only if:*

1. $\Sigma \cup \neg S$ *is satisfiable;*
2. $\Sigma \models (\neg S) \Rightarrow M$*;*
3. *No proper subset of S has properties 1 and 2.*

The CMS is supposed to incrementally receive arbitrary clauses from the PS. Σ represents the set of clauses stored in the CMS at a certain moment. The PS can query then the minimal support clauses for some other arbitrary clauses (e.g. M above), not necessarily just for the ones supplied to the CMS. The importance of the minimal support clauses in problem solving and in particular in AI was argumented in [RdK87]. As a computational mechanism for the minimal support clauses Reiter and de Kleer propose the computation of prime implicates.[11]

Definition 2.3.2 *A* prime implicate *of a set Σ of clauses is a clause C such that: $\Sigma \models C$, and for no proper subset C' of C does $\Sigma \models C'$.*

The characterization of the minimal support clauses in terms of prime implicates states (cf. [RdK87]):

[11] The notion of prime implicate is the dual of the well-known prime implicants, which play an important role in Boolean minimization of switching circuits. Namely, a prime implicant of ϕ is a minimal conjunction of literals that entails ϕ. The prime implicants / implicates have the same properties modulo the duality between $\wedge$ and $\vee$.

Theorem 2.3.3 *$S \in MinSupp(C, \Sigma)$ if and only if*
$S \in \Delta(C, \Sigma)$ and no clause of $\Delta(C, \Sigma)$ is a proper subset of S,
where $\Delta(C, \Sigma) = \{\Pi - C \mid \Pi$ is a prime implicate of Σ and $\Pi \cap C \neq \{\}\}$.

The literature contains many algorithms for the computation of prime implicants / implicates (cf. [dK92a, Qui59]), however, the problem is known to be NP-complete. The basic ATMS was characterized to compute:

Theorem 2.3.4 *Let $\mathcal{J}, \mathcal{A}$ be the set of justifications and assumptions transmitted to the* ATMS*. Let $\{n\}$ be a query, where n is a propositional symbol. The answer to this query is:*

$$\{A_1 \wedge \ldots \wedge A_k \mid A_i \in \mathcal{A} \text{ and } (\neg A_1 \vee \ldots \vee \neg A_k) \in MinSupp(n, \mathcal{J})\}.$$

2.3.6 The lazy ATMS

The LazyATMS (known as the LazyRMS, cf. [KvdG93]) was designed due to the dissatisfaction with the high computational costs of using the basic ATMS in scheduling and constraint satisfaction problems, but also in model-based diagnosis (cf [KvR92]).

The LazyATMS behaves like the basic ATMS, except that it employs a lazy strategy for computing the node labels. Unless the PS explicitly shows interest at a certain time in the label of a certain node, the LazyATMS does not update the label of that node. Since the ATMS maintains the labels in all the possible consistent contexts and since there can be an exponential number of such contexts, the ATMS labeling requires in most of the cases prohibitive amounts of time and memory. The LazyATMS avoids the amounts of ATMS work that are not relevant for the PS, and assumes that not all of the nodes are interesting at each moment for the PS. In [KvR92, KvdG93] the authors show significant reductions in both time and memory over the use of the basic ATMS.

In order to keep trace of which labels have already been computed and what might be in an incomplete / inconsistent state the LazyATMS maintains at each node a boolean mark which indicates that. The addition of a justification implies by default only the maintenance of the marks. When a justification is added all of the followers of the consequent in the justification network must be marked as having possibly unupdated labels. When the PS queries the label (belief) of some node the LazyATMS recursively steps backwards in the justification network until it reaches nodes with updated labels

and incrementally propagates the changes to the nodes that are required to answer the query. Afterwards some of the marks will be changed to indicate that the labels are complete and consistent.

In order to recall which label updates were not considered by each justification, the justification data structure of the LazyATMS may include another slot - the $delayedConstraints$. The $delayedConstraints$ can store a set of triples $(J, n, envSet)$ with the meaning: "the addition of the environments $envSet$ to $n.label$ was not propagated further by J", where n is an antecedent node of the justification J. When a justification is marked "required" the label updates from $delayedConstraints$ must be reconsidered.

As until now, we specify the constraints that the labeling algorithms have to maintain in the dependency network. Except the mark and the label, during query answering the nodes and the justifications can be marked as "required" or not:

1. The LazyATMS is not answering a query:

   - if a justification is added or if the mark of an antecedent of a justification is set to "unupdated" then the consequent of the justification is marked "unupdated", unless the consequent is a premise;

2. The LazyATMS answers a query about the label of a node:

   - by default the nodes and the justifications are not "required", but the queried node is "required";
   - if a node with an "unupdated" mark is "required" then also all the incoming justifications and their antecedents are "required";
   - if a node with an updated label is required then none of its incoming justifications are "required";
   - a justification that is not "required" imposes no constraints on the labels of the nodes it connects. A "required" justification imposes the usual ATMS constraints between the labels of the connected nodes (see 2.3.4).

   After a query is processed the "unupdated" mark is removed from all the nodes that were "required".

Obeying the above constraints can guarantee that after the query of a node its label will be sound, minimal and complete. The consistency, however, is not fully guaranteed. In order to guarantee the consistency also one

has to query the contradiction node(s) each time a query of a different node is posed. This observation suggests that, differing from the basic ATMS, in the LazyATMS one can explore also inconsistent contexts (if that makes sense).

While the LazyATMS avoids to perform work that is irrelevant for the PS, it, obviously, requires that the PS has a strategy of selecting what is *not* relevant. If the PS is not careful about this aspect and, for instance, always queries the labels of all nodes, then the LazyATMS would provide no advantage over the basic ATMS. It is also true that, even in this worst case, the additional work of maintaining the marks in the LazyATMS is not very substantial.

The above point indicates that the consumer-based architecture of the PS is not going to effectively take advantage of the lazy strategy. The consumer architecture is driven by the changes in the belief state of the nodes which (almost) constantly requires to maintain updated labels. There seems to be currently no general strategy for driving the PS inference on top of the LazyATMS (the solutions of [KvR92, KvdG93] are appropriate for the particular applications addressed).

Since the consistency check and the entailment check are still based on computing minimal supporting environments, the worst-case complexity for providing these services remains exponential in the LazyATMS .

More details about the LazyATMS are given in Appendix A.6.

2.3.7 The focusing ATMS

The focusing ATMS (cf. [DF90, FdK88]) is another descendant of the basic ATMS that attempts to reduce the high computational costs of labeling. As opposed to the basic ATMS, the focusing ATMS (fATMS) avoids to work in *all* the consistent contexts in parallel. While it still can work in several contexts, the focusing contexts must be selected by the PS, like in the JTMSset. This requires that the PS be more selective about what combinations of assumptions to keep in the focus at a certain time.[12]

The focus can be specified extensionally, by enumerating the current focusing environments, or intensionally, by providing a general predicate that decides at each moment which environments are in focus and which not. The

[12] In the generic PS architecture presented in Appendix A.1 this task was solved by the PS contradiction handling module, which appears in A.1 under the name *CandidateGenerator*.

basic idea of the fATMS is to delay the propagation of those environments that are out-of-focus. In this respect, the node data structure can include an additional slot - the *blockedLabel* - used to store the environments whose addition to the node label and further propagation in the network is delayed because they do not agree with the focus. When the focus changes, however, the previously delayed label updates that enter the focus must be reprocessed.

The label completeness and consistency are weaker than in the basic ATMS, namely they are ensured only *relative* to the current focus:[13]

1. *Completeness w.r.t. focus:* For any set of assumptions $\mathcal{A}$ such that $\mathcal{A} \cup \mathcal{J}$ is consistent and $\mathcal{A} \cup \mathcal{J} \models n$ and $Focus \models \mathcal{A}$ there exists an environment $e \in n.label$ such that $\mathcal{A} \supseteq e$;
2. *Consistency w.r.t. focus:* For any $e \in n.label$ s.t. $Focus \models e$ we have: $e \cup \mathcal{J}$ is consistent;

The fATMS can guarantee that it only finds those nogoods that are relevant to the focus, i.e. that invalidate (at least a part of) the focus. Appendix A.7 gives more details about the operation of the fATMS.

In the COCO architecture of [DF90] more sophisticated environment control mechanisms were proposed. The justifications have so called guards attached that control which environments they propagate and which not. The control using the guards can be done at a finer granularity: they can use global information or local one. It is less clear what degree of consistency and completeness one gets if the guards use different local decision strategies, but the proposed control mechanism can be very strong and flexible.

The worst case complexity of the focusing ATMS remains exponential. By maintaining relatively few contexts in the focus the average costs of labeling was shown to be tremendously reduced in comparison with the basic ATMS in diagnostic problems (cf.[DF90, FdK88]). For complex problems the costs remain very high. In Chapter 4 we aim to further reduce them by: *(a)* combining the focusing control with the lazy label evaluation; *(b)* separating the algorithms that provide the entailment and consistency checks from the expensive minimal supporting environment computation.

[13]The label soundness and minimality have the same formulation as in the basic ATMS.

2.4 Discussion

We have presented a series of RMSs with different degrees of complexity and expressiveness: Some of them work in single contexts, others in some set of contexts, others in all the contexts in parallel (see Table 2.1). Some of them allow to directly represent negations and arbitrary clauses (the LTMS and the CMS), others do not.[14] Some of them offer automatic conflict resolution (the nmJTMS and the nmATMS), others need external mechanisms for doing this (the monotonic RMSs). Some of them address tractable problems (the LTMS and JTMS), others address NP-hard problems (the ATMS family and the nmRMSs).

How to decide if an RMS would be useful for a specific application, and, if so, which RMS to take, does not seem to be easy questions. In the context of diagnosis we try to address them in Chapter 4. Relevant questions for taking a decision are:

- How expensive are the inferences?
- Is it important to explain the reasoning?
- How many solutions are wanted: one, several, all?
- How many context changes (backtrackings) are likely to be performed until a solution is usually found?
- In case of inconsistency (or failure), are the (minimal) conflicts found small enough to be informative?
- Is there a significant overlapping among the possible worlds?
- Is the parallel exploration of several alternatives required or useful?
- Is it possible to identify a preference among the solutions?

We end this chapter with a discussion of the services that the different RMSs can provide to their users from a logical point of view.

Both the label propagation in the JTMS (JTMSset) and in the LTMS is equivalent with unit resolution. However, the JTMS is restricted to work with propositional Horn clause sets, while the LTMS works with arbitrary propositional clause sets (cf. [FdK93], pp. 280-282, 467-468). As we know (cf. [HW74]), unit resolution provides a sound and complete check for the

[14] Negation and even arbitrary clauses can be encoded in all the RMSs, but not directly. Usually, in order to do this, several nodes, justifications and rules of inference must be added by the problem-solver in order to overcome the logical incompleteness of the RMSs (see [FdK93, dK86b, dK88, Dre88]).

satisfiability of Horn propositional clause sets. The BCP of the LTMS achieves this property also for non-Horn clause sets that can be rewritten as Horn sets by substituting literals with their negations (see also [Lew78]). However, the completeness of the satisfiability check is not guaranteed in general for non-Horn sets in an LTMS.

It is also known that unit resolution in propositional Horn theories is a sound and complete inference procedure for deciding the entailment of positive propositions. This is both true for the JTMS and for the LTMS, while the LTMS guarantees this as well for clause sets that can be rewritten as Horn, as mentioned above. This services are supplied efficiently, i.e. the worst-case complexity of BCP is linear in the number of literals in the clauses.

The labeling in the ATMS is equivalent with the computation of some prime implicates, respectively some minimal support clauses (cf. [RdK87]). This is a service that goes beyond the power of the JTMS or of the LTMS, i.e. based on the computation of the prime implicates (minimal support clauses) also the satisfiability checks and the positive literal entailment can be supplied, but not vice-versa. However, the prime-implicate generation can come at significant additional costs: the computation of prime implicates is known to be NP-complete. The CMS, the basic and the focusing ATMS differ, however, in the amount of prime implicates computed.

The following points summarize this discussion.[15]

- the JTMS operates on a set of enabled assumptions $\mathcal{A}$ (standing for positive literals) and a set $\mathcal{J}$ of justifications (Horn clauses) and supports:
  - the (positive) entailment check, i.e. whether $\mathcal{A} \cup \mathcal{J} \models p$, where p is a positive literal. In this case p has an associated node in the JTMS and the answer is directly provided by the node's label.
  - the satisfiability check for $\mathcal{A} \cup \mathcal{J}$. This is realized as a side-effect of the above service, by asking if the contradiction node is entailed by $\mathcal{A} \cup \mathcal{J}$.
- the JTMSset may operate with several sets of enabled assumptions $\mathcal{A}_i \in Focus$. It provides the same services as the JTMS for each $\mathcal{A}_i \cup \mathcal{J}$.
- the LTMS operates on propositional clause sets $\mathcal{C}$ and a set of enabled assumptions $\mathcal{A}$ (that can be negative or positive literals) and supports:

[15] The negative clauses are represented as justifications to the contradiction node in the JTMS, JTMSset, and in the ATMS family.

- the (positive) entailment check, i.e. whether $\mathcal{A} \cup \mathcal{C} \models p$. The answer to this question is true if the node p is labeled `true`;
- the satisfiability check for $\mathcal{A} \cup C$;

The answers of the LTMS are always sound, but may not be complete. The completeness is guaranteed only for Horn clauses, or for clause sets that can be rewritten as Horn by substitution;

- the CMS operates on propositional clause sets C. It can answer, for any clause M the set: $MinSupp(M, C)$ (see 2.3.5);
- the ATMS operates on a set of assumptions $\mathcal{A}$ (standing for positive literals) and a set of justifications (propositional Horn clauses) $\mathcal{J}$. For any node n it computes

$$MinSupp_{\mathcal{A}}(n, \mathcal{J}) := \{(\neg a_1 \vee \neg a_2 \vee \ldots) \in MinSupp(n, \mathcal{J}) \mid a_i \in \mathcal{A}\}.$$

When the above n is replaced with the contradiction node, the set of minimal negative prime implicates consisting of assumptions is computed - corresponding to the minimal nogoods. Both the satisfiability check and the positive entailment check are supported in the ATMS using the above service for any $\mathcal{A}' \subseteq \mathcal{A}$ (i.e. by inspecting the node labels and the nogoods).

- the lazy ATMS is providing the same services like the basic ATMS, but the labels (i.e. the sets $MinSupp_{\mathcal{A}}(n, \mathcal{J})$) are computed by request, while in the ATMS they are always updated;
- the focusing ATMS has the same services like the basic ATMS, but the sets $MinSupp_{\mathcal{A}'}(n, \mathcal{J})$ are computed for those sets of assumptions $\mathcal{A}'$, $\mathcal{A}' \subseteq \mathcal{A}$ such that $\mathcal{A}'$ are in the focus of the ATMS.

Chapter 3

Fundamentals of Model Based Diagnosis

3.1 Introduction

This chapter presents some of the basic work formalizing the model-based diagnostic tasks. The reader familiar with the field may skip this chapter.

As we have seen in the first chapter, at the top level there are two tasks that must be accomplished: *candidate elaboration* and *candidate discrimination*. The candidate elaboration step must decide what diagnostic candidates (hypotheses) should the diagnostic agent believe, given a set of observations about the system under diagnose. The candidate discrimination must decide what "experiments" the diagnostic agent should perform next in order to discriminate between the current hypotheses.

3.2 Preliminary notions

For most of this chapter we only need first order logic.

Definition 3.2.1 *A* system *is a pair* $Sys \equiv (SD, Comps)$ *where:* (i) SD, *the* system description, *is a set of first-order sentences; and* (ii) $Comps$, *the* components, *is a set of constants.*

Definition 3.2.2 *A* diagnostic problem *associates to a system* $(SD, Comps)$ *a set* Obs *of first order sentences representing the observations about the system. We note with* $(SD, Comps, Obs)$ *a diagnostic problem.*

When the system under diagnosis is a physical system then the system description usually models the behavior of the individual components and the structure of the physical system in terms of components and connections. To denote that a certain component is of a certain type commonly a predicate denoting that type is used (e.g. $adder(AD273)$, or the membership is denoted in the manner usual in object oriented representations.

Each type (class) of components is characterized by a set of variables that usually stand for the parameters of a physical model of the component. The parameters may be denoted by specific logical functions applied to the component instances (e.g. $in1(AD273)$, or $in(AD273, 1)$). Some of the parameters are called *port variables* others are intern variables. The port variables represent the interface through which a component interacts with the environment and are used to describe the connections with other components, or the inputs and outputs of the system. Furthermore, it is useful to capture aspects like observability and controllability of system's parameters.

The connections are denoted using specific predicates for this purpose, e.g. $conn(in1(AD273), out(M17))$. A connection type is associated with a certain interaction model that constraints the values of the variables belonging to the connected ports: For instance a logical connection states the equality of two variables (or of the corresponding port variables); while an electrical connection enforces a more complicated constraint according to Kirchhoff's laws.

More details about the structure of the system under diagnosis can be captured by the system description, for instance part-of hierarchies or information about the geometrical lay-out. Since such kind of information (usually) does not have causal effects on the behavior of the system, most of the current diagnostic programs do not make use of it. However, these aspects are important when one develops repair plans, or when the diagnosis must also cover undesirable component interactions like bridge faults in electronics (cf. [Dav84, Cla95]).

The behavior of the components is described using the, so-called, *behavioral modes*. A behavioral mode characterizes a particular aspect of the correct or faulty behavior of a component. In general there can be several correct and several faulty modes of behavior, although most of the approaches assume that there is a single correct mode and assume that a component can be in one and only one mode of behavior at a certain time (i.e. assume that the behavioral modes are exclusive). Another common assumption is that the modes of behavior do not change during diagnosis. In this case

the mode of behavior assumed by a component can be denoted by a specific predicate for each mode (e.g. $ok(AD273)$) or using an intern variable of a component (e.g. $mode(AD273) = ok$). Typical fault modes are, for instance, the stuck-at-one, stuck-at-zero faults, in logical circuits, or the broken and shorted to ground wires in electrical circuits. A mode of behavior of a component is associated a constraint among the component's parameters, e.g. $ok(c) \wedge adder(c) \Rightarrow out(c) = in1(c) + in2(c)$.

3.3 Candidate elaboration

In most of the papers this task alone is referred to as (model-based) diagnosis, since the candidates elaborated are possible diagnoses and must conform to a certain definition of what diagnoses are. There exists a spectrum of different definitions in the literature. In order to have any pragmatic value a theory of diagnosis must provide means to (efficiently) compute the possible diagnoses. But, one of the problems faced, is that there are usually a huge set of possible diagnoses whose direct computation is in practice too expensive. Different approaches try to deal with this problem in different ways, for instance:

- some approaches try to characterize concisely the set of all candidates, using *descriptions* that stand for possibly large sets of candidates (for instance the minimal diagnoses of [Rei87], or the kernel diagnoses of [dKMR92]);
- some definitions put more constraints on what a candidate is meant to be (e.g. the use of "alibis" in [Rai92], or abductive explanation versus consistency in [CT91]);
- some approaches characterize only a subset of candidates, e.g. the one that are more plausible (cf. [DS92]).

Another dimension on which the approaches differ is the expressiveness of the knowledge taken into account, for instance:

- only the correct mode of behavior is considered (cf. [dKW87, Rai92, Rei87]);
- only the fault modes are considered (cf. [CDT89]);
- both the correct and the defect behavior are considered (cf. [CT91, dKMR92, dKW89, DS92]).

In order to characterize the diagnoses all approaches presented here require notions like logical consistency or logical entailment, which are in general undecidable.

3.3.1 Minimal diagnoses

The work of Reiter (cf. [Rei87]) was one of the most influential in the field of model-based diagnosis. His theory provides a logical foundation for the GDE (cf. [dKW87]).

Reiter's definition of diagnosis is based on the notion of *abnormal*. A literal $AB(c), c \in Comps$ is supposed to be true if and only if the component c is behaving abnormally, i.e. it represents the negation of the predicate denoting the correct mode of behavior.

A diagnosis is a set of components that, if abnormal, make the system description consistent with the observations. A minimal diagnosis appeals to a principle of parsimony, namely it is a minimal (w.r.t. set inclusion) set of components that must be abnormal in order to achieve the consistency.

Definition 3.3.1 *A* diagnosis *for* $(SD, Comps, Obs)$ *is a set* $\Delta \subseteq Comps$ *such that* $SD \cup Obs \cup \{AB(c) \mid c \in \Delta\} \cup \{\neg AB(c) \mid c \in Comps - \Delta\}$ *is consistent. A* minimal diagnosis *is a minimal set of components satisfying the above requirement.*

In order to generate the possible minimal diagnoses the set of conflicts plays an important role:

Definition 3.3.2 *A* conflict *for* $(SD, Comps, Obs)$ *is a set* $\{c_1, \ldots, c_k\} \subseteq Comps$ *such that* $SD \cup Obs \cup \{\neg AB(c_1), \ldots, \neg AB(c_k)\}$ *is inconsistent. A* minimal conflict *is a minimal set of components satisfying the above requirement.*

Reiter proved that a minimal diagnosis is a minimal set of components such that it intersects each minimal conflict. In [Rei87] an algorithm for the computation of the minimal diagnoses using the minimal conflicts is given (an algorithm that works correctly also when the conflicts are not minimal is given in [GSW89]). The candidates generated by the GDE (cf. [dKW87]) are minimal diagnoses. GDE uses only the description of the correct behavior of the components, i.e. each component has a correct mode and an unknown mode of behavior. Reiter proposed to use a theorem prover in order to detect

the set of conflicts. The GDE uses an ATMS in order to compute the minimal conflicts. In addition, GDE used probabilities in order to generate only the most probable minimal diagnoses.

As shown in [dKMR92], if in the clausal form of the system description every occurrence of an AB literal is positive – which is the case in the GDE – then the minimal diagnoses characterize the set of all possible diagnoses, i.e. every superset of a minimal diagnosis is a diagnosis.

3.3.2 Prime diagnoses: culprits without alibis

Raiman considered in [Rai92] the case when there are negative occurrences of the AB literals in the clausal form of the system description. In such cases he showed that one can exonerate some components. A set of components can provide an *alibi* for a suspected component if the correctness of the alibi set implies the correctness of the suspected component, given the system description and the observations. In such cases the minimal diagnoses, as defined by Reiter, do not characterize the set of all possible diagnoses. Namely, not each superset of a minimal diagnosis is a diagnosis.

Raiman defined the notion of *trial diagnosis* which is a set Δ of components that intersects each conflict and, in addition, Δ intersects each alibi of each $c \in \Delta$ (i.e. the possible culprits are suspects without valid alibis) .

The trial diagnoses characterize the set of all diagnoses if $SD \cup Obs$ is a Horn theory. In such a case each diagnosis is a union of some trial diagnoses, and each union of trial diagnoses is a diagnosis. However, the second part of the previous statement does not hold in general for non-Horn theories.

3.3.3 Fault models: GDE+ and Sherlock

Systems like GDE (cf. [dKW87]) constrained only the correct behavior of the components, i.e. the component descriptions included only the correct mode of behavior and an *unknown* fault mode. This style of modeling has the advantage that one does not need to acquire knowledge about the faulty behavior. It also removed the limitation of assuming *complete* knowledge about the faulty behavior.

On the other side, as argued in [SD89], allowing the faults to behave arbitrarily leads to the construction of diagnostic candidates without any *physically* sensible meaning. The diagnostic engine may "invent" faults that do not exist and cannot exist.

In order to overcome this problem a few years later the GDE was extended to work with fault models by two related successors: GDE+ (cf. [SD89]) and Sherlock (cf. [dKW89]). In GDE+ complete knowledge about the faulty behavior is assumed. When complete knowledge about the faulty behavior can be assumed it is possible to exonerate components: if none of the fault modes of a component is consistent with the system description and the observations then the component must be correct. GDE+ uses this kind of inferences to conclude the correctness of the components, when possible. In such cases the concept of minimal diagnosis is no longer adequate to characterize the space of possible diagnoses, namely, not all supersets of a minimal diagnosis are diagnoses.

Sherlock still uses an unknown fault mode, and so it does not introduce the assumption about the completeness of the knowledge about faulty behavior. In this case the minimal diagnoses still characterize the space of all possible diagnoses (cf. [dKMR92]). But, since the faults are still allowed to behave arbitrarily, there is no logical means to prohibit the generation of physical impossible diagnoses. This problem is alleviated in Sherlock by the use of probabilistic knowledge: the unknown faults are usually assigned a very small probability such that, for instance, they will only be considered when no candidate using known fault models is consistent with the observations.

The candidates in Sherlock are complete mode assignments to the components, i.e. a candidate assigns a unique mode of behavior to each component of the system. The information about the probabilities of the behavioral modes of each component is used in Sherlock in order to generate only some most probable candidates (called the *leading* diagnoses) at a time. GDE+ also avoided to generate all the possible diagnoses by focusing on, for instance, single faults, double faults or using other heuristic focusing strategies. The candidate generation in Sherlock and GDE+ uses the minimal conflicts between the modes assigned to the components. The conflicts are identified using a focusing ATMS in Sherlock and GDE+.

3.3.4 Kernel diagnoses

An important work at characterizing diagnoses, which generalizes the minimal and the trial diagnoses, is provided by [dKMR92]. The minimal diagnoses characterize the space of all diagnoses only if the clausal form of $SD \cup Obs$ contains no negative occurrences of AB literals. The trial di-

agnoses characterize the space of all possible diagnoses if each clause from $SD \cup Obs$ contains at most one negative AB literal, i.e. if $SD \cup Obs$ is Horn. However, neither the minimal nor the trial diagnoses characterize the whole space of diagnoses in general.

A diagnosis is an explicit assertion of positive and negative AB literals for system's components consistent with $SD \cup Obs$. The conflicts in [dKMR92] generalize the conflicts of [Rei87] and the alibis of [Rai92], namely a conflict is defined to be any clause entailed by $SD \cup Obs$ and containing only AB literals.

A partial diagnosis is an assignment of AB-literals that does not necessarily assert an AB-literal for each component of $Comps$. The components for which no assertion is made in a partial diagnosis can be consistently assumed to have either a positive or a negative AB-assertion. A *kernel* diagnosis is a partial diagnosis with a minimal number of AB literals. [dKMR92] concluded the following remarkable results:

- The *minimal* diagnoses of $(SD, Comps, Obs)$ are the prime implicants of the set of *positive* minimal conflicts of $(SD, Comps, Obs)$;
- The *kernel* diagnoses of $(SD, Comps, Obs)$ are the prime implicants of the minimal conflicts of $(SD, Comps, Obs)$.

Although important from a theoretical point of view, the kernel diagnoses are from a practical point of view too expensive to compute.

3.3.5 Preferred diagnoses

In this paragraph we outline the work of Dressler and Struss from [DS92, DS94]. A preference order among the modes of each component induces a preference among the diagnoses. The preference relation reflects usually some notion of plausibility or criticality.

Assume, without loss of generality, that the mode of behavior of a component is an intern variable of the component denoted by $m(c), c \in Comps$. The mode of behavior of each component c can take mutually exclusive values from a fixed domain $modes(c)$, where $modes(\cdot)$ is a function associating to each component a finite set of constants denoting the modes. One of the modes of $modes(c)$ for each $c \in Comps$ is ok denoting the correct mode of behavior. The mode assigned to a component c is denoted using equality: $m(c) = m_i$, where $m_i \in modes(c)$.

The modes of each component are ordered according to a *preference partial order:* $\preceq \subseteq modes(c) \times modes(c)$. As usual, $x \prec y \Leftrightarrow x \preceq y \wedge \neg(y \preceq x)$. We note two that modes are equally preferred by: $x \simeq y \Leftrightarrow x \preceq y \wedge y \preceq x$. The correct mode of behavior is strictly preferred over all the other modes of a component: $ok \prec m, \forall m \in modes(c) - \{ok\}$.

A mode-assignment for a set of components $\Delta \subseteq Comps$ is a set of literals $\{m(c) = m_{ci} \mid c \in \Delta, m_{ci} \in modes(c)\}$.

Furthermore, in [DS92] it was assumed that the diagnostic engine can adopt several working hypotheses. The working hypotheses can be simplifying assumptions for instance. Let $Whyp$ be the set of literals denoting the working assumptions. Any subset $w \in 2^{Whyp}$ is called a world in [DS92] and represents the set of working assumptions that the diagnostic engine decides to adopt at a certain time.

Definition 3.3.3 *Let $w \in 2^{Whyp}$ be a world. A mode assignment for $Comps$, i.e. $\Pi = \{m(c) = m_{ci} \mid c \in Comps\}$, where $m_{ci} \in modes(c)$, is a* diagnosis *in world w iff $SD \cup Obs \cup w \cup \Pi$ is satisfiable.*

Definition 3.3.4 *Let $\Pi = \{..m(c) = m_{ci}..\}$, $\Pi' = \{..m(c) = m'_{ci}..\}$ be two mode-assignments for a set Δ of components.* Π *is* preferred *to* Π'*, i.e. $\Pi \leq \Pi'$ iff $m_{ci} = m'_{ci} \vee m_{ci} \prec m'_{ci}$ for all $c \in \Delta$.*

If two mode-assignments assert two distinct but equally preferred modes to some component ($m_{ci} \simeq m'_{ci} \wedge m_{ci} \neq m'_{ci}$), then the mode-assignments are not comparable, i.e. none is preferred to the other, irrespective how the other modes relate.[1]

Definition 3.3.5 *A diagnosis Π is a* preferred diagnosis *in the world w iff no diagnosis in world w is strictly preferred over it: $\forall \Pi'$, where Π' is a diagnosis in world w we have $\Pi \leq \Pi' \Rightarrow \Pi = \Pi'$*

Subsequently, Dressler and Struss characterized the preferred diagnoses in default logic. A (normal) default (cf. [Rei80]) is an inference rule $a : b \ / \ b$, with the meaning "if a is derived and it is consistent to assume b, then derive b". The preferences among the modes of each component are expressed using normal defaults. For each mode $m_i \in modes(c)$ a default is

[1] This is a minor correction to the definition from [DS92, DS94], where the preference was defined to be $\Pi \leq \Pi'$ iff $m_{ci} \preceq m'_{ci}$ for all $c \in \Delta$. The preferred diagnoses computed in those papers are according to the above definition.

created:[2] $\left(\bigwedge_{m_j \in pre(c,m_i)} \neg(m(c) = m_j)\right) : (m(c) = m_i) \ / \ (m(c) = m_i)$, where $pre(c, m_i) = \{m \in modes(c) \mid m \prec m_i\}$. For instance, for the correct mode of behavior the rule $: (m(c) = ok) \ / \ (m(c) = ok)$ is added for each component.

A first order logic set of statements P has a unique deductive closure $ctx(P) = \{p \mid P \models p\}$. A *default* theory (D, P), where D is a set of defaults and P is a set of first order formulae (the premises), can have several *extensions*, which contain, besides the monotonically derivable formulae, also the consequences of *maximal sets* of applicable defaults. Since there can exist different maximal sets of applicable defaults there can exist several extensions of a default theory. The following theorem characterizes the preferred diagnoses in default logic:

Theorem 3.3.6 *Let D be the set of defaults resulting by encoding the preferences among the modes of each component as specified above.* Π *is a preferred diagnosis under w iff $ctx(SD \cup Obs \cup w \cup \Pi)$ is an extension of the default theory $(D, SD \cup Obs \cup w)$.*

Besides this result, [DS92] showed how the preferred diagnoses can be generated using the nmATMS. For the negation of each literal p that appears in the non-monotonic part of a default (e.g. in $a : b/b$, for $\neg b$), an out-assumption $(\neg b)_{out}$ is created. An out-assumption x_{out} encodes the assumption that there is no derivation for x (cf. [Dre88, Dre90], see also Appendix A.8). A default $a : b/b$ is encoded in the nmATMS as the justification $a \wedge (\neg b)_{out} \rightarrow b$. The nmATMS computes then extensions of the non-monotonic dependency network, which correspond to the extensions of the encoded default theory.

3.4 Candidate discrimination

Compared to the work relevant to candidate elaboration, candidate discrimination received less attention until now. In this section we are going to sketch the work done in [MR92] which formalized the notion of relevant and discriminating test in hypothetical reasoning and the work from [dKW87, dKW89, dK91, dKRS91] which addressed the problem on choosing the next point of observation based on the information gain. Further work on test pattern proposal for discrimination purposes is in [Str94].

[2] This is the encoding given in [DS94] which corrected the one from [DS92].

3.4.1 Discriminating tests for hypothetical reasoning

The work described in [MR92, McI94a] characterizes the tests that might confirm or refute a certain hypothesis and that might, or are guaranteed to discriminate among a set of competing hypotheses. The characterization is, again, in terms of prime implicates of a first-order theory, thus allowing ATMS-based implementations.

The formalism assumed a first-order propositional language. Σ is a set of first-order sentences denoting the background knowledge. In the case of diagnosis, more precisely of the candidate discrimination, Σ corresponds to the system description and the current set of observations (i.e. $\Sigma = SD \cup Obs$). Moreover, it is assumed that there exists a fixed set of hypotheses Hyp, and that each element of the hypothesis set can be consistently added to Σ. In diagnosis these hypotheses are the possible candidate diagnoses, i.e. the result of the candidate elaboration. Usually, each element of Hyp is a conjunction of literals from the language, e.g. a candidate diagnosis is a conjunct of mode assignments or of AB-literals.

The formalism further defines two sets of literals: the set of, so called, *achievable* literals and the set of *observable* literals. The achievable literals correspond to properties that an external agent can, at some time, enforce in the world.

The observable literals correspond to properties whose truth an external agent can observe, given some achievable conditions in the world, and whose value of truth can be added to the theory Σ.

Definition 3.4.1 *A* test *is a pair* (A, o) *where* A *is a conjunction of achievable literals and* o *is an observable.*

Definition 3.4.2 *The* outcome of a test (A, o) *is one of* $o, \neg o$.

The tests as defined above determine the truth value of a literal. Important is, however, that the outcome of a test is one of several mutually inconsistent values.

Definition 3.4.3 *The* outcome α of a test (A, o) confirms $H \in Hyp$ *iff* $\Sigma \cup \{A\} \cup \{H\}$ *is satisfiable, and* $\Sigma \cup \{A\} \models H \Rightarrow \alpha$. α refutes H *iff* $\Sigma \cup \{A\} \cup \{H\}$ *is satisfiable, and* $\Sigma \cup \{A\} \models H \Rightarrow \neg\alpha$.

The requirement that $\Sigma \cup \{A\} \cup \{H\}$ is consistent makes sense, since in general not all the conjuncts of achievable literals are consistent with the theory Σ and a certain hypothesis.

A refuting outcome for H allows one to reject H as a possible hypothesis. However, in general, a confirming outcome for H is not strong enough to definitely accept or to reject H - at most, in a probabilistic framework, this might increase the probability of H.

Theorem 3.4.4 *The outcome of a test (A, o) confirms (refutes) $H \in Hyp$ iff*

1. *There is a* prime implicate *of Σ of the form $\neg A' \vee \neg H' \vee \alpha$ ($\neg A' \vee \neg H' \vee \neg\alpha$) where A' is a subconjunct of A and H' is a subconjunct of H; and*
2. *No prime implicate P of Σ subsumes $\neg A \vee \neg H$, i.e. $\neg(P \Rightarrow \neg A \vee \neg H)$.*

The requirement that no prime implicate of Σ subsumes $\neg A \vee \neg H$ implies that $\Sigma \cup \{A\} \cup \{H\}$ must be consistent.

The theorem tells us that in order to find a test that might confirm or refute a certain hypothesis one can search certain prime implicates of Σ. Since o is propositional one can search some minimal support clauses of o given Σ (cf. [RdK87]). Such a task can be accomplished using an ATMS. Namely, if the achievable literals and the literals that appear in the hypotheses are assumptions in an ATMS then one can inspect the label of the node o in order to find the minimal support clauses for o. Thus, the outcome α of (A, o) confirms (refutes) H if the label of the node corresponding to α ($\neg\alpha$) contains a subset of $A \cup H$.

Definition 3.4.5 *A test (A, o) is a* discriminating *test for the hypothesis set Hyp iff $\Sigma \cup \{A\} \cup \{H\}$ is satisfiable for each $H \in Hyp$, and there exist $H_i, H_j \in Hyp$ such that the outcome α of the test refutes H_i if $\alpha = o$ or H_j if $\alpha = \neg o$.*

Thus, a *discriminating test* is guaranteed to refute at least one of the competing hypotheses, irrespective of the outcome of the test. A weaker kind of test is a *relevant test*. A relevant test can confirm or refute some hypotheses from a set, depending on its outcome. The relevant and discriminating tests were characterized in terms of *prime implicates* in [MR92].

3.4.2 Choosing the next measurement

In [dKW87, dKW89, dKRS91] a decision theoretic methodology for choosing the best next measurement is presented. The goal of diagnosis is to identify,

as far as possible, the actual candidate diagnosis. Usually, a single measurement is not enough to achieve that and a sequence of measurements are required. The method makes the simplifying assumption that probing has equal cost and that the computation cost is much smaller than the probing cost.[3] In these conditions the best sequence of measurements is one that has minimal length and achieves the desired effect. In order to guarantee the optimum usually *full* lookahead is required, i.e. all measurement sequences and all possible measurement outcomes must be considered. The full lookahead is, however, very expensive. As some empirical results show (cf. [dKRS91, Out93]) an approximate criterion for probe selection which is much cheaper to compute behaves very well, namely, the one-step lookahead based on the information entropy. The information entropy is a measure of the uncertainty in a space of hypotheses. The method always proposes the measurement that most decreases the information entropy about the possible diagnoses.

The approach uses probabilistic information about the modes of behavior of the components and assumes further that these probabilities are independent. In this conditions the (prior) probability of a candidate is given by the product of the probabilities of the candidate's mode assignments.

The entropy of the current set of competing candidates is given by:

$$H(Hyp, Obs) = -\sum_{c \in Hyp} p(c|Obs) \log p(c|Obs), \tag{3.1}$$

where $p(c|Obs)$ is the probability that the candidate c is the actual diagnosis given the current observations Obs. The proposed measurement is at a variable x_i where the outcome α of the measurement decreases, on the average, at most the above entropy, i.e. where $H(Hyp, Obs \cup \{\alpha\})$ is minimum on the average. Consider that the variable x_i takes values from a domain $Dom(x_i)$. The *average* entropy after measuring x_i is given by:

$$H(x_i) = \sum_{v_{ik} \in Dom(x_i)} p(x_i = v_{ik}|Obs) H(Hyp, Obs \cup \{x_i = v_{ik}\}), \tag{3.2}$$

where $p(x_i = v_{ik}|Obs)$ is the likelihood that the outcome of the measurement will be v_{ik} and $H(Hyp, Obs \cup \{x_i = v_{ik}\})$ is the entropy after the measurement, computed as in the formula 3.1.

[3] In many cases these assumptions, especially the first one, do not hold in practice.

The best measurement to make is the one that minimizes $H(x_i)$ (for more details see [dKW89]).

[dK90b] showed how the entropy computation can be further simplified under additional restrictions, such as the assumption that one does not exactly know the prior mode probabilities, but one can partition these in several classes between which the probabilities differ with orders of magnitudes.

Chapter 4

Aspects of Efficiency in an RMS-Based Diagnostic Engine

4.1 Introduction

The RMS systems provide useful services at the propositional level like, consistency checking, entailment checking, prime implicate generation (see 2.4), notions that are at the basis of the different formalizations of diagnosis, as we have seen in the previous chapter. Moreover, they offer attractive features like, incremental operation, support for dependency-directed backtracking, explanation, etc. It is thus no wonder that many of the model-based diagnostic engines developed so far are RMS-based (e.g. GDE [dKW87], GDE+ [SD89], Sherlock [dKW89], XDE [Ham91b], DDE [DS94]).

In this chapter we discuss:

- the efficiency of the RMS; and
- the efficiency of candidate generation.

4.1.1 The contents of this chapter

Section 4.1.2 "loosely" specifies an RMS-based diagnostic engine that provides the framework for the discussions of this chapter.

Section 4.2 suggests a new kind of RMS, namely the 2vRMS, that has the power required for fulfilling the tasks relevant for diagnosis but offers more flexible control facilities.

Section 4.3 discusses an efficient algorithm for the generation of a few most plausible /critical diagnostic candidates. In 4.3.4 we analyze the properties of the candidate generator with respect to the framework of propositional logic. We also discuss there the usefulness of increasing the formula completeness of the candidate generator. We conclude that, in case candidate generation is distributed among several modules, the increase of the formula completeness could be useful in order to detect minimal conflicts among choices committed by other modules, and we give an efficient algorithm for this purpose.

Section 4.4 analyses the combination of the 2vRMS and the candidate generator. It shows that the combination is a more powerful propositional reasoner that can support (heuristically-guided) satisfiability checking and (multiple) model construction for *arbitrary* propositional clause sets.

Section 4.5 discusses the relationship with other work. In 4.5.2 we also suggest an extension of the candidate generator with, so-called, "secondary choice-sets", i.e. choice sets where no preference among the elements is specified. The search in the secondary choice spaces is similar to the techniques used in constraint satisfaction, and the relationship is discussed there.

4.1.2 The framework

Figures 4.1 and 4.2 describe a simple focusing diagnostic engine that we can use as a framework for our discussions. The engine interleaves prediction, conflict detection and candidate generation in a similar way as Sherlock (cf. [dKW89]) does. Figure 4.2 gives an RMS-based description of the task of candidate testing. Firing an RMS-consumer is supposed here to implement value (constraint) propagation (for more details, see Appendix A.1). While the RMS alone usually works only at the level of propositional logic, the consumer mechanism provides the mean to link more general inference processes on top of the RMS and to go beyond the expressiveness limitations of the propositional logic.

Algorithm DiagnosticEngine ()
Focus is a global data structure containing a list of diagnostic candidates; The *Focus* is maintained by the procedure *CandidateGeneration*. *CandidateTesting* removes from the *Focus* those candidates that failed to "explain" the observations.

(1) initialize the data structures corresponding to the system description and the initial observations; initialize the *Focus*;

(2) **while** not *ResultDecided*() **do:**

;; candidate elaboration

(3) **repeat**

(4) **call** *CandidateGeneration*() to update the *Focus* candidates;

(5) **call** *CandidateTesting*();

until "enough" competing consistent candidates are found;

;; candidate discrimination

(6) **while** not enough "good" tests are found and "it is worth" searching for a better one **do:**

(7) generate a discriminatory (relevant) test for the current focus;

endwhile

(8) propose the best test(s); acquire the actions and the observations performed by the repair and testing agent;

endwhile

Figure 4.1: A simple diagnostic engine.

Procedure CandidateTesting ()
CandidateTesting use the focus candidates to predict new values for them and to check if they are consistent with the observations. The *Focus* candidates that fail the testing are removed from the focus. The conflicts discovered by the RMS may be communicated to the candidate generator in order to guide the further proposals. The following is an RMS-based description of this function. *Agenda* is a data structure containing an ordered list of activated consumers.

(1) focus the RMS on the *Focus* candidates;
(2) **while** *Agenda* is not empty and *Focus* is not empty **do:**
  (3) pick a consumer $\mathcal{P}$ and its associated RMS node n from the *Agenda*;
  (4) **if** n holds in the RMS focus **then**
    (5) fire $\mathcal{P}(n)$;
    else
    (6) reattach $\mathcal{P}$ to n;
  (7) **if** a contradictory node of the RMS holds in the current focus **then**
    (8) compute (minimal) nogoods for the current focus;
    (9) remove the inconsistent candidates from *Focus* and communicate the conflicts to the candidate generator;

endwhile

Figure 4.2: RMS-based candidate testing.

4.2 Reducing the costs of reason maintenance

4.2.1 On choosing the RMS for diagnosis

Apart from application dependent particularities, diagnostic problems usually require many context changes. They also require in general the elaboration of several diagnostic candidates, when several can be elaborated - either for the sake of guiding the information providing actions for candidate discrimination, or for providing the user a better picture of what is known and what is still unknown about the system under diagnosis. These features stress the preference for the multiple-context RMSs.

The supposition that usually a large number of context changes would be required until the final solution is identified, stresses the importance of the dependency-directed backtracking. This, and the efficiency of the procedures that generate candidates, depend strongly on the identification of the *minimal nogoods*. In this respect the ATMS-like RMSs seem to be more attractive.

The basic ATMS and the CMS must be ruled out due to the prohibitive costs of using them in almost anything else but toy-problems. The search in all of the possible contexts in parallel is first too expensive, and second not needed. In order to guide the action-proposal usually only a few alternative diagnoses need to be elaborated. Also, more than being interested in all of the possible diagnoses, since they are usually too many, a user is more interested in some of the most plausible of them. A focusing ATMS seems to be the most attractive from this perspective.

It is questionable, however, if in diagnosis one needs at all times and for all the nodes the full information that a focusing ATMS maintains in a label. During candidate elaboration, as long as some focusing candidates are inconsistent, the discovery of the minimal conflicts is important. For this purpose it is sufficient to maintain the label completeness and consistency for the contradictory nodes only. But, what about the rest of the nodes? Is it really necessary to maintain by default the complete and consistent (focusing) ATMS labels for them? In our opinion the answer is no. The conflicts usually involve a relatively small part of the components. Propagating labels in the justification network built using components not involved in the nogoods is not really relevant. Moreover, other tasks performed by a diagnostic engine, like the candidate discrimination, or the elaboration of repair plans, build inference structures in consistent contexts. For such inferences, the compu-

tation of the detailed minimal supporting environments might again not be relevant, even if it makes sense to cache these inferences in the RMS network. According to our experience, a significant part of the labeling effort spent by the focusing ATMS is not relevant for the diagnostic engine.

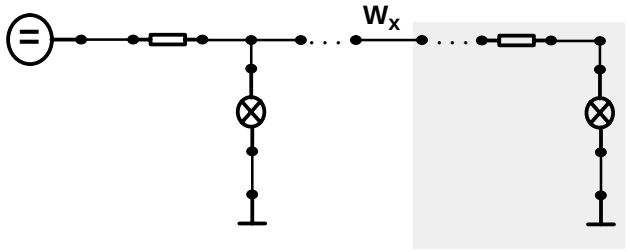

Figure 4.3: Irrelevant label computations.

Example 4.2.1 *Consider Figure 4.3 in which an electrical circuit containing power supplies, wires, bulbs, etc. is partially depicted. Let the focus of the diagnostic engine contain the candidate assuming that the wire W_x is broken. If this candidate is consistent with the observations then no value predicted as a consequence of this fault need be labeled. Suppose further that the assumption broken(W_x) is contradicted by a measurement from the right part of the circuit (the shadowed area). In this case, (a part of) the consequences of this fault must be labeled in the right network, but, at least, they need not be labeled in the left part of the circuit.*

In the context of a diagnostic engine like the one from the previous section, the RMS is expected to support the following tasks:

- consistency check for sets of clauses $\mathcal{A}_i \cup \mathcal{J}$, where $\mathcal{A}_i$ is (a superset) of a focusing candidate and $\mathcal{J}$ represents (part of) the logical description of the system under diagnosis and of the observations;
- in case of consistency, the entailment check is important for observation and test-pattern proposal, and also for triggering the activation of consumers that can implement more general inference rules in the engine;
- in case of inconsistency, there must exist some negative prime implicates of $\mathcal{J}$ that involve literals from $\mathcal{A}_i$ (i.e. the conflicts). The RMS

should be able to compute one (all) such conflict(s) for each inconsistent candidate. However, in many cases in practice, only a relatively small part of the justification network is involved in the derivation of the inconsistency - the computation of labels outside the responsible subnetwork seems to be an overzealous effort.

The problem of the focusing ATMS is that it bases the consistency and the entailment checks on the expensive computation of minimal supporting environments. An LTMS, JTMS, and a JTMSset solve these tasks much more efficiently.

The improved RMS that we are going to describe in the following combines the features of the JTMSset, of the focusing, and of the lazy ATMS. It can solve the consistency and the entailment checks with an efficiency comparable with the one of an JTMS, while also being able to support at request the more expensive computation of the minimal supporting environments, usually more efficient than any of the discussed members of the ATMS family.

4.2.2 The 2vRMS: combining focusing with lazy label evaluation

We assume that the problem-solver (PS) does not need the full (focusing) ATMS node labels for *all the nodes at all times*, but it might be interested to compute this information for *particular nodes at particular times*. More precisely, the requirements that we would like to be satisfied by our RMS are:

- it should support reasoning in a specified set of focusing contexts;
- for each node, at all times, the PS is interested if the node holds in the current focus, but not necessarily in the detailed dependence on the assumptions;
- for some nodes, at some times, the PS is interested in the detailed dependence on the assumptions, but only with respect to the current focus.

The hope is that using these relaxed requirements the RMS algorithms will become less expensive. The focusing ATMS was already shown to reduce tremendously the time and space required in diagnostic problems over the basic ATMS (see [FdK88, DF90]). However, since the fATMS maintains by default the completeness and consistency with respect to the focus for all

of the nodes, it does more work than necessary in order to fulfill the above requirements. The LazyATMS computes node labels only by request and seems appropriate to fulfill the third requirement listed above. Significant reductions of effort due to the usage of the LazyATMS compared to the basic ATMS are reported in [KvR92, KvdG93]. However, a LazyATMS cannot answer whether a certain node holds in the current focus (the second requirement listed above) without querying the node label. If one has to constantly query the node labels in order to find the support in the focus, then the lazy label evaluation brings no advantage. Thus, a direct combination of the focusing idea and of the lazy label evaluation is not going to solve all of the requirements that we want to fulfill in a convenient way.

In this respect, the proposed architecture integrates another RMS, namely a JTMSset. The JTMSset is used to decide the membership to the focusing contexts, i.e. without computing the detailed dependence on the assumptions as in an ATMS-like RMS. When a query about the detailed dependence on the assumptions of a certain node is posed, the 2vRMS acts like a combination of a focusing and a lazy ATMS. However, the 2vRMS is more than a sum of three different RMSs. More than just sharing the nodes and the justifications, the labeling procedures are integrated. The information computed by the cheaper JTMSset view is used during query answering to *control* the computations performed in the lazy-focusing ATMS view. This way, a tight control on the environment propagation is achieved, and, as our empirical experience with an implementation of these ideas showed, the costs of reason maintenance can be further reduced in diagnostic problems.

The two-view RMS

The 2vRMS (two-view RMS) tightly integrates the view of a JTMSset, called in the following the *focus* view, and the one of a combination of a focusing and a lazy ATMS, called in the following the *detailed* view. The views share the nodes and the justifications but attach distinct labels to the nodes: the *fLabel*, respectively, the *dLabel*. The $fLabels$ can be regarded as an abstraction of the $dLabels$ and the 2vRMS exploits this logical dependence for control purposes.

The 2vRMS works in a specified set of focusing contexts, specified by the PS in the same way as it was done in the JTMSset (see 2.3.3), i.e. by enumerating a set of *focusing environments*.

The $fLabel$ is the label that a JTMSset would attach to its nodes, i.e. it contains the set of (identifiers) of the focusing contexts where the node

is believed. The constraints among the $fLabels$ are exactly those of the JTMSset:

$$n.fLabel = \bigcup_{J \in n.justifications} J.fLabel;$$

$$J.fLabel = \bigcap_{n \in J.antecedents} n.fLabel.$$

The $dLabel$s are like the labels computed by a focusing ATMS. However, their maintenance is done by request, like in the LazyATMS. The addition of a justification triggers the update of the $fLabel$s, but not of the $dLabel$s. When a certain $dLabel$ is queried, the 2vRMS steps backward in the network of dependencies as long as delayed environment propagations exist, like the LazyATMS did. During query processing, some delayed $dLabel$ propagations are restarted, but their propagation is restricted to the set of justifications that are required to answer the query.

Maintaining the relationship between the two views

In order to keep track of which computations have already been done in the detailed view and which not, the 2vRMS attaches another slot to the nodes: the $fStatus$. The $fStatus$ of a node tells for which focusing contexts the $dLabel$ of the node might not be currently updated.[1] It is clear that for each node the $fStatus$ is a subset of the $fLabel$. If a certain focusing context is mentioned in a node's $fStatus$ this means that the $dLabel$ of that node might not be updated with respect to that context. If a node has an empty $fStatus$ this means that the $dLabel$ is complete with respect to the focus. The $fStatus$ makes the connection between the two views, its value being influenced by both of them. The addition of new justifications and the addition of new focusing environments cause the addition of new elements to the $fLabel$s and to the $fStaus$es (but not to the $dLabel$s). A $dLabel$ query causes the update of some $dLabel$s and the deletion of some elements from the $fStatus$es (but no change in the $fLabel$s).

We specify in the following the constraints that determine the value of the $fStatus$ slots. There are two ways of delaying the environment propagation in the 2vRMS. These are inherited from the fATMS and the LazyATMS (see also Section 2.3.6 and 2.3.7):

[1] The role of the $fStatus$ is similar to the one of the *marks* attached by the LazyATMS to the nodes, but these "marks" are maintained here independently for each focusing context.

- the environments propagated to a node, that do not hold[2] in the current focus are stored in the node's *blockedLabel*, like in the fATMS;
- the propagation tasks $(J, n, envs)$ mentioning a justification J that is not required for answering the current query, are stored in the justification's *delayedConstraints*, like in the LazyATMS.

In the fATMS the elements of the blocked labels were moved to the labels as soon as the focus changed and the blocked environments entered the new focus. We decided in the following to delay as well this operation until a node is required for query answering. We can now specify the constraints that govern the computation of the $fStatus$es:

$$n.fStatus = propagS(n) \cup localS(n); \tag{4.1}$$

$$J.fStatus = propagS(J) \cup localS(J); \tag{4.2}$$

$$propagS(n) = \bigcup_{J \in n.justifications} J.fStatus;$$

$$propagS(J) = J.fLabel \bigcap \left(\bigcup_{n \in J.antecedents} n.fStatus \right);$$

$localS(n) = \{i \mid \exists e \in n.blockedLabel$ s.t. e holds in the focusing context $i\}$;
$localS(J) = \{i \mid$ exists an environment e mentioned in $J.delayedConstraints$ s.t. e holds in the focusing context $i\}$.

In fact, the above $fStatus$ specification is quite strong: $i \in n.fStatus$ means not only that "the *dLabel* of n *might* not be updated w.r.t. the focusing context i", but also that: "an antecedent (node or justification) of n *really* blocked the propagation of an environment holding in the context i". One may relax the $fStatus$ constraints 4.1, 4.2 as bellow,

$$n.fStatus \supseteq propagS(n) \cup localS(n); \tag{4.3}$$

$$J.fStatus \supseteq propagS(J) \cup localS(J); \tag{4.4}$$

without losing the completeness of the *dLabel* computation. In the relaxed version, $i \in n.fStatus$ has the meaning that "an antecedent of n *might* have

[2]We say here that an environment holds in a context if all of the environment's assumptions are in that context.

blocked the propagation of an environment holding in the context i". It seems that there is a trade-off between making the $fStatus$ update easy (i.e. making the update of the focus view easy), on the one side, and making the query answering (i.e. the update of the detailed view) easy, on the other. For instance, the tighter $fStatus$ constraints require more checks, but no node / justification is marked "unupdated" without a real reason; while some relaxed $fStatus$ constraints can be easier enforced, but may mark as "unupdated" more nodes and justifications than necessary. In this respect, there is a possible spectrum of 2vRMS implementations that choose a different degree of "tightness" for the $fStatus$ constraints.

fStatus update at justification addition

When a new justification is added, its $fStatus$ is set equal with its $fLabel$, since the justification blocks any environment propagation by default. The 2vRMS further propagates the $fStatus$es, as well as the $fLabel$s, incrementally, i.e. reusing the old labels and propagating at the followers only the local incremental changes, if any. Also we avoid to compute the sets $localS$, when possible.

The degree of completeness and consistency and query answering

The PS may add and delete arbitrary environments to, respectively from, the focus. Like in the JTMSset, when an environment is deleted from the focus its identifier is simply added to the set $tms.obsoleteFocus$. Because the focus view is maintained by default, the activation of the consumers is triggered by the changes of the $fLabel$s. Like in the JTMSset, we stop the consumer activation in the contexts that are known to be inconsistent, or that were removed from the focus by the PS. In the JTMSset we used the set $tms.activeFocus$ containing the focusing environments that were still consistent and not obsolete. In the 2vRMS we store independently for each view its own active set of focusing environments: the $fActiveFocus$ for the focus view, and the $dActiveFocus$ for the detailed view. The motivation for this decision is that it supports more flexible control strategies over the environments computed and the nogoods searched in inconsistent contexts. The semantics for these two foci is more precisely stated as:

1. No obsolete focusing environment is in $fActiveFocus \cup dActiveFocus$;
2. The consumers are activated for the nodes holding in the $fActiveFocus$;

3. If a node is queried (i.e. its $dLabel$ is computed) the completeness of the $dLabel$ must be ensured with respect to the $dActiveFocus$;
4. If a focusing environment is discovered inconsistent in the focus view it will be removed from $fActiveFocus$, but not from $dActiveFocus$;
5. If one nogood, several, or all minimal nogoods - depending on the control strategy - is found for some inconsistent contexts (i.e. due to a query of a contradictory node), then the focusing environments that are supersets of the nogood(s) are removed from the $dActiveFocus$;

Because the focus view is maintained by default, while the detailed one is not, always a context will be discovered inconsistent first in the focus view and only eventually, if the contradictory nodes are queried (i.e. if the PS "wants" that) the search for (minimal) nogoods will be initiated and the contexts will be "discovered" inconsistent by the detailed view. This means that at all times we have $fActiveFocus \subseteq dActiveFocus$.

At query time, the subnetwork of nodes and justifications that are relevant for answering the query is determined and the $dLabels$ are incrementally updated in the relevant network. Like in the LazyATMS, we will first mark the nodes and the justifications that are required. However, different focusing contexts have different relevant subnetworks. Due to this fact, the "required" mark is, for each node and justification, a *set* with the meaning: "the node / justification is required for computing (d)labels in the context i, if i is in the set". Thus these slots are also a subset of the $fLabel$s, actually a subset of the $fStatus$es. The constraints that specify the assignment of these slots during query answering are:

- by default each node and justification stores an empty set in the field req, but for the queried node n, we have
  $n.req = n.fStatus \cap tms.dActiveFocus$;
- for all justifications $J.req = J.fStatus \cap J.consequent.req$;
- for all nodes

$$n.req = n.fStatus \bigcap \left(\bigcup_{J \in n.consequences} J.req \right).$$

A node / justification "required" in context i restarts the delayed environment propagation for the context i, i.e. the constraints among the $dLabels$ of the nodes connected by justifications required in a focusing context i are

the same as the constraints among the node labels of a focusing ATMS having the context i in focus (see Section 2.3.7). After the query is processed and the required $dLabels$ are updated, the $fStatus$es for the required net are updated by removing from the $fStatus$ the req set. Afterwards, the req fields are set to the empty set.

For reasons of flexibility we do not maintain by default the consistency of the detailed view. We do maintain the consistency of the $dLabels$ with respect to the nogoods stored in the nogood data base, but we do not necessarily query the contradictory nodes immediately when the focus view discovers a context inconsistent. Of course, the default $dLabel$ consistency maintenance is easy to perform: the function that signals the contradictions in the focus view just has to query the $dLabel$ of the contradictory node. However, we leave the decision of ensuring this degree of consistency to the PS because sometimes it is more efficient to delay the search for the nogoods (e.g. when a set of justifications have to be added in block), or the PS may not necessarily want to find the nogoods. In exchange, the 2vRMS supplies a function to the PS that, on demand, checks and ensures the consistency of the detailed view and performs the search for (minimal) nogoods in the inconsistent contexts.

Context changes

Due to the embedded JTMSset, the justifications also store the set of (focusing) contexts where they provide a well founded support for the consequent.

As in the JTMSset, the retraction of assumptions is performed at the time when a new focusing environment replaces an old obsolete focus environment. This helps to process in an incremental way the relabeling performed at context changes. As usual, when an assumption is retracted from a certain focusing context, first the identifier of that context is removed from all the assumption's followers whose well founded support depends on the assumption. Afterwards, the nodes touched at the previous step are examined once more for additional support in that context (the operations are similar to the ones performed in the JTMSset).

While the $fLabel$ propagation at context changes raises no specific problems, the $fStatus$ computation deserves a closer look. When the focusing context i changes, a new environment E_{new} replaces an obsolete focusing environment E_{old}. The assumptions from $E_{old} - E_{new}$ must be retracted in the context i, the assumptions from $E_{new} - E_{old}$ must be enabled in the context i, all the other assumptions remain as they were before.

fStatus update at assumption retraction

At assumption retraction, if one sticks to the tight $fStatus$ constraints (i.e. 4.1, 4.2), the $fStatus$ update cannot be done incrementally in some cases. The following example shows such a case.

Example 4.2.2 *Suppose we have a justification network including the assumptions A, B, the derived nodes p, q, and the justifications $J_1 : A \rightarrow p$, $J_2 : B \rightarrow p$, and $J_3 : p \rightarrow q$. Suppose the current state of the dLabels is as follows: $A.dLabel = \{\{A\}\}$, $B.dLabel = \{\{B\}\}$, $p.dLabel = \{\{A\}, \{B\}\}$, $q.dLabel = \{\{A\}\}$, i.e. J_3 delayed the propagation of $\{B\}$ from p to q. Suppose we have a single focusing environment (with identifier 1), namely $\{A, B\}$. In such a case, all the fLabels are set to $\{1\}$, and all the fStatuses are empty, except those of J_3 and q which are set to $\{1\}$. Furthermore, assume that J_1 supplies the well-founded support to p in the context 1 (this is important!).*

Suppose we retract now the assumption B from the focusing environment. 1 is removed from the sets $B.fLabel$, and $J_2.fLabel$, but no incremental change has to be performed at p. However, the tight $fStatus$ constraints would require to remove 1 from the $fStatus$ of J_3 and q, thus would require to inspect even the followers of the nodes where no incremental change was done. Failing to remove 1 from the above fStatuses would violate the strong $fStatus$ constraints, but not the relaxed ones.

Of course, if we agree to use a relaxed form for the $fStatus$ constraints, the $fStatus$ update can be performed incrementally. The above case appears relatively seldom, and we think that the amount of additional work required to impose the tight $fStatus$ constraints after assumption retraction does not justify the somewhat smaller effort required to answer a query that involves some nodes marked unnecessarily as unupdated. In the detailed description of the 2vRMS that will follow, we chose to keep the incremental $fStatus$ update during assumption retraction.

fStatus update at assumption enabling

When an assumption a is enabled in the context i, i has to be added to the $fLabel$ of a and the $fLabel$ change should be incrementally propagated at the followers. Although after assumption enabling the $fStatus$es can only grow, the $fStatus$ update at a and its followers cannot be performed incrementally

in all the cases if we want to work with the tight $fStatus$ constraints. The following example shows such a case.

Example 4.2.3 *Suppose we have a justification network like in Example 4.2.2, i.e. including the assumptions A, B, the derived nodes p, q, and the justifications $J_1 : A \rightarrow p$, $J_2 : B \rightarrow p$, and $J_3 : p \rightarrow q$, and having the current state of the dLabels as follows: $A.dLabel = \{\{A\}\}$, $B.dLabel = \{\{B\}\}$, $p.dLabel = \{\{A\}, \{B\}\}$, $q.dLabel = \{\{A\}\}$, i.e. J_3 delayed the propagation of $\{B\}$ from p to q. Suppose we have a single focusing environment (with identifier 1), namely $\{A\}$. Assume, all the $fLabels$ are set to $\{1\}$, except that of B and J_2 which are empty, and all the $fStatuses$ are empty - of course, this is a consistent labeling.*

Suppose that in this state we enable the assumption B in the context 1 and that we work with the tight $fStatus$ constraints. The incremental $fLabel$ propagation adds 1 to the $fLabel$ of B and J_2 and stops there. No change in the $fStatus$ is required at B, J_2 or p. If we would perform only the incremental addition of elements to the $fStatuses$, we could not notice that 1 must be added to the $fStatus$ of J_3 (from there on the $fStatus$ can be propagated incrementally)! Note that if we fail to add 1 to the $fStatus$ of J_3 and q, the consequences might be dramatic: we loose the dLabel completeness in the context 1.

There are several ways to deal with this situation:

- Use the relaxed $fStatus$ constraints during assumption enabling: If when enabling an assumption a in a context i, we add i to the $a.fLabel$ as well as to $a.fStatus$, this is like if we added a justification for a that has i in its $fLabel$. The $fLabel$ and $fStatus$ update can be performed incrementally like at justification addition. The drawback of this solution is that it marks a and some (or all, in the worst case) of its followers as unupdated in i, even when this is not the case;
- Keep the strong $fStatus$ constraints, but find all the nodes or justifications that block the propagation of an environment containing the just enabled assumption. i must be added to the $fStatus$ of these nodes and justifications, but this time the addition can be performed incrementally. We used this variant in the detailed description that follows.

2vRMS data structures

2vRMS instance :

assumptions, contradictions : store the set of assumptions, respectively, contradiction nodes, like in all the RMSs;

focusEnvironments : an ordered collection of environments. The index of the focusing environments in this collection is used as an identifier for that environment (like in the JTMSset);

obsoleteFocus : the (bit) set with the identifiers of the focus environments that are no longer in the focus of the PS, like in the JTMSset;

fActiveFocus : the (bit) set with the identifiers of those focusing environments that are consistent in the focus view and are not obsolete (like the $activeFocus$ of the JTMSset);

dActiveFocus : the (bit) set with the identifiers of those focusing environments that are not obsolete and are either consistent, or if they are inconsistent, the contradictory node(s) was (were) not yet queried, i.e. in other words, this set specifies those focusing environments that are not obsolete and are not superset of any nogood stored in the nogood database;

checkForSupportStack : a stack of nodes temporarily used during assumption retraction. The well founded support with respect to some focusing context must be checked for these nodes, like in the JTMSset;

sleepingConsumers : a collection of associations (i, n), where i is the identifier of a focusing context that is not in $fActiveFocus$ and n is a node that holds in that context and has attached consumers; like in the JTMSset.

envDB, nogoodDB : the environment and the nogood databases - like in the ATMS (see Appendix A.5). family;

checkStack : like in the ATMS family. It contains those incremental label updates that must be processed by the labeling algorithm (in the detailed view). This slot contains a collection of triples $(J, n, newEnvs)$ specifying the justifications where the addition of a set of environments $newEnvs$ to the $dLabel$ of the antecedent node n was not propagated to the consequent. (see also this slot in the ATMS);

reqJusts, reqNodes : during query answering this slots contain the justifications, respectively the nodes, that are needed in order to compute the$dLabel$ of the queried node;

2vRMS node : the *datum, justifications, consequences, contradiction, assumption, consumers* have the usual meaning like in any RMS (e.g. see the JTMS in Appendix A.2);

fLabel : like the JTMSset label, i.e. the (bit) set with the identifiers of the focusing contexts where the node is derivable;

dLabel : minimal and "consistent"[3] set of environments. It is updated only when a query is processed and the node is used to compute the query;

fStatus : a subset of the *fLabel*. Indicates the focusing contexts with respect to which the *dLabel* of the node might not be updated;

blockedLabel : as in the fATMS. A set of environments whose addition to the node's *dLabel* was not performed because they did not hold in the focus. A design choice is whether to update this field when the focus changes, or to wait until the node is needed for some query. We decided for the second variant in the following.

req : during the query answering this slot contains the set of focusing contexts with respect to which the *dLabel* completeness must be ensured in order to answer the query. It is always a subset of the $fStatus$;

2vRMS justification : except the usual *antecedents* and *consequence* it contains:

fLabel : as in the JTMSset, i.e. the intersection of the *fLabel*s of justification's antecedents;

wfSupport : as in the JTMSset, i.e. the (bit) set with the focusing contexts in which this justification is currently providing a well founded support to the consequence (it is a subset of *fLabel*);

fStatus : as discussed above, i.e. the (bit) set of focusing contexts with respect to which the consequent might not have updated label because: *(a)* some antecedents of this justification do not have updated labels, or because *(b)* this justification did not propagate some label updates;

delayedConstraints : as in the LazyATMS. A collection of unprocessed (d)label updates at the antecedents of this justification. It is a collection of triples $(J, n, newEnvs)$ with the same meaning as the entries in the 2vRMS's *checkStack* (see above);

[3]The consistency is ensured only with respect to the nogoods stored in the nogood database.

req : during query processing it contains the set of focusing contexts with respect to which the justification must propagate the label updates in order to ensure the *dLabel* completeness of the queried node.[4]

Basic Operations

The operations performed at the addition of a justification (Figure 4.4, 4.5, 4.6) have similarities with the ones performed in the JTMSset (for updating the $fLabels$) and in the LazyATMS (for updating the $fStatus$). The function *NewConstraint* is similar to the one of the LazyATMS: it pushes the update either on the *tms.checkStack* for immediate processing in case the justification involved is required for a query, or, otherwise, it delays the update by storing it in the *delayedConstraints* of the justification. In addition, here it is considered that a justification could be required for only some subset of focusing contexts (i.e. the ones mentioned in the set $J.req$) and consequently only the updates relevant to those contexts are immediately processed.

The procedure *SetFViewBelief* (Figure 4.5) performs the incremental propagation of the $fLabel$s and $fStatus$es. The procedure also updates the well founded support, signals the contradictions and activates the consumers if necessary. Line (3), respectively (13) and (14), of $SetFViewBelief$ add to the $fStatus$ the incremental set propagated from the ancestors, while the lines (4) and (15), eventually check the local $fStatus$. Since this test (see *ChkNodeStatus*, *ChkJustStatus* in Figure 4.6) can be relatively expensive, it is avoided to perform it when not necessary: the test is done only for the contexts newly added to the $fLabel$ that are not already in the $fStatus$, if any.

Note that if one chooses to work with the relaxed $fStatus$ constraints during the assumption enabling, then in all the calls of $SetFViewBelief$ we have $newFLabel \subseteq newFStatus$. In such a case no check for the local $fStatus$ is needed and the lines (4), (14) and (15) can be removed from $SetFViewBelief$ since always $newFLabel - n.fStatus$ and $newFL - J.fStatus$ will be empty.

The function that adds a new environment to the focus (Figure 4.7) is almost identical with the one of the JTMSset (Figure A.10). The only difference appears at the lines (10-13) because here the $fStatus$ must be updated

[4]The filed is analogous to the same one of the LazyATMS, but there it was a simple boolean. Here the same information is stored for each focusing context individually, thus the field is a set.

as well. Lines (12) and (13) are present only if we work with the strong $fStatus$ constraints during the assumption enabling. We can dispense with them if we work with the relaxed variant, in which case the third parameter in the call to $SetFViewBelief$ must be equal with the second one in line (10).

The function $WakeUpConsumers$ is identical with the one of the JTMSset, and we do not reproduce its code once more (see Figure A.10). The functions $RemoveBelief$ and $CheckAdditionalSupport$ (Figure 4.8) are similar to the ones of the JTMSset (Figure A.11) with slight differences introduced by the management of the $fStatus$.

The function that queries a node label (Figure 4.9) has similarities with the one of the LazyATMS (Figure A.15). This function is more complicated here because the nodes and justifications can be required for different contexts independently.

The functions $VerifyConstraints$, $NewEnvsForConseq$ and $Nogood$, relevant for the environment propagation in the detailed view, are identical with the ones of the basic ATMS (see Figure A.12 and A.13). The function $WakeUpReqConstraints$ reconsiders the delayed environment propagation within the required subnetwork. The $dLabel$ propagation performed by $VerifyConstraints$, is restricted to the required nodes, justifications and contexts (see in this respect the functions $NewConstraint$ - in Figure 4.4, and $SetBelief$ - in Figure 4.11).

The procedure $SetBelief$ which is almost identical with the one of the fATMS, also updates the active focus for the detailed view. Note that, although the $dActiveFocus$ may change during the environment propagation performed while querying a contradictory node, the contents of the req fields does not change during environment propagation! The function $FindReqNet$ uses the $dActiveFocus$ at the beginning of the query and finishes to mark the "required" net before the environment propagation is done. This justifies the correctness of the fact that after the environment propagation is finished we can subtract the req sets from the $fStatus$ (the functions responsible for the environment propagation, i.e. $NewConstraint$ and $SetBelief$ work with the individual "req" fields of the nodes and the justifications and not with the global $dActiveFocus$).

The procedure $FindNogoods$ starts the search for (minimal) nogoods in the inconsistent contexts for which no contradiction was queried yet. The PS has the liberty to call this procedure whenever it evaluates this operation as cost effective.

Procedure AddJustification (J, tms)
J is a 2VRMS justification

(1) Add J to the slots *consequences* of each node from $J.antecedents$ and to the slot $justifications$ of the node $J.consequent$;

(2) set $J.fLabel := J.fStatus := \bigcap_{n \in J.antecedents} n.fLabel$;

(3) set $J.wfSupport := J.fLabel - J.consequent.fLabel$;

(4) set $J.req$ to the empty set;

(5) **call** $NewConstraint(J, \texttt{nil}, \{\{\}\}, tms)$;

(6) **call** $SetFViewBelief(J.consequent, J.wfSupport,$
$J.fStatus - J.conesquent.fStatus, tms)$;

Procedure NewConstraint $(J, n, envs, tms)$
$J.req$ is a non-empty set only during query processing. The addition of the environments $envs$ at the $dLabel$ of n was not propagated through J. Splits the set of environments $envs$ in two disjoint subsets: the updates that hold in the focus contexts mentioned in $J.req$ are pushed to $tms.checkStack$ for immediate processing; the rest of updates are delayed and stored in $J.delayedConstraints$.

(1) **if** $\exists a \in J.antecedents$ s.t. $a.dLabel$ is empty **then return** ;

(2) let $fEnvs$ be the subset of $envs$ containing the environments that hold in the focusing contexts $J.req$; let $dEnvs := envs - fEnvs$;

(3) **if** $fEnvs \neq \{\}$ **then** push $(J, n, fEnvs)$ to $tms.checkStack$;

(4) **if** $dEnvs \neq \{\}$ **then** push $(J, n, dEnvs)$ to $J.delayedConstraints$;

Figure 4.4: Adding a justification in the 2VRMS

Procedure SetFViewBelief (n, $newFLabel$, $newFStatus$, tms)
$newFLabel$ is a set of *new* focus environments that have to be added to n's fLabel; $newFStatus$ is a set of *new* focus environments that have to be added to n's fStatus. Checks for contradictions, eventually activates consumers, and propagates the incremental change further.

(1) **if** $(newFLabel = \{\} \wedge newFStatus = \{\}) \vee n$ is premise **then return** ;
(2) $n.fLabel := n.fLabel \cup newFLabel$;
(3) $n.fStatus := n.fStatus \cup newFStatus$;
(4) $newFStatus := newFStatus \cup$
$ChkNodeStatus(n, newFLabel - n.fStatus, tms)$;
(5) **if** $n.contradiction$ is true **then**
(6) **if** $newFLabel \cap tms.fActiveFocus \neq \{\}$ **then**
(7) **call** $SignalContradiction(n, newFLabel \cap tms.fActiveFocus)$;
(8) $tms.fActiveFocus := tms.fActiveFocus - newFLabel$;
(9) **return** ;
(10) **for** $J \in n.consequences$ **do:**
;; Propagate the update further
(11) $newFL := newFLabel \cap \bigcap_{a \in J.antecedents} a.fLabel$;
(12) $J.fLabel := J.fLabel \cup newFL$;
(13) $J.fStatus := J.fStatus \cup (newFStatus \cap J.fLabel)$;
(14) $J.fStatus := J.fStatus \cup propagS(J, newFL - J.fStatus, tms)$;
(15) **call** $ChkJustStatus(J, newFL - J.fStatus, tms)$;
(16) $newFL := newFL - J.consequent.fLabel$;
(17) $J.wfSupport := J.wfSupport \cup newFL$;
(18) **call** $SetFViewBelief(J.consequent, newFL,$
$J.fStatus - J.conesquent.fStatus, tms)$;
endfor
(19) **if** $newFLabel \cap tms.fActiveFocus \neq \{\}$ **then**
(20) activate all consumers from $n.consumers$; empty $n.consumers$;
else
(21) **if** $n.consumers$ is not empty **then**
(22) add (i, n) to $tms.sleepingConsumers$ for each $i \in newFLabel$;

Figure 4.5: Propagating the updates in the focus view in the 2VRMS

Function ChkNodeStatus $(n, fSet, tms)$
checks the "local" $fStatus$ of n w.r.t. the focusing contexts named by $fSet$.

(1) let $retValue := \{\}$;

(2) **for** $i \in fSet$ **do:**

  (3) **if** $\exists env \in n.blockedLabel$ s.t. env is a subset of the focusing environment i **then**

    (4) add i to $retValue$;

  endfor

(5) $n.fStatus := n.fStatus \cup retValue$; **return** $retValue$;

Function ChkJustStatus $(J, fSet, tms)$
checks the "local" $fStatus$ of J w.r.t. the focusing contexts named by $fSet$.

(1) let $retValue := \{\}$;

(2) **for** $i \in fSet$ **do:**

  (3) **if** $\exists (J, n, envs) \in J.delayedConstraints$ s.t. $\exists env \in envs$ s.t. env is a subset of the focusing environment i **then**

    (4) add i to $retValue$;

  endfor

(5) $J.fStatus := J.fStatus \cup retValue$; **return** $retValue$;

Function propagS $(n, fSet, tms)$
returns the $fStatus$ propagated from n's justifications (for the contexts $fSet$).

(1) **if** $fSet = \{\}$ **then return** $\{\}$;

(2) **return** $\left(\bigcup_{J \in n.justifications} (J.fStatus \cap fSet)\right)$;

Function propagS $(J, fSet, tms)$
returns the $fStatus$ propagated from J's antecedents (for the contexts $fSet$).

(1) **if** $fSet = \{\}$ **then return** $\{\}$;

(2) **return** $(\bigcup_{n \in J.antecedents} (n.fStatus \cap fSet))$;

Figure 4.6: Checking the "local" and the "propagated" fStatus in the 2vRMS.

Procedure RemoveFromFocus $(fEnv, tms)$
$fEnv$ is an old focusing environment of tms.

(1) let i be the index of $fEnv$ in $tms.focusEnvironments$;

(2) remove i from $tms.fActiveFocus$ and from $tms.dActiveFocus$;

(3) add i to $tms.obsoleteFocus$;

Procedure AddToFocus $(fEnv, tms)$
$fEnv$ is a set of assumptions defining a new focusing environment for tms;

(1) **if** $tms.obsoleteFocus$ is empty **then**

(2) let i be the index of the next free entry in $tms.focusingEnvironments$;

(3) initialize $oldFEnv$ with the empty set;

else

(4) let $oldFEnv$ be an arbitrary element of $tms.focusingEnvironments$ whose index is in $tms.obsoleteFocus$; let i be the index of $oldFEnv$ in $tms.focusingEnvironments$;

(5) $assToRetract := oldFEnv - fEnv$; $assToEnable := fEnv - oldFEnv$;

(6) store $fEnv$ in $tms.focusEnvironments$ at the index i;

(7) remove i from $tms.obsoleteFocus$;

(8) add i to $tms.fActiveFocus$ and $tms.dActiveFocus$;

(9) **call** $RemoveBelief(n, i, tms)$ for each assumption $n \in assToRetract$;

(10) **call** $SetFViewBelief(n, \{i\} - n.fLabel, \{\}, tms)$
for each assumption $n \in assToEnable$;

(11) **call** $CheckAdditionalSupport(i, tms)$;

(12) **call** $SetFViewBelief(n, \{\}, \{i\} - n.fStatus, tms)$ for each node n s.t. $\exists env \in n.blockedLabel, env \cap assToEnable \neq \{\}, env \subseteq fEnv$;

(13) **call** $SetFViewBelief(J.consequent, \{\},$
$ChkJustStatus(J, \{i\}, tms) - J.consequent.fStatus, \quad tms)$
for each J s.t. $\exists env \in J.delayedConstraints, env \cap assToEnable \neq \{\}$;

(14) **if** $i \in tms.fActiveFocus$ **then call** $WakeUpConsumers(i, tms)$;

Figure 4.7: Changing the focus in the 2VRMS.

Procedure RemoveBelief (n, i, tms)
n is a node that must be retracted in the context i.

(1) remove i from $n.fLabel$ and $n.fStatus$;
push n to $tms.checkForSupportStack$;

(2) **for** $J \in n.consequences$ **do:**

(3) remove i from $J.fLabel$ and $J.fStatus$;

(4) **if** $i \in J.wfSupport$ **then**

(5) remove i from $J.wfSupport$;

(6) **call** $RemoveBelief(J.consequent, i, tms)$;

endfor

Procedure CheckAdditionalSupport (i, tms)
looks for support in context i.

(1) **while** $tms.checkForSupportStack$ is not empty **do:**

(2) pop node n from $tms.checkForSupportStack$;

(3) **for** $J \in n.justifications$ **do:**

(4) **if** $i \in n.fLabel$ **then break** the for-loop;

(5) **if** $i \in J.fLabel$ **then**

(6) add i to $J.wfSupport$;

(7) **call** $SetFViewBelief(n, \{i\}, propagS(n, \{i\}, tms), tms)$;

endfor

endwhile

Figure 4.8: Removing and checking for support in the 2vRMS.

Function QuerryLabel (n, tms)
incrementally updates the dLabel of n with respect to the $dActiveFocus$.

(1) **call** $FindReqNet(n, n.fStatus \cap tms.dActiveFocus, tms)$;

(2) **call** $WakeUpReqConstraints(tms)$; **call** $VerifyConstraints(tms)$;

(3) $J.fStatus := J.fStatus - J.req$; $J.req := \{\}$,
for all $J \in tms.reqJusts$; empty $tms.reqJusts$;

(4) $n.fStatus := n.fStatus - n.req$; $n.req := \{\}$,
for all $n \in tms.reqNodes$; empty $tms.reqNodes$;

(5) **return** $n.dLabel$;

Procedure FindReqNet $(n, relevCtxs, tms)$
steps back in the justification network and determines the justifications and nodes that are required to compute the dLabel of n in the contexts $relevCtxs$. The relevant justifications and nodes are stored in the $tms.reqJusts$, respectively $tms.reqNodes$. The slots req of the required nodes and justifications contain subsets of $relevCtxs$ stating individually with respect to which contexts should the delayed constraints be reprocessed.

(1) **if** $relevCtxs$ is empty **then return** ;

(2) **if** $n.req$ is empty **then** push n to $tms.reqNodes$;

;; this ensured that each node appears at most once in $tms.reqNodes$;

(3) $n.req := n.req \cup relevCtxs$;

(4) **for** each $J \in n.justifications$ **do:**

(5) let $newRelevCtxForJ := (J.fStatus \cap relevCtxs) - J.req$;

;; the update of the req fields is also performed incrementally;

(6) **if** $newRelevCtxForJ \neq \{\}$ **then**

(7) **if** $J.req = \{\}$ **then** push J to $tms.reqJusts$;

(8) $J.req := J.req \cup newRelevCtxForJ$;

(9) $FindReqNet(a, (a.fStatus \cap newRelevCtxForJ) - a.req, tms)$
for $a \in J.antecedents$;

endfor

Figure 4.9: Querying a node label in the 2vRMS.

Procedure WakeUpReqConstraints (tms)
For each required justification the collection of unprocessed dLabel updates $J.delayedConstraints$ relevant to $J.req$ is moved to the $tms.checkStack$. For each required node the collection of blocked environments $n.blockedLabel$ implied by $n.req$ is moved to n's dLabel (and is further propagated). The consistency of the environments that are restarted is checked here.

(1) **for** $n \in tms.reqNodes$ **do:**
  (2) remove any inconsistent environment from $n.blockedLabel$;
  (3) $fEnvs := \{env \in n.blockedLabel \mid env$ holds in $n.req\}$;
  (4) $n.blockedLabel := n.blockedLabel - fEnvs$;
  (5) **call** $SetBelief(n, fEnvs, tms)$;
endfor

(6) **for** $J \in tms.reqJusts$ **do:**
  (7) remove any superset of a nogood from $envs$,
    where $(J, n', envs) \in J.delayedConstraints$;
  (8) move all entries from $J.delayedConstraints$ to $tmpStack$;
  (9) **call** $NewConstraint(J, n', envs, tms)$
    for all $(J, n', envs) \in tmsStack$
endfor

Figure 4.10: Reconsidering delayed updates in the detailed view of the 2vRMS.

Procedure SetBelief $(n, newEnvs, tms)$
adds the elements of $newEnvs$ that hold in the contexts $n.req$ to n's $dLabel$, respectively to n's blocked label; creates more triples $(J, n, newEnvs)$ for the incremental dLabel updates at the outgoing justifications.

(1) remove from $n.dLabel, n.blockedLabel$ the supersets of any member of $newEnvs$;
(2) remove from $newEnvs$ the supersets of any member of $n.blockedLabel$;
(3) let $fEnvs := \{e \in newEnvs \mid e \text{ holds in } n.req\}$;
(4) $n.blockedLabel := n.blockedLabel \cup (newEnvs - fEnvs)$;
(5) **if** $fEnvs$ is empty **then return** ;
(6) **if** $n.contradiction$ is true **then**
  (7) **call** $Nogood(e, tms)$ for each $e \in fEnvs$;
  (8) **for** $i \in n.fLabel \cap tms.dActiveFocus$ **do:**
    (9) **if** $\exists e \in fEnvs$ s.t. e is included in the focusing environment i **then**
      (10) $tms.dActiveFocus := tms.dActiveFocus - \{i\}$;
    endfor
  (11) **return** ;
(12) add the elements of $fEnvs$ to $n.dLabel$;
(13) call $NewConstraint(J, n, fEnvs, tms)$ for each $J \in n.consequences$;

Figure 4.11: $dLabel$ update in the 2vRMS.

Procedure FindNogoods (tms)
checks and ensures the consistency of the detailed view and finds the (minimal) nogoods for the set of inconsistent contexts from $dActiveFocus$.

(1) **call** $QueryLabel(n, tms)$, for all $n \in tms.contradictions$ s.t. $n.fStatus \cap tms.dActiveFocus \neq \{\}$;

Figure 4.12: Searching for the nogoods in the 2vRMS.

4.2.3 Advanced control techniques in the 2vRMS

The 2vRMS can ensure the label completeness with respect to some *focusing* contexts *on request*, while offering cheaper means for deciding the membership of the nodes to the focusing contexts.

In this section we explore alternatives for further controlling the labeling tasks performed in the 2vRMS.

After a certain node is queried, and if the detailed view is currently in a consistent state with respect to the focus then the 2vRMS behaves towards the PSlike a focusing ATMS. This means that: the dLabel will be minimal, sound, complete and consistent with respect to the active focus. It is questionable if even this degree of completeness is sometimes not too strong; namely, there can be in principle a large number of minimal supports for a node even in one single context. Finding all of them may be both: *(a)* very expensive, and *(b)* not relevant.

These arguments motivate our interest in a strategy that computes the supports with respect to the focus in an *incremental* manner. This way, the PS will be given the possibility to control also the *number* of (minimal) environments per context computed in response to a node query.

Computing one minimal size minimal environment per context

In the following we present the changes that have to be done in the 2vRMS such as to compute always one (more) minimal environment for each focusing context for a queried node. Moreover, it may also be of interest that the minimal environment also has the smallest size across the not yet computed supports. If several queries are successively posed to a node then eventually all the minimal environments supporting the node in the focus may be computed (starting with the ones with the smallest size), but this decision is up to the PS.

In order to implement this control strategy we add two other slots to the *tms* data structure: the *qResidualFocus* and the *qOriginalFocus*.

The $tms.qResidualFocus$ holds during query processing the set of focusing contexts for which the (next) minimal environment was not yet computed. Its value changes dynamically during query processing and controls the environment propagation, together with the local contents of the "req" fields stored in the relevant nodes and justifications.

The $qInitialFocus$ stores the value of the initial value of the *qResidual-*

Focus. At the beginning of the query for a node n the *tms.qResidualFocus* and the *tms.qInitialFocus* are initialized with $n.fStatus \cap tms.dActiveFocus$.

The functions *FindReqNet* and *WakeUpReqConstraints* do not have to be changed. However, the functions *NewConstraint* (Figure 4.4) and *SetBelief* (Figure 4.11) must take the set *tms.qResidualFocus* into account: Instead of filtering the environments that hold in the contexts *J.req* (line (2) of *NewConstraint*), respectively *n.req* (line (3) of *SetBelief*), they should filter the environments holding in: $J.req \cap tms.qResidualFocus$, respectively in $n.req \cap tms.qResidualFocus$.

Additionally, $SetBelief$ should remove from *qResidualFocus* the contexts for which some environments were propagated to the queried node. Each time when the $qResidualFocus$ is reduced, the stack with incremental (d)label updates (i.e. *tms.checkStack*) can be filtered by removing the sets of environments that are no longer in the *qResidualFocus*. The tuples $(J, n, envs)$ that ran out of the *qResidualFocus* can be again delayed by storing them in the slots *J.delayedConstraints*.

The changes performed so far ensure that the environment propagation in a queried context stops as fast as a first environment for the queried node is computed. In the following we describe: *(a)* how to ensure that the propagated environment is minimal and has a smallest size; and *(b)* how to correctly update the $fStatus$ fields.

In order to ensure the minimality of the computed environments the function *VerifyConstraints* (Figure A.12) has to be changed such as to perform a *best first* propagation. At each step of the cycle *VerifyConstraints* should pick up only one environment with a smallest size to propagate it further.

One still cannot be sure that the first environment propagated in this way to the queried node is a minimal environment. In order to ensure that, $VerifyConstraints$ should continue the best-first propagation until it propagates all the environments smaller then the best new environment computed for the queried node.

Note, however, that in the worst case computing one (more) *minimal* environment at a query is as hard as computing all the minimal environments. The requirement of minimality alone makes the problem hard. A single, not necessarily minimal, support is easy to find, e.g. just following the well-founded supports in the queried contexts. Thus, the requirement of minimality should be regarded as a requirement that could be traded against efficiency in query processing.

Because the environment propagation can end without performing all the

updates that are relevant for the $qInitialFocus$, it is no longer correct to remove the fields req from the $fStatus$ at the end of a query - as was done in the lines (3-4) of $QueryLabel$ (Figure 4.9). Of course, one could start and completely recompute the $fStatus$ for the whole subnetwork relevant for the query. However, even this computation can be optimized. If, for instance, there is only one environment computed by the query even without the control strategy described here, then the use of this control strategy would just add more burden because of the recomputation of the $fStatus$. This does not sound very intelligent.

During the query evaluation, the nodes and justifications at which the procedures $NewConstraint$ and $SetBelief$ block (delay) updates which are in the $qInitialFocus$ should be stored for later inspection. At the end of the environment propagation the $fStatus$ can be updated as follows:

1. for all the relevant network the req sets are subtracted from the $fStatus$ sets;
2. the nodes and justifications with delayed updates, stored previously, have to be checked for the local $fStatus$ that has to be incrementally propagated to the followers.

4.2.4 Experimental results

We have performed a series of tests with two prototypical implementations of the 2vRMS in diagnostic problems. The results are depicted in Table 4.1 and 4.3.

Table 4.1 shows the results of running the candidate elaboration for several input-output settings in the system depicted in Figure 4.13. These tests were obtained using a tiny diagnostic engine developed during my research period at the Technical University of Cluj-Napoca.

In all cases described in Table 4.1 the diagnostic engine focused only on the first most probable candidate from the preferred ones (cf. Section 4.3), but several diagnoses were added to focus if they were equally probable. The table compares the performance of using the focusing ATMS vs. the 2vRMS. Column 2 compares the total number of environments, i.e. the sum of the environment database size and of the nogood database size at the end of diagnosis. Column 3 compares the average length of the labels, computed as the total label ($dLabel$) length divided by the total number

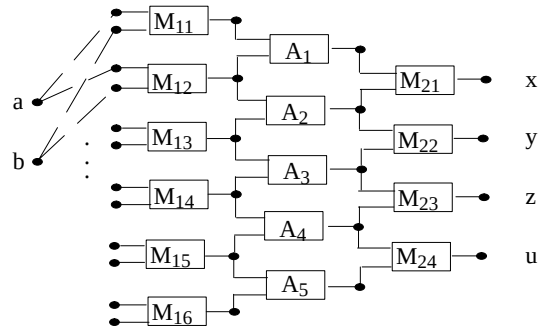

Figure 4.13: A pyramid of multipliers and adders. The inputs a and b are linked to the left respectively the right input of the multipliers M_{1i}.

Test pattern	Environments			Avg. label length			Time in seconds		
	◯	△	ratio	◯	△	ratio	◯	△	ratio
1.	382	166	2.3	1.91	1.26	1.6	8.18	2.36	3.5
2.	263	42	6.3	3	0.8	3.8	4.23	0.5	8.5
3.	206	85	2.4	1.69	1.18	1.4	3	1	3
4.	342	67	5.1	2.4	0.85	2.8	6.8	0.77	8.8

Table 4.1: ◯ focusing ATMS △ 2vRMS.

of RMS nodes. Column 4 compares the total amount of time[5] spent within the RMS during candidate elaboration. The focusing ATMS and the 2vRMS discovered the same nogoods and received the same sequence of justifications in these problems. The test patterns used are given in Table 4.2. The 2vRMS prototype used for these tests did not embed all the ideas presented here and the implementation differed at several places from the description of this chapter.

More convincing results were observed for a series of more complex and practically relevant problems. These tests were performed using a significantly more elaborate diagnostic engine (MDS) developed at the Daimler-Benz Research Center in Berlin. The 2vRMS implementation used by MDS

[5]The tests were run on a PC286 under DOS. The 2vRMS prototype used in these tests was implemented in C++.

	Test pattern						Diagnosis
	a	b	x	y	z	u	
1:	2	3	144	36	27	108	$\{r(M_{14}), r(A_3)\}$
2:	2	3	144	108	81	108	$\{r(M_{14})\}$
3:	2	3	12	12	12	12	$\{s1(A_2), s1(A_4)\}$
4:	2	3	84	144	96	64	$\{s1(M_{11}), l(M_{15})\}$

Table 4.2: The test problems for Table 4.1. Every component had 7 modes of behavior: *ok* / probability 0.75 - the correct behavior; $s1$ / 0.1 - output stuck at 1; $s0$ / 0.05 - output stuck at 0; l / 0.04 - output equal with the left input; r / 0.04 - output equal with the right input; s / 0.018 - output equal with the correct result shifted with one bit to the left; u / 0.002 - unknown failure.

was also closer to the description given here.

In addition to the tasks of candidate elaboration and probe selection, commonly supported by most of the nowadays diagnostic engines, MDS also integrates test-pattern proposal in the diagnostic cycle. In addition to the behavior and structure descriptions, MDS makes use of descriptions of the *observable* and *controllable* parameters, descriptions of the situations in which an external agent is *allowed* to perform observation and control actions, as well as *situation-dependent costs* associated with these actions. Without going into details that go beyond the scope of this thesis, one could characterize the test-pattern proposal of MDS as a heuristic guided search in the space of possible input settings for a certain number of tests which provide a "good" balance between the information gain and the cost of performing the test (the cost include the cost of performing the control actions that change the inputs and the cost of performing the measurement action). Interesting to note here is the fact that this task is realized without computing detailed labels in the 2vRMS, i.e. only the information provided by the $fLabels$ is necessary.

The candidate elaboration of MDS uses the focusing strategy described later in this thesis, i.e. it focuses on a few preferred candidates with highest priority (cf. Section 4.3). Prediction is realized as constraint propagation, not necessarily for finite domains.

In addition to the on-line computation of diagnoses, MDS can be used in an off-line mode to compile diagnostic (decision) trees for a specified set of

considered faults. For this purpose the test-pattern proposal and the candidate testing are run recursively, candidate testing being performed for all of the possible outcomes of the best test proposed previously, until no discriminating test is found. During the diagnostic tree construction no candidate generation is done, i.e. the faults considered are only the ones that were initially specified, for instance, all the single faults. The test-pattern proposal is like during on-line diagnosis.

The test results presented here were obtained on an electrical system resembling the one depicted in Figure 4.14 which contains part of the electrical system for an ABS. The system contains 72 components such as: power supplies, wires, bulbs, resistors, plug connectors, diodes, switches, relays, and an electronic control unit. The faults considered were: broken wires, bulbs or connectors, shorted to ground wires, stuck to open or closed switches and relays, etc. A combination of qualitative, quantitative and interval-based modeling was used.[6] Control actions, with differing costs, were associated with switches, plug connectors, and some controllable settings of the electronic control units. Observation actions are, for instance, visual inspection, for instance for bulbs, or, under certain security conditions and with additional costs, voltage and resistance measurements at certain accessible ports.

Table 4.3 compares the performance of using the focusing ATMS vs. the 2vRMS. The second column compares the total number of environments defined in the RMS, i.e. the sum of the size of the environment and the nogood database. The third column compares the total label (dLabel) length (the sum of the $dLabel$ length over all of the nodes). The fourth column compares the time required to solve the problems.[7]

The problems CE_1-CE_4 represent candidate elaboration problems, the problems CD_1-CD_4 represent candidate discrimination (test-pattern proposal) problems. A pair candidate elaboration - candidate discrimination is performed at each diagnostic cycle in MDS. The problems CD_i were performed in the situation resulting after running CE_i; the problems CE_i $(i > 1)$ added the result of the best test proposed at the step CD_{i-1}.

T_1, T_2 were problems requiring the generation of a diagnostic tree for an initial set of 10, respectively 20, single faults. At the end of T_2, for

[6] The interval-based reasoning takes advantage of the "shadowing" techniques (cf. [Gol91], [Ham91a]), which use, however, only the focus view of the 2vRMS. The shadowing technique is also extended to inhibit the propagation of approximately equal values, where the approximately equal relation can be defined in a qualitative way.

[7] MDS is implemented in Smalltalk. The tests were run on a Sparc20.

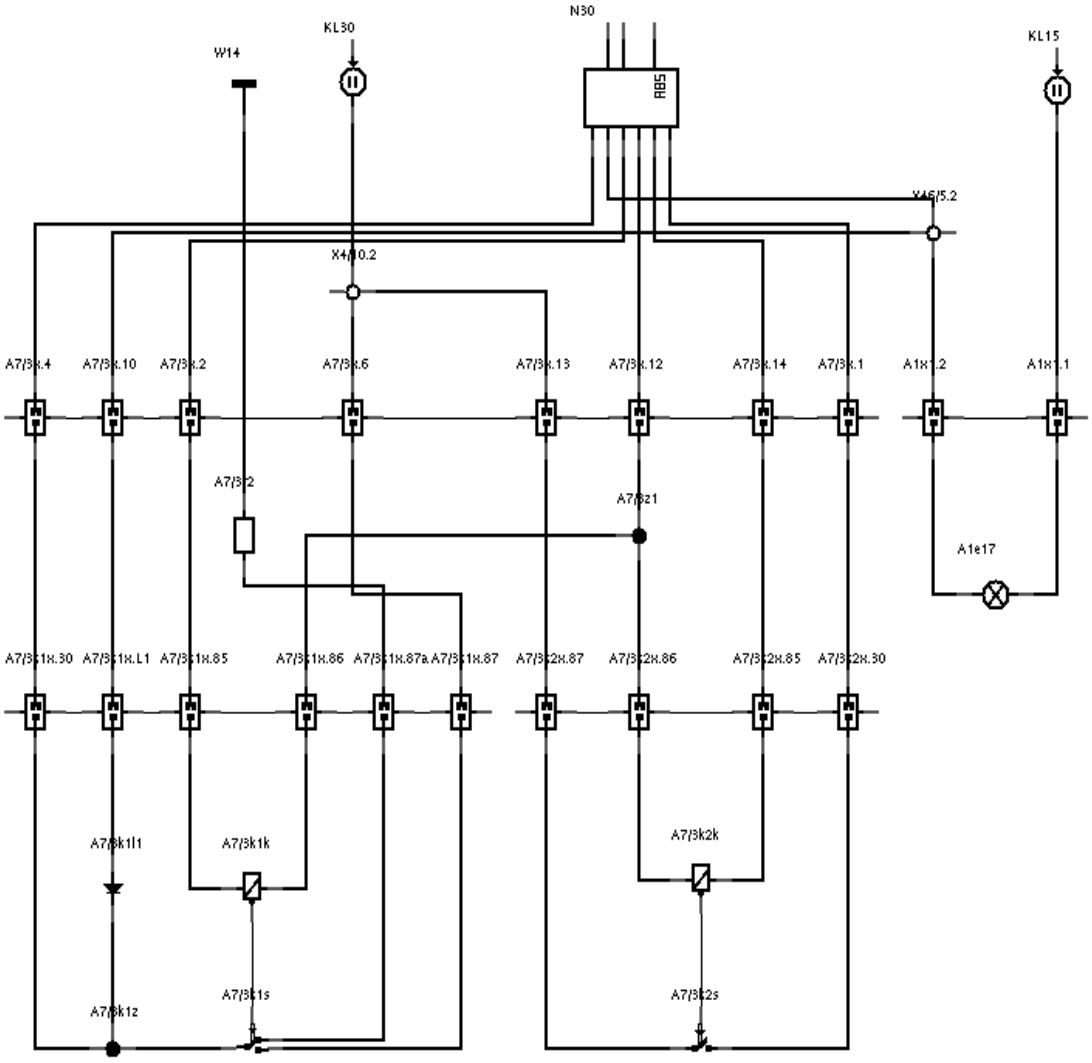

Figure 4.14: A part of the electrical system of an ABS.

instance, both the focusing ATMS and the 2vRMS discovered 77 nogoods, and the dependency networks contained 4293 justifications and 1495 nodes (i.e. this meant an average label length of 14.5 in the focusing ATMS, respectively 1.4 in the 2vRMS).

The results given here used a variant of the 2vRMS that computed all

Problem	Environments			Total label length			Time in seconds		
	◯	△	ratio	◯	△	ratio	◯	△	ratio
CE1	953	198	4.8	2960	252	11.7	28	5	5.6
CD1	985	198	5.3	4142	252	16.4	42	7	6
CE2	1379	256	5.4	4923	313	15.7	48	4	12
CD2	1514	256	5.9	5553	313	17.7	49	7	7
CE3	2164	490	4.4	6672	761	8.8	92	14	6.6
CD3	2222	490	4.5	7713	761	10.1	83	7	11.8
CE4	2620	701	3.7	8134	1162	7	64	12	5.3
CD4	2649	701	3.8	8299	1162	7.1	30	6	5
T1	2627	680	3.9	11105	1564	7.1	900	140	6.4
T2	5879	1243	4.7	21701	2140	10.1	3900	440	8.9

Table 4.3: ◯ focusing ATMS △ 2vRMS.

the minimal nogoods for an inconsistent context (i.e. without the control techniques described in 4.2.3), used the relaxed $fStatus$ constraints at assumption retraction and the tight ones at assumption enabling. The results reported here for the 2vRMS did not vary significantly in the above test problems when these control parameters were changed.

The 2vRMS achieved up to 10 times reductions of the time and memory consumed by the RMS in diagnostic tasks when compared to the fATMS.

The temporal representation used by MDS at the time the tests reported here were performed was a simplified form of the one required by the finite-state machine approach (cf. Chapter 5). The prediction sharing across time (or input settings) was an essential feature in order to obtain acceptable response times for the test-pattern proposal.

4.3 Efficient candidate generation

4.3.1 Introduction

The candidate generator receives knowledge about the different components and their possible modes of behavior. Additionally the candidate generator receives knowledge about the conflicts. Its main task is to propose a certain number of candidates that do not include any of the known conflicts, where by "candidate" we understand a set of mode assignments such that each component is assigned exactly one mode of behavior. Another desired feature refers to the incremental way of operation, since the set of discovered conflicts grows during diagnosis.

Usually in diagnosis there are very many possible candidates that are consistent with the observations, especially in the early diagnostic steps when there is not enough evidence about the system. It is not feasible to consider all of the possible candidates in parallel. A currently followed approach restricts the attention to a small set of candidates currently in the focus. A smaller focus requires smaller computational costs because the prediction of values and the RMS labeling can be performed only with respect to this focus. However the focus cannot be arbitrarily small: the elaboration of several alternatives is essential for guiding the further information gathering process. Current techniques of candidate generation attempt to generate a few of the most *plausible / critical* candidates from the ones that are possible.

We use a diagnostic framework where each component can be characterized with several modes of behavior. Each candidate chooses exactly one mode of behavior for each component. The plausibility / criticality information that we use to control the selection of the focus candidates is encoded by two relations: *preference* and *priority*. The algorithms we discuss can generate in an incremental way a few (say k) of the most preferred candidates having the highest priority.

The preference relation is as defined by Dressler and Struss (cf. [DS92]): a preference order among the modes of each component is used to induce a preference order among candidates. The partial order defined by the candidate preference imposes a lattice structure on the candidate space. The preferred diagnoses define a lower bound for the consistent candidates in the preference lattice. Our algorithms manipulate such lower bounds.

The preference alone can only encode knowledge about the plausibility of the modes of a single component, e.g. it can encode knowledge saying that:

"a wire is most probably correct, but if it is defect then it is more likely to be broken than shorted to ground". The focus selection heuristic based on the preference alone is in many cases not sharp enough: for large systems *the set of preferred candidates is very large*. In order to gain more control over the selection of the focusing candidates we define an additional *priority* relation among mode assignments. The priority relation increases the expressiveness of the control knowledge. Knowledge expressing that, for instance, *"the bulbs break more often than the switches, which, in turn, break more often than the wires"* cannot be expressed using the preference, but can be encoded using the priority.

The proposed algorithms generate a few of the most preferred candidates having the highest priority, *without computing all the preferred diagnoses and applying a filter afterwards*. Dressler and Struss used a default logic framework to characterize the preferred diagnoses and a non-monotonic ATMS to compute them (cf. [DS92]). Our algorithms are conceptually equivalent with their "incremental construction of the preferred diagnoses" from [DS94], but do not use the non-monotonic ATMS.

We analyze the properties the candidate generation algorithms with respect to the framework of propositional logic. The relations between the possible behavioral modes of a component can be abstracted to, what we call, *choice sets*, i.e. purely positive propositional clauses with additional control information encoded in the preference.[8] We used such choice sets instead of components and modes of behavior, because the relevance of the algorithms exceeds the domain of diagnosis. Namely, we argue that they provide *complementary* functions to the RMS in a general RMS-based problem-solver.

The 2vRMS (as well as the ATMS family and the JTMS) can perform satisfiability checking and model construction for Horn theories. The candidate generator can perform satisfiability checking and model construction for non-Horn theories consisting of purely positive and purely negative clauses. The combination 2vRMS (ATMS or JTMS) and the candidate generator can be used as a (heuristically guided) satisfiability checker and (multiple) model builder for *arbitrary* propositional theories, thus removing the expressiveness limitations of the RMS and of the candidate generator alone.

[8] In default logic this was encoded using a set of default rules.

4.3.2 Preferred candidates: basic definitions and properties

Definition 4.3.1 *A* choice space *is a tuple* $CS \equiv (L, Choices, Confl, \leq_{pri})$ *where:*

- L *is a finite set of symbols (the assumptions);*
- $Choices$ *is a set of* choice sets*:* $Choices = \{C_1, \ldots, C_N\}$*, where* $C_i \subseteq L$*. The elements of each choice set* $C_i = \{a_{i1}, \ldots a_{in_i}\}$ *are ordered according to a* total and strict *order (the preference among the elements of a choice set), i.e.* $a_{i1} <_{C_i} a_{i2} <_{C_i} \ldots <_{C_i} a_{in_i}$*;*
- $Confl$*, the set of conflicts, is a set of sets of symbols from* L*, i.e.* $Confl \subseteq 2^L$*;*
- $\leq_{pri}$*, the* priority order*, is a (partial) order relation on the sets of symbols, i.e.* $\leq_{pri} \subseteq 2^L \times 2^L$ *which agrees with the preference among the candidates as defined subsequently.*

The semantics of a choice set is that of a logical disjunction among the elements, i.e. at least one of them must be true. The semantics of a conflict is that of disjunction over the negation of the symbols, i.e. at least one assumption from a conflict must be false. In many cases the elements of a choice set are also mutually exclusive, case in which the set of conflicts contains also the combinations of the mutually inconsistent elements. Note that we did not require the choice sets to be disjoint, i.e. an assumption $a \in L$ can appear in several choice sets. This represents a generalization compared to the way the preference was defined in diagnosis (cf. [DS92]).

In the case of diagnosis each choice set corresponds to the possible behavioral modes that each component can have. In this case the choice sets are disjoint and the elements of a choice set are usually mutually exclusive. The conflicts correspond in diagnosis to the conflicts among the mode assignments.

Definition 4.3.2 (candidate) *A* candidate *for a choice space* $(L, Choices, Confl, \leq_{pri})$ *selects exactly one element from each choice set of* $Choices$*, i.e.* $A = \{(C_i, a_i) \mid C_i \in Choices\}$*, where* $a_i \in C_i$*.*

The space of candidates is the set product of the choice sets. The preference among the elements of the choice sets induces a preference among the candidates:

Definition 4.3.3 (preference) *Let CS be a choice space and A, B be two candidates for CS. A* is preferred to *B, noted $A \leq B$, iff*

$$\forall C_i \in Choices: \exists a_i, a_i' \in C_i \text{ s.t. } (C_i, a_i) \in A, (C_i, a_i') \in B \text{ and } a_i <_{C_i} a_i' \vee a_i = a_i'.$$

The priority order of a choice space is assumed to "agree" with the preference among the candidates, i.e. for any two candidates $A \leq B \Rightarrow B \leq_{pri} A$, i.e. a candidate that is preferred must also have a higher priority.[9]

Note that, although the preference among the elements of a choice set is total, the preference among the candidates is only a *partial order*. It is easy to see that the preference among the candidates imposes a lattice structure on the candidate space.

Example 4.3.4 *Figure 4.15 depicts the candidate lattice corresponding to a choice space with three choice sets, say C_1, C_2, C_3, each with three elements, i.e. $C_i = \{a_{i1}, a_{i2}, a_{i3}\}$, where $a_{i1} <_{C_i} a_{i2} <_{C_i} a_{i3}$. In the figure, an element ijk stands for the candidate $\{(C_1, a_{1i}), (C_2, a_{2j}), (C_3, a_{3k})\}$, for instance* 121 *represents the candidate $\{(C_1, a_{11}), (C_2, a_{22}), (C_3, a_{31})\}$.* 111 *is the bottom element (the smallest in the preference order) of the lattice,* 333 *is the top element. The lines between some candidates denote the preference order, e.g.* $121 \leq 221$, $121 \leq 131$, *etc.*

If $A \leq A'$ we sometimes say in the following that A' is a successor of A. Not all of the candidates are *consistent*. A candidate is defined to be consistent if and only if the set of selected atoms does not include any conflict:

Definition 4.3.5 (consistency) *A candidate A of a choice space $(L, Choices, Confl, \leq_{pri})$ is* inconsistent *if it "includes" a conflict, i.e. iff $\exists c \in Confl, c \subseteq \{a_j \mid (C_i, a_j) \in A\}$.*

In logical terms, each consistent candidate solves each positive clause corresponding to the choice sets, and does not violate any negative clause corresponding to the conflicts.

Example 4.3.6 *Figure 4.16 shows the candidate preference lattice for the choice space of Figure 4.15 with 3 conflicts. It is assumed here that the*

[9] We hope that the notation is not misleading: preference is noted as minimality, while "priority" is noted as maximality.

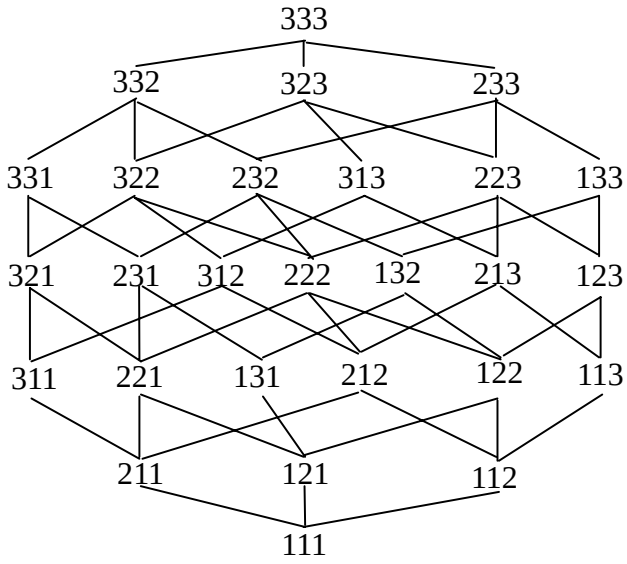

Figure 4.15: A candidate lattice.

three choice sets depicted are disjoint. The notation $k_1k_2k_3$*, when used to represent a conflict, has the interpretation:* $\{a_{jk_j} \mid k_j \neq 0, j \in \{1,2,3\}\}$*, for instance* 110 *represents the conflict* $\{a_{11}, a_{21}\}$*,* 001 *is the conflict* $\{a_{31}\}$*, and* 020 *the conflict* $\{a_{22}\}$*. The figure depicts the set of candidates, the consistent candidates and the preferred consistent candidates, given the three conflicts above.*

We are interested in the following to characterize the set of preferred and consistent candidates of a choice space. In fact, since we do not want to compute the whole set of preferred candidates from the consistent ones, we characterize *lower bounds* of the set of consistent candidates of a choice space.

Definition 4.3.7 (lower bound) *A* lower bound *of the consistent candidates of a choice space* CS *is a set of candidates* $LB = \{A_1, \ldots, A_k\}$ *such that all the consistent candidates of* CS *are successors of some member of* LB*, and no element of the lower bound is a successor of a different element of the lower bound, i.e.* (i) *for any consistent candidate* A *of* CS *there exists* $A_i \in LB$ *such that* $A_i \leq A$*, and* (ii) $\forall A_i, A_j \in LB, A_i \geq A_j \Rightarrow A_i = A_j$*.*

A lower bound in which all the elements are consistent corresponds to the

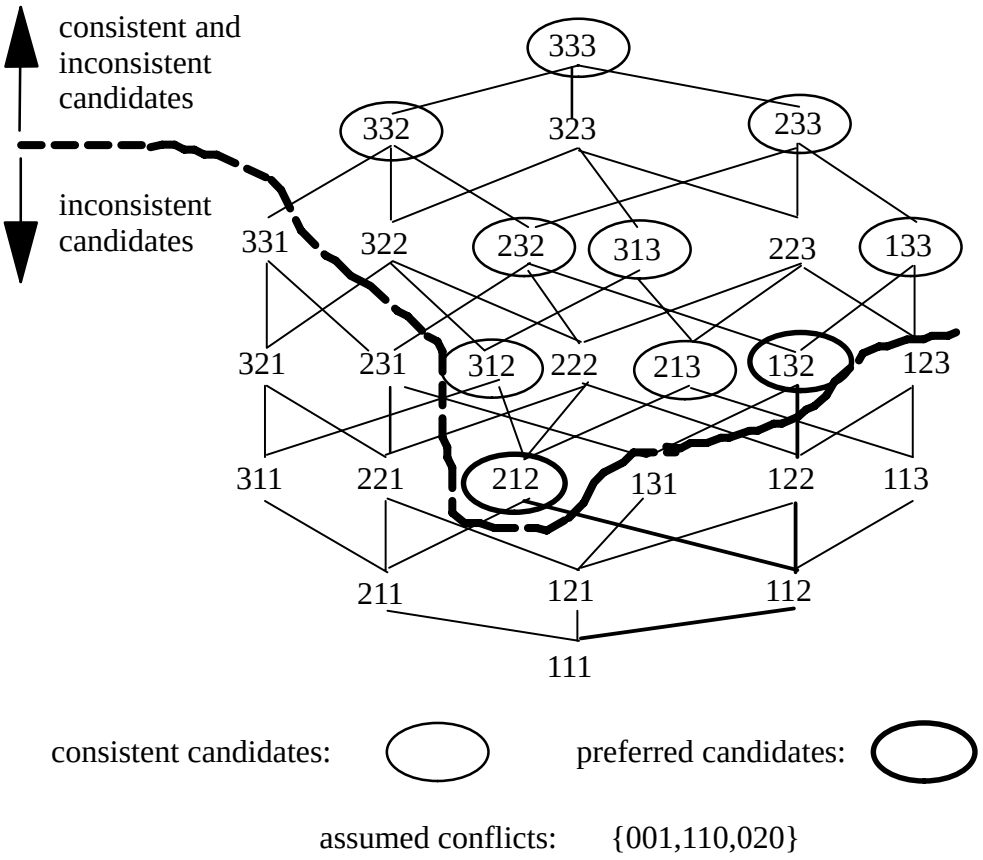

Figure 4.16: Preferred and consistent candidates.

preferred diagnoses, provided that the choice sets correspond to the modes of each component and the conflict set is complete.[10] One could note that the set of preferred consistent candidates represent a generalization of the minimal diagnoses (cf. [dKW87]) for the case where each component has several behavioral modes. When each choice set has two elements (corresponding in diagnosis to the correct, respectively the abnormal behavioral mode) the preference lattice collapses to the lattice of the minimal diagnoses (see also [dKW87]).

[10] This corresponds also to the definition of preferred diagnoses of Dressler and Struss from [DS92].

Example 4.3.8 *The set containing only the bottom candidate of the preference lattice always defines a lower bound for the consistent candidates, e.g.* $LB_1 = \{111\}$ *for the case depicted in Figure 4.16.* $LB_2 = \{211, 121, 112\}$, $LB_3 = \{311, 121, 112\}$, *and* $LB_4 = \{311, 121, 212, 113\}$, $LB_* = \{212, 132\}$ *are other possible lower bounds for the consistent candidates of the lattice, whereas the last one contains only consistent candidates and defines thus the set of preferred consistent candidates for this example.* $\{212, 132, 322\}$ *is not a lower bound because* $212 \leq 322$. $\{221, 212\}$ *is not a lower bound because not all of the consistent candidates are in the relation "* $\leq$ *" with some member of the set (e.g.* 132*).*

The lower bound can be "pushed" upward by replacing some inconsistent candidate from the lower bound with some of its direct successors. But, at this point we must state precisely what a *direct* successor is:

Definition 4.3.9 (direct successor) *Let* A *be a candidate of* CS. *A different candidate* A' *of* CS *is a* direct successor *of* A *(noted* $A' \in DirSucc(A)$*) iff* $A < A'$ *and there is no candidate* A'' *in between, i.e.*

$$\neg(\exists A'', A < A'' < A').$$

It is easy to construct all the direct successors of a given candidate. We note with $Next(a, C_i)$ the next preferred element of C_i after a:

$$Next(a, C_i) = \begin{cases} nil & \text{if } \neg(\exists a' \in C_i, a <_{C_i} a') \\ a' & \text{if } \exists a' \in C_i, a <_{C_i} a' \wedge \neg(\exists a'' \in C_i, a <_{C_i} a'' <_{C_i} a') \end{cases}$$

Because the preference within a choice set is total and strict $Next(a, C)$ is a function. The following is then obviously true:

Property 4.3.10 *Let* A *be a candidate in a choice space. Then* $A' \in DirSucc(A)$ *iff:* $\exists (C_i, a_i) \in A$ *s.t.* $Next(a_i, C_i) \neq nil$ *and*

$$A' = (A - \{(C_i, a_i)\}) \cup \{(C_i, Next(a_i, C_i))\}.$$

The following statement is also immediately following from the definitions:

Property 4.3.11 *Let* LB *be a lower bound of the set of consistent candidates of a choice space and* A *an* inconsistent *candidate* $A \in LB$. *Then* $LB' = (LB - \{A\}) \cup \{D \in DirSucc(A) \mid \neg(\exists A' \in LB - \{A\}, A' < D)\}$ *is another lower bound for the consistent candidates of the choice space.*

The above property tells us that one can replace an inconsistent candidate from a lower bound with its direct successors that are not successors of the rest of the lower bound and obtain a new bound. Note that one does not have to check if the newly inserted direct successors are more preferred than the old elements of the lower bound, since this cannot be the case given that their parent was in the lower bound before.

Example 4.3.12 *The lower bound LB_2 of example 4.3.8 is obtained from LB_1 by replacing* 111 *with its immediate successors. LB_3 is obtained from LB_2 by replacing* 211 *in LB_2; LB_4 is obtained by replacing* 112 *in LB_3 with its successors that are not less preferred than the rest of the lower bound. Continuing this operation one reaches in the end LB_*.*

This property can lay the foundation for an algorithm that computes all / some preferred consistent candidates. One can start with the bottom candidate of the preference lattice, since this alone always defines a lower bound, and then successively go upwards with the lower bound by replacing selected inconsistent candidates with the direct successors as stated above. However, as it is easy to imagine and as also our empirical results in practice showed (cf. [TI94a, TI94b]), this algorithm for computing some preferred consistent candidates is extremely expensive. This is why we were interested to further improve the algorithm, and we will do it successively:

The most significant result of this section is provided by the following lemma. It shows that one need not construct *all* the successors as above, but only a subset that also *resolve* one of the conflicts included the inconsistent candidate. The set $DirSucc(A, c)$ will be used to denote the set with the smallest successors of A that do not include c, where c is a conflict included in A. If the choice sets are disjoint then always $DirSucc(A, c) \subseteq DirSucc(A)$. But we did not require the choice sets to be disjoint. In the case when one assumption $a \in c$ from the conflict is selected by several choices from A then some elements of $DirSucc(A, c)$ will be deeper successors of A. In all this cases the semantic is the same: $DirSucc(A, c)$ contains the smallest (most preferred) successors of A that do not include c.

Definition 4.3.13 (direct successors w.r.t. a conflict) *Let A be a candidate of $(L, Choices, Confls, <_{pri})$ and $c \subseteq L$. The set of direct successors*

of A with respect to c is defined as: $A' \in DirSucc(A,c)$ iff $\exists a \in c$ s.t.

$$X_a = \{(C_i, a_i) \in A \mid a_i = a\} \neq \{\},$$
$$\forall (C_i, a) \in X_a : Next(a, C_i) \neq nil,$$
$$A' = (A - X_a) \cup \{(C_i, Next(a, C_i)) \mid (C_i, a) \in X_a\}$$

It is easy to see that each element of $DirSucc(A,c)$ does not include c. If the choice sets are disjoint then $DirSucc(A,c) \subseteq DirSucc(A)$. We now show that when replacing an inconsistent candidate with some of its successors only the one with respect to a conflict included in the candidate must be considered:[11]

Lemma 4.3.14 (conflict pruning I) *Let A be a candidate of $(L, Choices, Confls, <_{pri})$ and $c \in Confls$ be a conflict included in A. If*
S is a successor of A ($A < S$) that does not include c, then:
S is a successor of a direct successor of A with respect to c, i.e

$$\exists D \in DirSucc(A,c), s.t.\ D \leq S.$$

We are interested that our lower bounds still characterize the set of all *consistent* candidates. The previous lemma guarantees that when replacing inconsistent candidates with successors, only the ones that resolve one conflict of the parent are necessary.

Corollary to Lemma 4.3.14 *Let LB be a lower bound for the consistent candidates of a choice space. Let $A \in LB$ be an inconsistent candidate that includes a conflict c. Then*

$$LB' = (LB - \{A\}) \cup \{D \in DirSucc(A,c) \mid \neg(\exists A' \in LB - \{A\}, A' < D)\}$$

is another lower bound for the consistent candidates.

The above corollary achieves a pruning effect during the search for the consistent preferred candidates (compare it with Property 4.3.11). The number of candidates that replace an old inconsistent one in the lower bound depends now also on the *size* of the conflict included in the parent. The smaller the conflict, the more significant this pruning is. For instance, if all the conflicts have size one the lower bound will always contain one element, while the size of the lower bound without using the conflict pruning can grow

[11] The proofs are given in Appendix C.

to considerable sizes. Our empirical experiments have shown dramatic reductions of the time and space required by diagnosis due to the consideration of Lemma 4.3.14 in several diagnostic problems. In the worst case, when the conflicts have the size equal with the candidates there is no pruning at all.

Example 4.3.15 *Consider again the choice space of example 4.3.6 and 4.3.8. When* 111 *is replaced with its successors with respect to the conflict* 001 *in* LB_1*, one gets* $LB'_2 = \{112\}$*. When one replaces* 112 *with its successors with respect to the conflict* 110 *in* LB'_2*, one gets* $LB'_3 = \{212, 122\}$*. In the third step, when* 122 *is replaced in* LB'_3 *with its successors with respect to* 020 *we obtain* $LB_* = \{212, 132\}$*.*

Unfortunately, if a candidate contains more then one conflict, this pruning can be applied only with respect to one of them - otherwise the completeness of the search is lost.

The converse of the implication from Lemma 4.3.14 is not always true. A sufficient condition, which is satisfied in many cases in practice, is when the choice sets $C_i \in Choices$ are mutually disjoint:

Lemma 4.3.16 (conflict pruning II) *Let* A *be a candidate of* $CS = (L, Choices, Confls, <_{pri})$ *and* $c \in Confls$ *be a conflict included in* A*. Let the choice sets of* CS *be disjoint, i.e.*

$\forall C_i, C_j \in Choices, C_i \neq C_j \Rightarrow C_i \cap C_j = \{\}$*. Then if*

S is a successor of some $D \in DirSucc(A, c)$, i.e. $D \leq S$, then:

S is a successor of A that does not include c.

In diagnosis, when the choice sets correspond to the possible mode assignments of the components the choice sets are disjoint, the conditions of Lemma 4.3.16 are satisfied. This provides us with an interpretation of the meaning of the pruning achieved by using Lemma 4.3.14: by avoiding to insert in the lower bound other successors of A except $DirSucc(A, c)$ we avoid at later stages of search to generate other candidates that contained the same conflict c.

So far we did not take into account the priority order that is defined in a choice space. Having said that we want to build only some preferred candidates, and not all of them (because this is usually too expensive and usually not even necessary), the next question to ask, and to answer, is *which* of these consistent and preferred candidates to generate. At this point, our goal is to construct a lower bound that contains some consistent candidates that

have the highest *priority* within the consistent and preferred candidates, but *without* computing the complete set of preferred and consistent candidates. The following statement gives us a hint in this respect:

Lemma 4.3.17 *Let LB be a lower bound for the set of consistent candidates of a choice space and assume $A \in LB$ is a* consistent *element. Let also $H(A, LB)$ be the sets of elements from LB that have a strictly higher priority than A: $H(A, LB) = \{A' \in LB \mid A <_{pri} A'\}$. Then the following are true:*

1. *Let B be a* consistent *candidate that has a strictly higher priority that A, i.e. $A <_{pri} B$. Then either $B \in H(A, LB)$, or B is a successor of some element from $H(A, LB)$;*
2. *Let C be a* consistent *candidate that is a successor of some element from $LB - H(A, LB)$. Then C cannot have a higher priority than A has, i.e. $C \leq_{pri} A$.*

Corollary to Lemma 4.3.17 *Let LB be a lower bound for the set of consistent candidates of a choice space. Let $A \in LB$ be the candidate having the highest priority in LB. If A is consistent, then A has the highest priority across all the consistent candidates of the choice space.*

The corollary tells us that as soon as the element from a lower bound LB with the highest priority across LB is consistent, we can be sure that, irrespective if the rest of the LB contains consistent elements or not, the best element of LB is the best across all the consistent candidates (though, since $<_{pri}$ can be a partial order, the maximum may not be unique).

Lemma 4.3.17 suggests that in order to find a candidate having the highest priority one can conduct a best-first search in which at each step the element of a lower bound with the highest priority is replaced with its successors as long as the "best" element of the lower bound is inconsistent. If one wants to continue the search further, towards other preferred and consistent candidates having the highest priority, then the next "best" elements of the lower bound that are inconsistent should be successively replaced with (some of) their successors.

The improvement is that we avoid to build a lower bound where all of the elements are consistent in order to find only the best k of them - the search procedure can thus stop earlier.

Incremental addition of conflicts and choice sets

In practice the conflicts are discovered incrementally. In diagnosis, for instance, one first computes a first focus (some preferred candidates with high priority) and then by making prediction focused on those candidates new conflicts, not evident from the beginning, are found. The new conflicts make some of the old focus candidates inconsistent, and though, a new focus should be computed. Fortunately, the computation of the new focus can reuse the old lower bound:

Property 4.3.18 *Let LB be a lower bound for the consistent candidates of $(L, Choices, Confls, <_{pri})$. Then LB is also a lower bound for the set of consistent candidates of $(L, Choices, Confls \cup \{c\}, <_{pri})$, where $c \subseteq L$.*

The addition of new choice sets can also be handled in an incremental way:

Property 4.3.19 *Let LB be a lower bound of $(L, Choices, Confls, <_{pri})$. Let $C \subseteq L$ be a new choice set and let a_C be the smallest element from C with respect to the preference $<_C$. Then $LB' = \{A \cup \{(C, a_C)\} \mid A \in LB\}$ is a lower bound for the consistent candidates of $(L, Choices \cup \{C\}, Confls, <_{pri})$.*

4.3.3 Searching for the preferred candidates having the highest priority

Based on the properties of a choice space Appendix B.1 presents in more detail some algorithms for the incremental generation of the preferred consistent candidates with the highest priority.

Basically, a candidate generator works on an associated choice space and maintains a lower bound for it. The lower bound can be split into two sets: $Focus$ and $Candidates$.

- $Focus$ contains the preferred and consistent candidates (with the highest priority) that should be in the focus of the problem solver;
- $Candidates$ contains the rest of the candidates that together with $Focus$ define a lower bound for the consistent candidates of the choice space administrated by the candidate generator. The elements from $Candidates$ need not be consistent.

Table 4.4 shows the effects of the conflict pruning on four candidate elaboration problems for the electrical system from Figure 4.14. Although in

electrical systems usually the conflicts have relatively big sizes, and consequently the effects of the pruning are less dramatic, the results show significant reductions of the time and space required in diagnostic problems due to the consideration of the conflict pruning.

Problem	conflicts	size of lower bond		Time in seconds	
		◯	△	◯	△
1	6	150	30	13.5	10
2	11	327	32	12.6	8
3	21	860	103	31	7
4	30	1774	263	111	17

Table 4.4: Different candidate generators: ◯ no conflict pruning △ with conflict pruning.

Further improvements are obtained if we observe that the candidate generator need not store all of the conflicts in order to decide the consistency of the candidates. If the choice sets are disjoint, all conflicts expressing exclusive choice sets are not relevant (the algorithms implicitly deal with these conflicts). Furthermore, the conflicts that invalidate only candidates that are below the lower bound can be removed from the collection of stored conflicts.

Example 4.3.20 *Consider the choice space from Figure 4.16 and consider again the operations performed in order to find the preferred consistent candidates (shown in Example 4.3.15). We start with* $LB_1 = \{111\}$ *and have the conflicts* $\{001, 110, 020\}$. *After we insert the direct successors of* 111 *with respect to* 001, *we can dispense with the conflict* 001 *since no successor of* $LB_2 = \{112\}$ *can include* 001. *After we replace* 112 *with the successors with respect to* 110 *we can dispense with* 110 *as well. After the third step we reach* $LB_* = \{212, 132\}$. *Only* 020 *is still relevant, and only for the successors of* 212.

Removing from the candidate generator the non-relevant conflicts may increase the efficiency of the consistency test in case the number of conflicts gets very large.

Another improvement follows the observation that we usually have to deal with single faults, or with candidates accusing a small number of components as defective. Then in each candidate most of the choice sets select the best

assignment (the most preferred one). It saves space as well as time – consider for instance the operations that have to be performed when testing the preference between two candidates – to store internally in a candidate only the "exception" choice sets, i.e. the choice sets where not the best element is selected.

4.3.4 On the logical completeness of the candidate generator

In this section we address the following problems:

1. we show that, from a logical point of view, our candidate generator can be seen as a sound and complete satisfiability checker for propositional theories consisting of purely positive and purely negative clauses. Moreover, in case of consistency, our candidate generator builds one or several (preferred) models[12] of the propositional theory;
2. we analyze the usefulness of enhancing the candidate generator with hyperresolution rules in order to infer smaller implied conflicts and choice sets. GDE+ (cf. [SD89]) embeds such hyperresolution rules into its candidate generator. With respect to this point we conclude that:

   - due to the incremental way in which the space of diagnoses is explored and in which the conflicts are received by the candidate generator it does not seem very useful to enhance our candidate generator with such rules; but
   - if one enhances the candidate generator such as to build candidates consistent with a believed set of propositions Ctx, the extension with additional inference rules can be useful in order to infer more conflicts among the Ctx propositions in case of inconsistency. The Ctx propositions could represent working hypotheses in the spirit of [Str89], or choices already committed by other modules in a distributed candidate generation architecture (an example of a hierarchic organization of several candidate generation modules is given in Appendix B.2).

[12] By model, or interpretation, we mean here an assignment of boolean values to the propositions such that all the formulae of the propositional theory evaluate to true.

3. We give an efficient algorithm for computing the negative prime implicates among the Ctx propositions for a candidate generator extended as above.

Logical properties of the basic candidate generator

From a logical point of view, our a candidate generator is testing the satisfiability of the logical theory $\Sigma = \Sigma_p \cup \Sigma_n$, where:

- Σ_p consists of the positive disjunctions associated with the choice sets, and
- Σ_n consists of the negative disjunctions associated with the conflicts.

Each candidate is a conjunction of positive literals that solve each positive clause from Σ_p. The consistent candidates do not violate any of the negative clauses from Σ_n. We show in the following that Σ is satisfiable if and only if our candidate generator finds at least a consistent candidate in the associated choice space.[13]

Definition 4.3.21 *Let $CS = (L, Choices, Confls, <_{pri})$ be a choice space. Define $\Sigma(CS)$ to be the following set of propositional clauses: each set of symbols that appears in $Choices$ corresponds to a positive disjunction in $\Sigma(CS)$, and each set of symbols that appears in $Confls$ corresponds to a disjunction among the negations of the symbols in $\Sigma(CS)$.*

Lemma 4.3.22 *Let $CS = (L, Choices, Confls, <_{pri})$ be a choice space and $\Sigma(CS)$ its associated set of clauses (as in Definition 4.3.21). Let $v : L \rightarrow \{true, false\}$ be an assignment of boolean values to the symbols. Then:*

$$\textit{the formulae of } \Sigma(CS) \textit{ are satisfied by the assignment } v \Leftrightarrow$$

$$\exists A, \textit{ where } A \textit{ is a consistent candidate of } CS \textit{ s.t.}$$
$$\{a \mid \exists C_i \in Choices \textit{ s.t. } (C_i, a) \in A\} \subseteq \{a \in L \mid v(a) = true\}.$$

The following is then an immediate consequence of the above lemma:

Corollary 1 to Lemma 4.3.22 *Let CS be a choice space and $\Sigma(CS)$ be the propositional theory associated to CS as in Definition 4.3.21. Then: there is no consistent candidate for CS if and only if $\Sigma(CS)$ is not satisfiable.*

[13] The proofs are given in Appendix C.

In case of consistency, each consistent candidate built by the candidate generator provides (at least) one model for the propositional theory. According to the above lemma, if A is a consistent candidate, the assignment $v : L \rightarrow \{true, false\}$, $v(p) = true \Leftrightarrow p \in \{a \mid (C_i, a) \in A\}$ satisfies all the clauses from $\Sigma_p \cup \Sigma_n$.

On the usefulness of enhancing the formula completeness of the candidate generator

Another question that we want to clarify in this section is whether it would be useful to extend our candidate generator such as to infer more nogoods, when possible, from the knowledge about the existing nogoods and about the choice sets. Such additional nogoods can be inferred by hyperresolution:

Example 4.3.23 *Suppose a choice space has two choice sets:* $C_1 = \{a_1, a_2\}$, *and* $C_2 = \{b_1, b_2\}$ *and the conflicts:* $c_1 = \{a_1, b_1\}$, $c_2 = \{a_1, b_2\}$. *Applying a hyperresolution rule on* $b_1 \vee b_2$, $\neg b_1 \vee \neg a_1$, $\neg b_2 \vee \neg a_1$, *one finds* $c_* = \{a_1\}$ *as nogood.*

Model-based diagnostic systems like GDE+ (cf. [SD89]) embed such a hyperresolution rule into their candidate generator. In principle, these inference rules are superfluous if all we want to do is to decide consistency and to build (preferred) models of Σ. They would bring probably only computational overhead.

Corollary 2 to Lemma 4.3.22 *The set of consistent candidates of a choice space* CS *is not affected by the addition of conflicts that are implicates of* $\Sigma(CS)$.

Some of the new conflicts produced by hyperresolution can be subsets of several old ones (see Example 4.3.23). Since the complexity of the candidate generation is affected by the size of the conflicts, the use of smaller conflicts is likely to reduce the search effort. However, even if we ignore the additional effort required to perform the hyperresolution, due to the sequential way in which the conflicts are found, it is likely that the search effort cannot be reduced this way, as we try to illustrate in the following example:

Example 4.3.24 *Consider again the choice sets and the conflicts of Example 4.3.23. A candidate generator that starts the search using the conflict* c_* *would perform less operations than one using* c_1, c_2. *However, if*

first no conflict is known, the candidate generator provides the candidate $A = \{(C_1, a_1), (C_2, b_1)\}$. *Then the conflict* c_1 *is found and the candidate generator is proposing* $B = \{(C_1, a_1), (C_2, b_2)\}$. *Only now, after focusing on this candidate, the conflict* c_2 *is discovered. But now the addition of* c_* *would be performed too late: the operations that were useless have already been performed – in this case the generation of the candidate* B.

Increasing the completeness of the (prime) implicates computed by a candidate generator (module) makes sense, however, if the candidate generator is extended such as to build only candidates consistent with a "believed" set of literals Ctx. In case of inconsistency, the beliefs Ctx must be revised; for such a purpose the knowledge of the (minimal) conflicts among the Ctx literals might be essential.

Context-dependent candidates

The basic candidate generator described in the previous sections was able to generate a specified number of consistent and preferred candidates according to an additional priority order. However, there may be situations where this fixed scheme is not flexible enough. The centralized control over the generation of the candidates would not be appropriate, for instance, if a user of the system should cooperate interactively to the specification of the focusing candidates.

An interesting idea could be the one of *distributing* the choice sets among several candidate generation modules, which could use different generation strategies. One such module, for instance, could represent the user, if the user is supposed to play a more active role in guiding the search for a solution.[14] As a mean of communication, we assume that the modules that comprise the candidate generator can exchange focus candidates and conflicts.

Suppose we have two candidate generation modules CG_1, CG_2, each maintaining its own choice space. One useful service that one module, say CG_1, may ask the other one would be to extend a generated candidate Ctx of CG_1 with the "best" consistent candidates of CG_2. If no consistent extension exists, then this means that a conflict among the assumptions of Ctx exists

[14] For instance, in a reasoning system that is supposed to assist a human in design tasks, the decisions taken by the human should be followed by the system as long as they are consistent. The assistant would then play more the role of a consistency checker, but could as well accomplish some more routine tasks.

(as a consequence of Corollary 1 of Lemma 4.3.22), and this conflict should be communicated to CG_1.

In the following we extend the candidate generator such as to:

1. generate only candidates that are consistent with a certain specified set of symbols - possibly already chosen by other candidate generator modules, or representing working hypothesis in the spirit of [Str89]. More precisely, if *CS=(L, Ch, Confl,* $\leq_{pri}$*)* is a choice space and $Ctx \subseteq L$ is a set of literals, we are only interested in the candidates of CS that can consistently extend Ctx, i.e. in the candidates A such that:

$$\neg(\exists c \in Confls \text{ s.t. } c \subseteq Ctx \cup \{a_i \mid (C_i, a_i) \in A\}). \qquad (4.5)$$

   Of course, we are only interested in some of the preferred candidates from the above set, namely in the ones with the highest priority.
2. report the minimal conflicts among the Ctx symbols when Ctx cannot be consistently extended with any candidate of the module.

It is trivial to extend the candidate generator described in 4.3.3 (see also Appendix B.1) in order to construct candidates consistent with a certain set of literals Ctx:

- instead of testing if the generated candidates are consistent, the test from the equation 4.5 should be used. If there is a conflict $c \in Confls$ that violates the condition, then the successors of the candidate with respect to the set $c - Ctx$ should be computed.[15]

If a certain set of literals Ctx cannot be consistently extended with any candidate from a certain choice (sub-)space there must be a conflict among the literals of Ctx. The discovery of the minimal conflicts from Ctx may significantly increase the efficiency of the search in the modules that chose the Ctx literals.

The next subsection presents an efficient algorithm that computes the negative prime implicates among the Ctx literals. The algorithm avoids to generate the complete set of prime implicates or the complete set of negative prime implicates and to apply a filter afterwards. The application of this algorithm could be delayed until a candidate generator cannot consistently extend the set of literals Ctx. As long as Ctx still can be consistently extended, one cannot derive any conflict included in Ctx, of course.

[15] It is easy to see that the new candidate generator is more general than the old version, namely, the old version is obtained from the new one by making the associated set of literals Ctx an empty set.

An efficient algorithm for computing the conflicts among the Ctx literals

We note that:

1. It is possible to reuse some control techniques developed in the literature dedicated to the prime implicate (prime implicand) generation (cf. [KT90, Tis67, FdK93]) in order to avoid useless computations - without violating the completeness of the (prime) implicate generation;
2. Since we are, in fact, interested only in *some* negative prime implicates, namely in those involving only literals of Ctx, we show that we can further restrict the inference – this is going to give up the completeness of the (prime) implicate generation, but still guarantees the completeness with respect to the *negative* clauses involving *only* the Ctx literals.

Whenever all the elements from a choice set C are involved in some conflicts a hyperresolution rule can be applied in order to obtain new conflicts free of the literals from the choice set C. Namely, if one has a choice set $C_i = \{a_1, a_2, \ldots, a_k\}$ and each a_i is part of a conflict, i.e. there exist negative clauses having the form $\neg a_i \vee NC_i$, where NC_i are negative clauses, one gets:

$$R1: \quad \frac{a_1 \vee \ldots \vee a_k; \quad \neg a_i \vee NC_i, \text{ for all } i}{NC_1 \vee NC_2 \vee \ldots \vee NC_k} \tag{4.6}$$

The new clause represents a conflict that may be interesting if it is not a superset of an already discovered conflict. It is easy to implement R1 using an ATMS: for each member a_i of a choice set an associated node is created (having the semantics of the negation of the symbol, i.e. "$\neg a_i$"). Each conflict involving a_i, say $\{a_i, a_m, \ldots, a_n\}$ is used to create a justification: $a_m, \ldots, a_n \rightarrow$ "$\neg a_i$", and each choice set $C_i = \{a_1, a_2, \ldots, a_k\}$ is associated an additional justification: "$\neg a_1$", "$\neg a_2$", ..., "$\neg a_k$" $\rightarrow \bot$.

However, it is not necessary to apply this rule exhaustively. A conflict c should be considered at a choice point C_i only if $c \cap C_i \neq \{\}$. But, if $|c \cap C_i| > 1$, then the consideration of c at C_i is not going to derive any useful conflicts (the proofs are given in Appendix C).

Property 4.3.25 *Let $C = \{a_1, \ldots, a_k\}$ be a choice set and c be a conflict such that $|c \cap C| > 1$. Then any new conflict obtained by hyperresolution at C and involving c and possibly other conflicts is a superset of an already known conflict.*

Obvious sources of such conflicts are the exclusive choice sets, i.e. choice sets where the literals are pairwise inconsistent. As the above property shows, it is not necessary to encode these conflicts in the justification network that implements the hyperresolution.[16]

As proved in [Tis67, KT90] (see also Chapter 13 from [FdK93]) in order to obtain all the prime implicates of a set Σ of clauses one can use the following algorithm (which we refer in the following as PI):

1. Initialize Cls with the set of unsubsumed clauses of Σ;
2. Iterate on the symbols from Σ in some order, for each symbol a do:
   (a) Perform all the binary resolutions w.r.t. the symbol a (i.e. resolutions that resolve the clauses containing a positively with the ones that contain a negatively); Add the new unsubsumed clauses to Cls; Remove the subsumed clauses from Cls;
3. Return Cls.

The algorithm avoids to execute all the possible binary resolutions. This does not affect the completeness of the prime implicate generation because the same implicates can be derived in very many ways from an initial set of clauses - due to the fact that the binary resolution is often commutative and associative.

One could use PI to compute all prime implicates and then filter out the ones we are not interested in. But, since we are *not* interested in the complete set of prime implicates, we can do much better.

The function $NPI^*(CS, Ctx)$ (Figure 4.17) returns all the minimal conflicts among the Ctx symbols entailed by the logical theory $\Sigma(CS)$. The function applies only the hyperresolution rule R1, and thus derives only negative implicates, but not all of them.

The function iterates on the symbols from $Choices$ (the order of iteration is not important from a logical point of view, but it could let place for heuristics). For each symbol the conflicts involving that symbol are considered for the hyperresolution R1 at the choice sets involving that symbol. If the current symbol is not in Ctx, the conflicts considered at that step are removed from $Cfls$ and are prevented thus from being encoded later with respect to other unprocessed symbols (see lines 6,7 of NPI^*). The removing

[16] If the choice sets are disjoint then it is not necessary to communicate these conflicts to the candidate generator at all - they cannot be part of any generated candidate.

Function NPI^* (CS, Ctx)
$CS = (L, Choices, Confls, <_{pri})$ is a choice space, and $Ctx \subseteq L$. Returns the set of minimal conflicts among the symbols from Ctx entailed by $\Sigma(CS)$.

(1) let $ChSym := \{a_{ij} \in C_i \mid C_i \in Choices\}$;
(2) let $Cfls := \{c \in Confls \mid c \subseteq Ctx \cup ChSym\}$;
(3) **for** each symbol $a \in ChSym$ **do:**
  (4) **for** each conflict $c \in Cfls$ s.t. $a \in c$ **do:**
    (5) encode c for hyperresolution at all $C_i \in Choices$ s.t. $C_i \cap c = \{a\}$;
    (6) **if** $a \notin Ctx$ **then**
      (7) remove c from $Cfls$;
    endfor
  (8) add each new minimal conflict discovered by hyperresolution to $Cfls$; remove from $Cfls$ the supersets of the new conflicts;
  endfor
(9) **return** $Cfls$;

Figure 4.17: Computing the negative prime implicates of $\Sigma(CS)$ among the Ctx symbols.

of the conflicts performed in the steps 6,7 of NPI^* is the operation that prevents the completeness of the negative prime implicate generation.

If the choice sets are pairwise disjoint and do not share symbols with Ctx then it is easy to see that *each conflict is encoded at most at one* choice set (thus for each conflict, usually, at most *one* justification, as explained before, is created). As the following example shows, this procedure does not derive all the negative prime implicates of $\Sigma(CS)$:

Example 4.3.26 *Assume a choice space has the choice sets:* $C_1 = \{a_1, a_2\}$, $C_2 = \{b_1, b_2\}$ *and the conflicts:* $c_1 = \{a_1, a_2\}$, $c_2 = \{a_1, b_1, x\}$, $c_3 = \{a_1, b_2\}$, $c_4 = \{a_2, b_1, y\}$, $c_5 = \{a_2, b_2\}$. *Suppose* NPI^* *is called on this choice space and the set of literals* $Ctx = \{x, y, z\}$. *Suppose the order in which* NPI^* *iterates on the symbols is:* a_1, a_2, b_1, b_2. *Initially* $Cfls = \{c_1, c_2, c_3, c_4, c_5\}$. *In the first iteration of* NPI^* *the conflicts involving* a_1 *are considered. Only* c_2 *and* c_3 *are encoded for hyperresolution at* C_1 *(*c_1 *need not be encoded since* $|c_1 \cap C_1| > 1$*). After the first iteration* $Cfls = \{c_4, c_5\}$. *Next the remaining conflicts involving* a_2 *are considered and* c_4 *and* c_5 *are encoded at* C_1 *and are removed from* $Cfls$. *The rule R1 at* C_1 *derives new minimal conflicts:* $c_6 = \{b_1, x, y\}$ *and* $c_7 = \{b_2\}$. *After the second iteration* $Cfls = \{c_6, c_7\}$. *In the end, the function will return the set containing only the minimal conflict* $c_* = \{x, y\}$.

The execution of the function avoids to derive some of the negative prime implicates, in this case: $\neg a_1 \vee \neg x$ *and* $\neg a_2 \vee \neg y$.

Although, NPI^* avoids to derive the complete set of negative prime implicates of $\Sigma(CS)$, as the next corollary shows, NPI^* guarantees the completeness of the negative prime implicate generation with respect to the Ctx symbols. Of course, if one chooses Ctx to be a superset of the symbols appearing in $\Sigma(CS)$, then NPI^* generates all the negative prime implicates.

Theorem 4.3.27 *Let* Σ *be a set of clauses and* Ctx *be a set of symbols. Then the algorithm* $NPI(\Sigma, Ctx)$ *computes all the negative prime implicates of* Σ *among the symbols of* Ctx*, where* NPI *is defined as follows:*

1. *Initialize* Cls *with the set of unsubsumed clauses of* Σ*;*
2. *Iterate on the symbols from* Σ*, for each symbol* a *do:*

   (a) *Perform all the binary resolutions w.r.t. the symbol* a*; Add the new unsubsumed clauses to* Cls*; Remove the subsumed clauses from* Cls*;*

(b) If $a \notin Ctx$ then remove from Cls all clauses that mention a (positively or negatively); If $a \in Ctx$ then remove from Cls all clauses that mention positively a;

3. Return Cls.

Corollary to Theorem 4.3.27: *Let CS be a choice space. The algorithm $NPI^*(CS, Ctx)$ (Figure 4.17) computes all the minimal conflicts among the symbols from Ctx entailed by the set of clauses $\Sigma(CS)$.*

In practice there are situations where the fixed scheme of generating the most preferred candidates is not flexible enough. In this respect, the idea of distributing the candidate generation among several modules, with different generation strategies and different (competing / cooperating) "social" behaviors could represent an issue of further research. As an illustration of a possible organization of several candidate generation modules, Appendix B.2 gives an example of a hierarchic organization.

4.4 Putting the RMS and the candidate generator together

The 2vRMS (as well as other RMSs) was shown to check satisfiability and to construct some models for propositional Horn theories.

The candidate generator was shown to check satisfiability and to construct some models for non-Horn theories consisting of purely positive and purely negative propositional clauses.

Furthermore, we show in this section that:

- if the candidate generator is constantly updating the set of conflicts as they are found by the RMS, and if the RMS is constantly updating the focus as specified by the candidate generator, then the combination of the RMS and the candidate generator can be used as a satisfiability checker and (multiple) model builder for *arbitrary* propositional theories, thus removing the expressiveness limitations of the RMS and of the candidate generator.

When an RMS consumer is fired, it may add one or several propositional clauses to the existing propositional theory encoded in the RMS and the candidate generator. A Horn clause is added as a justification to the RMS. A non-Horn, purely positive clause, is added as a choice set to the candidate generator (eventually with control information regarding the preference and priority). A mixed clause, i.e. non-Horn and not purely positive, is "split" into two clauses: a Horn clause and a purely positive one, that refer to a new additional literal. A mixed clause:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow q_1 \vee q_2 \ldots \vee q_m \qquad (n \geq 1, m \geq 2)$$

can be encoded as:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \wedge \phi_{new} \rightarrow \bot$$
$$\phi_{new} \vee q_1 \vee q_2 \ldots \vee q_m$$

This encoding preserves the satisfiability property of the original theory.

It is not necessary to communicate all of the positive clauses to the candidate generator. The positive clauses that already evaluate to true, because one of their literals is implied by some selections in other disjunctions and the Horn theory, need not be encoded in the candidate generator. Since the

candidate generator can deal with the incremental addition of the positive clauses, an additional module of the reasoning architecture can select which positive disjunction to communicate next. A sensible strategy can select incrementally at each step one unsolved positive clause having the smallest entropy among the possible alternatives, thus trying to perform at each time the inferences that are most certain. This strategy selects first the choice sets where strong preferences are known (for instance, the mode of behavior is almost always *ok*), or where only a few possible ways of solving the disjunction exist.[17]

The rest of this section proves the correctness of the encoding and investigates more precisely the services that the RMS and the candidate generator together provide to the problem-solver from a logical point of view.

Candidate generator + RMS: a heuristically-guided satisfiability checker for propositional theories

Both the candidate generator and the 2vRMS operate at the propositional level. First we restate what kind of services each of the modules supports alone.

If one neglects the consumer activation, the 2vRMS provides the following services to its users:

- *consistency check for propositional Horn theories* $\mathcal{A} \cup \mathcal{J}$, where $\mathcal{A}$ represents a focusing environment[18] and $\mathcal{J}$ is the set of justifications.
- *entailment check*, i.e. given a certain node n one can find out if $\mathcal{A} \cup J \models n$.[19] The boolean assignment $v(n) = true \Leftrightarrow (\mathcal{A} \cup \mathcal{J} \models n)$ provides a model for $\mathcal{A} \cup \mathcal{J}$ in case of consistency.
- *minimal supporting set generation*: this service is provided by computing the labels in the *detailed* view of the 2vRMS, an operation similar to the label computation in a focusing ATMS.

The candidate generator operates on propositional theories Σ consisting of purely positive and purely negative clauses, i.e. $\Sigma = \Sigma_p \cup \Sigma_n$. Let us

[17] This corresponds to variable orderings in constraint satisfaction problems.

[18] In order to simplify the things, we considered only one environment in the RMS focus.

[19] The consistency and the entailment checks are supported by the (cheap) focus view of the 2vRMS. The labeling in the focus view of the 2vRMS is logically equivalent with unit resolution, an inference procedure that provides a sound and complete answer to the satisfiability and the entailment checks in Horn theories.

note the set of propositions that a candidate $A = \{(a_i, C_i) \mid C_i \in Choices\}$ selects, with $\mathcal{A}(A) = \{a_i \mid (a_i, C_i) \in A\}$. The candidate generator provides the following services:

- *consistency check* for propositional theories $\Sigma = \Sigma_p \cup \Sigma_n$. When no consistent candidate can be built by the candidate generator the theory is not satisfiable, otherwise it is. This provides a sound and complete check, as we proved in 4.3.4.
- when Σ is consistent, each focus candidate A is such that:
  - $\mathcal{A}(A) \models C$, for all $C \in \Sigma_p$, and
  - $\mathcal{A}(A) \cup \Sigma_p \cup \Sigma_n$ is consistent.

The candidates are generated incrementally, according to extra-logical control strategies expressed using the preference and the priority relations.

Consider the function SAT from Figure 4.18. As one can note, the diagnostic engine from Figure 4.1 performs during candidate elaboration a superset of the operations performed by SAT, namely, here we dispensed with the consumer activation. In order to simplify the analysis SAT works with only one candidate in the focus and we ignored the control issues regarding the preference and the priority. The following is then true (the proofs are given in Appendix C):

Lemma 4.4.1 *Let Σ_H be a finite set of Horn clauses, Σ_p be a finite set of positive clauses, and $\Sigma = \Sigma_H \cup \Sigma_p$. Then:*

- *$SAT(\Sigma)$ returns $failure$ if and only if Σ is not satisfiable;*
- *if Σ is satisfiable, then SAT returns a set of propositions $\mathcal{A}$ such that $\mathcal{A} \cup \Sigma$ is consistent and $\mathcal{A} \models C$, for all $C \in \Sigma_p$.*

From each set of propositions $\mathcal{A}$ returned by SAT one can easily construct a model, i.e. an assignment of boolean values to the symbols of $\Sigma_H \cup \Sigma_p$ such that all of the clauses evaluate to true. It is easy to see that the following assignment has this property:

$$v(n) = true \Leftrightarrow (\mathcal{A} \cup \Sigma_H \models n).$$

The question whether $\mathcal{A} \cup \Sigma_H \models n$ is answered simply by inspecting the focus label of the 2vRMS node n.

Function SAT (Σ)
Σ is a set of propositional clauses. It is assumed that Σ contains only Horn clauses and positive clauses. The function returns $failure$ if Σ is not satisfiable, or a set of positive literals $\mathcal{A}$ otherwise. cg is a candidate generator, rms is a $2vrms$.

(1) **for** each positive clause C of Σ **do:**
  (2) associate assumptions in the rms to the propositions from C;
  (3) add a choice set for C to the cg;
  endfor
(4) **for** each Horn clause of Σ **do:**
  (5) add a corresponding justification to the rms;
  endfor
(6) **repeat**
  (7) call the candidate generator cg to generate one candidate A, if one consistent candidate exists;
  (8) **if** the focus of cg is empty **then return** $failure$;
  (9) focus the rms on the environment defined by $\mathcal{A}(A)$;
  (10) **if** any contradiction node holds in the focus of the rms **then**
    (11) query the label of the contradictory nodes;
    (12) add the new conflicts to cg;
  until the focus of cg is not empty
(13) **return** $\mathcal{A}(A)$;

Figure 4.18: An RMS-based propositional reasoner.

SAT directly works on sets which contain Horn and positive clauses, i.e. it does not work on mixed clauses as defined above. However, for each clause set containing mixed clauses one can associate a set containing only Horn and positive clauses such as to preserve the satisfiability property:

Lemma 4.4.2 *Let* $C \equiv (\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n \vee q_1 \vee \ldots \vee q_m)$, $n \geq 1$, $m \geq 2$ *be a mixed clause, and* Σ *be a clause set. Then:*

$$\Sigma \cup \{C\} \textit{ is satisfiable} \quad \textit{if and only if} \quad \Sigma \cup \{C_1, C_2\} \textit{ is satisfiable,}$$

where: $C_1 \equiv (\neg p_1 \vee \ldots \vee \neg p_n \vee \neg \Psi)$, $C_2 \equiv (\Psi \vee q_1 \vee \ldots \vee q_m)$, *and* Ψ *is a new symbol not appearing in* Σ *or in* C.

Corollary 1 to Lemma 4.4.2 *Let* Σ, C, C_1, C_2 *and* Ψ *be as in Lemma 4.4.2. Let* F *be an arbitrary propositional formula not involving* Ψ. *Then:*

$$\Sigma \cup \{C\} \models F \quad \textit{if and only if} \quad \Sigma \cup \{C_1, C_2\} \models F.$$

Based on the above result, one can now transform any finite clause set into another finite one that preserves the satisfiability property (and the entailment property) and such that it contains only positive and Horn clauses. Then *SAT* can give a sound and complete answer to the satisfiability question of arbitrary clause sets.

Corollary 2 to Lemma 4.4.2 *Let* Σ *be a finite clause set, and* $\Sigma_p \cup \Sigma_H$ *be the transformation of* Σ *where each mixed clause is replaced as in Lemma 4.4.2. Then:*

1. Σ *is satisfiable* *if and only if* $\Sigma_p \cup \Sigma_H$ *is satisfiable;*
2. $\Sigma \models F$ *if and only if* $\Sigma_p \cup \Sigma_H \models F$, *where* F *is a propositional formula not involving any of the new literals introduced in* $\Sigma_p \cup \Sigma_H$.

However, in case of consistency, *SAT* computes more than just a boolean value, namely it returns a "consistent candidate". We have clarified previously the logical properties of the consistent candidates returned by *SAT* with respect to the encoded set of clauses $\Sigma_p \cup \Sigma_H$. What are the properties of the sets of symbols returned by *SAT* with respect to the original clause set Σ, is the question that we try to clarify in the following:

Lemma 4.4.3 *Let Σ be a finite and satisfiable set of clauses and $\Sigma_p \cup \Sigma_H$ its encoding, where each mixed clause is replaced as in Lemma 4.4.2. Let $\mathcal{A}$ be a set of symbols returned by SAT when called on $\Sigma_p \cup \Sigma_H$. Then:*

1. *$\Sigma \cup \mathcal{A}$ is satisfiable. The assignment $v(n) = true \Leftrightarrow (\mathcal{A} \cup \Sigma_H \models n)$ is a model of Σ;*
2. *if $\Sigma \models n$ then $\mathcal{A} \cup \Sigma_H \models n$, for any propositional formula n;*
3. *if $\mathcal{A} \cup \Sigma_H \models n$, then $\mathcal{A} \cup \Sigma \models n$, for any* positive literal *n.*

4.5 Related work and discussion

This chapter addressed problems related to basic reasoning tasks. Their relevance exceeds the domain of diagnosis. Connections can be drawn with work done in theorem proving, RMS-based problem solving, default reasoning, constraint satisfaction and, of course, diagnosis.

In our architecture, the role of the candidate generator can be also seen as "intelligently" decomposing a propositional theory into several simpler cases, in the spirit of [Cha72]. Namely, the theory is decomposed into several subtheories which are all Horn and can be efficiently checked for satisfiability.

Of course, not all of the positive clauses need to be communicated to the candidate generator. The positive clauses that already evaluate to true, because one of their literals is implied by some selections in other disjunctions and the Horn theory, need not be encoded in the candidate generator. Since the candidate generator can deal with the incremental addition of the positive clauses, an additional module of the reasoning architecture can select which positive disjunction to communicate next. A sensible strategy can select incrementally at each step one unsolved positive clause having the smallest entropy among the possible alternatives, thus trying to perform at each time the inferences that are most certain. This strategy selects first the choice sets where strong preferences are known (for instance, the mode of behavior is almost always *ok*), or where only a few possible ways of solving the disjunction exist.

The most important motivation behind the design of the 2vRMS and of the candidate generator was not to solve general satisfiability problems. There are many algorithms that address this problem, all of them being exponential since the problem is NP-complete. More important are the extra-logical features that guide the search for a solution, namely the preference and

priority concepts. These concepts can be used to encode *problem-specific control information*. For instance, in diagnosis, the probability distributions of the behavioral modes are used to guide the search for a diagnosis. Another important feature of the RMS and the candidate generator is the incremental way in which they operate: they reuse as much as possible the old results when new clauses are added to the theory.

What is the additional power that the RMS consumer architecture brings? The consumer activation mechanism makes possible to go beyond the expressiveness limitations of the propositional logic, or of the finite-domain constraint satisfaction. It makes possible to solve problems expressed in, for instance, first-order predicate logic (FOPL). How this is done is not described exactly in this thesis.[20] However, even for problems that have a finite encoding in propositional logic the consumer mechanism can prove extremely useful: it can be used as an additional *control* mechanism when the size of the complete propositional theory is too big. For instance, in diagnosis, the constraints associated to the faulty modes are not added to the theory as long as no focus candidate mentions them. Additional control strategies, except the one based on the preference and priority, can be encoded in the consumer architecture – which has the role to "unfold" the set of propositional clauses when necessary (see also [dK86c]).

4.5.1 Controlling the ATMS

The 2vRMS is able to provide all the services that the RMSs from the ATMS family can provide, i.e. consistency checking, entailment checking and minimal support set generation, but controls the operations in a more efficient way.

In [dK92b] it was also noted that even the focusing ATMS performs frequently unnecessary labeling work for diagnostic tasks. The HTMS (Hybrid TMS) used in [dK92b] and [dK91], was reported to combine some features of the focusing ATMS, with those of a JTMS, but, to the knowledge of the author, there is no publication that gives more details about the design of the HTMS so far. The HTMS was reported to find only one nogood for an inconsistent focusing context, but we do not have the information required for a more thorough comparison.

In [dK92b], the HTMS was used in conjunction with an LTMS. Based on the

[20] This is no wonder, since satisfiability in FOPL is not decidable.

observation that in practice in diagnosis most of the candidates are consistent, [dK92b] used by default the cheaper LTMS to validate a candidate, i.e. used the LTMS for the consistency check, and switched to the HTMS in order to find (minimal) conflicts only when the candidate was found inconsistent. Significant reductions of the time and memory spent for solving diagnostic problems were reported in this way, compared with the previously HTMS based implementation.

We believe that the 2vRMS can be seen as an improvement over the architecture described in [dK92b]: In [dK92b] the LTMS and the HTMS were loosely coupled, not *integrated*. In our architecture, the JTMSset view is used to further control the environment propagation in the (focusing and lazy) ATMS view. Also, since the JTMSset is a multiple-context TMS, as opposed to the LTMS and HTMS combination, the context switching within the focus worlds comes at no cost in the 2vRMS.

Of course, in the worst case the 2vRMS may perform as bad as the focusing ATMS,[21] which, in the worst case is as bad as the basic ATMS. However, in the problems we met this never happened. The 2vRMS constantly performed significantly better than the focusing ATMS, which in turn performed much better than the basic ATMS. Without giving up the requirement to compute *minimal* environments, the worst case behavior cannot be improved. It is not evident, however, how useful will prove the relaxation of the minimality in an overall reasoning architecture, i.e. including the PS, when other reasoning tasks performed by the PS depend critically on finding the minimal supports.

As a limitation of the 2vRMS we can note that, if the number of contexts that have to be simultaneously considered in the focus is very big, thus the size of the $fLabels$ gets significantly large, the management of the focus view can become harder than the management of the labels of a focusing ATMS. Also, the specification of the focus in the 2vRMS is less flexible than in the focusing ATMS: a focusing ATMS can work with an intensional focus specification (e.g. "all the single-faults"), while the 2vRMS requires, like the single-context RMSs, an extensional specification of the focusing environments. However, by setting a few control parameters in the 2vRMS, it is easy to configure it such as to behave like a basic, lazy, focusing ATMS, as a JTMSset, and of course as the 2vRMS. We think that this is a useful feature in order to customize the RMS for different application domains.

[21] This is not true if we give up the requirement to compute minimal environments for the $dLabels$, a variant that can be easily realized using the control techniques from 4.2.3.

4.5.2 CSP techniques and candidate generation

There is a close analogy between our problem-solving architecture and the (finite-domain) Constraint Satisfaction Problems (CSP) (cf. [Kum]; see also [dK89]). A finite-domain CSP is defined by:

- a finite set of variables each having a finite domain of possible values associated, and
- a finite set of constraints that limit the allowed assignments of values to some subsets of variables.

A solution to a CSP is a complete assignment of values to the variables that does not violate any of the constraints. Usually in CSP one is interested in only one arbitrary "consistent" assignment.

To see the analogy with our candidate generation problem one can associate to each variable and variable domain a choice set, and to each constraint from a CSP the prohibited sets of assignments as (minimal) conflicts. Note that our problem is usually more general than CSP in several ways:

- most important is the requirement of our candidate generation that the consistent candidates proposed *cannot be arbitrary*, i.e. a certain form of plausibility / optimality expressed by the preference and priority has to be ensured;
- another requirement coming from diagnosis is to construct several solutions, when several plausible ones exist, since this underlies the further process of investigation, namely the test-pattern, measurement and repair proposal.[22]

Most of the CSP algorithms select incrementally growing sets of variables for which attempt to build a partial solution. After selecting a new variable and a certain value assignment *forward checking* techniques test (in varying degrees) the consistency of the new partial solution. In case of inconsistency a varying effort is spent to identify the cause of the failure, and different algorithms perform a (more or less dependency directed) form of backtracking (see for instance the backjumping and dynamic backtracking techniques, cf. [Gin93]). Most of the CSP algorithms avoid to store and use the complete

[22] There are also other minor differences , e.g. the elements of our choice sets need not be exclusive (i.e. pairwise inconsistent), and our choice sets need not be disjoint, like it is usually the case in CSP.

set of conflicts found during the search, and compared with our problem solving architecture, may perform unnecessary search effort at the trade of consuming less memory.

We mentioned previously that it makes sense to interleave the selection of the next disjunction / choice set to solve in the candidate generator with the (partial) candidate testing in the RMS. The heuristic that chooses the next disjunction to consider can take advantage of *variable ordering* strategies developed in CSP. The preference order among the elements of a choice set is analogous to a fixed *value ordering* in CSP. Focusing the RMS on a partial candidate (solution) and checking consistency corresponds to forward checking in CSP.

It seems that the CSP community has addressed more the trade-off between the memory consumed and the amount of backtracking performed during the search for a solution. Because in the 2vRMS there is a finer control on the amount of labels and nogoods computed this trade-off seems to have now better chances to be addressed, as a possible future work. In an extreme case, when the 2vRMS is configured to work like a JTMSset, there is no need to store any nogood in the RMS. We mentioned in 4.3.3 that due to the ordered way of investigating the candidate space, the candidate generator can dispense during the search with many of the conflicts that become non-relevant. Different problem-solvers, and variants of "candidate generators" might choose to cache different amounts of conflicts.

When a certain degree of optimality has to be ensured, most of the CSP algorithms employ a "branch-and-bound" strategy, i.e. they *(a)* find a first (arbitrary) solution; *(b)* assert that the next solution has to have a better optimality measure; *(c)* continue the search further. This approach tends to be very sensible on the existence of good variable ordering strategies that select "more important / critical" variables first. In diagnosis, the modes of behavior of the components are such "more important" variables. However, there are usually very many such variables (for each component one), all of them having approximately the same "importance". In such a case it is likely that the "branch-and-bound" search strategy will perform worst than our search strategy because we build "optimal" solutions from the beginning. It is reasonable to believe that the CSP algorithms spend less effort to find the first (not necessarily optimal) solution, since when inconsistencies are found usually only one way of solving the inconsistency is considered. We spend more effort because we keep in our lower bounds all possible alternatives of dealing with the conflict. However, the CSP algorithms must spend

afterwards a considerable effort to find one (several) *optimal* solution(s), i.e. after they find a first solution. In our setting when we find a consistent solution we are guaranteed that it is "optimal".

Nevertheless, we could further improve our algorithms if we observe that not all of the choice-sets affect in an equal measure the optimality of the solution. Except the modes of behavior, where we usually have strong preferences, there may exist other choice sets (disjunctions) in the formulation of the problem, where we have no preference among the elements. In such a case, any solution with respect to these "secondary" choice sets might be as good as any other. We can then split our choice sets into two partitions:

- the *primary* choice sets, whose semantic is as before, and
- the *secondary* choice sets.

In a first step some preferred primary choice assignments could be generated using the techniques described in this chapter. For each primary choice assignment that comes into the focus an attempt to extend it consistently with a secondary choice assignment should be made. This can be done, as it is usually done in CSP, without considering simultaneously all of the possible ways of resolving the conflicts. If a primary choice assignment cannot be consistently extended with any secondary choice assignment this is an indicator that a conflict within the primary choice assignment exists. Hyperresolution rules, used in a similar way as described in 4.3.4, are sufficient for deriving the (minimal) conflicts among the primary choice assignments. Appendix B.3 describes an algorithm that searches one consistent extension of a set of symbols with an assignment for the secondary choice sets.

The introduction of the secondary choice assignments reduces the size of the lower bounds and can positively affect the efficiency of the candidate generation.

4.5.3 Increasing the completeness of the ATMS

Our RMS-candidate-generator architecture may be seen also as a way of increasing the completeness of the ATMS (cf. [dK86b, dK88, dK90a, Dre88, FdK93]). We begin by clarifying the terminology:

- If a certain algorithm is able to detect always the unsatisfiability of any logical theory from a certain class, we say that it is *refutation complete* for that class. If the algorithm signals unsatisfiability only when the

theory is logically unsatisfiable, then we say it is *refutation sound*. For instance, the JTMS, LTMS, ATMS, are refutation sound and complete for Horn theories.

- If a certain algorithm is able to detect always when a formula is entailed by a logical theory, we call it *formula complete*. The soundness is defined as above. The formula completeness and soundness can be relaxed to refer only to certain classes of formulae, for instance we talk about *literal completeness* when the above property holds for any literal.
- In an ATMS-like RMS, one can also talk about the *label completeness, soundness, minimality*, and *consistency* (see 2.3.4). This can be defined with respect to a larger class of logical theories, not only to the Horn ones. Note that, these properties can be regarded as instances of the *formula completeness* (see also the discussion from 2.4).

Among the different "completeness" properties the most important one is the refutation completeness (and soundness). The other degrees of "completeness" should be treated with great care since they can involve prohibitive amounts of work. Clearly, none of the techniques discussed here ever attempted to achieve formula completeness in general.

Neither the JTMS, JTMSset, LTMS, nor the (basic / focusing / lazy) ATMS, achieve any of the above forms of completeness in general. They all achieve the refutation completeness and the positive literal completeness for Horn theories. The ATMS family achieves formula completeness for certain classes of prime implicates in Horn theories.

As noted in [dK90a] (the subject is revisited in [FdK93]), there are two extreme techniques for coping with the incompleteness: One is based on performing search, by introducing, testing and retracting assumptions. The other one is based on pre-compiling the set of prime implicates of a theory. The second approach, followed in [dK90a], is based on the observation that if a set of propositional formulae is converted to their *prime implicates*, then boolean constraint propagation (BCP), the inference procedure underlying the LTMS, is a complete and efficient inference procedure. More precisely, one reaches the refutation completeness, the literal completeness and, of course, the prime implicate completeness. Unfortunately, the number of prime implicates of a formula is often exponential in the size of the formula. Therefore, unless the formula is small, this technique is impractical. Of course, the partial pre-compilation and the search can be combined. In [dK90a] it was

suggested to use the locality of knowledge in particular problem domains in order to guide how many, and which logical formulae to convert to their set of prime implicates. As noted in [dK90a], the search cannot, or should not, be totally eliminated in practice (similar arguments were also stated in the CSP context, as noted in [dK89] and [Mac87]).

The rest of the techniques aiming to increase the logical completeness of the RMSs, e.g. [dK86b, dK88, Dre88], as well as our algorithms for candidate generation, can be regarded as search-based approaches.

In [dK86b] a series of techniques that increase the logical completeness of the ATMS were introduced. They are based, as in our case, on the extension of the ATMS with disjunctions over assumptions (called *chooses*). The extended ATMS from [dK86b] achieved, except the refutation completeness, the node *label completeness and consistency* with respect to the *whole* propositional theory, i.e. including the justifications and the chooses. In our case we avoided to ensure by default even the (detailed) label completeness and consistency with respect to the Horn theory represented by the justifications in the 2vRMS. The additional degree of completeness and consistency comes at a significant additional cost, by integrating several hyperresolution rules on top of the basic ATMS.

In [dK88] the ATMS was extended with negations of assumptions. Very similar to this extension is also the one from [Dre88], where *out*-assumptions were introduced. Based on the negated assumptions, any propositional clause can be encoded. The extensions achieve the refutation completeness and the label consistency with respect to arbitrary propositional theories, but forgo the label completeness. Both extensions embed a nogood inference rule, similar to the RMS implementation of the hyperresolution rule R1 from 4.3.4. Although the label completeness was not achieved, a relaxed but still interesting form of label completeness was ensured by the extended ATMSs from [dK88, Dre88]: namely, the label completeness with respect to the *interpretation environments*.

Interpretation construction

The techniques of [dK86b, dK88, Dre88] were designed for the basic ATMS. The basic ATMS works in all possible consistent contexts in parallel. The maximal consistent environments are called *interpretations* (cf. [dK86b]). Most of the problem-solvers based on the basic ATMS constructed as solutions the set of interpretations. Algorithms that construct the whole set of interpre-

tations can be found in [dK86b] and in [FdK93] pp. 436-439. A simple (but not very efficient) scheme for constructing the interpretations can be based on the ATMS label propagation, where for each disjunction (choice set) a new "disjunction" node justified by each member of the choice set is created, and additionally a "goal" node justified by the conjunct of the "disjunction" nodes is created.[23] The label of the "goal" node contains the minimal sets of consistent assumptions that solve all of the positive disjunctions. These environments can then be extended in all possible ways with assumptions until the border with inconsistent environments in the environment lattice is reached. There is a certain correspondence between the *interpretations*, more precisely, between the environments from the label of the "goal" node, and our *consistent candidates*: both must solve (imply) all of the positive clauses (chooses, or choice sets). The difference between them is that, while the environments from the label of "goal" are *minimal support clauses* of the conjunct of positive disjunctions, our consistent candidates are just *support clauses* of the conjunct of disjunctions, but not necessarily minimal.[24]

As argued in [dK88], the extended ATMS from [dK88], and thus also the one from [Dre88], achieve the label completeness with respect to the interpretation environments, i.e. every interpretation I in which a node n holds is a superset of some environment from n's label. Although our consistent candidates are not interpretations, the (detailed) label completeness in the 2vRMS & candidate generator architecture holds with respect to the consistent candidates too. This service is ensured at request, i.e. only after focusing the 2vRMS on the candidate and after querying the detailed label. However, the 2vRMS does not need to query the detailed label to decide that the node holds given a consistent candidate, this information is provided, once focusing on that candidate, by the focus labels.

The interpretation construction in the ATMS was one of the most expensive services because it attempted to build all such maximal consistent environments. As opposed to it, our candidate generator, attempts to construct incrementally only a few candidates that satisfy the preference and the priority requirements.

[23] One can see the relationship with the construction of the *minimal extension bases* in the nmATMS (cf. [Dre90], see also Appendix A.8), where the disjunctions solved are $p \vee p_{out}$, and the "goal" node is noted Φ.

[24] A sufficient condition for reaching the above minimality is when the choice sets are disjoint and there are no implied assumptions.

4.5.4 Candidate generation in model-based diagnosis

While there are several papers that discuss focusing the ATMS and the constraint / value propagation (cf. [dK91, dK92b, DF90, FdK88, Gol91, Ham91a]), only a few discuss candidate generation in more detail.

Reiter (cf. [Rei87]) introduced the HS-tree for the computation of the diagnostic candidates. The HS-tree algorithm, improved later in [GSW89], computes the whole set of *minimal diagnoses*. de Kleer and Williams used in GDE (cf. [dKW87]) an algorithm that computed only a subset of the minimal diagnoses, namely some of the most probable ones. Mozetic (cf. [Moz92]) even gave a polynomial algorithm for the computation of an arbitrary minimal diagnosis - however, we think there is no practical relevance for computing just an arbitrary minimal diagnosis: Mozetic's algorithm could return a multiple-fault diagnosis even when single-fault diagnoses exist.

The concept of minimal diagnosis is only appropriate when one does not use knowledge about the fault modes (cf. [SD89]). Heuristic information about the frequent failures is usually used by human diagnosticians, and is essential for focusing on the most plausible diagnoses and for achieving efficiency in the process of investigation (i.e. observation / probing, and test-pattern proposal). The increase of expressiveness due to the consideration of several behavioral modes also leads to more dramatic complexity increases of the diagnostic tasks. In this context, as noted in [dKW89, dK91], focusing on a *small* number of plausible candidates became a key feature of the diagnostic engines.

Sherlock (cf. [dKW89, dK91]) focused on a small number of the most probable candidates (the "leading diagnoses"). There are, however, to the knowledge of the author, no published details about the algorithms for candidate generation used by Sherlock. It uses probably an algorithm that constructs in a best-first manner some interpretations in an ATMS. Our focusing strategy is to focus on the preferred diagnoses having the highest priority. The priority can be specified using the probability of the mode assignments, as we have done in practice (see some earlier reports of our work in [TI94a, TI94b]). In this case the proposed candidates are some most probable candidates chosen from the preferred ones, a concept very close to the one of the leading diagnoses. However, the most probable diagnoses do not coincide with the most probable preferred diagnoses in general. The reason is that some of the first k most probable diagnoses may not be preferred.

The closest relationship with our work has the work of Dressler and

Struss described in [DS92, DS94]. Their initial approach (cf. [DS92]), which proposed to compute all the preferred diagnoses, was recently improved in [DS94], where they also focus on a subset of the preferred diagnoses as we do. Dressler and Struss use default logic to characterize the preferred diagnoses and the nmATMS (cf. [Dre88, Dre90]) to compute them. A choice set $C = \{a_1, a_2, \ldots, a_n\}$, $a_1 <_C a_2 <_C \ldots <_C a_n$, is expressed in their formalism by the set of normal defaults:

$$: a_1/a_1; \quad \neg a_1 : a_2/a_2; \quad \ldots \quad \neg a_1 \wedge \neg a_2 \wedge \ldots \wedge \neg a_{n-1} : a_n/a_n.$$

The process of constructing the preferred diagnoses in the nmATMS uses an algorithm similar to the ones used for interpretation construction, but the maximal sets of consistent assumptions are generated in an incremental way. A similar pruning as that achieved by us (see Lemma 4.3.14, and Lemma 4.3.16), is also achieved in the implementation based on the nmATMS. The label of a special nmATMS node (noted Φ) is used to generate the candidate diagnoses and plays a similar role as our lower bounds. In the nmATMS, the justifications installed as a consequence of the *nogood inference rule* and the label propagation play a similar role as the insertion of the immediate successors with respect to a conflict in our algorithm.

Differing from our lower bounds, which may contain inconsistent candidates if those candidates do not have a high enough priority, all the environments from the label of Φ must be consistent with respect to the set of nogoods stored in the nmATMS.

The main advantage of not implementing the candidate generator within the (nm)ATMS is in our opinion the increased flexibility in tuning the control strategy and in using more specific data representations (see 4.3.3 and 4.3.4). For instance:

- we are not forced to impose the consistency of the whole lower bound, not even with respect to the nogoods already found in the RMS;
- the conflicts encoding the exclusive choice sets are implicitly dealt with;
- when testing the consistency we can dispense with considering the whole set of conflicts, and use in the candidate generator only the still relevant conflicts for the lower bound instead;
- the application of hyperresolution rules that increase the formula completeness of the RMS-candidate generator architecture may be controlled in a tighter way;

- the representation of the candidates can store only the "exception" choice sets;
- other special-purpose representations are conceivable, for instance, instead of representing extensionally the set of direct successors with respect to a conflict, an intensional representation that stores only the parent and the conflict could be used in order to further increase the efficiency.

The extension with secondary choice sets from 4.5.2 brings more control over the number of successors considered when a focusing candidate is found inconsistent. This strategy can be implemented in the nmATMS-based architecture of Dressler and Struss, by controlling the order and time of insertion of the justifications that are a consequence of the nogood inference rule. The effect of using a candidate generator with secondary choice sets, would correspond to an application of the nogood inference rule that creates *only one* justification for the nogoods that contain secondary assumptions.

Chapter 5

Model-Based Diagnosis with Dependent Defects

5.1 Introduction

The application of the current model-based diagnostic techniques in practice continues to face several difficulties. Among them, the assumptions that the faults are independent and that diagnosis and repair can be regarded as orthogonal tasks, does not seem to be reasonable. Although the current approaches are able to diagnose multiple faults, they fail to provide satisfactory solutions when there are cascading defects. For several reasons, the diagnostic session may provide as the most probable/preferred/minimal diagnosis one that includes only a subset of the actually broken components.[1] In such a case, the repair phase following diagnosis will be incomplete. The worst

[1] Among the reasons for such an outcome, one can note that:

- there is no guarantee that all the relevant symptoms have been observed. Some relevant symptoms may not even be exhibited in the current situation. Observing them would require the change of the test pattern.
- multiple faults are considered less plausible than single faults – the generation of supersets of a diagnosis explaining the current observations is explicitly avoided by approaches based on the notion of minimal/preferred diagnoses.
- some cascading defects can mask the cause, e.g. the failure $f_A \wedge f_B$ caused by f_A, can have exactly the same observable effects as the failure f_B – in such a case, the only way of discriminating between them requires the replacement of the component B, thus the interleaving of the diagnosis and the repair tasks.

is that, if the repair fails to eliminate the cause of the failure, the existing undetected defects will cause the already repaired components to break again (and again). Simply restarting the diagnosis as an independent task would probably end with the same unsatisfactory answer as before.

The scenario in which one finds a broken component and replaces it, but after a short time one discovers the same component broken again, is relatively common, and more difficult to diagnose.

Casual relations among failures are less frequent in the domain of digital circuits, but they flood the domain of analog circuits. The most common example is that of a fuse, whose correct behavior is exactly to break when there is an overcurrent, probably caused by a failure elsewhere in an electrical circuit.[2] In fact, there is a large range of components which break when exposed to certain abnormal conditions, e.g. bulbs, semiconductor junctions, insulators, resistors, capacitors, and many others.

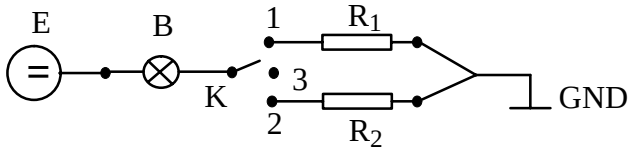

Figure 5.1: A simple electrical circuit with cascading defects

Example 5.1.1 *Consider the toy electrical circuit from Figure 5.1, containing a power-supply E, a bulb B, resistors R_1, R_2 and the switch K. Assume that:* (i) *a normal bulb breaks if exposed to an overcurrent as the one caused by $shorted(R_i)$;* (ii) *we observe $broken(B)$ in the situation $K = 1$. Without modeling the causal relations affecting the change of the bulb's modes, the diagnostic engine cannot infer that in the situation $K = 1$ the fault $shorted(R_1)$ forces a normal bulb to break. There is only one minimal/preferred diagnosis computed without this knowledge: $\{broken(B)\}$, while the (non-minimal / non-preferred) diagnosis $\{broken(B), shorted(R_1)\}$ would be assigned a wrong plausibility measure. Suppose we replace the bulb, but the above symptoms reappear. What information can one extract from that?*

[2] The purpose of this behavior is to protect other more expensive parts of a circuit.

Simply allowing the faults to be intermittent and performing temporal diagnosis is not enough: we are less interested in finding out that at the time point 1 the bulb was correct and at the time point 2 it is broken – we are more interested in the explanation of why the bulb changed its mode: did it actually behave intermittently, or was the mode-change a predictable event in the given situation?

In this chapter we present an approach in which the causes of the mode transitions can be modeled. Apart from a representation formalism that allows to capture the causality of the mode changes, we are interested in avoiding the introduction of unnecessary increases in complexity: the complexity of diagnosis should depend on the complexity of the problem at hand and not on the representation, as far as possible. That means that, there shouldn't be an increase of complexity just due to the use of the extended framework if the problem at hand does not require the newly added features, i.e. if there is no caused defect possible (modeled).

In the proposed approach each component is modeled as a (tiny) finite-state machine. The modes of behavior are regarded as states. This enables to model (at least partially) the causality that drives the mode changes. One can now easily integrate repair actions into the component models. Moreover, it is possible to continue a diagnostic session even after performing some repair actions, without making the implicit assumption that the newly repaired/replaced components are and will be correct forever. The representation can also be used to model dynamic components with memory, though the difficulties of diagnosing dynamic systems are well known and challenging (cf. [HD84, Ham91b, DJD+91]).

The problem of finding the diagnoses is closer to the problem of state identification in an automaton, namely, we map diagnosis to the problem of transition path identification. Our approach offers an attractive degree of prediction reusability if implemented on top of an RMS, namely it achieves prediction reusability across *contexts* - like any RMS-based problem solver - and also across *time*. It also allows the reuse of the mechanisms for candidate generation based on minimal conflicts discussed earlier in this thesis, without introducing unnecessary complexity in the cases when the system under diagnosis behaves statically.

We build the framework in a gradual way:

- The deterministic finite state machine framework (Section 5.3) assumes that all the state transitions that occur during diagnosis can be deter-

ministically predicted, and that one has exact temporal information about the observations and the durations of the mode transitions;

- The pseudo-static framework (Section 5.4) removes the assumption about the availability of exact temporal information about the observations, an assumption that is hard to keep in applications concerning the electrical domain. The pseudo-static framework introduces as well some new presuppositions in order to deal with the underconstrained nature of the new problem. Namely, it is assumed that the system always reaches a steady state in a finite number of mode transitions after each input change. It is also assumed that this steady state is reached very quickly, i.e. before another input change occurs.
- Section 5.5 discusses the possibility of reasoning "backwards" from certain states to possible situations that might have caused that state.

5.2 Repair and testing during diagnosis

As stated in the introduction, in the case of cascading defects the need to interleave diagnosis and repair is higher. Observing the system symptoms after a repair action is a valuable information source and it should not be neglected. Since the approach that we are going to describe in this chapter can capture the causality of the mode changes, it is very easy to model repair actions: some (special) input changes cause some mode transitions from defect modes to correct modes. The cost of changing such controllable inputs could reflect the cost of the repair.

The cases when faulty behavior persists even after performing the repair actions, should shift the focus of the diagnostic engine to other unconsidered alternatives, like, for instance, the existence of some cascading defects.

5.3 The Finite State Machine Framework

5.3.1 Basic presuppositions

We make the following assumptions about the system under diagnosis:

1. *The modeling assumption:* Each component is modeled as a deterministic finite-state machine, i.e. given the state and the input assignment for the current time point, the values of the outputs at the current time point, and the next state can be deterministically predicted. The behavioral mode is regarded as a state variable.
2. *The closed world assumption:* The only state transitions that are allowed to occur during diagnosis are the ones that can be predicted based on the behavioral descriptions and the knowledge of the inputs of the system.
3. *The synchronous composition assumption:* If several state variables could change their value at the time point t, it is assumed that all of them occur simultaneously.

The modeling assumption restricts the attention to the deterministic aspects of behavior of each component. If one ignores the causality of the mode changes it is relatively reasonable to assume that each component has a deterministic model. However, one cannot claim in general that it is feasible to capture all the causes that drive the mode changes. In practice, the causality of the mode changes can be at most partially described. We require that the models capture the deterministic (and frequent) causes of the mode changes, like the repair actions and the operation beyond the admissible operating regions. Of course, there are mode changes that are not captured by the models (for instance aging), but the closed world assumption, a relaxed version of the non-intermittence of modes assumption (cf. [RdKS91]), restricts them to occur only before the start of the diagnostic session. Unexplainable mode changes can represent *primary* causes of other cascading failures. The unexplainable primary causes are not supposed to occur during diagnosis, however, the defects caused by the primary ones and by the inputs applied can, of course, occur during the diagnostic session.

The synchronous composition assumption ensures that the composed behavior of the components is deterministic, given that the individual behavior of the components is deterministic.

5.3.2 Basic concepts

The definition of the system under diagnosis makes the one from Chapter 3 (cf. Definition 3.2.1) more particular, assuming that the components are finite state machines. It is assumed that the description of the behavior of a component can be done using constraints / equations / logical statements among a set of variables characterizing the component. Among these variables, apart from the usual inputs and outputs, there are two specified subsets with a special semantics: a subset of variables, $StateVars_{C_i}$, is used to denote the current state of the component C_i, while the variables from $NextStateVars_{C_i}$ are used to predict the next state of the component. Between the variables from $StateVars_{C_i}$ and the ones from $NextStateVars_{C_i}$ there is a one-to-one correspondence that is given in the system description.

Definition 5.3.1 (System) *A system is a tuple:*

$$Sys = (SD, Comps, CompVars, SysInputs), where:$$

- *SD describes the structure of the system and the components' behavior - it is a set of first-order sentences;*
- *$Comps$ is the set of components, i.e. a set of constants;*
- *$CompVars$ associates with each component C_i: the set of state variables $StateVars_{C_i}$, the set of variables used to predict the next state $NextStateVars_{C_i}$, other variables relevant for the component $OVars_{C_i}$, and a function $\mathcal{F}_{C_i} : NextStateVars_{C_i} \rightarrow StateVars_{C_i}$ providing a one-to-one association between the next-state and the state variables: $CompVars =$*

  $$\{(C_i, StateVars_{C_i}, NextStateVars_{C_i}, OVars_{C_i}, \mathcal{F}_{C_i}) \mid C_i \in Comps\};$$

- *$StateVars_{C_i}, NextStateVars_{C_i}, OVars_{C_i}$, and $SysInputs$ are disjoint sets of variables.*

As can be seen, SD embeds no concept of time. The behavioral description is *temporally generic*: it describes the relation among the inputs, other variables, states and the next state for a generic time point.

Example 5.3.2 *Consider the description of a fuse F using two modes of behavior: correct – ok and broken – bk. Suppose that the fuse breaks when*

the current exceeds a certain limit I_B. In addition to the usual variables holding the current and the voltage at the fuse's terminals the description uses one state variable (s), respectively one next-state variable (s_{next}) for the mode of behavior of F. The behavior description could look like:

$$\begin{array}{lcll} s = ok & \Rightarrow & (i_{left} = i_{right} \wedge u_{left} = u_{right}) & \wedge \quad (|i_{left}| > I_B \Rightarrow s_{next} = bk) \\ & & & \wedge \quad (|i_{left}| \leq I_B \Rightarrow s_{next} = ok); \\ s = bk & \Rightarrow & i_{left} = i_{right} = 0 \wedge s_{next} = bk; & \end{array}$$

If one wants to model the action of repairing a fuse, one possibility is to add another input to the model of a fuse, say rep, whose value being true controls the transition from the broken mode to the correct mode:

$$s = bk \Rightarrow (i_{left} = i_{right} = 0) \wedge (rep \Rightarrow s_{next} = ok) \wedge (\neg rep \Rightarrow s_{next} = bk).$$

We use the following additional notations:

- The global state of the system is given by the vector of component state assignments. We note with $StateVars$, $NextStateVars$ the union of the sets $StateVars_{C_i}, respectively, NextStateVars_{C_i}$.
- The global association between the system's state, respectively, next state variables is noted with: $\mathcal{F} : NextStateVars \rightarrow StateVars$,

$$\mathcal{F}(s') = \begin{cases} \cdots \\ \mathcal{F}_{C_i}(s'), & \text{if } s' \in NextStateVars_{C_i}; \\ \cdots \end{cases}$$

- A *state assignment* is a set $d = \{..s_i = v_j..\}$ where $s_i \in StateVars$. Analogously, input assignments are defined. A *complete* state (input) assignment assigns exactly one value to each variable from $StateVars$ (*SysInputs*). We use lowercase letters to denote partial state/input assignments and uppercase letters to denote complete assignments.
- An *input-observation sequence* is a sequence[3] $((i_1, o_1), \ldots, (i_k, o_k))$, where i_j are input assignments and o_j are sets of first order sentences.

What can one do with such descriptions? Assume an RMS-based prediction engine in which the constraints of SD are represented and the input assignments and the state assignments are RMS assumptions. The following questions can be answered:

[3]As usual, we sometimes note the sequences by enumerating their elements, i.e. $(e_1, e_2, \ldots, e_n)$, or using the dot notation, i.e. $(e_1.(e_2, \ldots, e_n))$. We note the empty sequence with ε.

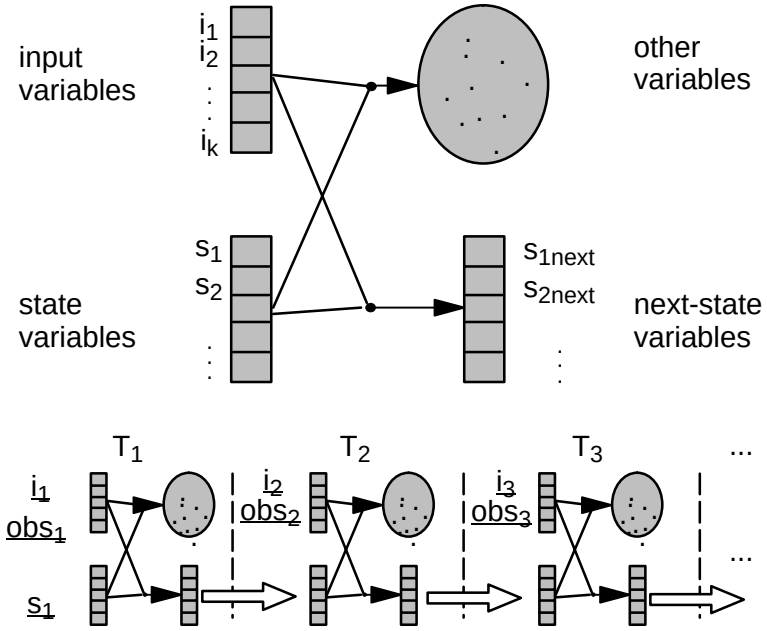

Figure 5.2: Prediction across several time points.

- *the consistency check:* are an input assignment I, a state assignment D and some observations o consistent with SD, i.e. is $I \cup D \cup o \cup SD$ consistent?
- *the entailment check:* given an input assignment I and a state assignment D, what other variable assignments are entailed? In particular, what assignments to the *next-state* variables are predicted?
- *the minimal-support-set computation:* given a certain variable assignment (in particular a next state assignment), under which minimal conditions - in terms of inputs and state assignments assumed - is the variable assignment predicted?

The RMS network "spreads the activation" from the currently enabled state and input assumptions to the rest of nodes. The next state can be retrieved by querying the next-state nodes "active" in the current time point context. The sequence of time points is not explicitly represented in the RMS network. To make predictions across several time points the diagnostic engine has to "simulate" the automaton by executing the cycle of focus changes corresponding to the sequence of inputs and states (see Figure 5.2). Prediction

reusability across time is achieved the same way as prediction reusability across contexts in an RMS.

Definition 5.3.3 (Next System State) *Let Sys be a system and $\mathcal{A}$ be a set of first order sentences, such that: $SD \cup \mathcal{A}$ is consistent. The next system state for $(Sys, \mathcal{A})$ is a state assignment: $(s_i = v_j) \in NextState(Sys, \mathcal{A}) \Leftrightarrow$*

$$SD \cup \mathcal{A} \models (s'_i = v_j), \quad s'_i \in NextStateVars \quad \text{and} \quad s_i = \mathcal{F}(s'_i).$$

As defined above, the sets $NextState(Sys, \mathcal{A})$ contain those assignments to state variables that are deterministically predicted for the next time point under some conditions (assumptions) $\mathcal{A}$. The assumption that a system is a deterministic finite state machine can be expressed formally:

Definition 5.3.4 (FiniteState Machine) *A system Sys is a deterministic finite state machine (FSM) if and only if:*

1. *each state variable takes values from a finite domain, and*
2. *for any complete state assignment D and for any complete input assignment I the next system state $NextState(Sys, I \cup D)$ is a* complete *state assignment if $SD \cup D \cup I$ is consistent.*

Definition 5.3.5 (Path Description; Transition Path) *Let Sys be a finite state machine. A sequence $((d_1, i_1, o_1), (d_2, i_2, o_2), \ldots, (d_k, i_k, o_k))$, where d_j are state assignments, i_j are input assignments, and o_j are sets of first order sentences is a* path description *for Sys if and only if:*

$$\begin{array}{ll} \forall j, & 1 \leq j \leq k, \quad SD \cup i_j \cup o_j \cup d_j \text{ is consistent, and} \\ \forall j, & 1 \leq j < k, \quad d_{j+1} = NextState(Sys, i_j \cup o_j \cup d_j). \end{array}$$

A transition path *is a path description in which all the state and the input assignments are complete.*

A path description characterizes several transition paths that extend the state and the input (partial) assignments to "consistent" complete assignments. The observations o_j that are part of the descriptions constrain the set of transition paths characterized by a path description. It can be seen that, due to the deterministic framework, for any state assignment d and for any input-observation sequence *ioseq* there exists at most one path description that has the initial state assignment d and takes the inputs and observations from *ioseq*.

Definition 5.3.6 (consistency / conflicts) *A state assignment d_1 is* consistent *with an input-observation sequence $((i_1, o_1), \ldots, (i_k, o_k))$ if and only if there exists a path description having the form $((d_1, i_1, o_1), \ldots, (d_k, i_k, o_k))$. d_1 is consistent with the empty sequence if and only if $SD \cup d_1$ is consistent.*

A state assignment is a conflict *for an input-observation sequence if and only if it is not consistent with that input-observation sequence. A conflict c is* minimal *if and only if no proper subset of c is a conflict.*

Property 5.3.7 *Let Sys be a system, d be a state assignment and ios be an input-observation sequence for Sys. Then:*
d is consistent with ios iff d includes no minimal conflict for ios.

Definition 5.3.8 *Let $seq = (e_1, \ldots, e_k)$, $seq' = (e'_1, \ldots, e'_k)$, be two sequences of length k. We note:*

$$seq \subseteq seq' \quad iff \quad e_j \subseteq e'_j \text{ for all } j, 1 \leq j \leq k.$$

If the elements e_j, e'_j are pairs, i.e. $e_j = (i_j, o_j)$, $e'_j = (i'_j, o'_j)$, then we note $(i_j, o_j) \subseteq (i'_j, o'_j)$ iff $i_j \subseteq i'_j$ and $o_j \subseteq o'_j$. If the elements of the sequences are triples, we note $(d_j, i_j, o_j) \subseteq (d'_j, i'_j, o'_j)$ iff $d_j \subseteq d'_j$, $i_j \subseteq i'_j$ and $o_j \subseteq o'_j$.

Property 5.3.9

1. *Any superset of a conflict for an input-observation sequence is a conflict for that input-observation sequence.*
2. *If c is a conflict for ios, then c is a conflict for any input-observation sequence $ios' \cdot tail$, where $ios \subseteq ios'$, $tail$ is an arbitrary input-observation sequence, and "$\cdot$" denotes the concatenation of two sequences.*

The proofs are given in Appendix D. We reformulate the diagnostic problem as follows:

Definition 5.3.10 (the FSM Diagnostic Problem) *Given an input-observation sequence $Ioseq = ((I_1, o_1), \ldots, (I_k, o_k))$, where I_j are complete input assignments, find all (or some "preferred") transition paths consistent with $Ioseq$, i.e. having the form $((D_1, I_1, o_1), \ldots, (D_k, I_k, o_k))$.*

As a strategy for finding the consistent transition paths we suggest here to search for the acceptable initial state assignments of the transition paths. For each candidate initial state assignment D_1 we try to construct a transition path consistent with the input-observation sequence in an incremental

manner, i.e. first taking into account only the first input-observation pair, then only the first two, and so on. Whenever a transition path cannot be consistently extended with the next input-observation pair a minimal conflict among the last state assignments is found. This minimal conflict is then mapped back to a minimal conflict among the initial state assignments.

In order to characterize how to map back the conflicts we need a function that has the "inverse" role of $NextState$. Namely, we want to find out under which *minimal* conditions, in terms of states, inputs and observations, a certain target state can be reached in one transition.

Definition 5.3.11 ("backward" transition set) *Let Sys be a system and d' be a state assignment for Sys. Define the function:* $BackTranz(Sys, d') :=$

$$\{(d, i, o) \mid \forall d'', i'', o'' \quad (d \subseteq d'' \wedge i'' \subseteq i \wedge o'' \subseteq o \wedge (SD \cup d'' \cup i'' \cup o'' \text{ is consistent }) \Rightarrow d' \subseteq NextState(Sys, d'' \cup i'' \cup o'')\}.$$

In order to compute the function $BackTranz$ we compute some minimal support clauses for conjunctions of literals (see Definition 2.3.1 from Section 2.3.5). In [RdK87] the sets $MinSupp(M, \Sigma)$ were characterized in terms of the prime implicates of Σ, but M was supposed to be a clause (i.e. a disjunction of literals). In our setting we are interested in M being a conjunction of literals. As the next theorem shows, the minimal support clauses of a conjunct can be determined using the minimal support of a literal with respect to a slightly modified Σ.

Theorem 5.3.12 *Let Σ be a set of clauses and M be a* conjunction *of literals. Let ϕ be a new positive literal not appearing in Σ or in M. Then:*

$$MinSupp(M, \Sigma) = MinSupp(\phi, \Sigma \cup \{M \Rightarrow \phi\}) - \{\neg\phi\}$$

The results of [RdK87] can then be reused in our setting. We can now compute $BackTranz(Sys, d')$ based on computing some minimal support clauses for the conjunct of next-state literals corresponding to the target state assignment d':

Theorem 5.3.13 *Let Sys be a system and d, d' be state assignments, i be an input assignment for Sys, and o be a set of first-order sentences. Then:*

$$(d, i, o) \in BackTranz(Sys, d')$$
$$\Leftrightarrow$$
$$NegCl(d \cup i \cup o) \in MinSupp(Conj(\mathcal{F}^{-1}(d')), SD),$$

where: $NegCl(\{l_1, l_2, \ldots\}) = (\neg l_1 \vee \neg l_2 \vee \ldots)$,
$Conj(\{l_1, l_2, \ldots\}) = (l_1 \wedge l_2 \wedge \ldots)$,
$\mathcal{F}^{-1}(\{..s_i = v_j..\}) = \{..\mathcal{F}^{-1}(s_i) = v_j..\} = \{..s'_i = v_j..\}$.

We can now characterize, in a recursive manner, the conflicts for input-observation sequences:

Theorem 5.3.14 (conflict mapping) *Let Sys be a system, $ioseq$ be an input-observation sequence for Sys and (i, o) be an input-observation pair for Sys. Let c be a state assignment for Sys. Then:*

$$c \text{ is a conflict for the sequence } ((i, o).ioseq) \quad \Leftrightarrow$$

either:

1. $SD \cup i \cup o \cup c$ *is not satisfiable, or*
2. *there exists c'' a minimal conflict for $ioseq$, and $i' \subseteq i$, $o' \subseteq o$, $c' \subseteq c$, such that $(c', i', o') \in BackTranz(Sys, c'')$.*

Corollary to Theorem 5.3.14 *Let Sys be a system, $ioseq$ be an input-observation sequence for Sys and (i, o) be an input-observation pair for Sys. Let c be a state assignment for Sys. Then:*

$$c \text{ is a minimal conflict for the sequence } ((i, o).ioseq) \quad \Leftrightarrow$$

c is a minimal set such that either:

1. $SD \cup i \cup o \cup c$ *is not satisfiable, or*
2. *there exists c' a minimal conflict for $ioseq$, and $i' \subseteq i$, $o' \subseteq o$ such that $(c, i', o') \in BackTranz(Sys, c')$.*

The characterization of the conflicts in terms of minimal support clauses is particularly interesting because as Reiter and de Kleer showed in [RdK87] this is what the ATMS systems are computing (see Section 2.3.5). For instance, in order to compute the function $BackTranz(Sys, d)$ with an ATMS, if the state, input and observation assignments are represented as assumptions, one simply has to: *(i)* find the assignment to the next state variables corresponding to d, i.e. $\mathcal{F}^{-1}(d)$; *(ii)* build a temporary node ϕ (as in Theorem 5.3.12) and justify it with the conjunct of the $\mathcal{F}^{-1}(d)$ nodes; *(iii)* retrieve the label of ϕ.

Since we can now compute conflicts for input-observation sequences, we can also generate consistent initial state assignments for the input-observation sequences. The generation of the consistent initial state assignments using the minimal conflicts can be done using the mechanisms described in the previous chapter (i.e. using preferences and priorities).

The next function tests a candidate initial state assignment against an input-observation sequence and returns either the associated path description, or a set of conflicts among the candidate's state assignments:

Function TestFSMCand $(Sys, d, ioseq)$
Sys is a FSM; d is a candidate initial state assignment;
$ioseq = ((i_1, o_1), \ldots, (i_k, o_k))$ is an input-observation sequence;
Returns: a pair $(bool, obj)$, where obj is either:
(i) a path description $((d, i_1, o_1), \ldots, (d_k, i_k, o_k))$ - if *bool* is True;
(ii) a set of conflicts included in d - if *bool* is False.

(1) Initialize TP with the empty sequence ε ;
(2) **for** $j = 1$ to k **do:**
  (3) **if** $SD \cup i_j \cup o_j \cup d$ is inconsistent **then**
    (4) **return** $MapFSMConflicts(Sys, d, i_j, o_j, TP)$;
  (5) append $((d, i_j, o_j))$ to TP; $d := NextState(Sys, i_j \cup o_j \cup d)$;
  endfor
(6) **return** $(True, TP)$.

Function MapFSMConflicts $(Sys, d_{cfl}, i_{cfl}, o_{cfl}, TP)$
Sys is a FSM; TP is a path description (transition path);
$d_{cfl}, i_{cfl}, o_{cfl}$ are state, input and observation assignments that should extend TP, but which are inconsistent with SD. Returns: a pair $(False, CflSet)$, where $CflSet$ is a set of minimal conflicts included in the initial state assignment of TP.

(1) let $CflSet$ be the set of minimal conflicts for $((i_{cfl}, o_{cfl}))$ included in d_{cfl};
(2) **while** TP is not the empty sequence **do:**
  (3) remove (d, i, o), the last element from the sequence TP ;
  (4) $MapBack := \{c \subseteq d \mid \exists (i' \subseteq i, o' \subseteq o, c' \in CflSet) :$
    $(c, i', o') \in BackTranz(Sys, c')\}$;
  (5) let $CflSet$ be the set of minimal elements w.r.t. set inclusion of $MapBack$;
  endwhile
(6) **return** $(False, CflSet)$.

5.4 The Pseudo Static Framework

5.4.1 Basic presuppositions

The FSM framework assumes that the input for a diagnostic problem is an input-observation sequence $((i_1, o_1), \ldots, (i_k, o_k))$. This is a strong assumptions for many application problems since it assumes precise temporal information about the observations and about the duration of the state transitions.

In electrical circuits the changes due to the cascading defects are usually very fast. They also occur asynchronously. Moreover, there is an order of magnitude difference between the time granularity at which the user performs (measurement / test / repair) actions and the granularity at which the internal state changes occur. The pseudo static framework builds on the FSM framework by adding more presuppositions about the system under diagnosis in order to address the uncertainty of the temporal information about the observations:

4. *The pseudo-static assumption:* The system can be modeled as a finite state machine such that: under any initial setting of inputs and states the system will go through a finite sequence of state transitions until it will finally reach a steady state, in which it would remain for ever, provided that no input changes its value.
5. *The time granularity assumption:* The granularity Δ at which input changes and user actions occur is order of magnitude higher than the granularity δ at which state transitions occur ($\Delta >> \delta$).

The above assumptions are made in order to ensure that the user actions and the observations are performed only in the steady states. This assumption seems to hold in electrical domains, where the changes due to cascaded defects are very fast, but it may not be natural for other domains.[4] Since the mode changes occur asynchronously and are extremely fast in electrical circuits one cannot hope to have timing information about the occurrence of these events. In order to deal with this timing problem we made the restriction that the observations refer to the equilibrium states only.

[4] In some cases when the time granularity assumption is not natural, it could be enforced artificially by postponing the user actions until the system reaches the equilibrium.

5.4.2 Basic concepts

The application of the results developed within the FSM framework to the diagnosis of systems that obey the pseudo-static and the time granularity assumptions, leads to some subtle difficulties. These are mainly due to the uncertainty about the exact time-span of the observations. The time when an observation is valid is constrained by the requirement that the system finds itself into a steady state. But, since the initial state of the system is not known, one cannot surely state how many state transitions have occurred between the initial state and the equilibrium state. We will come back at this discussion later.

The pseudo-static assumption can be expressed formally:

Definition 5.4.1 (Pseudo-Static System) *Let Sys be a FSM. Sys is a pseudo-static system (PSS) if and only if there exists an integer T such that:*[5] *for any transition path having the form*

$$((D_1, I, \{\}), (D_2, I, \{\}), \ldots, (D_T, I, \{\}), (D_{T+1}, I, \{\})),$$

we have $D_T = D_{T+1}$. The smallest integer T having the above property is called the transition order *of the system.*

The time granularity assumption is reflected in the reformulation of the diagnostic problem:

Definition 5.4.2 (the PSS Diagnostic Problem) *Given an input-observation sequence $Ioseq = ((I_1, o_1), \ldots, (I_k, o_k))$, where I_j are complete input assignments, find all (or some "preferred") transition paths having the form:*

$$\begin{array}{llll}
((D_{11}, I_1, \{\}), & (D_{12}, I_1, \{\}), & \ldots, & (D_{1(n_1-1)}, I_1, \{\}), \quad (D_{1n_1}, I_1, o_1), \\
(D_{21}, I_2, \{\}), & (D_{22}, I_2, \{\}), & \ldots, & (D_{2(n_2-1)}, I_2, \{\}), \quad (D_{2n_2}, I_2, o_2), \\
 & & \ldots & \\
(D_{k1}, I_k, \{\}), & (D_{k2}, I_k, \{\}), & \ldots, & (D_{k(n_k-1)}, I_k, \{\}), \quad (D_{kn_k}, I_k, o_k))
\end{array}$$

such that:

$$\begin{array}{ll}
\forall i, 1 \leq i \leq k & D_{in_i} = NextState(Sys, D_{in_i} \cup I_i), \quad \textit{and} \\
\forall i, 1 \leq i < k & D_{in_i} = D_{(i+1)1}.
\end{array}$$

[5] For any pseudo-static system Sys, $T \leq \mathcal{N}(Sys)$, where $\mathcal{N}(Sys)$ denotes the total number of states of the system – that is (upper bounded by) the product of the number of states of system's components.

As can be seen, the observations and the input changes are asserted only in the equilibrium states of the PSS. Since we assumed that there is an order of magnitude difference between the time granularity with which intern state changes occur and the time granularity with which observations and input changes occur, one can repeat for an arbitrary number of times the equilibrium states in a legal transition path and still obtain a legal transition path. However, we build only the shortest transition paths obeying the above constraints. The changes required by the function $TestFSMCand$ presented in 5.3.2 are straightforward:[6]

Function TestPSSCand $(Sys, D, ((I_1, o_1), \ldots, (I_k, o_k)))$
Return a transition path or a set of conflicts included in D

(1) Initialize TP with the empty sequence ε;

(2) **for** $j = 1$ to k **do:**

  (3) **repeat**

    (4) **if** $SD \cup I_j \cup D$ is inconsistent **then**

      (5) **return** $MapPSSConflicts(Sys, D, I_j, \{\}, TP)$;

    (6) $D_{old} := D; D := NextState(Sys, I_j \cup D)$;

    (7) **if** $D \neq D_{old}$ **then** append $((D_{old}, I_j, \{\}))$ to TP;

  until $D = D_{old}$;

  (8) **if** $SD \cup I_j \cup o_j \cup D$ is inconsistent **then**

    (9) **return** $MapPSSConflicts(Sys, D, I_j, o_j, TP)$;

  (10) append $((D, I_j, o_j))$ to TP; $D := NextState(Sys, I_j \cup o_j \cup D)$;

  endfor

(11) **return** $(True, TP)$.

It is less obvious, however, that also the function that maps the conflicts has to be changed. The requirement that the system reaches an equilibrium state does not guarantee that the dependencies on the initial state also reach the equilibrium. The minimal conflicts among the initial state assignments of D_1 can depend on the length of the transition paths, e.g. one can find different conflicts among the assignments of D_1 when considering the transition

[6] In this version of the function the candidate state assignment (D) and the input assignments (I_j) from the input-observation sequence must be complete, otherwise it is more problematic to ensure that the observations are asserted in the steady states of the PSS.

paths:

$$\begin{array}{llll} ((D_1, I, \{\}), & (D_2, I, \{\}), & \ldots & (D_n, I, o)); \\ ((D_1, I, \{\}), & (D_2, I, \{\}), & \ldots & (D_n, I, \{\}), \quad (D_n, I, o)); \\ ((D_1, I, \{\}), & (D_2, I, \{\}), & \ldots & (D_n, I, \{\}), \quad (D_n, I, \{\}), \quad (D_n, I, o)); \end{array}$$

where D_n is the equilibrium state under the input I. The difference arises from the fact that each transition path states not only the final state, but also the exact number of state transitions that occurred. We cannot say exactly after how many state transitions the observations started to be valid since we do not know the initial state and we assumed that we do not have exact timing information about the observations. However, there is an upper bound of the number of transitions after which the observations are valid irrespective of the initial state, namely after at most T transitions (cf. definition 5.4.1) counted from the last input change. The function $MapPSSConflicts$ takes this observation into consideration: at each point the function $ExtendPath$ checks if the current state (D) is an equilibrium state in the transition path; if so it acts "as if" more elements having the form $(D, I, \{\})$ were present in the transition path, up to a maximum of T points having the same input.

Function MapPSSConflicts $(Sys, D_{cfl}, I_{cfl}, o_{cfl}, TP)$
$(D_{cfl}, I_{cfl}, o_{cfl})$ should extend TP, but are inconsistent with SD. Returns a set of minimal conflicts included in the initial state of TP.

(1) let $CflSet$ be the set of minimal conflicts for $((I_{cfl}, o_{cfl}))$ included in D_{cfl};

(2) $CflSet := ExtendPath(Sys, D_{cfl}, I_{cfl}, TP, CflSet)$;

(3) **while** TP is not the empty sequence **do:**

  (4) remove (D, I, o), the last element from TP;

  (5) $MapBack := \{c \subseteq D \mid \exists (i' \subseteq I, o' \subseteq o, c' \in CflSet) :$
  $(c, i', o') \in BackTranz(Sys, c')\}$;

  (6) let $CflSet$ be the set of minimal elements w.r.t. set inclusion of $MapBack$;

  (7) $CflSet := ExtendPath(Sys, D, I, TP, CflSet)$;

  endwhile

(8) **return** $(False, CflSet)$.

Function ExtendPath $(Sys, D, I, TP, CflSet)$
Sys is a PSS of transition order T; $CflSet$ is a set of conflicts included in D; Extends if necessary TP with more triples $(D, I, \{\})$ if D is a steady state under the input I and maps the conflicts back.

(1) **if** $SD \cup I \cup D$ is consistent and $D = NextState(Sys, I \cup D)$ **then**
  ;; extend transition path
  (2) let l be the length of the final subsequence of TP having the input I;
  (3) **repeat**
    (4) $MapBack := \{c \subseteq D \mid \exists(i' \subseteq I, c' \in CflSet) :$
        $(c, i', \{\}) \in BackTranz(Sys, c')\}$;
    (5) remove from $MapBack$ the elements that are not minimal w.r.t. set inclusion ;
    (6) $OldSet := CflSet; CflSet := MapBack; l := l + 1;$
    until $l + 1 \geq T$ or $CflSet = OldSet$;
(7) **return** $CflSet$.

It seems, however, that in most of the cases (if not always) one need not consider transition paths having T transitions under each input in order to compute the correct minimal conflicts. It is easy to prove that when mapping back a conflict on transition paths having the form $((D, I, \{\}), \ldots, (D, I, \{\}))$ and having an increasingly greater length, the sets computed either oscillate in a cycle, or they converge to an "equilibrium" value after a finite number of steps.

Property 5.4.3 *Let Sys be a PSS. Let I be an input assignment, D be a steady state under the input I and o be a set of first order sentences such that $SD \cup I \cup D \cup o$ is inconsistent. Let TP_j be the family of transition paths: $TP_j = ((D, I, \{\}), \ldots, (D, I, \{\}), (D, I, o))$, where TP_j has the length $j \geq 1$. Let $CflSet_j$ be the set of minimal conflicts for the initial state of TP_j. Then: there exist two finite integers $L \geq 1, P \geq 1$ such that $\forall j, j \geq L$: $CflSet_{(j+P)} = CflSet_j$.*

All the real examples that we considered so far have the "convergence" property, i.e. the minimal "period" P from Property 5.4.3 is 1. It is, however, still a topic of research to define formally under which conditions the convergence is guaranteed. The termination criterion using the convergence (see the termination condition of the repeat-until statement of $ExtendPath$) is more convenient than the one using the system's transition order, since the last one seems hard to compute automatically (if possible at all cf. Definition 5.4.1).

5.5 Searching for the primary causes

This section attempts to explore "what happened before the diagnosis started". Being limited to the time period of the diagnostic session, the transition paths constructed within the FSM / PSS frameworks can explain only the faults that were forced to occur during the diagnostic session. However, these transition paths cannot explain why the components that were already defect at the beginning of the diagnosis broke, neither can they indicate what the primary[7] cause of failure was.

Definition 5.5.1 (possible explanation) *A path description*

$$exp = ((d_1, i_1, o_1), \ldots, (d_k, i_k, o_k))$$

is a possible explanation *for a state assignment m, i.e. $exp \in Expl(Sys, m)$ iff $m \subseteq NextState(Sys, i_k \cup o_k \cup d_k)$. A state assignment is a* cause *for m iff it is the initial state assignment of an explanation of m.*

If we search for the explanations of the defects present at the beginning of the diagnostic session, it is reasonable to assume that we are not interested in the explanations that require non-empty observation sets (we allowed non-empty observation sets in the above definition from reasons of generality). Moreover, we are interested only in those explanations that are consistent with the observations made during the diagnosis:

Definition 5.5.2 (consistent explanations) *Let Sys be a FSM, $ioseq$ be an input-observation sequence, and m be a state assignment. Define the set of explanations of m consistent with $ioseq$:*

$$Expl(Sys, m, ioseq) := \{((d_1, i_1, o_1), \ldots, (d_k, i_k, o_k)) \in Expl(Sys, m) \mid NextState(Sys, i_k \cup o_k \cup d_k) \text{ is consistent with } ioseq\}.$$

Of course, if m is not consistent with $ioseq$, there can be no explanation for m consistent with $ioseq$. Also, it is easy to see that $Expl(Sys, m, \varepsilon) = Expl(Sys, m)$. We partition the explanations according to their length:

$$Expl_k(Sys, m, io) := \{e \in Expl(Sys, m, io) \mid length(e) = k\}.$$

Next we characterize the consistent explanations in terms of minimal explanations:

[7] By primary cause we understand here a cause that cannot be further explained within the given model of the world.

Definition 5.5.3 (minimal explanations) *Let Sys be a FSM, and m be a state assignment. The* length-k minimal explanations *of m are a subset of the length-k consistent explanations of m: $MinExpl_k(Sys, m, io) :=$*

$$\{e \in Expl_k(Sys, m, io) \mid \forall e' \in Expl_k(Sys, m, io) : e' \subseteq e \Rightarrow e' = e\}.$$

The following is an immediate consequence of the definitions:

Property 5.5.4 $MinExpl_1(Sys, m, io) = \{((d, i, o)) \mid$

$$(d, i, o) \in BackTranz(Sys, m), \text{ and } d \text{ is consistent with } ((i, o).io)\}.$$

As can be seen, the computation of $MinExpl_1$ can again be based on the computation of minimal support clauses, a task that an ATMS can solve. The next theorem provides the foundation for an algorithm that computes the length-k minimal explanations.

Theorem 5.5.5 $\forall k \geq 1, MinExpl_{k+1}(Sys, m, io)$ *is the set of the minimal elements of the set:* $\{((d'_o, i_o, o_o), (d'_1, i_1, o_1), \ldots, (d'_k, i_k, o_k)) \mid$

$$\exists((d_1, i_1, o_1), \ldots, (d_k, i_k, o_k)) \in MinExpl_k(Sys, m)$$
$$s.t.\ ((d'_o, i_o, o_o)) \in MinExpl_1(Sys, d_1, ((i_1, o_1), \ldots, (i_k, o_k)) \cdot io), \text{ and}$$
$$\forall j, 0 \leq j < k : d'_{j+1} = NextState(Sys, i_j \cup o_j \cup d'_j)\},$$

where "$\cdot$" denotes the concatenation of two sequences.

The sets of (minimal) explanations and causes are too large to be computed completely, at least in an on-line manner. One could rank them according to a plausibility measure in order to get a means to select a subset of interest. The development of reasonable control strategies for the construction of explanations and causes is, however, still a subject of further research.

There are classes of systems for which the construction of the set of causes that are complete state assignments is conceivable. In this respect it is sufficient to construct the minimal explanations having the length less than L, where L is the length of the longest cycle-free path through the system state space.[8] In the worst case L equals $\mathcal{N}(Sys)$, where $\mathcal{N}(Sys)$ is the total number of states in the system, i.e. L can grow exponentially in the number of components of the system. But, for some classes of systems L grows, in the worst case, only linearly in the number of components. For instance assume that:

[8] For a PSS we have $L \geq T$, where T is the transition order of the PSS (cf. definition 5.4.1). T assumes a constant input, while L does not.

- No self repair components exist (i.e. all the transitions from defect states to correct states, if any, are controlled by inputs modeling the user's repair actions).
- All the explanations that assume repair/replacement actions (prior to the diagnostic session) are not interesting.
- After removing all transitions from defect states to correct states, each component state transition graph is free of cycles, excepting the length-one cycles - i.e. having the form $s_i \rightarrow s_i$.

Under the above assumptions, if one ignores the "repair transitions", the global *system* state transition graph is free of cycles, excepting the length-one cycles. Also, the length of the longest cycle-free path through the system state space (L) is upper bounded by $N \times l$, where N represents the total number of components in the system and l represents the length of the longest cycle-free path through any of the components' state space. For this class of systems the length of the explanations required to characterize the set of all complete causes grows, in the worst case, linearly in the number of components. In this case, and if the space of interesting input assignments is relatively small, the construction of the complete set of explanations for a certain combination of defects could be conceivable. Many systems that are normally static (i.e. the dynamic aspect is introduced only by the change of the behavioral modes), fall into this category. Electrical systems, like the ones used in the previous examples, containing bulbs, fuses, wires, switches, etc., are instances of this class.

Of course, the set of causes for specified sets of defects (e.g. for all the single faults) can be computed off-line. Such information can be used by some heuristics to augment the candidate generator, but the elaboration of such heuristics is an issue of future research.

5.6 Diagnosis with an RMS-based engine

The assignments to state variables, to input variables and the observations are represented as assumptions in the RMS. The rest of the predictions are represented as derived RMS nodes. Each time point is represented by a complete input and state assignment. A partial input and state assignment represents a description of a set of time points. Retrieving the values of the system parameters that hold at a certain time point means to retrieve the

nodes that hold in the context defined by the input and state assignment for that time point. Changing the time point corresponds to a context change (e.g. in a focusing ATMS or in the 2vRMS this accounts to a focus change). The RMS network "spreads the activation" from the currently enabled state and input assumptions to the rest of nodes. The next state can be retrieved by querying the next-state nodes "active" in the current time point context. The sequence of time points is not explicitly represented in the RMS network. To make predictions across several time points the diagnostic engine has to "simulate" the automaton by executing the cycle of focus changes corresponding to the sequence of inputs and states. The ATMS labels correspond to some minimal support clauses (cf. [RdK87]). Some of these labels are used in order to identify the minimal conflicts, to map the conflicts on the initial candidate and to search for the primary causes.

The characterization of the minimal conflicts and of the minimal explanations in a recursive "backward" manner suits the lazy label evaluation in the 2vRMS.

5.7 Related work

Qualitative reasoning

Much work in qualitative reasoning addressed the modeling for monitoring or diagnosis of (continuous) dynamic systems (cf. [DK92c, Ng90, OFK92, Pan84]). Among these works, Pan (cf. [Pan84]) addressed specifically systems with dependent failures. The system description used similar knowledge as input: state-transition graphs for the components (together with the associated behavior) and knowledge about the system structure. He uses a QSIM-based qualitative simulation (cf. [Kui86]) to build an envisionment of the evolution of the system after a certain stimulus. However, the paper addresses diagnosis only marginally: the proposed system can be used to refute some diagnostic candidates generated by some heuristic rules, but the topic is not elaborated. No attempt to identify conflicts was made in the work, thus no dependency-directed backtracking during the search for diagnoses can be imagined.

Mode-transition graphs in temporal diagnosis

In [CPDT92, FL91, NG94] several extensions to model-based temporal diagnosis using mode-transition graphs for the individual components are described. The mode-transition graphs constrain the non-deterministic mode changes that are allowed during diagnosis. The framework of [FL91] was proposed to address the problem of intermittent defects. The mode transitions have absolute probabilities associated. A temporal diagnosis in [FL91] associates to each failure the set of all time intervals when it is consistent to assume that failure (but no attractive mechanism for the computation of the temporal diagnoses is proposed there).

The framework of [CPDT92] is, in fact, very similar to that of [FL91], but the authors are concerned more to propose an efficient way of computing the temporal diagnoses. In this respect they separate the time dependent and the time independent aspects of behavior. A temporal diagnosis is decomposed into a set of atemporal diagnostic problems (one for each time point) whose solutions are then "composed" using the mode transition graphs.

The idea of decomposing a temporal diagnostic problem into several time independent problems is also present in our framework since the system description does not embed any logical model of time – SD is thus static. In our approach the mode transition graph is defined implicitly by the behavior descriptions. We couple the state assignments of the distinct problems using the next-state predictions.

In [NG94] a temporally abstract definition of diagnosis is presented. The authors use qualitative temporal relations (i.e. a subset of Allen's interval algebra – cf. [All83]) for: behavioral mode description, for observations' time span and for describing the possible component mode transitions. The diagnoses are sets of time intervals connected by qualitative temporal relations associated to the behavior modes of the components. The mode transition graphs used are similar to those of [CPDT92], but the relations between states are qualitative temporal relations.

In all these approaches (i.e. [CPDT92, FL91, NG94]) the transitions between the modes do not depend on any context-dependent information and thus remain purely non-deterministic. For instance, it is possible to say that a normal bulb will remain correct or will break, but not to say that this is affected by, for instance, the power dissipated by the bulb. This limits the expressiveness of the frameworks and makes them inappropriate for the diagnosis of dynamic devices with memory (e.g. sequential circuits) and / or

with dependent failures.

However, [CPDT92, FL91, NG94] model non-deterministic state evolutions, while we are unable to do this. An extension that would associate context-dependent probabilities to the transition graphs would probably bring our approach and those of [CPDT92, FL91] closer together.

Other work on temporal diagnosis

Many of the initial works on model-based diagnosis ignored the temporal aspects. A first step towards the consideration of time in dynamic systems was the introduction of time in static systems in [RdKS91]. In fact, in [RdKS91], observations made at several time points are used in order to filter out the diagnoses that assume intermittent behavior. The representation of time was based on introducing a set of special assumptions, namely temporal assumptions having the form $t = T$. The observations made at the time point T were then justified in the underlying ATMS with the corresponding temporal assumption. This representation offers the advantage that at the level of the inference engine it achieves prediction reusability among time points. However, the representation has strong limitations because it does not allow reasoning across several time points, thus it can only be applied to the modeling of static devices. Moreover, the prediction reusability was achieved only at a superficial level, namely at the level of the propagator (i.e. consumer activation mechanism): in the underlying ATMS the environments from the labels are duplicated for each time point – thus the labeling effort is not reused across time.

Systems like the ones described in [GSR92, Ham91b, Iwa94, Wil86] embed time in the predicates describing a domain, for instance $P(x) : t$, where $P(x)$ stands for a property of the world, and t is a time token, i.e. interval or point. The expressiveness of the representation is high, allowing the representation of static as well as dynamic behavior, but it leads unfortunately to high computational costs. A new time point is represented as a new set of grounded instances of the predicates. As argued also in [Dre94] and [TL93] such systems cannot achieve prediction sharing over time and require considerable amounts of space. The problem becomes really critical when temporal reasoning is coupled with assumption maintenance. In this case not only the predicates representing properties of the world are duplicated at different time points, but also significant parts of the justification network and of the labels in the ATMS.

Embedding temporal reasoning into the ATMS

Although not addressing diagnosis directly, comparisons can be made with the systems that attempted to integrate temporal reasoning with assumption maintenance, like in [JR90], [TL93], and [Dre94]. A major difficulty with these systems seems to be the control of reasoning.

HEART (cf. [JR90]) was a system that integrated the reasoning tasks performed by the TCP (cf. [Wil86]) with those of the basic ATMS. HEART, like TCP, used an interval-based representation of time. The datum of each node was an episode, a pair (dat, Int), where Int represents an interval of time where the datum dat holds. The environments were regarded as conjunctions of episodes that had to be conjointly assumed. HEART aimed to achieve minimal representations of the dynamic world by combining the minimal representations of the environments of an ATMS, with the maximal representation of the episodes of TCP. The tasks performed by HEART are, however, very complex due to this integration, and there was no empirical evaluation of the behavior of the system on realistic applications reported. It is very likely that the system would spend most of the resources on merging and splitting episodes, the relevance of all these operations for higher-level reasoning modules being questionable. Controlling the reasoning inside HEART, as well as on top of it, was not addressed in any way in [JR90]. Moreover, being based on the basic ATMS, we think the work from [JR90] is more relevant from a conceptual point of view than from a practical one.

Some of our early work (cf. [TL93]) also attempted to embed temporal reasoning into the ATMS. We extended the representation based on temporal assumptions used in [RdKS91], in order to make reasoning *across* time possible. In order to optimize some reasoning tasks the environments from the node labels were indexed after the temporal assumptions, which were treated in a special way (i.e. they were implicitly known to be mutually inconsistent, and were not part of the conflicts discovered). Although, like in [RdKS91], the reasoning at the level of the problem-solver achieved prediction reusability, this was not achieved at the deep level of labels in the ATMS: even with no change in the world, the set of environments is duplicated for each new time point. Thus, even if value propagation need not be redone at the top level, the environment propagation has to be redone for each time point.

There are similarities between an ATMS-based FSM implementation and the work of Dressler described in [Dre94]. The goal of [Dre94] was to achieve prediction sharing across time the same way prediction sharing across con-

texts is achieved in an ATMS.

Basically, [Dre94] suggests that a theory should be described in a temporal generic manner using formulae like: $p_1(t) \wedge ... \wedge p_n(t) \rightarrow q(\Delta(t))$, where $\Delta(t)$ is a delay function. The time is not supposed to be part of the predicates, rather it is an extra-logical means for indexing. For each proposition that has an extensional temporal extent defined, in [Dre94] a special assumption denoting (symbolically) that temporal extent is created. These temporal assumptions are propagated in the usual way in the ATMS labels, thus providing generic formulae about the temporal extent of dependent propositions.

Our system description is also temporally generic – i.e. the models describe only a current time point and the next state being given the input and state assignment. The propositions for which a temporal extent is provided are in our case the state and input assignments and the observations. We represent these propositions as assumptions in the RMS, thus we came to the same encoding as that of [Dre94]. Partial state and input assignments provide descriptions for sets of time points, in the same generic way as the temporal labels of [Dre94].

Event-based diagnosis:

Approaches like those of [CT94, SSL+94] represent the system as a transition graph between states.

As in our approach, in [SSL+94] each component is described as a finite state machine and the system is another FSM that aggregates the components. The aggregation is based on the shared events of the component FSMs and on a synchronous composition law. The synchronous composition law corresponds to our synchronous composition assumption from the section 5.3.1. Based on the system FSM and a global sensor map (giving associations between global system states and observable events) a "diagnoser" is built in [SSL+94]. The diagnoser is another finite state machine whose states model the states of the diagnostic process, e.g. the current set of hypotheses given the past observable events.

Relative to our component descriptions, the component FSMs of [SSL+94] represent "compiled" descriptions. The event sets must be defined extensionally, while we associate first order predicates or constraints for the conditions that control the transitions. Thus, we can theoretically work with infinite sets of events while this is not the case in [SSL+94]. Also, the construction of the sensor map requires knowledge acquisition in [SSL+94] – while we do

not need the sensor map because we derive implicitly the relation between the failures and their observable effects.

In [CT94] diagnosis is defined as the identification of some "preferred" sequences of events that are consistent with a sequence of observations. This definition of diagnosis has similarities with our definition. The formalism of [CT94] also allows to integrate repair actions, dependent failures and allows diagnoses to account also for unknown input sequences applied to the systems. The events in [CT94] can have nondeterministic effects. Differing from [CT94], in our formalism we distinguish between inputs, state variables and other variables, while these are all mixed in the vector of state variables in [CT94]. While [CT94] does not advocate a certain representation (intensional or extensional) for the state space and for the event relations, it also does not provide attractive means for the computation of the preferred diagnoses, nor does it describe how the system knowledge can be automatically generated starting from knowledge about the components and about the system's structure.

5.8 Discussion and future work

Approaches to model-based diagnosis that make the presupposition that the faults are independent, do not satisfactorily handle the cases when there are causal relations among the faults. We argued that in order to deal with caused defects one has to model, at least partially, the causes of the mode transitions.

Among the main features of the proposed approach we can mention that: (a) it is a specialization of the GDE-like approaches working with several behavioral modes (cf. [dKW89, SD89]); (b) it requires the partition of the parameters characterizing the system into inputs, state variables and other parameters, and assumes that the system can be modeled as a finite state machine; (c) the mode of behavior is regarded as a distinct state variable of each component; (d) it maps diagnosis to the problem of transition path identification in an automaton; (e) it allows the reuse of the candidate generation techniques based on minimal conflicts; (f) it allows RMS-based implementations to achieve attractive degrees of prediction reusability: across contexts and time.

The applicability of the proposed solution is limited by the assumptions made. The time granularity assumption seems to hold in electrical systems,

but is not a general property. This assumption, together with the pseudo-static assumption were introduced in order to constrain the situations in which the observations and the user actions should be asserted, since we do not have in general exact timing information about the mode changes. If this information were readily available one could in principle give up these assumptions and use the FSM framework instead of the PSS.

Of course, the diagnosis of dynamic systems remains a challenging issue. The finite-state machine framework cannot avoid the fact that when reasoning across several time points the conflicts tend to be very large and thus, not very informative. For some classes of systems, and under additional assumptions about the availability of measurements, diagnosis can be done without using the dynamic constraints at all, as discussed in [Dre96]. In our terms, this would mean that the relationship between the state and the next-state variables can be omitted. It is interesting to note that the RMS dependency networks of the finite-state machine and of the techniques from [Dre96] are very similar, for instance similar temporally generic nogoods would be found by both representations. It would be just a different interpretation of these nogoods and a different candidate generator required to switch between the finite-state machine framework and the framework from [Dre96].

There are many interesting directions in which the research presented here could be continued. The requirement that the system behaves deterministically is too strong in many cases. There are several aspects of non-determinism which must be dealt with: (i) the non-deterministic behavior of the individual components at the limits of the admissible operating conditions; (ii) the non-determinism due to the inaccuracies of the measurements and of the allowed tolerances for the components' parameters; and (iii) the non-determinism of the composed behavior when more than one state variable can change its value at the next time point.

The first two aspects of non-determinism could be modeled relatively easy using an RMS-based engine like the one described in the previous chapter. The individual components, instead of predicting deterministically an assignment to their next-state variables, should be able to predict disjunctions over several possible assignments, eventually also with associated probabilities or preferences. Such implied disjunctions could be handled by the candidate generator, as we have shown in 4.4. Associating these implied choices with primary vs. secondary choice sets (cf. 4.5.2), could control the degree of parallelism in exploring the non-deterministic branches. Another way of representing the same aspects of non-determinism is to allow the assignments

to state / next-state variables to represent sets, with the semantic that the actual value assigned is one of the members of the set, like in a hidden Markov chain (cf. [RJ86]). Compared with the previous representation, where the branches are explored individually, the set-based representation is less expressive, because the possible branches would be merged together in an indistinguishable way, but has computational advantages because it avoids more the combinatorial explosion. It would be interesting to allow to use both representations to express the non-determinism, and to elaborate opportunistic strategies for selecting from case to case between them.

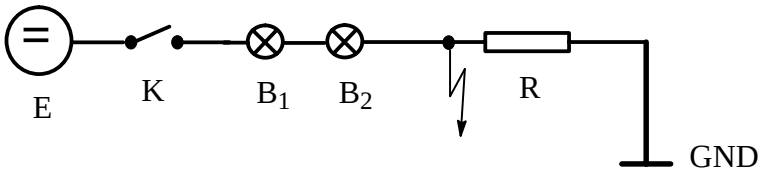

Figure 5.3:

The third aspect of non-determinism, however, cannot be handled with the set-based, or with the implied choice mechanisms, and should be dealt with in a different way. This form of non-determinism, prevented in our frameworks by the synchronous composition assumption, appears frequently in practice when modeling asynchronous processes.

Example 5.8.1 *Figure 5.8 shows a simple electrical circuit that can exhibit this form of non-determinism. Assume that all the components are correct, except the resistor R, which is shorted to the ground. In this situation we close the switch K. Since the bulbs are of the same type, their models, and our engine would predict that they will break simultaneously. However, in practice, there always exist small differences between the components of the same type. It could happen that one of the bulbs breaks earlier and prevents the other one to break. The lack of information forces us to regard the behavior of the system as non-deterministic in such a situation.*

Another direction of future work aims to associate durations to the state transitions. Currently this is possible to express in an inconvenient and limited way, namely by state replication. However, one should note that making predictions that "jump" over several time points, i.e. predict some

manifestations in the world after a certain delay, should be treated with great care: they contain hidden assumptions about the events happening in the world between the current time and the time when the prediction is made. Working with hidden assumptions is relatively dangerous, particularly in diagnosis. We saw in the pseudo-static framework, that even when the world appears to be static, the minimal dependencies on some aspects from the past might not be in a steady state.

Yet another open question refers to the way in which temporal abstraction (cf. [Ham91b, NG94]) could be conveniently dealt with in our approach. The pseudo-static framework, uses a very simple form of temporal abstraction, namely the one which condenses an infinity of time points with no change in the world to a single one. Another interesting idea could be to decompose the complex temporal behavior into several less complex interacting behaviors, using the fact that the global system space is indexed after the individual state variables. Abstract descriptions of behavior can be obtained by identifying static / cyclic behaviors in different state sub-spaces, for instance, a combination of two clocks, instead of being seen as a global mechanism performing a cycle in the global state space represented by the cross product of the individual state spaces, could simply be regarded as two smaller concurrent processes, synchronized in some way.

Chapter 6

Conclusion

In this thesis we have investigated two questions:

- How to improve the efficiency of reasoning in RMS-based diagnostic engines? We considered here two main aspects, namely the efficiency of the underlying RMS, and the efficiency of candidate generation.
- How to extend model-based diagnosis in order to represent and reason about dependent defects? Although important in practice, the problem of depending defects was not addressed until now in model-based diagnosis.

6.1 Contributions of this thesis

In this thesis we made the following contributions to the field of model-based diagnosis:

With respect to improving the efficiency of RMS-based diagnosis we have:

- introduced a new type of RMS (the 2vRMS) that supports more elaborate control strategies, and thus more efficient reasoning. The 2vRMS supports a large spectrum of services relevant for diagnosis in a more efficient way that the current ATMS-like RMSs : it separates the services than can be supported in polynomial time, like satisfiability checking in propositional Horn theories, from the services that require to solve NP-hard problems, like the generation of minimal supporting environments. The 2vRMS integrates features present at the focusing ATMS,

and at the lazy ATMS, with those of the JTMS, and was successfully tested in several realistic diagnostic problems.

- we have introduced an algorithm for computing in an incremental way some most plausible candidates. The plausibility is defined using qualitative preferences among the modes of each component, and an additional priority relation among combinations of mode assignments. The preference order structures the candidate space in a very convenient way. We have shown how to take advantage on this structure in order to optimize the search.
- we have investigated the formal properties of the reasoning architecture that combines the proposed algorithms for reason maintenance and candidate generation. We have shown that the integrated architecture is able to support services such as, for instance, the satisfiability check for arbitrary propositional clause sets. Since such a service, which is fundamental in most of the reasoning tasks in AI, is known to be NP-complete, it is hard to discuss the efficiency of such algorithms in general – they are all exponential in the worst case. However, important for the proposed architecture is the fact that based on the preference and priority concepts, on the RMS-consumer activation, and on the other control parameters, domain and problem-dependent control strategies can be implemented in order to reduce the complexity of reasoning for the *average* cases. In diagnosis, in particular, the consideration of the information about the plausibility of the modes of behavior proved to be an essential key in successfully diagnosing complex devices.

With respect to the problem of dependent defects we have introduced an approach in which it is possible to model the causes of the mode / state changes in a device. We regard each component and the whole system as discrete event systems. Dependent defects, dynamic devices with memory, as well as repair and replacement actions can be represented in the proposed framework. We have characterized how forward, as well as backward reasoning across time can be performed. From a technical point of view, two significant features of this approach are that:

- it allows the reuse of the same RMS-based architectures for diagnosis as the ones discussed in the first part of this thesis;
- RMS-based implementations of the diagnostic engine achieve attractive

degrees of prediction reusability: across contexts (i.e. diagnostic candidates), and across time.

6.2 Sugestions for further research

Efficiency of diagnosis and modeling aspects

Diagnosis is a hard task. Efficiency will continue to represent a topic of interest in model-based diagnosis.

Current RMS systems usually cache each and every inference made by the problem-solver and never "forget" it. This can save time because the problem-solver never has to make the same inference twice. It also makes the RMS a significant memory consumer. In the end also the time consumed can increase too much because bookkeeping for large amounts of data takes more time - consider, for instance, the time required when memory pages have to be swapped. This might happen due to the attempt not to forget anything, not even data that is clearly outdated. A pragmatic approach should allow also a controlled way to "forget" data and dependencies in the RMS and a systematic model of interaction between the problem-solver and the RMS for this case.

Even with respect to the problem of *efficient and flexible* candidate generation there is still room for improvement. The idea of distributing the candidate generation among several agents, possibly including a human, could prove appropriate for increasing the flexibility of the investigation process. Strategies for candidate generation that take into account pre-compiled possible causes of (cascading) defects are still an open research problem, as well as the elaboration of acceptable strategies for searching for the most plausible explanations and primary causes in the FSM framework.

A significant impact on the efficiency of diagnosis is expected from incorporating knowledge about, and strategies that take advantage of, model descriptions at different levels of detail, using abstractions, simplifications, functional and structural hierarchies. Although some work in this direction has been done (cf. [Cla93, Ham91b, Moz91, OR92, Out93, Str91b]), the elaboration of strategies that take advantage on this richer knowledge is still open. The problem is complicated because, in general, the test-pattern and the observation proposal need to use different abstraction levels than strictly an optimal candidate elaboration would use.

From the modeling point of view, working with several model views brings into discussion the availability of the richer knowledge: can the elaboration of appropriate abstract and simplified models be automated, or how can the knowledge acquisition at best be dealt with?

Extensions of the FSM framework

The FSM framework has a series of interesting features which we think are worth to investigate and to generalize further.

- From a practical point of view, an extension allowing operation with quantitative / qualitative durations of the transitions would make modeling easier, and the overall reasoning more efficient.
- The problem of efficient temporal reasoning is often brought into discussion in conjunction with the ability of performing abstract temporal reasoning. The way temporal abstraction can be conveniently performed within our approach represents another very interesting problem for future research.
- An extension that allows us to deal with non-deterministic behavior would enlarge the class of problems that can be addressed. Associating context dependent probabilities / plausibility measures to the possible branches would close the gap existing between formalisms that allow to model dynamic components with state and formalisms that allow purely non-deterministic mode-changes.
- Within a non-deterministic and probabilistic framework, the integration of models for the *wearing* of the components is conceivable. The framework would be then not only suitable to diagnose cascading faults, but also to predict and to estimate the moments when failures are likely to occur, i.e. before they really occur. The need for such predictive mechanisms is very high in practice: not-only in safety-critical domains, but, as well, in other domains, because in many cases a single defect may have extremely expensive consequences during the operation of a device or process.

Integration of diagnosis, testing and repair

The integration of diagnosis, testing and repair is another "hot" topic of research (cf. [Str94, SW93, McI94b, McI94a, FGN92, FN92]). The design of

plans for experiments that have the goal to discriminate or confirm hypotheses, or to bring the system into a safe state, are research issues that received little attention from the model-based diagnostic community until now. The support offered by the diagnostic systems to the technicians should also consider the costs of the proposed diagnostic actions (e.g. measurements, state-changing actions and repair actions). A nice feature of our approach is that it allows to easily integrate world-changing actions into the component or system models.

The need to combine monitoring with on-line diagnosis and with on-line reconfiguration is, in many cases, another requirement of safety-critical systems. The delivery of integrated monitoring and on-line diagnostic facilities together with the complex technical systems is a current trend in many manufacturing domains.

Integration with the design and the safety analysis tasks

While diagnosis, and the elaboration of testing plans are hard problems in general (i.e. undecidable, or NP-complete at best), especially for dynamic devices, another starting point for the research would be from the design tasks, namely: How should a device / process be designed such as to make its diagnosis and testing easy? The algorithms developed for model-based diagnosis and qualitative reasoning, as they are now, can be useful to automate the failure mode and effects analysis (FMEA) (cf. [JHL93]). The reverse is also true, namely, the output of the FMEA can be an important knowledge source for modeling for diagnosis (see also [PWT96, SMS95]). Also other questions relevant for safety analysis, like finding the minimal conditions under which some undesired events happen, could be addressed using the computational machinery developed for diagnosis. Other useful information that the model-based diagnostic algorithms could provide for the designers can be "a measure" of the average cost of diagnosing a device (given some prior fault likelihoods, controllability and observability information). Further support would be needed to suggest design reformulations that would make an artifact easier to diagnose. In particular, a first step, should address the questions: what design features make the testing and the diagnosis harder / easier in an artifact? What design reformulations achieve a reasonable balance between the cost, reliability and diagnosability of an artifact?

Appendix A

RMS Internal Operation

A.1 RMS-based problem-solving

In this paragraph we describe a generic architecture of the PS. The architecture is based on the concept of *consumer* described in more detail in [dK86c]. When used properly, the consumer architecture guarantees that no inference is performed twice when switching between contexts and that no inference in which the problem-solver expresses interest is left out.

We view here problem-solving as search in the space of possible inferences. We do not assume a particular knowledge representation and reasoning formalism, rather we assume that the problem-solver has a set of *inference rules*. We also assume that there is a method to determine if a certain proposition can be used by a certain inference rule to produce new data (e.g. pattern-matching or unification). If the PS has a certain context in focus then inferences in that context should be performed as long as the goal is not reached and the context is not discovered inconsistent.[1]

It depends on the problem-solver's strategy and on the RMS whether the search is performed in several contexts in parallel or in only one context at a time. A module of the architecture which we call *CandidateGenerator* is responsible to maintain and to communicate to the RMS and the focusing environments. When a focusing context is discovered inconsistent the *CandidateGenerator* is called to recompute the focus. It is the responsibility of this module to guarantee the completeness of the search in the possible worlds (see Section 4.3).

[1] In general there is not much point in exploring inconsistent contexts, but it is possible.

Algorithm PS ()
Opened and *Closed* are global variables holding lists of RMS nodes;
Agenda is a global variable holding the ordered list of active consumers;

(1) Create the RMS, initialize data structures, create the initial assumptions and premises and attach the *AllConseq* consumer to the created nodes, put all created nodes in *Opened*, etc.;

(2) **while** not *GoalAchieved*() **do:**

  (3) **call** *CandidateGenerator*() to update the focus environments;

  (4) **if** the focus is empty **then return** `Failure` ;

  (5) **while** not *GoalAchived*() and *Agenda* is not empty and no contradiction in the current focus was signaled **do:**

    (6) Pick a consumer $\mathcal{P}$ and its associated node n from *Agenda* ;

    (7) **if** n holds in focus **then**

      (8) fire $\mathcal{P}(n)$;

      else

      (9) Reattach the consumer $\mathcal{P}$ to the node n;

    endwhile

  endwhile

Figure A.1: PS description.

Our example defines only one type of consumer that is activated when a node holds in the current focus - procedure *AllConseq* (Figure A.2). At execution *AllConseq* takes the associated node and possibly other inferred nodes and applies all the inference rules in order to produce the immediate consequences of that node. The inferred nodes are maintained in two lists: the *Opened* list contains the nodes whose consequences have not yet been explored, and the *Closed* list contains the nodes whose consequences have already been explored.[2] During the execution of a consumer the node can only be combined with other nodes from *Closed* in order to match and fire the inference rules.

Figure A.1 and A.2 depict how the interaction PS - RMS basically looks like. In line 7 of PS it was necessary to test if n holds in focus because between the time when the RMS activates a consumer and the time when the PS decides to pick it form the agenda, the focus could have changed.

[2]These lists are similar to the lists maintained by the basic AI search algorithms (e.g. breadth-first, depth-first, A*, etc.) (cf. [Nil71]).

Procedure AllConseq (n)
n is an RMS node;
this procedure computes all the immediate consequences of n;

(1) Move n from *Opened* to *Closed* ;

(2) **for** each inference rule ri **do:**

(3) **for** each combination of nodes on which ri could fire and such that n is part of the combination and the other nodes belong to *Closed* **do:**

(4) Fire ri on the current combination; let o be the data produced;

(5) **if** no RMS node having the datum o already exists **then**

(6) Create an RMS node $oNode$ for the datum o ;

(7) Add $oNode$ to *Opened*; attach *AllConseq* to $oNode$;

else

(8) Let $oNode$ be the existing node having the datum o ;

(9) Add the belief constraint involving the current combination of nodes and $oNode$ to the RMS ;

endfor

endfor

Figure A.2: The consumer procedure for the algorithm PS.

The interaction can be refined to gain more efficiency by interleaving the firing of each inference rule with the consistency check. In this respect one could associate a new type of consumer to each inference rule. The consumer defined here should then, instead of directly firing the rules, attach the consumers that implement each inference rule.[3] This can prevent to unnecessarily apply all the inference rules if after applying only some of them the focus changes (e.g. due to an inconsistency). Other refinements of the above description could, for instance, structure the lists *Opened* and *Closed* according to some indexing mechanisms in order to gain more efficiency. For instance, in case the propositions refer to values assigned to variables, a natural and more efficient way of structuring the node lists would be to distribute the lists to the variables, i.e. such that each variable has its own *Opened* and *Closed* lists of assignments.

[3]There are two variants to realize this:

1. A consumer is attached to a combination of nodes that matches an inference rule. Look at [dK86c] *conjunctive consumer* to see how this can be implemented.
2. A consumer is attached to the same node, but the consumer must find all the matching combinations and fire the rule. In this case the consumer must have means to "recall" the contents of *Closed* at the time when the consumer was created.

A.2 The JTMS

We describe the operations performed when:[4]

- adding justifications;
- enabling assumptions;
- disabling assumptions.

All these operations are performed incrementally, i.e. it is assumed that before the operation the labels were correct, and the label update inspects as few nodes as necessary.

JTMS data structures

JTMS **instance** : the data structure associated to a JTMS instance could include: the set of nodes, the set of contradiction nodes, assumptions and justifications added to a JTMS. It should also provide the hooks for the PS contradiction handling and consumer activation mechanisms. The specification of the operations performed in the JTMS that follows this section uses a slot called *checkForSupportStack* to temporarily store some nodes during the label update.

JTMS **node** :

datum the PS proposition associated to this node;

label records the current belief in this node. `IN` indicates the belief, `OUT` indicates the lack of belief;

justifications references to the JTMS *justification* data structures in which this node appears as consequent;

consequences references to the JTMS *justification* data structures in which this node appears as an antecedent;

contradiction boolean. If true indicates that this is a contradiction node;

assumption boolean. If true indicates that this is an assumption node;

consumers contains the attached consumers that should be activated when the node is believed;

wfSupport if the node is labeled `IN` then this field refers to the justification currently providing a well-founded support. If the node is an enabled assumption this field contains the symbol `Assumed`.

[4] For a more detailed description, including a Lisp code implementation, see [FdK93]).

JTMS **justification** :

antecedents the list of antecedent nodes of this justification;

consequent the consequent node.

Basic operations

Figure A.3 depicts the operations performed when enabling assumptions and when new justifications are added.

When a contradictory node is labeled `IN` the procedure *SignalContradiction* is called. This procedure should mark the current context as inconsistent, should identify the set of enabled assumptions that support the contradiction, and should communicate them to the contradiction handler. In order to detect the enabled assumptions that support the contradiction one could, for instance, trace back the well-founded support of the contradictory node and collect all the enabled assumptions encountered.[5]

Figure A.4 depicts the operations performed when assumptions are disabled. In order to correctly update the labels after a previously enabled assumption is disabled, one has to proceed in two steps:

1. Label `OUT` all the nodes whose well-founded support depends on the retracted assumption;
2. Inspect all the nodes labeled `OUT` in step 1 to check if some of them cannot be labeled `IN`.

[5]This simple mechanism, however, is not enough to guarantee the minimality of the nogoods. In order to guarantee the minimality *all* the existing well-founded supports must be investigated.

Procedure AddJustification (J, tms)
J is a JTMS justification; tms is a JTMS instance.

(1) add J to the slot *consequences* of each node from $J.antecedents$ and to the slot $justifications$ of the node $J.consequent$;

(2) **call** $VerifyJust(J, tms)$;

Procedure VerifyJust (J, tms)
Checks the constraints among the labels of the nodes connected by J. Eventually initiates the label update.

(1) **if** all the nodes from $J.antecedents$ are `IN` and $J.consequent$ is not **then**

  (2) **call** $SetBelief(J.consequent, J, tms)$;

Procedure SetBelief $(n, reason, tms)$
n is a JTMS node; $reason$ is either a justification or the symbol {`Assumed`}.
Sets the label to `IN`, and the well-founded support to $reason$;
Checks for contradictions and eventually propagates the belief further.

(1) set $n.label :=$ `IN` ; set $n.wfSupport := reason$;

(2) **if** $n.contradiction$ is true **then**

  (3) **call** $SignalContradiction(n, tms)$; **return** ;

(4) **call** $VerifyJust(J, tms)$ for each $J \in n.consequences$;

(5) **if** the current context is consistent **then**

  (6) activate all consumers from $n.consumers$;

  (7) empty the list $n.consumers$;

Procedure EnableAssumption (n, tms)
n is a JTMS assumption that must be enabled in the JTMS instance tms.

(1) **if** $n.label =$ `IN` **then return** ;

(2) **call** $SetBelief(n,$ `Assumed`$, tms)$;

Figure A.3: Adding justifications and enabling assumptions in the JTMS

Procedure RetractAssumption (n, tms)
n is an enabled assumption that will be retracted in tms.
$tms.checkForSupportStack$ is used during the retraction operation.

(1) **call** $RemoveBelief(n, tms)$;

(2) **call** $CheckAdditionalSupport(tms)$;

Procedure RemoveBelief (n, tms)
n a JTMS node. Removes the support and the belief in n and the followers whose well-founded support depends on n.

(1) set $n.label :=$ `OUT` ; set $n.wfSupport :=$ `nil` ;

(2) push n to $tms.checkForSupportStack$;

(3) **for** each $J \in n.consequences$ s.t. $J = J.consequent.wfSupport$ **do:**

(4) **call** $RemoveBelief(J.consequent, tms)$;

endfor

Procedure CheckAdditionalSupport (tms)
Checks the support of the nodes from $tms.checkForSupportStack$.

(1) **while** $tms.checkForSupportStack$ is not empty **do:**

(2) pop n from $tms.checkForSupportStack$;

(3) **if** $n.label =$ `OUT` **then**

(4) **call** $VerifyJust(J, tms)$
for each $J \in n.justifications$ as long as $n.label =$ `OUT`;

endwhile

Figure A.4: Retracting an assumption in the JTMS

A.3 The LTMS

Again we present only the most important aspects. For more details see [McA82, FdK93].

LTMS data structures

LTMS instance : Like the JTMS instance. In addition it includes the *checkStack* and the *violatedStack* slots required by BCP. *checkForSupportStack* temporarily contains a stack of nodes and is used for the same purpose as in the JTMS.

LTMS node :

datum the PS proposition associated to this node;

label records the current belief. It is one of `true`, `false` or `unknown`;

posClauses the clauses in which the proposition appears positive;

negClauses the clauses in which the proposition appears negated;

assumption boolean. If true indicates that the current labeling should be treated as assumption (an assumption can be positive or negative);

tConsumers the consumers activated when the node is true;

fConsumers the consumers activated when the node is false;

wfExpl if the `assumption` slot is `true` then this field contains the symbol `Assumed`. Otherwise, if the node is not `unknown`, then this field refers to the clause currently providing a well-founded explanation.

LTMS clause :

posLiterals the nodes occurring positive in this clause;

negLiterals the nodes occurring negative in this clause;

conseq the node for whom this clause provides the well-founded explanation (if any);

sats, unkns for efficiency reasons these two slots cache the number of literals that are true, respectively unknown in the clause. A clause with $sats > 0$ is satisfied. A clause with $sats = 0 \wedge unkns = 1$ is unit-open, while one with $sats = 0 \wedge unkns = 0$ is violated.

Procedure AddClause (C, tms)
C is an LTMS clause; no proposition appears two times in C.

(1) Add C to the slot $posClauses$ of each node from $C.posLiterals$, and to the slot $negClauses$ of each node from $C.negLiterals$;

(2) initialize $C.sats$ respectively $C.unkns$;

(3) push C on $tms.checkStack$; **call** $VerifyConstraints(tms)$;

(4) **if** $tms.violatedStack$ is not empty **then call** $SignalContradiction(tms)$;

Procedure VerifyConstraints (tms)
checks the clauses from $tms.checkStack$ and sets belief if possible.

(1) **while** $tms.checkStack$ is not empty **do:**
  (2) pop C from $tms.checkStack$;
  (3) **if** $C.sats = 0 \wedge C.unkns = 1$ **then**
    (4) let p be the unknown node of C; set $C.conseq := p$;
    (5) **if** $p \in C.posLiterals$ **then**
      (6) **call** $SetBelief(p, \texttt{true}, C, tms)$;
    else
      (7) **call** $SetBelief(p, \texttt{false}, C, tms)$;
  endwhile

Figure A.5: Adding a clause in the LTMS.

LTMS basic operations

We again present in more detail: (1) the addition of a new clause (Figure A.5); (2) the enabling of an assumption (Figure A.6); and (3) the retraction of an assumption (Figure A.7). The main work of BCP is done by the *SetBelief* (Figure A.6).

The incremental label update after an assumption is disabled follows the same strategy as in the JTMS.

Procedure EnableAssumption $(n, bool, tms)$
n is an LTMS assumption; *bool* is one of `true, false`.

(1) **if** $n.label \neq$ `unknown` **then** signal error and return;

(2) **call** $SetBelief(n, bool, \texttt{Assumed}, tms)$; **call** $VerifyConstraints(tms)$;

(3) **if** $tms.violatedStack$ is not empty **then call** $SignalContradiction(tms)$;

Procedure SetBelief $(p, val, reason, tms)$
p is a node; *val* is one of `true, false` ; *reason* is a clause or the symbol `Assumed`; $chkSats$ and $chkUV$ are local variables containing lists of clauses. Sets the belief and the well-founded explanation, checks if the belief should be further propagated, activates consumers.

(1) set $p.label := val$; $p.wfExpl := reason$;

(2) **if** $val =$ `true` **then**

(3) activate consumers from $p.tConsumers$ and clear the list;

(4) $(chkSats, chkUV) := (p.posClauses, p.negClauses)$;

else

(5) activate consumers from $p.fConsumers$ and clear the list;

(6) $(chkSats, chkUV) := (p.negClauses, p.posClauses)$;

(7) increment $C.sats$ and decrement $C.unkns$ for each clause $C \in chkSats$;

(8) **for** each clause $C \in chkUV$ **do:**

(9) decrement $C.unkns$;

(10) **if** $C.sats = 0$ **then**

(11) **if** $C.unkns = 1$ **then** push C on $tms.checkStack$;

(12) **if** $C.unkns = 0$ **then** push C on $tms.violatedStack$;

endfor

Figure A.6: Enabling an assumption in the LTMS.

Procedure RetractAssumption (n, tms)
n is an assumption that must be retracted. Uses $tms.checkForSupportStack$.

(1) **if** $n.label =$ `unknown` **then return** ;

(2) **call** $RemoveBelief(n, tms)$; **call** $CheckAdditionalSupport(tms)$;

(3) **if** $tms.violatedStack$ is not empty **then call** $SignalContradiction(tms)$;

Procedure RemoveBelief (n, tms)
labels all nodes whose well-founded explanation depends on n to unknown;
$conseqCls, anteCls$ are local variables containing lists of clauses.

(1) **if** $n.label =$ `true` **then**

(2) $(conseqCls, anteCls) := (n.negClauses, n.posClauses)$;

else

(3) $(conseqCls, anteCls) := (n.posClauses, n.negClauses)$;

(4) set $n.label :=$ `unknown`; $n.wfExpl :=$ `nil`;

(5) push n to $tms.checkForSupportStack$;

(6) decrement $C.sats$, increment $C.unkns$ for each $C \in anteCls$;

(7) **for** each $C \in conseqCls$ **do:**

(8) increment $C.unkns$;

(9) **if** $C.conseq \neq$ `nil` **then**

(10) set $C.conseq :=$ `nil` ; **call** $RemoveBelief(C.conseq, tms)$;

endfor

Procedure CheckAdditionalSupport (tms)
checks the nodes from $tms.checkForSupportStack$.

(1) **while** $tms.checkForSupportStack$ is not empty **do:**

(2) pop n from $tms.checkForSupportStack$

(3) **if** $n.label =$ `unknown` **then**

(4) push all clauses from $n.posClauses \cup n.negClauses$ to $tms.checkStack$;

(5) **call** $VerifyConstraints(tms)$;

endwhile

Figure A.7: Retracting an assumption in the LTMS.

A.4 The JTMSset

JTMSset data structures

JTMSset instance : like the JTMS instance. In addition it includes the slots:

focusEnvironments : an ordered collection of environments defining the current focus. The index of a focus environment in this ordered collection is used as the identifier for that focus context.

obsoleteFocus : the (bit) set with the identifiers of the focus environments from *focusEnvironments* that the PS removed from the focus;

actualFocus : the (bit) set with the focusing environments that are neither obsolete, nor inconsistent;

checkForSupportStack : like in the JTMS;

sleepingConsumers : this is a collection of associations (i, n) where i is the identifier of an inactive focusing environment (i.e. obsolete or inconsistent) and n is a node that holds in the focusing context i and has attached consumers. For reasons of efficiency this data structure should be indexed after i (e.g. an array of node-lists).

JTMSset node : like the JTMS node. The *wfSupport* field is missing because it is more conveniently to store it in the justifications. The *label* contains the (bit) set of focusing contexts in which the node is derivable. The consumers of a node n will be activated whenever:
$tms.activeFocus \cap n.label \neq \{\}$.

JTMSset justification : as in the JTMS. In addition to the *antecedents* and *consequent* slots the JTMSset justification contains a slot called *wfSupport* containing the (bit) set of focusing contexts in which the current justification provides the well-founded support for the consequent node. For some operations it is also useful to cache the set of focusing contexts propagated by a justification (i.e. the intersection of the antecedents' labels) - this slot we call *fLabel*.

Basic operations

We present the operations performed when adding justifications and when changing the focus. The addition of a justification is very simple (Figure A.8 and A.9).

Procedure AddJustification (J, tms)
J is a JTMSset justification.

(1) Add J to the slots *consequences* of each node from $J.antecedents$ and to the slot $justifications$ of the node $J.consequent$;

(2) initialize $J.wfSupport$ to the empty set;

(3) **call** $VerifyJust(J, tms)$;

Procedure VerifyJust (J, tms)
checks if the constraint between J's antecedents and the consequent's labels is satisfied. If not, initiates the label update.

(1) set $J.fLabel$ to the set intersection of the nodes labels from $J.antecedents$;

(2) let $newFLabel := J.fLabel - J.consequent.label$;

(3) **if** $newFLabel \neq \{\}$ **then**

  (4) set $J.wfSupport := J.wfSupport \cup newFLabel$;

  (5) **call** $SetBelief(J.consequent, newFLabel, tms)$;

Figure A.8: Adding a justification in the JTMSset

The addition of a focusing environment is more complicated. When there exists an obsolete focus environment the new focusing environment will replace it. The assumptions from the old obsolete focusing environment that do not appear in the new focusing environment must be retracted. Since it is probable that several assumptions have to be retracted at such an operation the usual check for alternative well-founded support is delayed until all the retractions are done. See Figure A.10.

Procedure SetBelief $(n, newFLabel, tms)$
$newFLabel$ is a set of new focus environments that have to be added to n's label; activates consumers and propagates the update further if necessary.

(1) set $n.label := n.label \cup newFLabel$;
(2) **if** $n.contradiction$ is true **then**
  (3) **if** $newFLabel \cap tms.activeFocus \neq \{\}$ **then**
    (4) **call** $SignalContradiction(n,$
      $tms.activeFocus \cap newFLabel, tms)$;
    (5) $tms.activeFocus := tms.activeFocus - newFLabel$;
  (6) **return** ;
(7) **call** $VerifyJust(J, tms)$ for each justification $J \in n.consequences$;
(8) **if** $newFLabel \cap tms.activeFocus \neq \{\}$ **then**
  (9) activate all consumers from $n.consumers$; empty $n.consumers$;
  else
  (10) **if** $n.consumers$ is not empty **then**
    (11) add (i, n) to $tms.sleepingConsumers$ for each $i \in newFLabel$;

Figure A.9: Incremental label propagation in the JTMSset

Procedure RemoveFromFocus $(fEnv, tms)$
$fEnv$ is an old focusing environment of tms.

(1) let i be the index of $fEnv$ in $tms.focusEnvironments$;

(2) remove i from $tms.activeFocus$; add i to $tms.obsoleteFocus$;

Procedure AddToFocus $(fEnv, tms)$
$fEnv$ is a set of assumptions defining a new focusing environment for tms;

(1) **if** $tms.obsoleteFocus = \{\}$ **then**

(2) let i be the index of the next free entry in $tms.focusEnvironments$;

(3) initialize $oldFEnv$ with the empty set;

else

(4) let $oldFEnv$ be an arbitrary element of $tms.focusEnvironments$ whose index is in $tms.obsoleteFocus$; let i be the index of $oldFEnv$ in $tms.focusEnvironments$;

(5) let $assToRetract := oldFEnv \backslash fEnv$; $assToEnable := fEnv \backslash oldFEnv$;

(6) store $fEnv$ in $tms.focusEnvironments$ at the index i;

(7) remove i from $tms.obsoleteFocus$; add i to $tms.activeFocus$;

(8) **call** $RemoveBelief(n, i, tms)$ for each assumption $n \in assToRetract$;

(9) **call** $SetBelief(n, \{i\}, tms)$ for each assumption $n \in assToEnable$;

(10) **call** $CheckAdditionalSupport(i, tms)$;

(11) **if** $i \in tms.activeFocus$ **then call** $WakeUpConsumers(i, tms)$;

Procedure WakeUpConsumers (i, tms)
i is the index of a valid focusing environment. checks the nodes from $tms.sleepingConsumers$ and activates consumers if necessary.

(1) let $chkNodes$ be the collection of nodes from $tms.sleepingConsumers$ associated with the context i;

(2) **for** each $n \in chkNodes$ **do:**

(3) remove (i, n) from $tms.sleepingConsumers$;

(4) **if** $i \in n.label$ and $n.consumers$ is not empty **then**

(5) activate n's consumers; empty $n.consumers$;

endfor

Figure A.10: Changing the focus in the JTMSset.

Procedure RemoveBelief (n, i, tms)
n is an node that must be retracted in the context i.

(1) remove i from $n.label$; push n on $tms.checkForSupportStack$;
(2) **for** $J \in n.consequences$ **do:**
  (3) remove i from $J.fLabel$;
  (4) **if** $i \in J.wfSupport$ **then**
    (5) remove i from $J.wfSupport$;
    (6) **call** $RemoveBelief(J.consequent, i, tms)$;
  endfor

Procedure CheckAdditionalSupport (i, tms)
looks for support in context i.

(1) **while** $tms.checkForSupportStack$ is not empty **do:**
  (2) pop node n from $tms.checkForSupportStack$;
  (3) **if** $i \notin n.label$ **then**
    (4) **for** $J \in n.justifications$ **do:**
      (5) **if** $i \in J.fLabel$ **then call** $VerifyJust(J, tms)$;
      endfor
  endwhile

Figure A.11: Removing and checking for support in the JTMSset.

A.5 The basic ATMS

Basic data structures

ATMS **instance** : except the usual information (e.g. see the JTMS instance) the ATMS instance stores:

nogoodDB : the collection with the minimal nogoods discovered so far. Efficient representations of this structure are discussed in [dK86a] and are, for instance, a collection of environments indexed by the size of the environments, or discrimination trees. The nogoods of size one and two are usually handled more efficient (cf. [dK86a]).

envDB : for efficiency reasons the environments appearing in the node labels are stored in this database. This ensures that each distinct environment has a unique data structure. The entries in the *envDB* hold back pointers to the nodes in whose labels the associated environment appears (see below *envDB-Elm*). Also, due to efficiency reasons, this database should be indexed after the size of the environments (e.g. an array or hash-table of collections). These features makes the maintenance of the consistency more efficient - the supersets of a nogood are checked in the *envDB*, and not at every node.

checkStack : auxiliary data structure.[6] During the label update this stack contains the tuples $(J, n, newEnvs)$ specifying the justifications where an antecedent node n received in its label the set of new environments $newEnvs$, but the constraints of J were not yet checked. At the beginning and end of each PS - ATMS transaction this stack is empty, i.e. the dependency network is in a consistent state.

ATMS **node** : *datum, justifications, consequences, assumption, contradiction, consumers* have the same meaning as in the JTMS. The *label* holds the complete, minimal and consistent set of environments that entail the node given the justification structure. The well-founded support is not recorded.

ATMS **justification** : has *antecedents* and *consequent* as in the JTMS.

envDB-elm : entry in the environment database.

env : the data structure representing the environment;

[6] It plays the same role as the LTMS *checkStack* that held unprocessed constraints.

inLabels : references to the nodes whose labels contain the associated environment.

Basic operations

In the ATMS there is no correspondence to assumption enabling and assumption retraction. When an assumption is created its label is immediately propagated to the followers (i.e. they are from the start enabled), and when an inconsistent environment is detected, the supersets of the nogood are removed, thus restoring the consistency. We present in the following the operations performed when a justification is added (Figure A.12). The algorithms are the same but more detailed than the ones published in [dK88].

The labels are updated in an incremental way. When a set of new environments *newEnvs* is added to the label of a node n, the triple $(J, n, newEnvs)$ is added to the *checkStack*. The function *VerifyConstraints* (similar to the one of the LTMS) processes the entries from the *checkStack*. For each entry it verifies if the constraints among the labels of the nodes connected by the justification J are satisfied and, if not, computes the new environments that must be added to the consequent of J - this is computed by the function *NewEnvsForConseq*. In this procedure n is assumed to be an antecedent node of J or the symbol `nil`. The latter case appears only when a new justification is added.

The pending updates are processed in a depth-first manner since *checkStack* is a stack. By making it a queue one can process them in a breadth-first manner. It is easy (and more efficient) to keep this collection ordered according to some criteria, i.e. to perform the label propagation in a best-first manner, e.g. the propagations that involve smaller environments should be processed before the ones involving bigger ones, the justifications of contradictions should be processed before the others, etc. This would help finding the minimal nogoods as soon as possible, thus stopping the work in inconsistent contexts earlier.

Procedure AddJustification (J, tms)
J is an ATMS justification.

(1) Add J to the field *consequences* of each node from $J.antecedents$ and to the field $justifications$ of the node $J.consequent$;

(2) **call** $NewConstraint(J, \texttt{nil}, \{\{\}\}, tms)$;

(3) **call** $VerifyConstraints(tms)$;

Procedure NewConstraint $(J, node, envSet, tms)$
$envSet$ was added to $node.label$ where $node \in J.antecedents$ if $node \neq \texttt{nil}$; or J is a newly added justification if $node = \texttt{nil}$; in the basic ATMS this procedure simply pushes the update on $tms.checkStack$.

(1) push the triple $(J, node, envSet)$ to $tms.checkStack$;

Procedure VerifyConstraints (tms)
Checks the entries of the *checkStack* which contain unprocessed label updates.

(1) **while** $tms.checkStack$ is not empty **do:**
  (2) pop (J, n, upd) from $tms.checkStack$;
  (3) $updConseq := NewEnvsForConseq(J, n, upd)$;
  (4) **if** $updConseq$ is not empty **then**
    (5) **call** $SetBelief(J.conseq, updConseq, tms)$;
  endwhile

Procedure SetBelief $(n, newEnvs, tms)$
adds $newEnvs$ to n's label; checks consistency, creates more triples $(J, n, newEnvs)$ for the incremental label update and activates consumers.

(1) **if** $n.contradiction$ is true **then**
  (2) **call** $Nogood(e, tms)$ for each $e \in newEnvs$; **return** ;

(3) remove from $n.label$ the supersets of any member of $newEnvs$;

(4) install $newEnvs$ in $tms.envDB$ and add $newEnvs$ to $n.label$;

(5) activate the consumers of n and empty the list of attached consumers;

(6) call $NewConstraint(J, n, newEnvs, tms)$ for each $J \in n.consequences$;

Figure A.12: Adding a new justification in the ATMS.

Function NewEnvsForConseq (J, n, upd)
$\Longrightarrow$ returns the set of new environments that must be added to $J.consequent.label$; if $n \neq$ `nil` then $n \in J.antecedents$ and upd is the set of new environments that have been added to $n.label$ without being propagated to $J.consequent$. This function takes the union of every combination of environments from J's antecedents. If $n \neq$ `nil` than the set upd is used instead $n.label$.

(1) initialize $newUpd := upd$;

(2) **for** each antecedent node $a \in J.antecedents$ **do:**

  (3) **if** $a \neq n$ **then**

    (4) let tmp be the collection of environments resulting by unioning each environment from $newUpd$ with each environment from $a.label$;

    (5) remove from tmp each environment that is inconsistent, equal, or superset of another environment from tmp; $newUpd := tmp$;

  endfor

(6) remove from $newUpd$ the supersets of any element of $J.consequent.label$;

(7) **return** $newUpd$;

Procedure Nogood (e, tms)
e is a new minimal nogood. Notify the PS, remove supersets from the environment database, from the node labels and from the nogood database. The pending label updates from $checkStack$ must also be checked for consistency.

(1) inform the PS: **call** $NoticeNogood(e, tms)$;

(2) remove any superset from $tms.nogoodDB$; store e in $tms.nogoodDB$;

(3) remove any superset of e from the sets $newEnvs$ of the triples $(J, n, newEnvs)$ from $tms.checkStack$;

(4) **for** any $envEntry \in tms.envDB$ that stores a superset of e **do:**

  (5) remove $envEntry.env$ from the label of each node $n \in envEntry.inLabels$;

  (6) remove the entry $envEntry$ from $tms.envDB$;

  endfor

Figure A.13: Computing incremental label updates and processing nogoods.

A.6 The lazy ATMS

The Lazy ATMS data structures

LazyATMS **instance** : like the ATMS instance. Additionally we use:

reqJusts : during query answering this stack contains the justifications that are required in order to compute the label of the queried node.

LazyATMS **node** : like the ATMS node. In addition it contains the slot:

mark : If true signals that the label of the node might not be updated;

LazyATMS **justification** : except the usual *antecedents* and *consequent* it contains:

delayedConstraints : This collection stores unprocessed incremental label updates $(J, n, newEnvs)$ for this justification (like the $tms.checkStack$) but which are not in effect due to the lazy label evaluation. The lazy label evaluation is implemented thus as a lazy justification-constraint evaluation. When a node is queried some delayed unprocessed constraints are moved from some justifications' *delayedConstraints* to the tms *checkStack* where they are processed.

req : boolean. If true, the justification is required for the current query.

envDB-**entry** : like in the ATMS. Additionally it could include references to those justifications whose slot *delayedConstraints* mentions this environment. This entry is optional. If present it makes the removal of the supersets of a nogood from the pending label updates more efficient. Alternatively, this operation can be also postponed until the delayed updates are restarted. In the following we take the last alternative.

Basic operations

The addition of a justification raises no difficulty (Figure A.14). It just propagates the mark to the followers of the consequent as long as the mark is not already set. It also installs delayed label propagation constraints. The procedure $AddJustification$ avoids to set the marks if the addition of the justification would not produce any change in the label of the consequent.

The procedure that queries a node label (Figure A.15) first determines the set of justifications whose constraints must be fully satisfied (they are called required justifications). For each required justification the label update

Procedure AddJustification (J, tms)
triggers only the mark update. The incremental label update for the consequent is computed, but instead of processing it it is placed on $J.delayedConstraints$.

(1) Add J to the slots *consequences* of each node from $J.antecedents$ and to the slot $justifications$ of the node $J.consequent$;

(2) **if** $NewEnvsForConseq(J, \texttt{nil}, \{\{\}\})$ is not empty **then**

(3) **call** SetMark(J.consequent);

(4) **call** $NewConstraint(J, \texttt{nil}, \{\{\}\}, tms)$;

Procedure SetMark (n)
n is marked as having an "unupdated" label.

(1) **if** $n.mark = \texttt{true} \vee n$ is premise **then return** ;

(2) set $n.mark := \texttt{true}$;

(3) **call** $SetMark(J.consequent)$ for each $J \in n.consequences$;

Procedure NewConstraint $(J, n, envs, tms)$
unless $J.req$ is set to true stores the unprocessed label update under $J.delayedConstraints$. Only during query processing some justifications are marked required and some delayed constraints are moved to the $tms.checkStack$.

(1) **if** $J.req$ is true **then**

(2) push $(J, n, envs)$ to $tms.checkStack$;

else

(3) push $(J, n, envs)$ to $J.delayedConstraints$;

Figure A.14: Adding a justification in the LazyATMS.

constraints that were delayed in $J.delayedConstraints$ are moved to the tms *checkStack* where they will be processed by $VerifyConstraints$. The functions *VerifyConstraints, NewEnvsForConseq, SetBelief* and *Nogood* have the same description as the ones of the basic ATMS! Since *SetBelief* uses *NewConstraint* to signal the updates that it performs and *NewConstraint* delays only the propagations of the non-requested justifications the completeness of the queried node label is ensured.

It is easy to see that this implementation of the LazyATMS differs from the one of the basic ATMS just in the order and moment when the label updates are performed.

In [KvdG93] the computation of the labels of a cycle of justifications was performed in a special, optimized way.[7] In principle, the function *VerifyConstraints* can do more work than simply sequentially processing the entries of *checkStack* - as we have already mentioned when we discussed the basic ATMS. So this procedure could (and should) analyze and reorganize the update tasks in order to gain more efficiency. For instance, several update tasks having the form (J, n, upd_i) could be combined in one $(J, n, \bigcup_i upd_i)$.

[7] However, in order to correctly process a query one does not need to update the labels of a whole cycle. To see why this is true, recall that each environment in a node label is based on a well-founded support, and that each well-founded support is *acyclic*! Thus, also the procedures shown above mark more justifications as required than strictly necessary. See also [Tat94].

Function QuerryLabel (n, tms)
computes and returns the complete, sound and minimal label for n (not necessarily consistent, unless the contradictory nodes are also queried - see the text).

(1) **call** $FindReqNet(n, tms)$;

(2) **call** $WakeUpReqConstraints(tms)$; **call** $VerifyConstraints(tms)$;

(3) $J.req :=$ `false` for each $J \in tms.reqJusts$; empty $tms.reqJusts$;

(4) **return** $n.label$;

Procedure FindReqNet (n, tms)
steps back in the justification network and determines the justifications that are required to compute the label of n. These justifications are stored in the $tms.reqJusts$ for later processing and their field req is set true.

(1) **if** $n.mark =$ `false` **then return** ;

;; change node mark to indicate that (after processing) the label is updated

(2) set $n.mark :=$ `false`;

(3) **for** each $J \in n.justifications$ **do:**

(4) push J to $tms.reqJusts$; set $J.req :=$ `true`;

(5) **call** $FindReqNet(n', tms)$ for $n' \in J.antecedents$;

endfor

Procedure WakeUpReqConstraints (tms)
For each required justification J the collection of unprocessed label updates $J.delayedConstraints$ is moved to the $tms.checkStack$ where they will be processed by $VerifyConstraints$. At this time, the consistency of the environments that are restarted must be checked.

(1) **for** each $J \in tms.reqJusts$ **do:**

(2) remove any superset of a nogood from $envs$,
where $(J, n', envs) \in J.delayedConstraints$;

(3) move all entries from $J.delayedConstraints$ to $tms.checkStack$;

endfor

Figure A.15: Querying a node label in the LazyATMS.

A.7 The focusing ATMS

fATMS data structures

fATMS instance : like the ATMS instance. Additionally it must include a slot that specifies the current focus. This can be either an extensionally defined set of focusing environments (like it was in the JTMSset) or a PS provided predicate that decides if an arbitrary environment is in the current focus. We take here the second alternative since it is more general.

fATMS node : like the one of the basic ATMS. Additionally it has:

blockedLabel : holds the environments whose propagation is delayed because they do not hold in the current focus.[8]

fATMS justification : like the one of the basic ATMS.

envDB-Elm : like the one of the basic atms, but including the slot *inBlockedLabels* - stores the references to those nodes where this environment appears in the *blockedLabel*. This helps to perform the focus change more efficiently.

Basic operations

We describe the addition of a justification and the focus change. The addition of a justification is very similar to the ones of the basic and the lazy ATMS. The functions *AddJustification, VerifyConstraints, NewConstraint, NewEnvsForConseq* are identical with the ones of the basic ATMS (see Figure A.16 and A.12). The procedure *SetBelief* is different in that it must decide which label updates to store in the node labels and which in the blocked labels. The procedure *NewConstraint* need not test if the environments of its argument hold in focus or not because it is only called with sets of environments that hold in focus (*SetBelief* ensures this). The *Nogood* function also has a minor change because it must remove the inconsistent environments from

[8]This field has a similar role as the slot *delayedConstraints* of a LazyATMS justification. However, while in the LazyATMS it is more natural to store the delayed updates at justifications, since the justifications "decide" if they are relevant, in the fATMS it is more natural, and more efficient, to store the delayed updates at nodes, since a delayed environment is delayed for all consequent justifications and is even not added to the node label.

the blocked labels too.[9]

When the focus changes the previously delayed updates that enter the focus must be restarted. This is done by sweeping the environment table, detecting the environments that are in the new focus but appear in blocked node labels, and by moving them to the node labels and propagating them further (Figure A.16).

[9] This is not really necessary. If the function that decides what holds in focus never answers true to an inconsistent environment then the inconsistent elements of the blocked labels remain for ever pending there. Removing them can save some space.

Procedure SetBelief $(n, newEnvs, tms)$
adds the elements of $newEnvs$ to n's label or blocked label; checks consistency, creates more triples $(J, n, newEnvs)$ for the incremental label update.

(1) remove from $n.label, n.blockedLabel$ the supersets of any member of $newEnvs$;

(2) remove from $newEnvs$ the supersets of any member of $n.blockedLabel$;

(3) let $benvs := \{e \in newEnvs \mid e$ is not in the tms's focus$\}$;

(4) add $benvs$ to $n.blockedLabel$; $newEnvs := newEnvs - benvs$;

(5) **if** $newEnvs$ is empty **then return** ;

(6) **if** $n.contradiction$ is true **then**

(7) **call** $Nogood(e, tms)$ for each $e \in newEnvs$; **return** ;

(8) add the elements of $newEnvs$ to $n.label$;

(9) activate the consumers of n and empty the list of attached consumers;

(10) call $NewConstraint(J, n, newEnvs, tms)$ for each $J \in n.consequences$;

Procedure ChangeFocus $(newFocus, tms)$
restarts the propagation of the environments from the node blocked labels

(1) update tms' focus slot with $newFocus$;

(2) **for** each contradiction node $n \in tms.contradictionNodes$ **do:**

(3) $newNgs := \{e \in n.blockedLabel \mid e$ is in tms' focus $\}$;

(4) **call** $Nogood(e, tms)$ for each $e \in newNgs$;

endfor

(5) **for** $envEntry \in tms.envDB$ s.t. $envEntry.inBlockedLabels$ is not empty and $envEntry.env$ holds in focus **do:**

(6) **for** $n \in envEntry.inBlockedLabels$ **do:**

(7) remove $envEntry.env$ from $n.blockedLabel$;

(8) **call** $SetBelief(n, \{envEntry.env\}, tms)$;

endfor

(9) set $envEntry.inBlockedLabels$ to the empty list;

endfor

;; one could optimize the $tms.checkStack$ before the next call

(10) **call** $VerifyConstraints(tms)$;

Figure A.16: The incremental changes of the fATMS relative to the ATMS.

A.8 Non-Monotonic RMSs

Although the reasoning performed within the monotonic RMSs is monotonic, once one adds the *contradiction handler* to the architecture, the reasoning becomes *non-monotonic*. The contradiction handler decides what is "best" to believe next and could base its decision on a variety of reasons. It adds and retracts assumptions, and thus, the reasoning it performs is non-monotonic. In fact, one could implement a non monotonic RMS just by embedding a (systematic) contradiction handler into a monotonic RMS and, eventually, extending the interface .

The distinguishing feature of a non-monotonic RMS is that it allows to express the dependence of the belief in a node also on the *lack* of belief in other nodes. Non-monotonic and default reasoning underlies common sense reasoning, design, decision making, and many other tasks ([Doy79] gives several arguments for the significance and usefulness of a non-monotonic reasoner). Both Doyle's nmJTMS and Dressler's nmATMS have common features which we present in the following:

The belief constraints are non-monotonic justifications: $p_1 \wedge \ldots \wedge p_k \wedge out(q_1) \wedge \ldots \wedge out(q_l) \rightarrow r$, where p_i, q_i, r are propositional symbols. The interpretation of the nm-justification is: Whenever we believe $p_1 \wedge ... \wedge p_k$ and we have no reason to believe any of $q_1 \ldots q_l$ it is justified to believe r. We note with $ma(J)$ (i.e. the monotonic antecedents of J) the set of antecedents that must be `IN`, and with $nma(J)$ the set of antecedents that must be `OUT` such that J constraints the belief of the consequent. A justification with an empty set of antecedents forces the unconditioned belief in the consequent, thus making the consequent a premise.

Let a dependency network be $\mathcal{D} = (\mathcal{N}, \mathcal{A}, \mathcal{J})$, where $\mathcal{N}$ is the set of nodes, $\mathcal{A}$ is the set of assumptions, and $\mathcal{J}$ is the set of nm-justifications. Given a set $S \subseteq \mathcal{N}$ of nodes it is said that J *is valid in* S if and only if $ma(J) \subseteq S \wedge nma(J) \cap S = \Phi$.

The *well-founded support* of a certain node n *with respect to a set* S is providing a non-cyclic and non-redundant proof for n and plays an important role in defining the properties of the nmRMSs. A well-founded support for n w.r.t. a set of nodes S is a sequence of justifications $J_1, \ldots, J_k$ such that:

1. each J_i *is valid* in S;
2. J_k justifies n or n is an enabled assumption;

3. all the monotonic antecedents of J_i are either enabled assumptions or are justified earlier in the sequence;
4. no node has more than one justification in the sequence; an enabled assumption has no justification in the sequence.

A set $S \subseteq \mathcal{N}$ is called *closed* if and only if any consequent of a valid justification in S is also in S (i.e. J is valid in $S \Rightarrow J.consequent \in S$).

A set $S \subseteq \mathcal{N}$ is called *grounded* if and only if any node of S has a well-founded support in S.

A set $S \subseteq \mathcal{N}$ is *an extension* of $\mathcal{D}$ if and only if S is both grounded and closed.

We can now formally characterize what the nmJTMS and the nmATMS compute:[10]

- Given $\mathcal{D} = (\mathcal{N}, \mathcal{A}, \mathcal{J})$ the nmJTMS interprets $\mathcal{A}$ to be the set of *enabled* assumptions. The nmJTMS computes *one extension* of $\mathcal{D}$ and labels with `IN` the nodes from the extension and with `OUT` the rest of the nodes.
- Given $\mathcal{D} = (\mathcal{N}, \mathcal{A}, \mathcal{J})$ the nmATMS interprets each element of $\mathcal{A}$ to be either an enabled assumption or a derived node. The nmATMS computes for *each* consistent set $E \in 2^A$ of enabled assumptions *all* the possible extensions.

In [RDB89] it has been shown that there is a strong relationship between the extensions computed by the nmJTMS and nmATMS and the ones computed in autoepistemic and default logic.

At this point we make some remarks about the relationship between the monotonic and non-monotonic RMSs that also clarifies some confusing terminology that exists in the literature. First we should note that the assumption set $\mathcal{A}$ from the above definition of a dependency network is not really needed in a nmRMS. There are *implicit* assumptions inside a nmRMS also when $\mathcal{A} = \Phi$. These assumptions are implicitly added by basing the belief on the *assumption* that some nodes have no derivation. Any belief that is based on `OUT` nodes could be regarded as an assumption. The nmRMSs labeling algorithms automatically handle the "enabling and retraction" of these implicit assumptions. The PS handles the assumptions $\mathcal{A}$, if any. In several papers

[10] This characterization does not cover the conflict resolution in the nmJTMS or nmATMS as we shall see.

the above set $\mathcal{A}$ is, a bit misleading, called the set of premises. Towards the PS this set plays the same role as the set of assumptions in a monotonic RMS. However, relative to the implicit assumptions of a nmRMS they are like premises since it is considered preferable to revise the labels before the assumptions $\mathcal{A}$.

Since the nmRMSs can be seen as manipulating the implicit assumptions about outness they also must offer means to detect and recover from inconsistencies. Inconsistency is simply detected when a contradiction node becomes believed. In such a case the nmRMSs chooses to retract the (implicit) assumption that one of the out-labeled nodes is in fact out. Thus, one of the out nodes underlying the contradiction is picked and is justified with the situation that produced the contradiction. The nmJTMS conducts a search until one such node is found and then it justifies it, while in the nmATMS all the ways of solving the contradiction are considered.

This way of solving the inconsistencies introduces more justifications in the dependency net. The nmJTMS and the nmATMS continue to find the extensions with respect to the *total* set of justifications, but not with respect to the *original* justifications that the PS supplied. This makes the relationship between nmRMSs and default logic less clear: The new justifications are a source of ungroundness with respect to the original set of justifications.

The nmATMS

The nmATMS (cf. [Dre88, Dre90]) is relatively simple in comparison to the nmJTMS. It is in fact a basic ATMS which makes explicit the assumptions about the out-nodes that appear in some justification (thus it makes explicit the assumptions that were implicit in the nmJTMS) and is extended with two supplementary inference rules and with a mechanism to construct the extensions.

Let $\mathcal{D} = (\mathcal{N}, \mathcal{A}, \mathcal{J})$ be a non-monotonic dependency network. Besides the PS assumptions $\mathcal{A}$ the nmATMS uses another set of assumptions $\mathcal{A}'$ - the out-assumptions. For each node p that is mentioned in the non-monotonic antecedents of some justification the nmATMS automatically creates an *out*-assumption $p_{out} \in \mathcal{A}'$. The label of p_{out} represents the situations where there is no justified belief in p. p and p_{out} are constrained not to believed in the same context. The, so-called, *consistent-belief rule* makes inconsistent the contexts where both p and p_{out} are believed. For each pair of nodes p, p_{out} this rule introduces the justification $p \wedge p_{out} \rightarrow \bot$. The context defined by a

set of assumptions E is noted $cxt(E)$.

Each environment of PS assumptions $E \in 2^{\mathcal{A}}$ can have an extension. In the terminology of the nmATMS, $S \subseteq \mathcal{N}$ is an extension *of an environment* E if and only if there exists an environment of out-assumptions $B \subseteq \mathcal{A}'$ such that: $S = cxt(E \cup B) \wedge (\forall p_{out} \in \mathcal{A}' : p_{out} \in B \Leftrightarrow \neg(p \in S))$. The environment $E \cup B$ defining an extension is called an *extension-base*. An extension S is inconsistent if $\bot \in S$.

The nmATMS has a mechanism to construct the consistent extension-bases. The extensions are easy to retrieve as the contexts of the extension-bases. The extension-bases are constructed as follows:

1. For each pair of nodes p, p_{out} a new node α_p is created and is justified with two justifications: $p \rightarrow \alpha_p; p_{out} \rightarrow \alpha_p$.
2. An additional node ϕ is created and it is justified with the conjunct of the α-nodes: $\alpha_1 \wedge \ldots \wedge \alpha_k \rightarrow \phi$.

In [Dre90] it is proven that the consistent extension-bases are exactly the elements of the label of ϕ and the consistent supersets of these environments.[11]

A problem-solver based on the nmATMS would only be interested in the environments that have a consistent extension. These environments are called *coherent*. As noted in [Dre88, Dre90] the coherence is non-monotonic with respect to the addition of assumptions and justifications, e.g. an incoherent environment could have a coherent superset. This makes the test for coherence more difficult than the one for consistency. In [Dre90] it is shown how one could test in the nmATMS if a environment is coherent or if it can have coherent supersets using the environments from ϕ's label.

When an inconsistency is detected (an environment is propagated to the label of a contradiction node) a minimal nogood is recorded as in the ATMS and the supersets of the nogood are removed from the node labels. The mechanism of contradiction handling of the nmATMS makes use of one more inference rule, the *nogood inference rule*. In fact, the nmATMS also embeds the semantic of *negated* nodes. The encoded relations among the nodes, negated nodes and out-nodes are: $p \wedge \neg p \rightarrow \bot$, $p \wedge p_{out} \rightarrow \bot$, $\neg p \rightarrow p_{out}$. From any nogood involving out-assumptions a new negative clause is built by

[11] It can be seen that the nmATMS maps the odd loops to inconsistencies while for the even loops it constructs several extension bases. The simplest example of an odd loop is $p_{out} \rightarrow p$. The simplest even loop is $p_{out} \rightarrow q$, $q_{out} \rightarrow p$.

replacing the out-assumptions from the nogood with the negative nodes, e.g. from the nogood $\{p_{out}, q_{out}\}$ the negative clause $\neg p \vee \neg q$ is inferred. These negative clauses together with the implicit knowledge that $p \vee \neg p$ are used to install a series of justifications:

$$\frac{\neg A_1 \vee \ldots \neg A_k \vee \neg p; \qquad p \vee \neg p}{A_1 \wedge \ldots \wedge A_k \rightarrow p} \tag{A.1}$$

As noted at the beginning of this section these additional justifications introduce a source of ungroundness with respect to the justifications provided by the PS. Thus, some of the extension-bases computed after the usage of the nogood inference rule are not necessarily grounded from PS's perspective.[12] In the nmATMS this aspect is stressed because incoherence is mapped to inconsistency. This is obvious in the case of an odd-loop: $p_{out} \rightarrow p$ will lead to the discovery of the nogood $\{p_{out}\}$ due to the consistent belief rule, which triggers the addition of the justification $(\rightarrow p)$ that makes p a premise due to the nogood inference rule.

[12] A check for groundness is described in [Dre88] but it does not look computationally attractive.

Appendix B

Algorithms for Candidate Generation

B.1 A basic candidate generator

A candidate generator must have means to explicitly or implicitly store and access the elements of its choice space, i.e. the choice sets, the conflicts and the priority order. The problem solver must have means to create and destroy candidate generators, to add new conflicts and choice sets and to communicate and change the heuristic that controls the number of focus candidates that a candidate generator should compute. A candidate generator must communicate to the problem solver the current focus candidates.

In this respect, a candidate generator maintains at each moment a lower bound. The lower bound is split into two sets:

- *Focus* contains the preferred and consistent candidates (with the highest priority) that should be in the focus of the problem solver;
- *Candidates* contains the rest of the candidates that together with *Focus* define a lower bound for the consistent candidates of the choice space administrated by the candidate generator. The elements from *Candidates* need not be consistent.

Both *Focus* and *Candidates* can be stored as lists of candidates ordered by priority.

Basic data structures

candidate generator instance :

Choices a collection of choice sets (see below);

Confls stores, explicitly or implicitly, the set of (minimal) conflicts among the assumptions of the choice space. In an ATMS based problem solver this slot can (but need not necessarily) refer to the ATMS since the ATMS maintains the conflicts;

Priority stores a function that takes two candidates and returns true if the first one has a higher priority than the second one, and false otherwise.

FocusController specifies a certain heuristic that controls how many of the preferred candidates should enter the focus (in [dK91] such a heuristic was proposed in a probabilistic framework).

Focus stores the current focus candidates, i.e. a list of candidates ordered according to the *Priority* (see below the structure of a candidate). The focus candidates should not include any conflict from $Confls$.

Candidates stores a collection of candidates ordered by *Priority* that together with the $Focus$ defines a lower bound for the consistent candidates of the choice space administrated by this candidate generator.

choice set contains an ordered collection (e.g. a list or an array) of symbols.[1] The order in a choice set is according to the preference order of that choice set. A choice set must be able to answer which is the *next* element after a certain member, if one exists.

candidate instance A candidate should specify for each choice set a selected element. If the preference order in each choice set reflects some form of plausibility then it should be the case that, on the average, most of the consistent candidates select few elements that are not the most preferred in each choice set. In such a case it makes sense to store in (the internal format of) a candidate only the pairs (C, a) where the element $a \in C$ is not the most preferred in C, as we will do in the following.

Figures B.1, B.2, B.3 and B.4 describe the operations performed when a new candidate generator is created.

[1] A "symbol" is in general just some data structure, and may have, in fact, a richer semantics in the problem solver, but the candidate generator treats these data as propositional.

Function MakeCandGen $(Choices, Confls, Priority, FocusController)$
creates and returns a data structure associated to the new candidate generator.

(1) let cg be a new instance of a candidate generator data structure; initialize the fields of cg with the parameters of the function accordingly;

(2) let $bottomCandidate := \{\}$;

;; this selects implicitly for each choice the most preferred element;

(3) initialize $cg.Focus := \{\}$; $cg.Candidates := \{bottomCandidate\}$;

(4) **call** $ComputeFocus(cg)$; **return** cg;

Function ComputeFocus (cg)
cg is a candidate generator instance; computes a number of consistent preferred candidates with the highest priority according to the heuristic $cg.FocusController$. Returns the collection of focus candidates.

(1) **while** $cg.FocusController$ returns false and $cg.Candidates \neq \{\}$ **do:**
  (2) remove $bestCand$, the first element from $cg.Candidates$;
  (3) let $c := ChkConsistency(bestCand, cg)$;
  (4) **if** $c = nil$ **then**
    ;; $bestCand$ is consistent
    (5) insert $bestCand$ into $cg.Focus$ according to the order $cg.Priority$;
  else
    ;; c is a conflict included in $bestCand$
    (6) **call** $InsertSuccessors(bestCand, c, cg)$;
  endwhile
(7) **return** $GetFocus(cg)$;

Figure B.1: Initializing the candidate generation and computing the focus

Procedure InsertSuccessors $(cand, c, cg)$
$cand$ is a candidate of cg that includes c. The elements from the set $DirSucc(cand, c)$ that are not successors of some candidate from $cg.Candidates \cup cg.Focus$ are inserted into $cg.Candidates$.

(1) **for** each $a \in c$ **do:**
  (2) let A be a new copy of $cand$;
  (3) **while** $A \neq nil$ and $\exists C_i \in cg.Choices$ s.t. either $(C_i, a) \in A$, or
    a is the first element of C_i and $\neg(\exists a' \in C_i$ s.t. $(C_i, a') \in A)$ **do:**
    (4) **if** a is the last element in the ordered collection C_i **then**
      (5) $A := nil$;
      else
      (6) remove the pair (C_i, a), if one exists, from A;
      (7) add $(C_i, Next(a, C_i))$ to A;
    endwhile
  (8) **if** $A \neq nil$ and $CheckIfSubsumed(A, cg) = false$ **then**
    (9) insert A in $cg.Candidates$ according to the order $cg.Priority$;
  endfor

Figure B.2: Computing direct successors

Function CheckIfSubsumed $(cand, cg)$
returns true if $cand$ is a successor of some element of $cg.Focus$ or $cg.Candidates$.

(1) **for** each candidate $A \in cg.Focus \cup cg.Candidates$ **do:**
  (2) let $isSuccessor := true$;
  (3) **for** each pair $(C_i, a_i) \in A$ as long as $isSuccessor$ is true **do:**
    (4) **if** exists a pair for C_i, i.e. $(C_i, a_i') \in cand$ **then**
      (5) **if** a_i' precedes a_i in C_i **then** $isSuccessor := false$;
      else
      (6) $isSuccessor := false$;
    endfor
  (7) **if** $isSuccessor = true$ **then return** $true$;
  endfor
(8) **return** $false$;

Figure B.3: Preference checking

Function GetFocus (cg)
This is the function that the problem solver uses in order to access the focus of a candidate generator cg. Returns the current focus of cg. The candidates are communicated in the external format, i.e. the "missing" pairs (C, a), where $C \in cg.Choices$ and $a \in C$ is the most preferred element of C, which are not stored in the internal representation of a candidate, are added.

(1) let $result := \{\}$;

(2) **for** each candidate $A \in cg.Focus$ **do:**

  (3) let A' be a copy of A;

  (4) add the pair (C_i, a_i) to A' for each $C_i \in cg.Choices$ that is not mentioned in A, where a_i is the first (most preferred) element of C_i;

  (5) add A' to result according to $cg.Priority$;

  endfor

(6) **return** $result$;

Figure B.4: Accessing the external representation of the focus

Function AddNewConflict (cfl, cg)
adds a new conflict to the candidate generator cg; recomputes and returns the new focus.

(1) add cfl to $cg.Confls$;

(2) **for** each $A \in cg.Focus$ such that A includes cfl **do:**

  (3) remove A from $cg.Focus$;

  (4) **call** $InsertSuccessors(A, cfl, cg)$;

  endfor

(5) **return** $ComputeFocus(cg)$;

Function AddNewChoice (C, cg)
adds a new non-empty choice set. Recomputes and returns the new focus.

(1) add C to $cg.Choices$;

  ;; now each element of the lower bound implicitly selects the best element of C;

(2) move the elements from $cg.Focus$ to $cg.Candidates$;

(3) **return** $ComputeFocus(cg)$;

Figure B.5: Adding new conflicts and choice sets

Function ChangePriority (*Priority*,*cg*)
Priority defines a new priority order for *cg*.

(1) $cg.Priority := Priority$;

(2) move all elements of $cg.Focus$ to $cg.Candidates$; reorder $cg.Candidates$ according to $cg.Priority$;

(3) **return** $ComputeFocus(cg)$;

Function ChangeFocusController ($FocusController$, cg)
resets the heuristic controlling the size of the focus.

(1) $cg.FocusController := FocusController$;

(2) move all elements from $cg.Focus$ to $cg.Candidates$ according to the priority;

(3) **return** $ComputeFocus(cg)$;

Figure B.6: Changing the control strategy

Figure B.5 depicts the incremental processing of new conflicts and of new choice sets. Figure B.6 depicts the operations performed when the priority or the focus controller are dynamically changed.

B.2 A hierarchic organization of several candidate generator modules

As an illustration of a possible organization of several candidate generation modules, we sketch here a hierarchic organization. By "hierarchic organization" we mean here that the global set of choice sets $Choices$ is partitioned among several sets $Choices = \bigcup_{i=1}^{n} Choices_i$, and $Choices_i \cap Choices_j = \{\}$, for all $i \neq j$. Moreover, there is a strict priority across the partitions, say, $Choices_1 > Choices_2 > \ldots > Choices_n$. Each set $Choices_j$ is administrated by a candidate generator module cg_j. It is expected that each module cg_{j+1} is activated after the preceding modules in the sequence already defined a consistent assignment for the choice sets $Choices_1^j = \bigcup_{i=1}^{j} Choices_i$. The module cg_{j+1} should find a preferred and consistent (sub)candidate that consistently extends the assignment to the choice sets $Choices_1^j$. If this is possible than the next following module (cg_{j+1}) is activated, otherwise, the previous module (cg_{j-1}) is supplied with additional conflicts and is expected to attempt

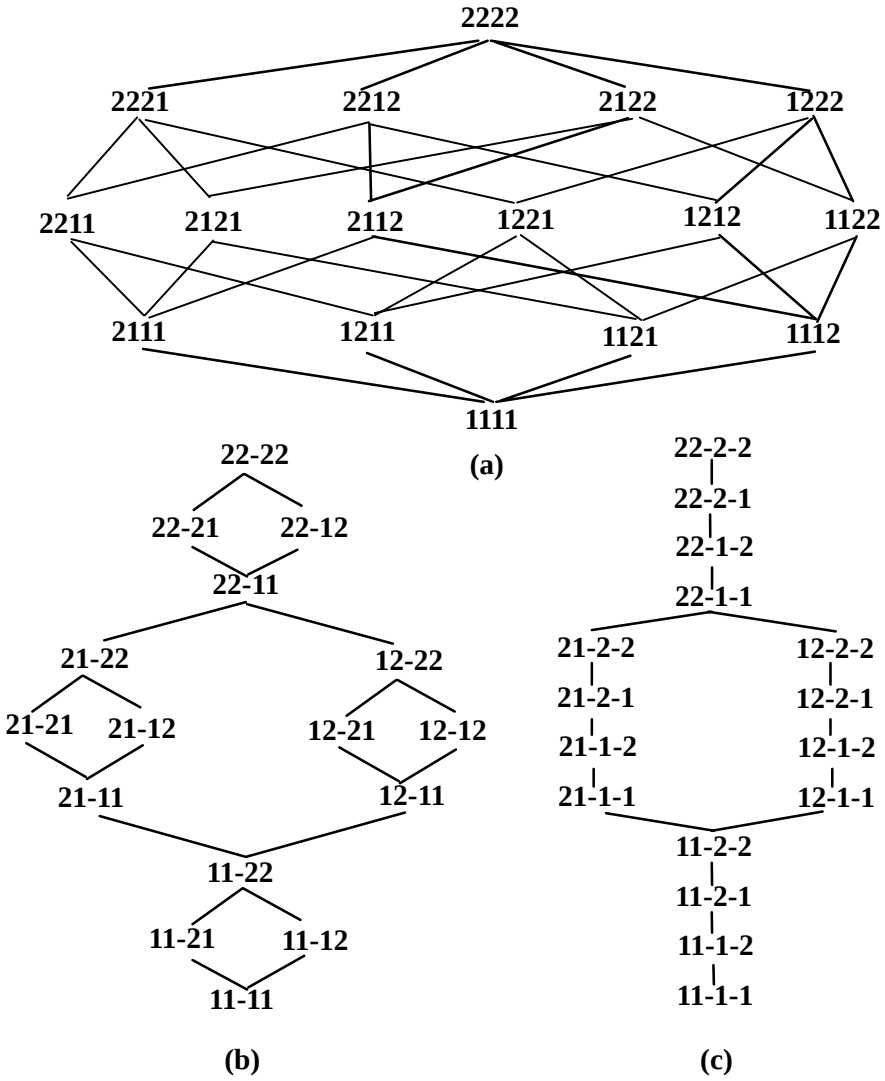

Figure B.7: Preference lattice in a choice space with four choice sets, each with two elements. The candidates are depicted using the same conventions as in Figure 4.15. The lattice (a) corresponds to a centralized choice administrator. The figures (b) and (c) show the order induced by distributing the choice sets among several modules organized hierarchically. Figure (b) has the organization: $\{C_1, C_2\} > \{C_3, C_4\}$. Figure (c) has the organization $\{C_1, C_2\} > \{C_3\} > \{C_4\}$.

to revise the proposed (sub)candidate.

Such a scheme affects the order in which the candidates are investigated. Figure B.7 (a) shows the preference order among the choice assignments of a choice space with four choice sets C_1, C_2, C_3, C_4 when the choice space is administrated by a single global module. Figure B.7 (b) shows the order among the choice assignments when the choice space is partitioned in two sub-spaces, each having two choice sets, namely $\{C_1, C_2\} > \{C_3, C_4\}$.

An extreme case of the hierarchic organization is obtained when each module is given a single choice set to administrate. In such a case, there will be no branching in the preference order, i.e. the induced order will be total.

As can be seen, the hierarchic organization reduces the "width" of the lattice and increases the depth of it, i.e. "induces" additional orderings among the candidates that were not comparable in the preference lattice of the global candidate generator. The induced order can be used to "compile" more of the priority orderings into the induced preference in some cases. In effect, the size of the lower bounds is going to be smaller. This can positively affect the efficiency of the candidate generation in some situations.

B.3 Searching in secondary choice spaces

Figure B.8 depicts an algorithm that searches *one* consistent extension of a set of symbols with an assignment for the secondary choice sets. The algorithm iterates on the secondary choice sets and attempts to assign a symbol for each choice set in turn. The order in which the choice sets are chosen is a place for heuristics, as well as the order in which the elements of each choice set are tried. As soon as a set of symbols cannot be further consistently extended, new conflicts are derived by hyperresolution (see also 4.3.4).

One can see the relationship with the way the search order was changed by the introduction of the "hierarchically organized candidate generators" in Appendix B.2. Figure B.7 (c) shows also the search order in a choice space where C_1, C_2 are primary choice sets, while C_3 and C_4 are secondary choice sets. Of course, not all of the points of the candidate space are inspected. The search takes advantage of the (induced) preference order and on the minimal conflicts in order to prune some sub-spaces of inconsistent candidates. For instance, if in the candidate space of Figure B.7 (c) the conflict 01-1-0 is added (noted using the conventions from the example 4.3.15), the lower bound {11-1-1} is going to be replaced with {11-2-1}. If further, the conflict 01-2-0 is

discovered, the lower bound is going to be replaced directly with {12-1-1}, thus pruning the whole sub-space assigning 21-x-x.

While reducing the size of the lower bounds can positively affect the efficiency, also the reduction of the potential parallelism may be a desirable feature in diagnosis in some situations. For instance, one might not be interested to discriminate between the diagnoses that assign different fault modes to the same components.

Example B.3.1 *Assume each component C_i has the modes of behavior ok_i, $f^i{}_1, \ldots, f^i_{ni}$, where f^i_j are fault modes. Associating to each component a primary choice set $\{ok_i, f^i{}_1, \ldots, f^i_{ni}\}$ makes possible to select simultaneously in the focus candidates containing mode assignments like: $\{..(C_i, f^i_1), (C_j, f^j_2), ..\}$ and $\{..(C_i, f^i_2), (C_j, f^j_1)..\}$. This enables the candidate discrimination to propose tests that distinguish between the above candidates.*

If, however, in a certain application, only the set of faulty components has to be identified, irrespective of the specific fault modes assumed, the following representation can ensure it:

For each component C_i another generic fault mode is created, say ab_i, and a primary choice set $\{ok_i, ab_i\}$ is created. In addition, the relations $ab_i \Rightarrow (f^i{}_1 \vee \ldots \vee f^i_{ni})$ can be represented into the RMS *and the candidate generator as in Lemma 4.4.2, i.e. using an "implied choice set" $C^{ab}_i = \{\Psi_i, f^i{}_1, \ldots f^i_{ni}\}$. Representing the implied choice sets C^{ab}_i as* secondary *choice sets in the candidate generator, guarantees that the focus always contains candidates that accuse distinct sets of faulty components.*

Function SExtension $(ctx, SecChoices, CS)$
ctx is a set of symbols, possibly chosen by a primary choice assignment; $SecChoices$ is the set of choice sets for which an assignment consistent with ctx should be found, if possible. CS is a choice space and is used here just as a repository of the conflicts.
Returns: a pair $(success, retV)$, where: $success$ is a boolean value; if $success$ is true than $retV$ represents a consistent extension for ctx with assignments for $SecChoices$. If $success$ is false, then no consistent extension of ctx exists, and $retV$ represents a set of conflicts that are included in ctx.

(1) **if** ctx is inconsistent **then return** $(false, \{c \in CS.Confls \mid c \subseteq ctx\})$;

(2) **if** $SecChoices$ is empty **then return** $(true, \{\})$;

(3) choose a choice set $C \in SecChoices$; choose an order among the elements of C, let it be $a_1, a_2, \ldots, a_{k_C}$;

(4) **for** $i := 1$ to k_C **do:**

(5) **if** $ctx \cup \{a_i\}$ is inconsistent **then**

(6) consider the conflicts $\{c \in CS.Confls \mid (c \subseteq ctx \cup \{a_i\}) \wedge (c \cap C = \{a_i\})\}$ for hyperresolution at the choice set C;

else

(7) $(success, retV) :=$
$SExtension(ctx \cup \{a_i\}, SecChoices - \{C\}, CS)$;

(8) **if** $success$ **then**

(9) **return** $(true, retV \cup \{(C, a_i)\})$;

else

(10) **if** ctx contains any conflict from $retV$ **then**

(11) **return** $(false, \{c \in retV \mid c \subseteq ctx\})$;

else

(12) consider the conflicts $\{c \in retV \mid c \cap C = \{a_i\}\}$ for hyperresolution at the choice set C;

endfor

;; no assignment for C can consistently extend ctx;

(13) activate hyperresolution at the choice set C; let $newCfls$ be the set of new minimal conflicts derived; add $newCfls$ to $CS.Confls$;

(14) let $retV := \{c \in newCfls \mid c \subseteq ctx\}$;

;; we know that $retV$ cannot be empty

(15) **return** $(false, retV)$.

Figure B.8: Searching for a consistent secondary choice assignment.

Appendix C

Proofs of Chapter 4

C.1 Lemma 4.3.14

Let A be a candidate of $(L, Choices, Confls, <_{pri})$ and $c \in Confls$ be a conflict included in A. If

S is a successor of A ($A < S$) that does not include c, then:

S is a successor of a direct successor of A with respect to c, i.e

$$\exists D \in DirSucc(A, c), s.t.\ D \leq S.$$

Proof: Let $A = \{(C_1, a_1), \ldots, (C_N, a_N)\}$, $c \in Confls$, $c \subseteq A$, i.e. $\forall a \in c, \exists (C_i, a_i) \in A$ s.t. $a = a_i$. $S = \{(C_1, s_1), \ldots, (C_N, s_N)\}$ is a successor of A. By definition, for all i, $a_i = s_i \vee a_i <_{C_i} s_i$. By hypothesis, we know that c is not included in S, i.e. $\exists a \in c$ s.t. $\neg(\exists (C_i, s_i) \in S$ s.t. $a = s_i)$. But c is included in A gives: $\exists (C_x, a_x) \in A \wedge a_x = a$. Let then $X = \{(C_j, a_j) \in A \mid a = a_j\}$. From the above relations we know that $s_j \neq a_j$ for all $(C_j, a_j) \in X$, and further that $a_j <_{C_j} s_j$ for all $(C_j, a_j) \in X$. Now we construct one of the direct successors of A w.r.t. c:

$$D := (A - X) \cup \{(C_j, Next(a_j, C_j)) \mid (C_j, a_j) \in X\}.$$

Since for all $(C_j, a_j) \in X : a_j <_{C_j} s_j$, and since this means also that $Next(a_j, C_j) <_{C_j} s_j \vee Next(a_j, C_j) = s_j$ it is easy too see that $D \leq S$ given also that $\forall i, a_i = s_i \vee a_i <_{C_i} s_i$. **q.e.d.**

C.2 Corollary to Lemma 4.3.14

Let LB be a lower bound for the consistent candidates of a choice space. Let $A \in LB$ be an inconsistent candidate that includes a conflict c. Then

$$LB' = (LB - \{A\}) \cup \{D \in DirSucc(A, c) \mid \neg(\exists A' \in LB - \{A\}, A' < D)\}$$

is another lower bound for the consistent candidates.

Proof: It is clear that no element from $DirSucc(A, c)$ is preferred to any element in LB (for otherwise if $\exists D \in DirSucc(A, c)$, $A' \in LB$ s.t. $D \leq A'$ this, together with $A < D$ would imply $A < A'$ which contradicts the definition of a lower bound). The hypothesis eliminates the elements of $DirSucc(A, c)$ that are less then some element of $LB - \{A\}$. We only have to prove that each consistent candidate is successor of some element of LB'.

Assume to the contrary that S is a consistent candidate such that $\neg(\exists A' \in LB'$ s.t. $A' \leq S)$. But we know that $\exists A'' \in LB$ s.t. $A'' \leq S$. This means that $A'' \in (LB - LB')$, which means $A'' = A$. Thus $A \leq S$. It cannot be that $A = S$ since S is consistent and A is not. Then $A < S$. Now, because c is not included in S we are in the case of Lemma 4.3.14, and thus $\exists D \in DirSucc(A, c)$ s.t. $D \leq S$. Since no element of LB' is a predecessor of S it means that $D \in DirSucc(A, c) - LB'$. But only the elements of $DirSucc(A, c)$ that are successors of some element of $LB - \{A\}$ are not present in LB'. Thus $S \leq D$, $D \in DirSucc(A, c)$ and $\exists A' \in LB - \{A\}$ s.t. $A' < D$. But then $A' < S$ and $A' \in LB'$ which contradicts the hypothesis of the proof. **q.e.d.**

C.3 Lemma 4.3.16

Let A be a candidate of $CS = (L, Choices, Confls, <_{pri})$ and $c \in Confls$ be a conflict included in A. Let the choice sets of CS be disjoint, i.e.

$\forall C_i, C_j \in Choices, C_i \neq C_j \Rightarrow C_i \cap C_j = \{\}$. Then if

S is a successor of some $D \in DirSucc(A, c)$, i.e. $D \leq S$, then:

S is a successor of A that does not include c.

Proof: Let $D \in DirSucc(A, c)$ where c is included in A, i.e. by definition there exists a *unique* $a \in c$ such that:

$$D = (A - X) \cup \{(C_i, Next(a_i, C_i)) \mid (C_i, a_i) \in X\}, \text{where}$$

$X = \{(C_i, a_i) \in A \mid a = a_i\}$. Since the choice sets are disjoint we conclude that X is a singleton, i.e. there exists a *unique* pair $(C_k, a_k) \in A$ s.t. $a = a_k$. We also know that $Next(a_k, C_k) \neq nil$. Let $S = \{(C_1, s_1), \ldots, (C_N, s_N)\}$ be a successor of D. Clearly, S is also a successor of A. We still have to prove that c is not included in S. Assume by contrary that c is included in S. Then there exists a choice set $C_w \in Choices$ such that $(C_w, s_w) \in S \wedge s_w = a$. But we know that $a \in C_k$. Since we assumed the choice sets to be disjoint we conclude that $C_w = C_k$. Because we assumed S to be a successor of D we have: $Next(a_k, C_k) <_{C_k} s_k \vee Next(a_k, C_k) = s_k$. In either case $a_k <_{C_k} s_k$. But this contradicts the fact that we assumed that $a = a_k = s_k$ since the preference order $<_{C_k}$ is a strict order. **q.e.d.**

C.4 Lemma 4.3.17

Let LB be a lower bound for the set of consistent candidates of a choice space and assume $A \in LB$ is a consistent *element. Let also $H(A, LB)$ be the sets of elements from LB that have a strictly higher priority than A: $H(A, LB) = \{A' \in LB \mid A <_{pri} A'\}$. Then the following are true:*

1. *Let B be a* consistent *candidate that has a strictly higher priority that A, i.e. $A <_{pri} B$. Then either $B \in H(A, LB)$, or B is a successor of some element from $H(A, LB)$;*
2. *Let C be a* consistent *candidate that is a successor of some element from $LB - H(A, LB)$. Then C cannot have a higher priority than A has, i.e. $C \leq_{pri} A$.*

Proof:

Part 1. Assume to the contrary that $A <_{pri} B$ and that neither $B \in H(A, LB)$, nor is B a successor of some element of $H(A, LB)$. Then B is either in, or a successor of $LB - H(A, LB)$, since B is consistent (see the definition of a lower bound). It cannot be

the case that $B \in LB - H(A, LB)$ since $A <_{pri} B$ would violate the definition of H. Thus $\exists A' \in LB - H(A, LB)$ s.t. $A' \leq B$. The definition of a choice set assumed that the preference and the priority "agree", i.e. $A' \leq B \Rightarrow B \leq_{pri} A'$. Due to the definition of $H(A, LB)$ we know that $A' \leq_{pri} A$. Due to the transitivity of the priority order we conclude $B \leq_{pri} A$ which contradicts the assumption that $A <_{pri} B$.
Part 2. We actually proved this already in the case above. **q.e.d.**

C.5 Lemma 4.3.22

Let $CS = (L, Choices, Confls, <_{pri})$ be a choice space and $\Sigma(CS)$ its associated set of clauses (as in Definition 4.3.21). Let $v : L \rightarrow \{true, false\}$ be an assignment of boolean values to the symbols. Then:

the formulae of $\Sigma(CS)$ are satisfied by the assignment $v \Leftrightarrow$

$\exists A$, *where A is a consistent candidate of CS s.t.*
$\{a \mid \exists C_i \in Choices \text{ s.t. } (C_i, a) \in A\} \subseteq \{a \in L \mid v(a) = true\}$.

Proof: $\Rightarrow$ ("completeness"):
Suppose that the boolean values assigned by v satisfy all the formulae of $\Sigma(CS)$. Then the assignment satisfies all the positive clauses of $\Sigma(CS)$. This means that for each $C_i \in Choices$, $\exists a_i \in C_i$ s.t. $v(a_i) = true$. Let us build a candidate $A = \{(C_i, a_i) \mid C_i \in Choices, v(a_i) = true\}$. We still have to prove that A does not include any conflict from $Confls$. Suppose, by contrary, that A includes a conflict from $Confls$. Then all the symbols from the conflict are assigned $true$ by v. But this would violate the negative clause of $\Sigma(CS)$ associated with the conflict.

$\Leftarrow$ (soundness):
Suppose we have a consistent candidate A of CS. We show that the following assignment of boolean values to the literals satisfies each formula of $\Sigma(CS)$:

$$v(a) = \begin{cases} true & \text{if } \exists (C_i, a) \in A; \\ false & \text{otherwise.} \end{cases}$$

It is obvious that each positive clause from $\Sigma(CS)$ is satisfied since v assigns true to at least one literal from each choice set. Because A is consistent in CS, i.e. no conflict of $Confls$ is included in A, it turns out that each conflict $c \in Confls$ must involve at least one literal $a_c \in c$ such that

$$\neg(\exists C_i \in Choices \text{ s.t. } (C_i, a_c) \in A)$$

But then in each negative clause from $\Sigma(CS)$ there exists at least one symbol that is assigned $false$ by v, and so, each negative clause from $\Sigma(CS)$ is satisfied as well. **q.e.d.**

C.6 Corollary 2 to Lemma 4.3.22

Let $CS = (L, Choices, Confls, \leq_{pri})$. The set of consistent candidates of CS is identical with the set of consistent candidates of any choice space $CS' = (L, Choices, Confls \cup \{ic_1, \ldots, ic_k\}, \leq_{pri})$ where ic_j are conflicts such that $\Sigma(CS) \models NegCl(ic_j)$, where $NegCl(\{a_1, \ldots, a_n\}) \equiv \neg a_1 \vee \ldots \vee \neg a_n$.

Proof: Because they have the same set of choice sets, CS and CS' have the same set of candidates. Because CS' has a bigger set of conflicts than CS it is clear that the set of consistent candidates of CS' cannot be a superset of CS. We show that any consistent candidate of CS is also a consistent candidate of CS'. Assume, by contrary, that there is a candidate A that is consistent in CS, but not in CS'. This means that A contains a conflict ic such that $\Sigma(CS) \models NegCl(ic)$. Due to Lemma 4.3.22 (the "soundness" part) we know that we can construct an assignment of boolean values to L, $v : L \rightarrow \{true, false\}$ such that $\exists(C_i, a) \in A \Rightarrow v(a) = true$, and such that the clauses of $\Sigma(CS)$ are satisfied. Of course then v provides a mapping that satisfies all the formulae of $\Sigma(CS) \cup \{a \mid (C_i, a) \in A\}$. But $\Sigma(CS) \cup \{a \mid (C_i, a) \in A\} \models NegCl(ic)$ and then also $\Sigma(CS) \cup \{a \mid (C_i, a) \in A\} \cup \{NegCl(ic)\}$ must be satisfiable, and this contradicts the fact that ic is included in A **q.e.d.**

C.7 Property 4.3.25

Let $C = \{a_1, \ldots, a_k\}$ be a choice set and c be a conflict such that $|c \cap C| > 1$. Then any new conflict obtained by hyperresolution at C and involving c and possibly other conflicts is a superset of an already known conflict.

> **Proof:** Suppose $|c \cap C| > 1$. To apply the hyperresolution rule (see the equation 4.6) one must take for each literal of C a negative clause $\neg a_i \vee NC_i$. Since c is used at least once, there must exist at least one index j such that $NC_j = c - \{a_j\}$. But, since $|c \cap C| > 1$, we know that there exists al least a different literal $a_k, a_k \neq a_j$ such that $a_k \in c \cap C$, i.e. $a_k \in NC_j$. But then the resulting clause is subsumed by the clause $a_k \vee NC_k$, which was an input clause in the inference. The derived conflict is a superset of an already known one. **q.e.d.**

C.8 Theorem 4.3.27

Let Σ be a set of clauses and Ctx be a set of symbols. Then the algorithm $NPI(\Sigma, Ctx)$ computes all the negative prime implicates of Σ among the symbols of Ctx, where NPI is defined as follows:

1. *Initialize Cls with the set of unsubsumed clauses of Σ;*
2. *Iterate on the symbols from Σ, for each symbol a do:*
   (a) *Perform all the binary resolutions w.r.t. the symbol a; Add the new unsubsumed clauses to Cls; Remove the subsumed clauses from Cls;*
   (b) *If $a \notin Ctx$ then remove all the clauses from Cls that mention a (positively or negatively); If $a \in Ctx$ then remove all the clauses that mention positively a from Cls;*
3. *Return Cls.*

> **Proof:** We use the fact that the algorithm PI generates the complete set of prime implicates of a set of propositional clauses. We prove that in order to generate the negative prime implicates of Σ among the Ctx symbols we can dispense with some classes of binary resolutions performed by PI, and what is left is what

the NPI algorithm performs. NPI differs from the PI just in the additional step 2.b that removes some clauses from the clause set. It is easy to see that no negative prime implicate among the Ctx symbols can result from resolutions performed later and involving these clauses:

Suppose PI chose a symbol a and performed all the binary resolutions w.r.t. a. Suppose $a \notin Ctx$. From now on, no resolutions w.r.t. a are performed by PI. But then, without losing the completeness of the prime implicate generation w.r.t. the Ctx symbols, one can remove the Π_a clauses mentioning a from the clause set, since all the resolvents generated later using a Π_a clause will still contain the symbol a.

If $a \in Ctx$ then, once all the resolutions w.r.t. a are performed, all the clauses containing a positively can be removed, without losing the completeness of the *negative* prime implicate generation w.r.t. the Ctx symbols. The resolvents generated later using these clauses will involve a positively, and thus are not interesting. **q.e.d.**

C.9 Corollary to Theorem 4.3.27

Let $CS = (L, Choices, Confls, <_{pri})$ be a choice space and $Ctx \subseteq L$. Then the algorithm $NPI^*(CS, Ctx)$ (Figure 4.17) computes all the minimal conflicts among the symbols from Ctx entailed by the set of clauses $\Sigma(CS)$.

Proof: Let S be the set of symbols from $\Sigma(CS)$ and let $ChSym$ be the set of symbols from the choice sets of CS (i.e. the symbols appearing in the positive clauses of $\Sigma(CS)$). Define also the set $S_1 := S - (ChSym \cup Ctx)$.

The symbols from S_1, if any, are only involved in negative clauses in $\Sigma(CS)$, thus there are no resolutions with respect to these symbols. The clauses containing symbols from S_1 can be removed from $\Sigma(CS)$ without losing the completeness of the prime implicate generation w.r.t. the Ctx symbols. This justifies step 2 of NPI^*. The rest of NPI^* is a realization of NPI. **q.e.d.**

C.10 Lemma 4.4.1

Let Σ_H be a finite set of Horn clauses, Σ_p be a finite set of positive clauses, and $\Sigma = \Sigma_H \cup \Sigma_p$. Let SAT be the algorithm depicted in Figure 4.18. Then:

- *$SAT(\Sigma)$ returns $failure$ if and only if Σ is not satisfiable;*
- *if Σ is satisfiable, then SAT returns a set of propositions $\mathcal{A}$ such that $\mathcal{A} \cup \Sigma$ is consistent and $\mathcal{A} \models C$, for all $C \in \Sigma_p$.*

Proof: If Σ_H is not satisfiable then the property is trivially true since the 2vRMS alone is a sound and complete satisfiability checker, i.e. the empty environment will be discovered as nogood and SAT will return $failure$. Assume in the following that Σ_H is consistent.

The proof proceeds as follows:

- we prove that SAT always terminates;
- we prove that each time SAT returns $failure$, Σ is inconsistent;
- finally we prove that each time SAT returns a set of propositions $\mathcal{A}$, Σ is consistent; moreover $\mathcal{A} \cup \Sigma$ is consistent, and $\mathcal{A}$ implies each positive clause from Σ.

At each iteration i, the candidate generator is called on the set of choice sets defined by Σ_p and on a set of conflicts Σ_{ni}. Σ_{ni} monotonically grows with i, i.e. during the execution of SAT more conflicts are added to cg, but no one is removed. Thus, the number of distinct candidates that cg could build for $\Sigma_p \cup \Sigma_{ni}$ decreases with the increase of i. Anyway, the number of distinct candidates that cg could build at each step is lower than the number of consistent candidates for Σ_p, which is a finite number. Next we show that there cannot be two iteration steps i, j, $i < j$, such that cg generates the same candidate A. Suppose, by contrary, that such two steps and such a candidate exist. $\mathcal{A}(A)$ must be inconsistent with Σ_H for otherwise the iteration would have ended at i. But then, there must exist at least one negative prime implicate of Σ_H, s.t. $\Sigma_H \models ng$, where the set of symbols from ng is a subset of $\mathcal{A}(A)$. These negative prime implicates will be discovered in the 2vRMS by querying the dLabel of the

contradictory nodes and they will be added to the conflicts of cg. Thus A contains the conflict ng. Then $ng \in \Sigma_{nj}$ for all $j > i$. But then cg cannot generate the candidate A at the step j, since A is not consistent with Σ_{nj}. This completes the proof that SAT always terminates.

Suppose SAT returns $failure$. This can only happen because at a certain iteration i, cg cannot find any consistent candidate, i.e. because $\Sigma_p \cup \Sigma_{ni}$ is inconsistent. But then also $\Sigma_H \cup \Sigma_p \cup \Sigma_{ni}$ is inconsistent. Since all the clauses from Σ_{ni} are implicates of Σ_H, we conclude that $\Sigma_H \cup \Sigma_p$ must be inconsistent.

Suppose SAT returns a set of symbols $\mathcal{A}$. This can only happen if there exists an iteration i such that $\Sigma_p \cup \Sigma_{ni}$ is consistent, cg generates a candidate A such that $\mathcal{A}(A) \cup \Sigma_p \cup \Sigma_{ni}$ is consistent, $\mathcal{A}(A)$ implies all clauses from Σ_p, and $\mathcal{A}(A) \cup \Sigma_H$ is consistent. But then $\mathcal{A}(A) \cup \Sigma_H \cup \Sigma_p$ is consistent, and so is $\Sigma_H \cup \Sigma_p$. **q.e.d.**

C.11 Lemma 4.4.2

Let $C \equiv (\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n \vee q_1 \vee \ldots \vee q_m)$, $n \geq 1$, $m \geq 2$ *be a mixed clause, and* Σ *be a clause set. Then:*

$$\Sigma \cup \{C\} \textit{ is satisfiable} \quad \textit{if and only if} \quad \Sigma \cup \{C_1, C_2\} \textit{ is satisfiable,}$$

where: $C_1 \equiv (\neg p_1 \vee \ldots \vee \neg p_n \vee \neg \Psi)$, $C_2 \equiv (\Psi \vee q_1 \vee \ldots \vee q_m)$, *and* Ψ *is a new symbol not appearing in* Σ *or in* C.

Proof:

$\Leftarrow$ Suppose $\Sigma \cup \{C_1, C_2\}$ is satisfiable. C is an implicate of C_1 and C_2. Then $\Sigma \cup \{C_1, C_2, C\}$ is satisfiable. Obviously then $\Sigma \cup \{C\}$ is satisfiable.

$\Rightarrow$ Suppose $\Sigma \cup \{C\}$ is satisfiable. Then there must exist an assignment of boolean values to the symbols L of $\Sigma \cup \{C\}$, i.e. $v : L \rightarrow \{true, false\}$, such that each clause from $\Sigma \cup \{C\}$ evaluates to true. Then the following must be true:

$$(\exists p_i \in C, \text{ s.t. } v(p_i) = false) \vee (\exists q_i \in C, \text{ s.t. } v(q_i) = true).$$

Then either C_1 or C_2 or both evaluate to true under the assignment v. If both C_1 and C_2 evaluate to true then we are done since then all the clauses from $\Sigma \cup \{C_1, C_2\}$ evaluate to true. In case only C_1 evaluates to true, since v does not assign any value to Ψ, one can extend v by the assignment $(\Psi, true)$. Under the new assignment all the clauses from $\Sigma \cup \{C_1, C_2\}$ evaluate to true. In case only C_2 evaluates to true under v, then one can extend v with the assigment $(\Psi, false)$. In all cases one can build an assignment of boolean values to the symbols such that all the clauses from $\Sigma \cup \{C_1, C_2\}$ evaluate to true, thus the set of clauses is satisfiable. **q.e.d.**

C.12 Corollary 1 to Lemma 4.4.2

Let Σ, C, C_1, C_2 and Ψ be as in Lemma 4.4.2. Let F be an arbitrary propositional formula not involving Ψ. Then:

$$\Sigma \cup \{C\} \models F \quad \textit{if and only if} \quad \Sigma \cup \{C_1, C_2\} \models F.$$

Proof: $\Sigma \cup \{C\} \models F$ if and only if $\Sigma \cup \{C\} \cup \{\neg F\}$ is not satisfiable. Applying Lemma 4.4.2 we get that $\Sigma \cup \{\neg F\} \cup \{C_1, C_2\}$ is not satisfiable, i.e. $\Sigma \cup \{C_1, C_2\} \models F$. **q.e.d.**

C.13 Corollary 2 to Lemma 4.4.2

Let Σ be a finite clause set, and $\Sigma_p \cup \Sigma_H$ be the transformation of Σ where each mixed clause is replaced as in Lemma 4.4.2. Then:

1. *Σ is satisfiable if and only if $\Sigma_p \cup \Sigma_H$ is satisfiable;*
2. *$\Sigma \models F$ if and only if $\Sigma_p \cup \Sigma_H \models F$, where F is a propositional formula not involving any of the new literals introduced in $\Sigma_p \cup \Sigma_H$.*

Proof:
1: A simple induction on the number of mixed clauses from Σ.
2: Immediate consequence of the first point.
q.e.d.

C.14 Lemma 4.4.3

Let Σ be a finite and satisfiable set of clauses and $\Sigma_p \cup \Sigma_H$ its encoding, where each mixed clause is replaced as in Lemma 4.4.2. Let $\mathcal{A}$ be a set of symbols returned by SAT when called on $\Sigma_p \cup \Sigma_H$. Then:

1. *$\Sigma \cup \mathcal{A}$ is satisfiable. Moreover, the assignment of boolean values $v(a) = true \Leftrightarrow (\mathcal{A} \cup \Sigma_H \models a)$ to the symbols from Σ is satisfying all the clauses from Σ;*
2. *if $\Sigma \models n$ then $\mathcal{A} \cup \Sigma_H \models n$, for any propositional formula n;*
3. *$\mathcal{A} \cup \Sigma_H \models n$ does not imply that $\Sigma \models n$;*
4. *if $\mathcal{A} \cup \Sigma_H \models n$, then $\mathcal{A} \cup \Sigma \models n$, for any* positive *literal n.*

Proof: Let us note with L_Φ the set of new symbols introduced in $\Sigma_p \cup \Sigma_H$ due to the encoding of the mixed clauses from Σ.
1: We know that (cf. Lemma 4.4.1) $\mathcal{A} \cup \Sigma_p \cup \Sigma_H$ is consistent. Each clause from Σ is either in $\Sigma_p \cup \Sigma_H$, or is a resolvent of two clauses from $\Sigma_p \cup \Sigma_H$. Then $\mathcal{A} \cup \Sigma_p \cup \Sigma_H \cup \Sigma$ is consistent, and so is $\mathcal{A} \cup \Sigma$. Moreover, since we know that the boolean assignment satisfies each clause from $\Sigma_p \cup \Sigma_H$, and since each clause from Σ is an implicate of $\Sigma_p \cup \Sigma_H$, we conclude that the above assignment v satisfies each clause from Σ.
2: Assume $\Sigma \models n$. Since Σ is consistent, clearly n cannot involve literals from L_Φ. As Corollary 2 to Lemma 4.4.2 states, then $\Sigma_p \cup \Sigma_H \models n$. But we know that $\mathcal{A} \models C$, for all $C \in \Sigma_p$. Then also $\mathcal{A} \cup \Sigma_H \models n$.
3: It is easy to give counterexamples, even when n does not mention symbols from L_Φ. Consider, for instance the theory $\Sigma \equiv \{a \vee b\}$, and $\mathcal{A} = \{a\}$;
4: Let n be a symbol such that $\mathcal{A} \cup \Sigma_H \models n$. Taking in account the way Σ was replaced with $\Sigma_p \cup \Sigma_H$, we conclude that Σ_H can be partitioned into two disjoint sets, $\Sigma_H = \Sigma_{H\Sigma} \cup \Sigma_{Hn}$, where $\Sigma_{H\Sigma} \subseteq \Sigma$. All the clauses from Σ_{Hn} are purely negative. Then, applying the resolution, it is clear that any positive literal derived from $\mathcal{A} \cup \Sigma_{H\Sigma} \cup \Sigma_{Hn}$ is derived using only clauses from $\mathcal{A} \cup \Sigma_{H\Sigma}$ (the purely negative clauses are not relevant for the derivation of positive literals). But then $\mathcal{A} \cup \Sigma \models n$.
q.e.d.

Appendix D

Proofs of Chapter 5

D.1 Property 5.3.9

We first prove two additional properties:

Property D.1.1 *Let Sys be a system and $\mathcal{A}, \mathcal{A}'$ be sets of first-order sentences. Then*

$$\mathcal{A} \subseteq \mathcal{A}' \Rightarrow NextState(Sys, \mathcal{A}) \subseteq NextState(Sys, \mathcal{A}').$$

Proof: Let $(s_i = v_j) \in NextState(Sys, \mathcal{A})$. According to Definition 5.3.3, it means that $\exists s'_i \in NextStateVars$ s.t. $s_i = \mathcal{F}(s'_i)$ and $SD \cup \mathcal{A} \models (s'_i = v_j)$. Since $\mathcal{A} \subseteq \mathcal{A}'$, we have that $SD \cup \mathcal{A}' \models (s'_i = v_j)$, and thus $(s_i = v_j) \in NextState(Sys, \mathcal{A}')$. **q.e.d.**

Property D.1.2 *Let Sys be a system, d be a state assignment and $ioseq$ be an input-observation sequence for Sys such that d is consistent with $ioseq$, i.e. there exists a path description pd associated with d and $ioseq$. Then for any $d' \subseteq d$ and for any $ioseq' \subseteq ioseq$ we have:*

- *d' is consistent with $ioseq'$, i.e. there exists a path description pd' associated with d' and $ioseq'$;*
- *$pd' \subseteq pd$.*

Proof: It is easy to prove this by induction on the length k of the sequences.

If the length is zero, i.e. the sequences $ioseq, ioseq', pd, pd'$ are empty, the property is trivially true since $SD \cup d'$ is satisfiable if $SD \cup d$ is satisfiable and $d' \subseteq d$. If the length is 1, then $ioseq = ((i,o))$ and $pd = ((d,i,o))$. Since $SD \cup d \cup i \cup o$ is satisfiable, $d' \subseteq d$ and $(i',o') \subseteq (i,o)$ we have that $SD \cup d' \cup i' \cup o'$ is satisfiable, thus $pd' = ((d',i',o'))$ is a path description for d' and $ioseq'$. Moreover, $pd' \subseteq pd$.

Induction hypothesis: Assume that the property is true for all state assignments and for all input-observation sequences of length k, $k \leq n$. We show that the property is satisfied for any state assignments and input-observation sequence of length $k = n+1$.

Assume d_0, d_0' are state assignments, $d_0' \subseteq d_0$, and ios_{n+1}, ios'_{n+1} are input-observation sequences of length $n+1$, $ios'_{n+1} \subseteq ios_{n+1}$, and d_0 is consistent with ios_{n+1}. Let

$$ios_{n+1} = ((i_o,o_o),(i_1,o_1),\ldots(i_n,o_n)),$$
$$ios'_{n+1} = ((i'_o,o'_o),(i'_1,o'_1),\ldots(i'_n,o'_n)).$$

Since d_0 is consistent with ios_{n+1} we have a path description $pd_{n+1} = ((d_o,i_o,o_o),(d_1,i_1,o_1),\ldots(d_n,i_n,o_n))$, where $d_{j+1} = NextState(Sys, d_j \cup i_j \cup o_j)$ for $0 \leq j < n$. Obviously, $SD \cup d'_o \cup i'_o \cup o'_o$ is consistent. Cf. Property D.1.1, $d'_1 = NextState(Sys, d' \cup i'_o \cup o'_o) \subseteq d_1$. Then, due to the induction hypothesis there exists a length n path description $pd'_n = ((d'_1,i'_1,o'_1),\ldots(d'_n,i'_n,o'_n))$, $pd'_n \subseteq pd_n$, where $pd_n = ((d_1,i_1,o_1),\ldots(d_n,i_n,o_n))$. But the sequence $pd'_{n+1} = ((d'_o,i'_o,o'_o).pd'_n)$ is a transition path for d'_o and ios'_{n+1}. Moreover, $pd'_{n+1} \subseteq pd_{n+1}$. **q.e.d.**

We can now prove **Property 5.3.9**:

1. *Any superset of a conflict for an input-observation sequence is a conflict for that input-observation sequence.*
2. *If c is a conflict for ios, then c is a conflict for any input-observation sequence $ios' \cdot tail$, where $ios \subseteq ios'$, $tail$ is an arbitrary input-observation sequence, and "$\cdot$" denotes the concatenation of two sequences.*

Proof: We prove both points together, namely assuming that c is a conflict for ios, we prove that any superset c^* of c is a conflict for any input-observation sequence $ios^* = ios' \cdot ios''$ if $ios \subseteq ios'$.

Assume, by contrary, that there exist a certain superset c^*, $c \subseteq c^*$ which is consistent with some $ios^* = ios' \cdot ios''$. Then there exists a path description pd^* for c^* and ios^*. Of course, pd^* can be split into two sequences, i.e. $pd^* = pd' \cdot pd''$, where pd' is a path description for ios' and c^*. But then c^*, ios' and c, ios are as in Property D.1.2, thus c is consistent with ios, which contradicts the fact that c is a conflict for ios. **q.e.d.**

D.2 Theorem 5.3.12

Let Σ be a set of clauses and M be a conjunction *of literals.*
Let Ψ be a new positive literal not appearing in Σ or in M. Then:

$$MinSupp(M, \Sigma) = MinSupp(\Psi, \Sigma \cup \{M \Rightarrow \Psi\}) - \{\neg\Psi\};$$

Proof: We show that:

1. $\forall S \in MinSupp(M, \Sigma)$:
   $S \in MinSupp(\Psi, \Sigma \cup \{\neg M \vee \Psi\}) - \{\neg\Psi\}$;
2. $\forall S \in MinSupp(\Psi, \Sigma \cup \{\neg M \vee \Psi\}) - \{\neg\Psi\}$:
   $S \in MinSupp(M, \Sigma)$;

Proof of 1:
Let $S \in MinSupp(M, \Sigma)$. This means that (cf. Def. 2.3.1):

$$\Sigma \cup \neg S \text{ is satisfiable;} \tag{D.1}$$

$$\Sigma \models S \vee M; \text{ and} \tag{D.2}$$

no proper subset of S has the first two properties. Because Ψ does not appear in Σ or in M, Ψ cannot appear in S, for otherwise the minimality of S would be violated. Thus, it is clear that $S \neq \neg\Psi$. Since Ψ does not appear in M, it is clear that $\neg M \vee \Psi$ is satisfiable. Furthermore, since Ψ does not appear in $\Sigma \cup \neg S$, from D.2 we have that

$$\Sigma \cup \neg S \cup \{\neg M \vee \Psi\} \text{ is satisfiable.} \tag{D.3}$$

From D.2 we get: $\Sigma \cup \{\neg M \vee \Psi\} \models S \vee M$, and further

$$\Sigma \cup \{\neg M \vee \Psi\} \models S \vee \Psi \tag{D.4}$$

Thus, from D.3 and D.4 we know that S is a support clause of Ψ w.r.t. the set of clauses $\Sigma \cup \{\neg M \vee \Psi\}$. Since $S \neq \neg\Psi$ what is left to be proved is that no proper subset of S has properties D.3 and D.4. Assume, by contrary that there exists $S' \subset S$ such that $\Sigma \cup \neg S' \cup \{\neg M \vee \Psi\}$ is satisfiable; and $\Sigma \cup \{\neg M \vee \Psi\} \models S' \vee \Psi$. This means that

$$\Sigma \cup \neg S' \text{ is satisfiable;} \tag{D.5}$$

$$\Sigma \cup \{\neg M \vee \Psi, \neg S', \neg\Psi\} \text{ is contradictory} \tag{D.6}$$

The last equation gives that: $\Sigma \cup \{\neg M, \neg\Psi, \neg S'\}$ is contradictory. But since Ψ does not appear in Σ, M or S', we conclude that

$$\Sigma \cup \{\neg M, \neg S'\} \text{ is contradictory} \tag{D.7}$$

which means that $\Sigma \models S' \vee M$. The last statement together with D.5 contradicts the minimality of S w.r.t. properties D.1 and D.2.

Proof of 2:
Let $S \in MinSupp(\Psi, \Sigma \cup \{\neg M \vee \Psi\}) - \{\neg\Psi\}$. Cf. Definition 2.3.1:

$$\Sigma \cup \neg S \cup \{\neg M \vee \Psi\} \text{ is satisfiable;} \tag{D.8}$$

$$\Sigma \cup \{\neg M \vee \Psi\} \models S \vee \Psi; \text{ and} \tag{D.9}$$

no proper subset of S has the above two properties. Since $\neg\Psi$ is a minimal support clause for Ψ, and S is a different one, we conclude that S does not include $\neg\Psi$. But S does not include Ψ as well, for otherwise, $S - \{\Psi\}$ would satisfy D.8 and D.9, thus contradicting the minimality of S. From D.8 and D.9 we have:

$$\Sigma \cup \neg S \text{ is satisfiable; and} \tag{D.10}$$

$$\Sigma \cup \{\neg M \vee \Psi, \neg S, \neg\Psi\} \text{ is contradictory;} \tag{D.11}$$

The last equation means that $\Sigma \cup \{\neg M, \neg S, \neg\Psi\}$ is contradictory. Furthermore, since neither Σ, nor M, nor $\neg S$ mention Ψ this means that $\Sigma \cup \{\neg M, \neg S\}$ is contradictory; which is:

$$\Sigma \models S \vee M; \tag{D.12}$$

From D.10 and D.12 we see that S is a support clause of M w.r.t. Σ. We still have to prove the minimality of S. Assume, by contrary, that exists $S' \subset S$ such that:

$$\Sigma \cup \neg S' \text{ is satisfiable; and } \Sigma \models S' \vee M;$$

Clearly, S' does not contain Ψ or $\neg\Psi$. Using the same arguments as in the first part of the proof we conclude that S' is a support clause of Ψ w.r.t. $\Sigma \cup \{\neg M \vee \Psi\}$, which contradicts the minimality of S w.r.t. the properties D.8 and D.9.

q.e.d.

D.3 Theorem 5.3.13

Let Sys be a system and d, d' be state assignments, i be an input assignment for Sys, and o be a set of first-order sentences. Then:

$$(d, i, o) \in BackTranz(Sys, d')$$
$$\Leftrightarrow$$
$$NegCl(d \cup i \cup o) \in MinSupp(Conj(\mathcal{F}^{-1}(d')), SD),$$

where: $NegCl(\{l_1, l_2, \ldots\}) = (\neg l_1 \vee \neg l_2 \vee \ldots)$,
$Conj(\{l_1, l_2, \ldots\}) = (l_1 \wedge l_2 \wedge \ldots)$,
$\mathcal{F}^{-1}(\{..s_i = v_j..\}) = \{..\mathcal{F}^{-1}(s_i) = v_j..\} = \{..s'_i = v_j..\}$.

Proof:
If we make more explicit the definition of $BackTranz$ we get: $(d, i, o) \in BackTranz(Sys, d')$ iff

$$SD \cup d \cup i \cup o \text{ is satisfiable;} \tag{D.13}$$

$$d' \subseteq NextState(Sys, d \cup i \cup o); \tag{D.14}$$

$$\text{No proper subsets of } d, i, o \text{ have properties D.13, D.14;} \tag{D.15}$$

If we make more explicit the definition of $MinSupp$ we get: $NegCl(d \cup i \cup o) \in MinSupp(Conj(\mathcal{F}^{-1}(d')), SD)$ iff

$$SD \cup d \cup i \cup o \text{ is satisfiable;} \tag{D.16}$$

$$SD \models (NegCl(d \cup i \cup o) \Rightarrow Conj(\mathcal{F}^{-1}(d'))); \tag{D.17}$$

No proper subsets of d, i, o have properties D.16, D.17; (D.18)

We show that the relations D.14 and D.17 are equivalent.
$d' \subseteq NextState(Sys, d \cup i \cup o) \Leftrightarrow$
$(\forall(s_i = v_j) \in d', \exists s'_i \in NextStateVars$ s.t.
$s_i = \mathcal{F}(s'_i) \wedge SD \cup d \cup i \cup o \models (s'_i = v_j))$.
Since $\mathcal{F}$ is bijective, the last relation is equivalent with:
$(\forall(s'_i = v_j) \in \mathcal{F}^{-1}(d'), SD \cup d \cup i \cup o \models (s'_i = v_j))$.
This is further equivalent with:
$SD \cup d \cup i \cup o \models Conj(\mathcal{F}^{-1}(d'))$, i.e.
$SD \models (NegCl(d \cup i \cup o) \Rightarrow Conj(\mathcal{F}^{-1}(d'))$. **q.e.d.**

D.4 Theorem 5.3.14

Let Sys be a system, $ioseq$ be an input-observation sequence for Sys and (i, o) be an input-observation pair for Sys. Let c be a state assignment for Sys. Then:

$$c \text{ is a conflict for the sequence } ((i, o).ioseq) \quad \Leftrightarrow$$

either:

1. *$SD \cup i \cup o \cup c$ is not satisfiable, or*
2. *there exists c'' a minimal conflict for $ioseq$, and $i' \subseteq i$, $o' \subseteq o$, $c' \subseteq c$, such that $(c', i', o') \in BackTranz(Sys, c'')$.*

Proof:

$\Rightarrow$: Let c be a conflict for $((i, o).ioseq)$. Assume, by contrary, that $SD \cup i \cup o \cup c$ is satisfiable and for any minimal conflict c'' for $ioseq$ and for any $i' \subseteq i$, $o' \subseteq o$, $c' \subseteq c$, we have that $(c', i', o') \notin BackTranz(Sys, c'')$. Then (cf. Def. 5.3.11) for any minimal conflict c'' of $ioseq$ we have $c'' \not\subseteq NextState(Sys, c \cup i \cup o)$. This means that $NextState(Sys, c \cup i \cup o)$ is not a conflict for $ioseq$, i.e. there exists a path description pd for $ioseq$ whose initial state assignment is $NextState(Sys, c \cup i \cup o)$. But then $((c, i, o).pd)$ is a path description for $((i, o).ioseq)$, which contradicts the assumption that c is a conflict for $((i, o).ioseq)$.

$\Leftarrow$: If $SD \cup i \cup o \cup c$ is not satisfiable, then c is a conflict for $((i, o))$ and cf. Property 5.3.9 c is a conflict for $((i, o).ioseq)$.

Suppose $SD \cup i \cup o \cup c$ is satisfiable and there exists a conflict c'' for $ioseq$, and $i' \subseteq i$, $c' \subseteq c$, $o' \subseteq o$ such that $(c', i', o') \in BackTranz(Sys, c'')$. Then, according to Definition 5.3.11, $c'' \subseteq NextState(Sys, c \cup i \cup o)$. Then $NextState(Sys, c \cup i \cup o)$ is a conflict for $ioseq$, and thus c is a conflict for $((i, o).ioseq)$.

q.e.d.

D.5 Property 5.4.3

Let Sys be a PSS. Let I be an input assignment, D be a steady state under the input I and o be a set of first order sentences such that $SD \cup I \cup D \cup o$ is inconsistent. Let TP_j be the family of transition paths: $TP_j = ((D, I, \{\}), \ldots, (D, I, \{\}))$, where TP_j has the length $j \geq 0$. Let $CflSet_j$ be the set of minimal conflicts for the initial state of TP_j extended with the pair (D, I, o). Then: there exist two finite integers $L \geq 0, P \geq 1$ such that $\forall j, j \geq L : CflSet_{(j+P)} = CflSet_j$.

Proof: According to Corollary of Theorem 5.3.14, since $SD \cup D \cup I$ is consistent, we have for any $j \geq 0$:

$c \in CflSet_{j+1} \Leftrightarrow c$ is a minimal element of $MapBack(CflSet_j)$

where, $MapBack(CflSet_j) := \{c \subseteq D \mid \exists (i' \subseteq I, c' \in CflSet_j) : (c, i', o') \in BackTranz(Sys, c')\}$;
Clearly, $CflSet_j \in \mathcal{P}(D)$, where $\mathcal{P}(D)$ denotes the power set of D. This means that the series $CflSet_0, CflSet_1, \ldots$ takes values from a finite domain. Then there exist two positive integers n, m, $n < m$ such that $CflSet_n = CflSet_m$. But then $CflSet_{(n+1)} = MapBack(CflSet_n) = MapBack(CflSet_m) = CflSet_{(m+1)}$. By induction it is trivial to show now that

$$\forall j \geq n : CflSet_{(j+m-n)} = CflSet_j.$$

The role of L, P from the statement of the property can be played by n respectively $m - n$. **q.e.d.**

D.6 Theorem 5.5.5

We first give:

Property D.6.1 *Let Sys be a system, and m be a state assignment for Sys. Let seq, seq' be two path descriptions for Sys having the length $k \geq 1$. Then:*

$$(seq \subseteq seq' \wedge seq \in Expl_k(Sys, m)) \Rightarrow seq' \in Expl_k(Sys, m).$$

Proof: An immediate consequence of the definitions and of Property D.1.1 **q.e.d.**

The theorem states:

$\forall k \geq 1, MinExpl_{k+1}(Sys, m, io)$ *is the set of the minimal elements of the set:* $\{((d'_o, i_o, o_o), (d'_1, i_1, o_1), \ldots, (d'_k, i_k, o_k)) \mid$

$$\exists((d_1, i_1, o_1), \ldots, (d_k, i_k, o_k)) \in MinExpl_k(Sys, m)$$
$$s.t.\ ((d'_o, i_o, o_o)) \in MinExpl_1(Sys, d_1, ((i_1, o_1), \ldots, (i_k, o_k)) \cdot io),\ and$$
$$\forall j, 0 \leq j < k : d'_{j+1} = NextState(Sys, i_j \cup o_j \cup d'_j)\},$$

where "$\cdot$" denotes the concatenation of two sequences.

Proof: The proof has two parts:

1. we prove that for any sequence $seq = ((d'_o, i_o, o_o), (d'_1, i_1, o_1), \ldots, (d'_k, i_k, o_k))$ if

   $$\exists((d_1, i_1, o_1), \ldots, (d_k, i_k, o_k)) \in MinExpl_k(Sys, m, io) \text{ s.t.}$$
   $$((d'_o, i_o, o_o)) \in MinExpl_1(Sys, d_1, ((i_1, o_1), \ldots, (i_k, o_k)) \cdot io),$$
   $$\forall j, 0 \leq j < k : d'_{j+1} = NextState(Sys, i_j \cup o_j \cup d'_j)\},$$

   then seq is an explanation of m consistent with io, but not necessarily minimal.
2. we prove that for any $seq \in MinExpl_{k+1}(Sys, m, io)$ we have that $seq = (d'_o, i_o, o_o), \ldots, (d'_k, i_k, o_k))$ and

   $$\exists((d_1, i_1, o_1), \ldots, (d_k, i_k, o_k)) \in MinExpl_k(Sys, m, io) \text{ s.t.}$$
   $$((d'_o, i_o, o_o)) \in MinExpl_1(Sys, d_1, ((i_1, o_1), \ldots, (i_k, o_k)) \cdot io),$$
   $$\forall j, 0 \leq j < k : d'_{j+1} = NextState(Sys, i_j \cup o_j \cup d'_j)\}.$$

Proof of 1: Let $seq = ((d'_o, i_o, o_o), (d'_1, i_1, o_1), \ldots, (d'_k, i_k, o_k))$ such that

$$\exists((d_1, i_1, o_1), \ldots, (d_k, i_k, o_k)) \in MinExpl_k(Sys, m, io) \text{ s.t.}$$
$$((d'_o, i_o, o_o)) \in MinExpl_1(Sys, d_1, ((i_1, o_1), \ldots, (i_k, o_k)) \cdot io),$$
$$\forall j, 0 \leq j < k : d'_{j+1} = NextState(Sys, i_j \cup o_j \cup d'_j)\},$$

Then we know that $d_1 \subseteq d'_1$ and that d'_1 is consistent with $((i_1, o_1), \ldots, (i_k, o_k)) \cdot io)$. This means that there exists a path description for $((i_1, o_1), \ldots, (i_k, o_k)) \cdot io)$ whose initial state assignment is d'_1. Then this path description is $((d'_1, i_1, o_1), \ldots, (d'_k, i_k, o_k)) \cdot pd_{io}$, where pd_{io} is a path description for the sequence io whose initial state is $NextState(Sys, i_k \cup o_k \cup d'_k)$. Cf. Property D.6.1 then $((d'_1, i_1, o_1), \ldots, (d'_k, i_k, o_k)) \in Expl_k(Sys, m)$. Then $seq \in Expl_{k+1}(Sys, m)$, and since $NextState(Sys, i_k \cup o_k \cup d'_k)$ is consistent with io, $seq \in Expl_{k+1}(Sys, m, io)$. This completes the first part of the proof.

Proof of 2: Let $seq = ((d'_o, i_o, o_o), (d'_1, i_1, o_1), \ldots, (d'_k, i_k, o_k)) \in MinExpl_{k+1}(Sys, m, io)$. Of course, $\forall j, 0 \leq j < k : d'_{j+1} = NextState(Sys, i_j \cup o_j \cup d'_j)$. Clearly, $seq'_k = ((d'_1, i_1, o_1), \ldots, (d'_k, i_k, o_k)) \in Expl_k(Sys, m, io)$. Then there must exist a minimal path description $seq_k = ((d_1, i'_1, o'_1), \ldots, (d_k, i'_k, o'_k)) \in MinExpl_k(Sys, m, io)$ s.t. $seq_k \subseteq seq'_k$. We show that $i'_j = i_j, o'_j = o_j$ for all $1 \leq j \leq k$.

Assume, by contrary, that this is not true, i.e. at least for a j we have $i'_j \subset i_j$ or $o'_j \subset o_j$. Let us then take a look at the following sequence: $seq_{k+1} = ((d'_o, i_o, o_o), (d''_1, i'_1, o'_1), \ldots, (d''_k, i'_k, o'_k))$, where $d''_{j+1} = NextState(Sys, i'_j \cup o'_j \cup d''_j)$ for all $j, 1 \leq j < k$, and $d''_1 = d'_1 = NextState(Sys, i_o \cup o_o \cup d'_o)$. Cf. Property D.1.2 then seq_{k+1} is a path description, $seq_{k+1} \subset seq$. Since $d_1 \subseteq d''_1 = d'_1$, $seq_k \subseteq seq''_k = ((d''_1, i'_1, o'_1), \ldots, (d''_k, i'_k, o'_k))$. Then (cf. Property D.6.1) $seq''_k \in Expl_k(Sys, m)$, and then $seq_{k+1} \in Expl_{k+1}(Sys, m)$. Since $seq_{k+1} \subset seq$ (cf. Property D.1.1), $NextState(Sys, d''_k \cup i'_k \cup o'_k) \subseteq NextState(Sys, d'_k \cup i_k \cup o_k)$. Then, since $NextState(Sys, d'_k \cup i_k \cup o_k)$ is consistent with io we conclude (cf. Property D.1.2) that $NextState(Sys, d''_k \cup i'_k \cup o'_k)$ is consistent with io. From this and the fact that $seq_{k+1} \in Expl_{k+1}(Sys, m)$

we conclude that $seq_{k+1} \in Expl_{k+1}(Sys, m, io)$. The last statement, together with the fact that $seq_{k+1} \subset seq$, contradict the fact that $seq \in MinExpl_{k+1}(Sys, m, io)$.

Thus, there exists $seq_k \in MinExpl_k(Sys, m, io)$, s.t. $seq_k = ((d_1, i_1, o_1), \ldots, (d_k, i_k, o_k))$ and $d_1 \subseteq d'_1$. Of course, $((d'_o, i_o, o_o)) \in Expl_1(Sys, d_1, ((i_1, o_1), \ldots, (i_k, o_k)) \cdot io)$. We still have to show that $((d'_o, i_o, o_o))$ is a minimal element of that set.

Assume, by contrary, that there exists $((d^*_o, i^*_o, o^*_o)) \subset ((d'_o, i_o, o_o))$, s.t. $((d^*_o, i^*_o, o^*_o)) \in Expl_1(Sys, d_1, ((i_1, o_1), \ldots, (i_k, o_k)) \cdot io)$. Then there exists a path description
$seq^* = ((d^*_o, i^*_o, o^*_o), (d^*_1, i_1, o_1), \ldots, (d^*_k, i_k, o_k))$ such that
$NextState(Sys, i_k \cup o_k \cup d^*_k)$ is consistent with io. Cf. Property D.1.2, since $d_1 \subseteq d^*_1$, we get
$seq_k \subseteq seq^*_k = ((d^*_1, i_1, o_1), \ldots, (d^*_k, i_k, o_k))$, and then (cf. Property D.6.1) $seq^*_k \in Expl_k(Sys, m)$. Moreover, since
$NextState(Sys, i_k \cup o_k \cup d^*_k)$ is consistent with io:
$seq^*_k \in Expl_k(Sys, m, io)$. Then $seq^* \in Expl_{k+1}(Sys, m, io)$ and this violates the minimality of seq, since $seq^* \subset seq$.

q.e.d.

Appendix E

Extended Abstract in German

Bibliography

[All83] J. Allen. Maintaining knowledge about temporal intervals. *Communications of ACM*, 26(11):832–843, 1983.

[BB92] Rene Bakker and M. Bourseau. Pragmatic Reasoning in Model-Based Diagnosis. In *Proc. 10th European Conference on Artificial Intelligence (ECAI'92)*, pages 734–738, Vienna, Austria, 1992.

[Byl89] Tom Bylander. Complexity of model-based diagnosis. Technical Report 89-TB-COMPMOD, Ohio State University, 1989.

[Byl90] Tom Bylander. Some causal models are deeper than others. *Artificial Intelligence in Medicine*, 2(3):123–128, 1990.

[CDT89] Luca Console, D.T Dupré, and P. Torasso. A theory of diagnosis for incomplete causal models. In *Proc. 11th International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 1311–1317, Detroit (USA), 1989.

[Cha72] C. Chang. The decomposition principle for theorem proving systems. In *Proc. 10th Annual Allerton Conf. on Circuit and System Theory*, University of Illinois, 1972.

[Cla93] Claudia Böttcher and Oskar Dressler. Diagnosis process dynamics: Holding the diagnostic trackhound in leash. In *Proc. Intern. Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 1460–1465, 1993.

[Cla95] Claudia Böttcher and Matthias Schick. Diagnosing structural faults by detecting hidden interactions. In *Proc. Intern. Joint Conference on Artificial Intelligence (IJCAI'95)*, Montreal, Canada, 1995. Also in Proc. DX'94.

[CPDT92] Luca Console, Luigi Portinale, D.T Dupré, and P. Torasso. Diagnostic reasoning accross different time-points. In *Proc. 10th European Conference on Artificial Intelligence (ECAI'92)*, pages 369–373, Vienna (Austria), 1992.

[CT91] Luca Console and P. Torasso. A spectrum of logical definitions of model-based diagnosis. *Computational Intelligence*, 7(3):133–141, 1991. Also in: [HCdKe92].

[CT94] Marie-Odile Cordier and Thiebaux. Event-based diagnosis for evolutive systems. In *Proc. 5th Internatinal Workshop on Principles of Diagnosis (DX'94)*, New Platz (USA), 1994.

[Dav84] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24(1):347–410, 1984. Also in: [HCdKe92].

[DF90] Oskar Dressler and Adam Farquhar. Putting the problem solver back in the driver's seat: Contextual control of the atms. In *Lecture Notes in AI 515*. Springer Verlag, 1990.

[DJD+91] Philippe Dague, O. Jehl, P. Deves, P. Luciani, and P. Taillibert. When oscillators stop oscillating. In *Proc. 12th Int. Joint Conf. on Artificial Intelligence (IJCAI '91)*, pages 1109–1115, Sydney, Australia, 1991. Also in: [HCdKe92].

[dK86a] Johan de Kleer. An assumption based truth maintenance system. *Artificial Intelligence*, 28:127–162, 1986.

[dK86b] Johan de Kleer. Extending the atms. *Artificial Intelligence*, 28(2), 1986.

[dK86c] Johan de Kleer. Problem solving with the atms. *Artificial Intelligence*, 28(2), 1986.

[dK88] Johan de Kleer. A general labeling algorithm for assumption-based truth maintenance. In *Proc. AAAI'88*, pages 188–192, Saint Paul, MN, 1988.

[dK89] Johan de Kleer. A comparison of atms and csp techniques. In *Proc. 11th IJCAI'89*, Detroit, USA, 1989.

[dK90a] Johan de Kleer. Exploiting locality in a tms. In *Proc. AAAI'90*, pages 254–271, 1990.

[dK90b] Johan de Kleer. Using crude probability estimates to guide diagnosis. *Artificial Intelligence*, 45:381–392, 1990. Also in: [HCdKe92].

[dK91] Johan de Kleer. Focusing on probable diagnoses. In *Proc. AAAI'91*, pages 842–848, Anaheim CA, 1991.

[dK92a] Johan de Kleer. An improved incremental algorithm for generating prime implicates. In *Proc. 1th AAAI'92*, San Jose (CA), USA, 1992.

[dK92b] Johan de Kleer. Optimizing focusing model-based diagnosis. In *Proc. 3rd Internatinal Workshop on Principles of Diagnosis*, pages 26–29, Rosario WA, 1992.

[DK92c] D. Dvorak and B. Kuipers. Model-based monitoring of dynamic systems. In Walter Hamscher, Luca Console, and J. de Kleer, editors, *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[dKMR92] Johan de Kleer, A. Makworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56, 1992. Also in: [HCdKe92].

[dKRS91] Johan de Kleer, Olivier Raiman, and Mark Shirley. One step lookahead is pretty good. In *Proc. 2nd Internatinal Workshop on Principles of Diagnosis*, pages 136–142, Torino Italy, 1991. Also in: [HCdKe92].

[dKW87] Johan de Kleer and Brian Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987. Also in: [HCdKe92].

[dKW89] Johan de Kleer and Brian Williams. Diagnosis with behavioral modes. In *Proc. 11th International Joint Conference on Artificial Intelligence (IJCAI '89)*, pages 1324–1330, Detroit MI, 1989. Also in: [HCdKe92].

[Doy79] BJ. Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[Dre88] Oskar Dressler. An extended basic atms. In *Proc. 2nd Non-monotonic Reasoning Workshop (Lecture Notes in AI 346)*. Springer Verlag, 1988.

[Dre90] Oskar Dressler. Problem solving with the nm-atms. In *Proc. 9th European Conference on Artificial Intelligence (ECAI'90)*, pages 252–258, Stockholm, Sweden, 1990.

[Dre94] Oskar Dressler. Prediction sharing over time and contexts. In *Proc. AAAI'94*, Seattle, USA, 1994.

[Dre96] Oskar Dressler. On-Line Diagnosis and Monitoring of Dynamic Systems based on Qualitative Models and Dependency-recording Diagnosis Engines. In *Proc. 12th European Conference on Artificial Intelligence (ECAI'96)*, Budapest, Hungary, 1996.

[DS92] Oskar Dressler and Peter Struss. Back to Defaults: Characterizing and Computing Diagnoses as Coherent Assumption Sets. In *Proc. 10th European Conference on Artificial Intelligence (ECAI'92)*, pages 719–723, Vienna, Austria, 1992.

[DS94] Oskar Dressler and Peter Struss. Model-Based Diagnosis with the Default-Based Diagnosis Engine: Effective Control Strategies that Work in Practice. In *Proc. 11th European Conference on Artificial Intelligence (ECAI'94)*, Amsterdam, the Nederlands, 1994.

[FdK88] K. Forbus and J. de Kleer. Focusing the atms. In *Proc. AAAI'88*, pages 193–198, 1988.

[FdK93] K. Forbus and J. de Kleer. *Building Problem Solvers*. MIT Press, 1993.

[FGN90a] Gerhard Friedrich, Georg Gottlob, and Wolfgang Nejdl. Generating efficient diagnostic procedures from model-based knowledge using logic programming techniques. *Computers Math. Applic.*, 20(9110):57–72, 1990.

[FGN90b] Gerhard Friedrich, Georg Gottlob, and Wolfgang Nejdl. Physical impossibility instead of fault models. In *Proc. AAAI'90*, pages 331–336, Boston, USA, 1990. Also in: [HCdKe92].

[FGN92] Gerhard Friedrich, Georg Gottlob, and Wolfgang Nejdl. Formalizing the repair process. In *Proc. ECAI'92*, Vienna, Austria, 1992.

[FL91] Gerhard Friedrich and F. Lackinger. Diagnosing temporal misbehavior. In *Proc. IJCAI'91*, pages 1116–1122, Sydney, Australia, 1991.

[FN92] Gerhard Friedrich and Wolfgang Nejdl. Choosing observations and actions in model-based diagnosis/repair systems. In *Proc. 3rd Intern. Conference on Principles of Knowledge Representation and Reasoning (KR'92)*. Morgan Kaufmann, 1992.

[Gin93] M. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[Gol91] D.J. Goldstone. Controlling inequallity reasoning in a tms-based analog diagnosis system. In *Proc. 9th Nat. Conf. on AI*, pages 512–517, Anaheim, USA, 1991. Also in: [HCdKe92].

[GSR92] T. Guckenbiehl and G. Schaefer-Richter. SIDIA: Extending prediction Based Diagnosis to dynamic models. In W. Hamscher, L. Console, and J. de Kleer, editors, *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[GSW89] R. Greiner, B. Smith, and R. Wilkerson. A correction to the algorithm in reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989. Also in: [HCdKe92].

[Ham91a] Walter Hamscher. Acp: Reason maintenence and inference control for constraint propagation over intervals. In *Proc. 9th Nat. Conf. on AI*, pages 506–511, Anaheim, USA, 1991. Also in: [HCdKe92].

[Ham91b] Walter Hamscher. Modeling digital circuits for troubleshooting. *Artificial Intelligence*, 51(1-3), 1991. Also in: [HCdKe92].

[HCdKe92] Walter Hamscher, Luca Console, and J. de Kleer (eds). *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[HD84] Walter Hamscher and Randall Davis. Diagnosing circuits with state: An inherently underconstraint problem. In *Proc. 4th Nat. Conf. on AI*, pages 142–147, Austin, USA, 1984. Also in: [HCdKe92].

[HP88] J. Hunt and C. Price. Explaining qualitative diagnosis. *Engineering Applications of AI*, 1(3), 1988.

[HP89] J. Hunt and C. Price. Towards a generic qualitative-based diagnostic architecture. In *Proc. Avignon'89 Conference on Second Generation Expert Systems*, 1989.

[HW74] L. Henschen and L. Wos. Unit refutations and horn sets. *Journal of the ACM*, 21:590–605, 1974.

[Iwa94] Sebastian Iwanowski. An algorithm for model-based diagnosis that considers time. *Annals of Mathematics and Artificial Intelligence*, 11(1-4), 1994.

[JHL93] C. Price J. Hunt and M. Lee. Automating the fmea process. *Intelligent Systems Engineering*, 2(2), 1993.

[JR90] C. Joubel and O. Raiman. How time changes assumptions. In *Proc. 9th European Conference on Artificial Intelligence (ECAI'90)*, pages 378–383, Stokholm, Sweden, 1990.

[KT90] A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9:185–206, 1990.

[Kui86] B. Kuipers. Qualitative simulation. *Artificial Intelligence*, 29(3):289–338, 1986.

[Kum] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, spring 1992:32–44.

[KvdG93] Gerry Kelleher and Linda van der Gaag. The lazy rms: Avoiding work in the atms. *Computational Intelligence: An International Journal*, 9(3):239–253, 1993.

[KvR92] Gerry Kelleher and T. van Rij. The Application of Lazy RMS in Automated Diagnosis. In *Proc. ECAI 1992 Workshop on Applications of RMS*, Vienna, Austria, 1992.

[Lac91] Franz Lackinger. Model-Based Troubleshooting: Qualitative Reasoning and the Impacts of Time. Technical Report CD-TR 91/32, Vienna University of Technology, Vienna, Austria, 1991.

[Lew78] H. Lewis. Renaming a set of clauses as a horn set. *Journal of the ACM*, 25:134–135, 1978.

[Mac87] A. Mackworth. Constraint satisfaction. In S.C Saphiro, editor, *Encyclopedia of Artificial Intelligence*, pages 205–211. John Wiley and Son, 1987.

[Mar90] João Martins. The truth, the whole truth, and nothing but the truth: an indexed bibliography to the literature of truth maintenence systems. *AI Magazine*, pages 7–25, 1990.

[McA80] D. McAllester. An Outlook at Truth Maintenance Systems. Technical Report AIM-551, MIT, AI Lab., Cambridge, MA, 1980.

[McA82] D. McAllester. Reasoning Utility Package. Technical Report AIM-667, MIT, AI Lab., Cambridge, MA, 1982.

[McA90] D. McAllester. Truth maintenance. In *Proc. AAAI-90*, pages 1109–1116, 1990.

[McD91] D. McDermott. A general framework for reason maintenance. *Artificial Intelligence*, 50:289–329, 1991.

[McI94a] Sheilla McIlraith. Generating tests using abduction. In *Proc. 4th Conf. on Principles of Knowledge Representation and Reasoning (KR'94)*, pages 449–460, 1994.

[McI94b] Sheilla McIlraith. Towards a theory of diagnosis testing and repair. In *Proc. 5th Int. Workshop on Principles of Diagnosis (DX'94)*, New Platz, USA, 1994.

[Moz90] Igor Mozetiĉ. Diagnostic efficiency of deep and surface knowledge in kardio. *Artificial Intelligence in Medicine*, 2(2):67–83, 1990.

[Moz91] Igor Mozetiĉ. Hierarchical model-based diagnosis. *Int. Journal of Man-Machine Studies*, 35(3):329–362, 1991. Also in: [HCdKe92].

[Moz92] Igor Mozetiĉ. A polynomial-time algorithm for model-based diagnosis. In *Proc. 10th European Conference on Artificial Intelligence (ECAI '92)*, pages 729–793, Vienna, Austria, 1992.

[MR92] Sheilla McIlraith and Raymond Reiter. On Tests for Hypothetical Reasoning. In W. Hamscher, L. Console, and J. de Kleer, editors, *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[Ng90] T. Ng. Model-based Multiple Fault Diagnosis of Time-varying, Continuous Devices. In *Proc. 6th IEEE Conf. on AI Applic.*, Santa Barbara, USA, 1990. Also in: [HCdKe92].

[NG94] W. Nejdl and J. Gamper. Harnessing the power of temporal abstractions in model-based diagnosis of dynamic systems. In *Proc. ECAI'94*, Amsterdam, Netherlands, 1994.

[Nil71] N. Nilsson. *Problem Solving Methods in AI*. McGraw-Hill, 1971.

[OFK92] O. Oyeleye, F. Finch, and M. Kramer. Qualitative modeling and fault diagnosis of dynamic processes by midas. In W. Hamscher, L. Console, and J. de Kleer, editors, *Readings in Model-Based Diagnosis*, pages 262–275. Morgan Kaufmann, 1992.

[OR92] D.-J. Out and R.T. Rikxoort. On the construction of hierarchic models. In *Proc. ECAI'92 Workshop on Model-Based Reasoning*, Vienna, Austria, 1992.

[Out93] Dirk-Jan Out. *Strategies for efficient model-based troubleshooting*. PhD thesis, University of Twente, the Netherlands, 1993.

[Pan84] J. Pan. Qualitative reasoning with deep-level mechanism models for diagnosis of mechanism failures. In *Proc. 1st IEEE Conf. on AI Applic.*, Denver, USA, 1984. Also in [HCdKe92].

[PL88] C. Price and D. Lee. Deep Knowledge: Tutorial and Bibliography. Technical Report IKBS 3/26/048, The University College of Whales, 1988.

[PWT96] C. Price, M. Wilson, and J. Timmis. Generating fault trees from fmea. In *Proc. 7th International workshop on Principles of Diagnosis (DX96)*, Val Morin, Canada, 1996.

[Qui59] W. Quine. On cores and prime implicants of truth functions. *American Math. Monthly*, 66:755–760, 1959.

[Rai92] Olivier Raiman. Diagnosis as a Trial: The Alibi Principle. In W. Hamscher, L. Console, and J. de Kleer, editors, *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[RDB89] Michael Reinfrank, O. Dressler, and G. Brewka. On the relation between truth maintenence and autoepistemic logic. In *Proc. Intern. Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 1206–1213, 1989.

[RdK87] Raymond Reiter and Johan de Kleer. Foundations of assumption-based truth maintenance systems: Preliminary report. In *Proc. AAAI'87*, pages 183–188, 1987.

[RdKS91] Olivier Raiman, Johan de Kleer, and Vijav Saraswat. Characterizing non-intermittent faults. In *Proc. AAAI'91*, pages 849–854, Anaheim, USA, 1991. Also in: [HCdKe92].

[Rei80] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–96, 1987. Also in: [HCdKe92].

[Rei89] Michael Reinfrank. *Fundamentals and Logical Foundations of Truth Maintenance*. PhD thesis, Linköping University, Dept. of Computer and Information Science, dissertations 221, 1989.

[RJ86] L. Rabiner and B. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, 3(1):4–16, 1986.

[SD89] Peter Struß and O. Dressler. Physical negation: Integrating fault models into the general diagnostic engine. In *Proc. 11th Int. Joint Conference on Artificial Intelligence (IJCAI '89)*, pages 1318–1323, Detroit, USA, 1989. Also in: [HCdKe92].

[SMS95] P. Struss, A. Malik, and M. Sachenbacher. Qualitative modeling is the key. In *Proc. 6th Int. Workshop on Principles of Diagnosis (DX'95)*, Goslar, Germany, 1995.

[SSL+94] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. A discrete event systems approach to failure diagnosis. In *Proc. 5th Int. Workshop on Principles of Diagnosis (DX'94)*, New Platz, USA, 1994.

[Str89] Peter Struß. Diagnosis as a process. In *Working Notes of the Workshop on Model-Based Diagnosis*, Paris, France, 1989. Also in: [HCdKe92].

[Str91a] Peter Struß. Knowledge-based systems - the second generation sets to work. In *Proc. 4th GI Congress on Knowledge-based Syatems*, Munchen, Germany, 1991.

[Str91b] Peter Struß. What's in sd? towards a theory of modeling for diagnosis. In *Proc. 2nd Int. Workshop on Principles of Diagnosis*, 1991. Also in: [HCdKe92].

[Str92] Peter Struß. Knowledge-based diagnosis: An important challenge and touchstone for ai. In *Proc. ECAI'92*, pages 863–874, Vienna, Austria, 1992.

[Str94] Peter Struß. Testing for discrimination of diagnoses. In *Proc. 5th Int. Workshop on Principles of Diagnosis (DX'94)*, New Platz, USA, 1994.

[SW93] Y. Sun and D. Weld. A framework for model-based repair. In *Proc. AAAI '93*, pages 182–187, 1993.

[Tat92] Mugur Tatar. Tehnologii de realizare a sistemelor expert. Technical Report (referat pentru doctorat), Technical University of Cluj, Cluj-Napoca, Romania, Nov. 1992. (romanian).

[Tat94] Mugur Tatar. Combining the lazy label evaluation with focusing techniques in an atms. In *Proc. ECAI '94*, Amsterdam,the Netherlands, 1994.

[TI94a] Mugur Tătar and Sebastian Iwanowski. Aspects of efficient focusing. In *Proc. 5th Int. Workshop on Principles of Diagnosis (DX'94)*, New Platz, USA, 1994.

[TI94b] Mugur Tătar and Sebastian Iwanowski. Efficient Candidate Generation in a Model-Based Diagnostic Engine. Technical Report F3S-95-003, Daimler-Benz Research, Berlin, Germany, 1994.

[Tis67] P. Tison. Generalized consensus theory and application to the minimization of boolean functions. *IEEE Transactions on Electronic Computers*, 4:446–456, 1967.

[TL93] Mugur Tătar and Alfred Leţia. Embedding temporal reasoning into the atms framework. In *Proc. 2nd German Conference on Expert Systems (XPS'93)*, Hamburg, Germany, 1993.

[Wil86] B.C. Williams. Doing time: putting qualitative reasoning on firmer ground. In *Proc. 5th National Conf. on Artificial Intelligence*, pages 105–112, Philadelphia, USA, 1986.