Nostrum: Constraint Directed Diagnosis

Thesis

NOSTRUM:

Constraint Directed Diagnosis

Simon Nuttall

Thesis submitted in partial fulfillment of
requirements for Ph.D. in Artificial Intelligence.

HCRL Technical Report No. 62
May 1990

Abstract: This thesis describes the design, implementation and use of NOSTRUM, a computer program that diagnoses faults in electrical and mechanical devices. The diagnosis is driven from a model of the components of the device. The model itself represents the operating principles of the component parts of the device. By choosing to model the operating principles on a constraint based system it is easy to predict the consequences of infringement of those operating principles, and also to back propagate from an observed symptom to a hypothesized fault.

NOSTRUM interacts with the user, proposing tests to perform on the device and asking if the predicted consequences of a hypothesis are observed. NOSTRUM allows experiential knowledge in the form of fault models to be added to modify and speed up its search strategy.

At a general level NOSTRUM can perform diagnosis in novel situations, pinpointing a break in the structure of the device whilst not necessarily being able to describe the nature of that break.

Keywords: Fault Diagnosis, Constraint Propagation.

Date of submission: 29 September 1989

Date of award: 25 April 1990

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

nostrum (nos'trəm) *n.* [L., ours] **1.** a quack medicine **2.** a pet scheme for solving some problem

Contents

1. Introduction 1
1.1. Motivation 2
1.2. Overview 3
1.3. Organization 5
2. Capabilities of Existing Techniques 7
2.1. Representing Device Behaviour 7
2.1.1. Diagnosing Multiple Faults 7
2.1.2. Constraint Suspension 13
2.1.3. Qualitative Diagnosis 19
2.2. Qualitative Reasoning 21
2.2.1. Confluence Based QR 22
2.2.2. Qualitative Process Theory 26
2.2.3. Qualitative Simulation 31
2.2.4. Helix - An example of the use of QSIM in diagnostics. 33
2.2.5. Summary of Qualitative Techniques 36
2.3. Knowledge based Diagnostic Strategies 37
2.3.1. CRIB 37
2.3.2. Meta Level Reasoning 40
2.4. Summary of Diagnostic Methods. 41
3. Diagnosis: A Mixture of Skills 46
3.1. The Skills 46
3.1.1. Fault Recognition. 47
3.1.2. Symptom recognition 48
3.1.3. Tracing 48
3.1.4. Hypothesize and Test 48
3.1.5. Repair Action 52
3.2. The Diagnostic Process 54

4. Roots of a Representation 58
4.1. Nostrum - Rationale 59
4.1.1. How it works diagrams 60
4.2. Christmas Tree Lights 63
4.2.1. Constraint model of a light bulb 64
4.2.2. Circuit Constraint Model 67
4.2.3. The Switch Constraint Model 69
5. Using Nostrum to Diagnose Christmas Tree Lights .. 71
5.1. Defining the Device Structure 71
5.2. Transfer to a Frame Language 72
5.3. Production rules interpret circuit Structure 74
5.4. The Constraint Model is built. 74
5.5. Initializing the Model 76
5.6. Describing the Symptom 78
5.7. The constraint model generates hypotheses 78
5.8. NOSTRUM requests the user to perform tests 82
5.9. A Repair action is suggested. 86
6. Implementing NOSTRUM 88
6.1. Circuit Buff 89
6.1.1. Using Circuit Buff to Draw the Christmas Tree light set .. 90
6.1.2. Defining your own icons 94
6.1.3. Behind the Scenes 96
6.1.4. Defining Constraints 97
6.1.5. Concoctions 100
6.2. The Solder Knowledge Base 102
6.2.1. Constraint Propagation in Solder 106
6.2.1.1. Causal Propagation 106
6.2.1.1.1. Implementation 107
6.2.1.2. Diagnostic Constraint Propagation 109
6.2.1.2.1. Implementation 109
6.2.2. Using Heuristic Knowledge 112

7. Application of Nostrum to other domains 114
7.1. Diagnosing a Car Starting Mechanism. 114
7.1.1. Case 1: Engine turns over weakly. 119
7.1.2. Case 2: Engine Does not start. 132
7.2. Doorlock 134
8. Conclusion 140
8.1. Summary 141
8.2. Implication for Explanations 141
8.3. Limitations 142
8.4. Future Work 143
9. References 145

Acknowledgements

Many people have had to put up with me during the production of this thesis. The main sufferers are the members of the Gardiner Building at Walton Hall and the residents of Tinkers Bridge.

But in particular I would like to mention:

Marc Eisenstadt for providing a 'hackers paradise' at the Open University, and his light handed approach to supervision.

John, Tim and Enrico for their guidance on the 'AI style'.

Jitu and Arthur who were going through the same agony as myself during the completion of this thesis.

Rae Sibbitt for making encouraging comments on earlier draughts of this thesis.

Hank Kahney and his sons for providing a wealth of displacement activities.

Tony Hasemer for many cups of tea!

John Watkins for sharing an interest in pool and old cars.

Tony Willoughby for several useful discussions regarding the content of the thesis.

Clive Bailey who unwittingly taught me a lot about diesel engines and fault diagnosis.

The members of KEG.

The other residents of the Gardiner Building for providing a link to the real world.

Scientia for letting me use their machines in the final stages of preparation of this thesis.

Maradona for knocking out England in the 1986 World Cup and enabling me to win the Gardiner Building sweepstake.

This research was supported by a Science and Engineering Research Council CASE award in collaboration with British Telecommunications PLC and The Open University. The work has been carried out within The Human Cognition Research Laboratory, at Walton Hall, Milton Keynes.

I should also like to thank Sperry (now UNISYS) for providing an Explorer lisp machine on which early work for this thesis was done. Bill Pilgrim provided several interesting domains which got me started in the area of knowledge engineering.

I should also like to thank my cat Sushi for showing up and having kittens: . Eric, Emma, Harry, Nelson, Peshwari, Tikka and Dansak.

Dedication

This thesis is dedicated to all the devices that have been thrown away that could have been repaired.

Chapter One

Introduction

Most faults are frustratingly simple, yet the study of fault diagnosis is interesting because it challenges our understanding of the way devices work. NOSTRUM is a computer program designed to perform fault diagnosis on a range of electrical and mechanical devices. One of the primary goals of NOSTRUM is to diagnose a device from an understanding of the operating principles of its components and as such the challenge is to find a representation of the behaviour of devices to facilitate this.

Fault diagnosis has often been the driving force of research in Artificial Intelligence. It fueled the MYCIN [Shortliffe 1976] project which used approximately 500 production rules to diagnose illnesses in the domain of bacterial infections. It was dissatisfaction with MYCIN's generality and applicability to other related domains that led researchers into the area of model based reasoning. Several authors [Bobrow 1984] devised various qualitative representations (see chapter 2) that simulated and envisioned the behaviour of physical systems whilst at the same time being heralded as 'the way experts think'. Problems with the loss of resolution in making qualitative approximations have obsessed researchers [Raiman 1986, Struss 1988, Kuipers and Chiu 1987, and Willoughby 1989] and the original goals of diagnostic reasoning have been lost.

In parallel to the evolution of model based reasoning other researchers [Breuker and Weilinga 1985, Schreiber et al 1987, Keravnou and Johnson 1986, Clancey 1985] have concentrated on the strategies experts use to perform diagnosis. Much of this work has studied how

experts organize their knowledge and reorganize it in the light of experience. Such systems aim to reinforce the psychological validity of modelling using rule based systems by, for instance generating theories of explanation. They also approach the problem of limited applicability of expert systems by defining 'meta-rules' and 'meta-levels of reasoning' in which goals and strategies of diagnosis can be used in different domains.

Because of this history few diagnostic systems actually make use of a 'deeper model' to drive diagnosis, and those that do have largely been implemented in the well defined areas of digital electronics [DeKleer and Williams 1987, Davis 1984.], described in chapter 2. NOSTRUM is unique in working from the symptom described by the user back along the causal pathways in a device to find faults. Fundamental to the operation of NOSTRUM is the model of the device. To achieve a model that can both simulate the behaviour of the device and work back from a symptom to find a fault NOSTRUM is novel in its use of the operating principles of the components. These operating principles are implemented with a constraint network. NOSTRUM views faults as infringements of the operating principles rather than as specific breakdowns. So rather than saying that a car's fan belt has snapped NOSTRUM will say that the power link between the crankshaft pulley and the cooling system has failed. NOSTRUM does allow associations between faults like that just described and past experience, but by offering diagnoses in this style is able to cover a wider range of cases. For instance, in the preceding example, the fan-belt may not have actually broken, and instead the pulley that drives the cooling system may have started to slip on its spindle.

1.1. Motivation

NOSTRUM developed from my dissatisfaction with the way that others had tried to build diagnostic systems. I did not want to follow the shallow implementations of rule based systems because in order to

implement them all possible diagnoses have to be listed beforehand. By doing that, one can only diagnose the device in those ways and it seems that the program is not doing any work, just pattern matching. Systems like that cannot cope with new situations or invent novel diagnoses. However it appears that there should be a place for heuristic based knowledge in a good diagnostic system. Such knowledge helps speed up diagnosis.

Some diagnostic systems [eg. Helix see section 2.2.5] choose to ignore the symptom description given to them and try to reproduce the faulty behaviour of the device by systematically assuming each element in the model is the cause of the fault. Such systems simply assume the device has 'gone wrong' and try to isolate the faulty part to a replaceable unit somewhere in a structured description of the device. A lot can be learned from the symptom description. On telling a car mechanic about a new noise developing in the engine they will often ask questions like 'Does the noise only appear at high revs?', 'Has the noise been getting gradually worse?' or 'Does it only happen when you are in reverse gear?' and so on. Subtle differences in the observed symptom greatly affect the way a mechanic starts to look for faults.

NOSTRUM is a computer program designed to avoid these deficiencies. By using a model of the device being diagnosed, NOSTRUM can pin-point faults that may never have been suspected before. The model is sufficiently versatile that descriptions of the symptom help direct Nostrums' search strategy.

1.2. Overview

This thesis proposes that fault diagnosis in any domain and in novel situations can be performed by analysing the *operating principles* of the components of the device. The operating principles of a component define its behaviour, for instance they describe how a light bulb's brightness increases with voltage, and that beyond a certain voltage the bulb will fuse. The operating principles capture the physics of the

component and describe its interactions with other components. In Nostrum the operating principles are modelled using a system of constraints and constraint propagation. This has three advantages for doing fault diagnosis. Firstly the constraint mechanism can simulate the device, enabling consequences of a hypothesized fault to be predicted and compared with the real device. Secondly the constraint mechanism provides a means of following causal pathways from the manifestation of the symptom towards the fault. Finally the constraint models form a library of building blocks that can be connected together to represent the behaviour of any device.

Diagnosis of a device using NOSTRUM is a series of operations. Firstly the structure of the device is drawn in a graphical interface. The user draws icons from a palette that represents a library of components whose operating principles are known to NOSTRUM. The icons are placed on the screen and connected together with links. Next NOSTRUM uses pattern recognition rules to identify components and sub-systems of the device for which it knows the operating principles. When these have been identified NOSTRUM builds a constraint model of the device by assembling known constraint models of the components. The final effort of the user is to set up the constraint model of the device to reflect the state of the actual device. NOSTRUM then starts to do diagnosis from the point at which there is an observed difference between the real and modelled devices.

The search element of the diagnostic process is a local propagation around the constraint network of the observed difference between the actual device and the model. As the new values are proposed, by NOSTRUM, for nodes in the constraint network that might explain the difference, the effects of the new value can be explored. For instance if NOSTRUM is trying to explain why a car won't start and it is testing the idea that the petrol tank is empty then it will also expect that the fuel gauge has a zero reading. In practice these avenues are followed up by

creating hypothetical worlds in which the node has the hypothesized fault value.

By asking the user about observable or testable values Nostrum is able to pick among the hypothetical situations those which can explain the fault. Some predicted fault values may coincide with known failure modes of certain devices, for instance, if a voltage is proposed to be zero this can be explained by a flat battery. Such cases are suggested to the user who may accept or reject the hypothesis. Rejected hypotheses cause NOSTRUM to propagate further around the constraint network until an explanation can be found or until propagation comes to an end.

Nostrum doesn't claim to always find the fault in a device. The cases in which it can't are those in which the device has broken in such a way that the structure of the device is radically changed. However it can usually pin-point the location in the device where the fault has occurred. For example it will be able to recognize that force isn't being transmitted through a lever although it won't be able to explain as to why.

Nostrum requests the user to make measurements on the device. However the tests it decides to ask the user to do depend upon how easy they are to perform and how significantly the test will split the search tree.

This thesis shows how Nostrum has been applied to three diagnostic problems: A Christmas tree light set, a doorlock and the starting system of a car.

1.3. Organization

Chapter 2 of this thesis describes work already done in the area of fault diagnosis. Systems, highly algorithmic in nature, are described that have been devised to do fault diagnosis in digital electronics. There are also examples of knowledge based systems that attempt to model the inferences of an expert in making a diagnosis. I also discuss the introduction of qualitative representations of device behaviour.

In chapter 3 I begin to discuss some of the motivations behind the design of NOSTRUM - what are the steps in a diagnosis and what demands those steps make on a representation of devices. This process is continued in chapter 4 where examples of the representation are given.

Chapter 5 describes in detail how NOSTRUM is used to diagnose a simple set of Christmas tree lights. The implementation details and explanations of NOSTRUM's constraint propagation procedures are described in chapter 6. Chapter 7 describes how NOSTRUM can be applied to other domains and Chapter 8 concludes.

Chapter Two

Capabilities of Existing Techniques

In the coming sections I describe some of the representations and techniques that have been used to model devices for diagnosis. To begin with I review the work on functional representations of devices. The examples used by workers in this area are usually taken from the domain of digital electronics where the motivation is to diagnose faults in VLSI chips. Section 2.2 describes approaches to qualitative modelling of device behaviour. Finally I look at the efforts of various workers to model the strategies experts use in diagnosis.

2.1. Representing Device Behaviour

Devices can be represented as a hierarchy of sub-devices. Each sub-device can be regarded as a *black box* which takes some inputs and produces various outputs. Electronic devices are particularly amenable to this type of modelling and some techniques are described below.

2.1.1. Diagnosing Multiple Faults

DeKleer and Williams [1987] have shown how an Assumption Based Truth Maintenance (ATMS) can be used to find multiple faults in electronic devices. A functional model of the device is defined, consisting of a set of constraints which define the behaviour of each part of the device. A constraint is a mechanism used to compute the inputs and outputs of a 'black box' of known functionality. For instance Figure 2.1 shows a black box which adds the values at inputs A and B to produce the sum at C.

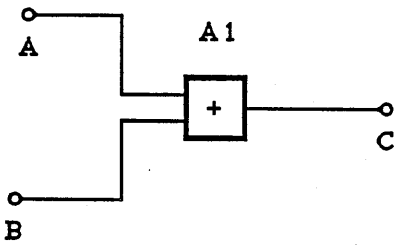


Figure 2.1. A simple two-input one output black box adder.

So for instance if the value at A is 3, and 4 at B then we expect the value at C to be 7. Conversely if C is known to be 2 and A has the value 5 then B is expected to be -3. The behaviour of the constraint is to compute the unknown value as soon as enough information is known at the other *ports*. This is known as *completing the constraint*. Such constraints can be linked together with 'wires' so that values derived at the constraints are passed as inputs to others. This process is called *constraint propagation*[1].

Simulation of a target circuit is performed by propagating values through the resulting network, but a record of which devices have been used to derive the values is also maintained for each datum see Figure 2.2. (Note: Figures 2.2, 2.3 & 2.5 are screen snapshots produced by the constraint propagation system Circuit Buff which is described in chapter 6.)

---

[1]This phrase has different interpretations in different applications, see [Stefik 1981], where propagation refers to propagation of the constraint itself through a planning network rather than propagation of actual values.

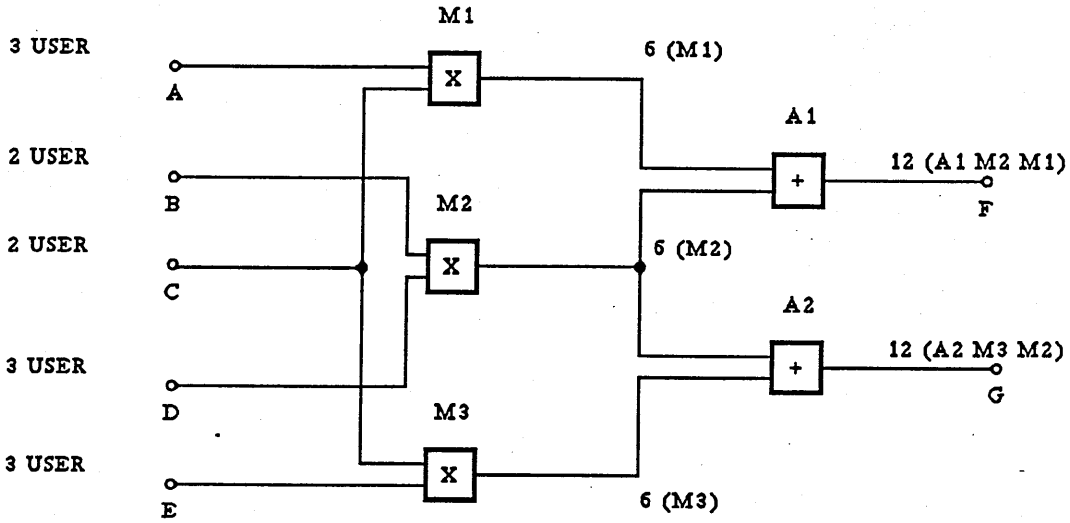


Figure 2.2. A circuit modelled with constraints. The circuit consists of three multipliers, M1-3 and two adders A1 and A2. Values on wires are shown alongside the information used to derive them. For instance the output of A1 has the value 12 assuming that A1, M1 and M2 are all working.

In the circuit shown a value of 12 is predicted at both outputs, F and G. However measurement on the real circuit might give the value 10 at F. This fact is then asserted at wire F, noting that the fact has been measured by the user. This causes a propagation of values around the circuit according to the rules of constraint propagation to give the state shown in Figure 2.3.

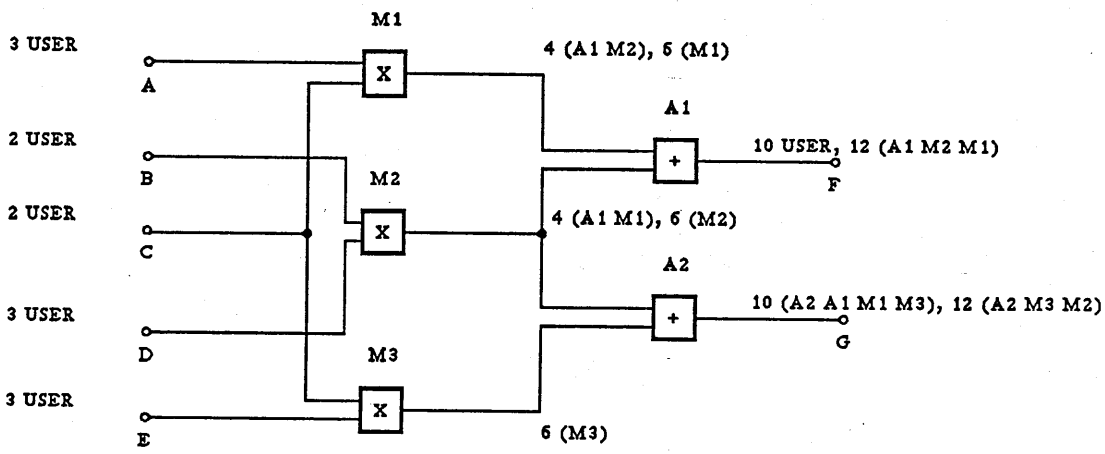


Figure 2.3. The state of the circuit model after observation that F=10. Wires are now shown with more than one potential value - depending on which sets of assumptions are believed.

To generate these values the constraints have behaved as follows:

"The value at F is 10, this is in contradiction to the predicted value of 12 which assumes {A1 M1 M2}. So either A1 is faulty or one of the inputs is wrong.

Assuming A1, and M2 are working then I predict a value of 10-6=4 at X. This value is in conflict with the value 6 at the output to M1, so either M1 is broken or its inputs are wrong. The inputs to M1 are user-measured and so have to be believed; constraint propagation down this channel halts.

Assuming A1, and M1 are working then as above I predict a value of 4 at Y. This value cannot be back propagated through M2 because its inputs are user asserted, but the 4 on Y can propagate with the 6 at Z to predict a new value of 10 at G with assumptions (A1 M1 A2 M3)."

The work of the constraint propagator is merely to produce the new values and the assumptions associated with them. The ATMS generates candidates by analysing which combinations of correctly functioning devices lead to inconsistencies. For instance from the above discussion it is apparent that not all of {A1 M1 M2} can be working simultaneously as this allows deduction that X=6 and X=4 (among other contradictions). This constitutes a *conflict set*. Any superset of this will also be a conflict and so the root set is often referred to as a *minimum conflict* set. Diagnosis with the ATMS is an incremental process and the conflict sets produced after various observations are used to produce candidates. This is known as *candidate generation*:

"... any previous minimal candidate which does not explain the new conflict is replaced by one or more superset candidates which are minimal based on this new information. This is accomplished by replacing the old minimal candidate with a set of new tentative minimal candidates each of which contains the old candidate plus one assumption from the new conflict." [DeKleer and Williams 1987 p104-105.]

Before measurement that F=10 there were no known candidates and the minimal candidate was {}. Measurement produced the conflict set {A1 M1 M2} and so according to the candidate generation mechanism

this leads to the minimal candidates {A1}, {M2} and {M3}. The candidate space is shown in Figure 2.4.

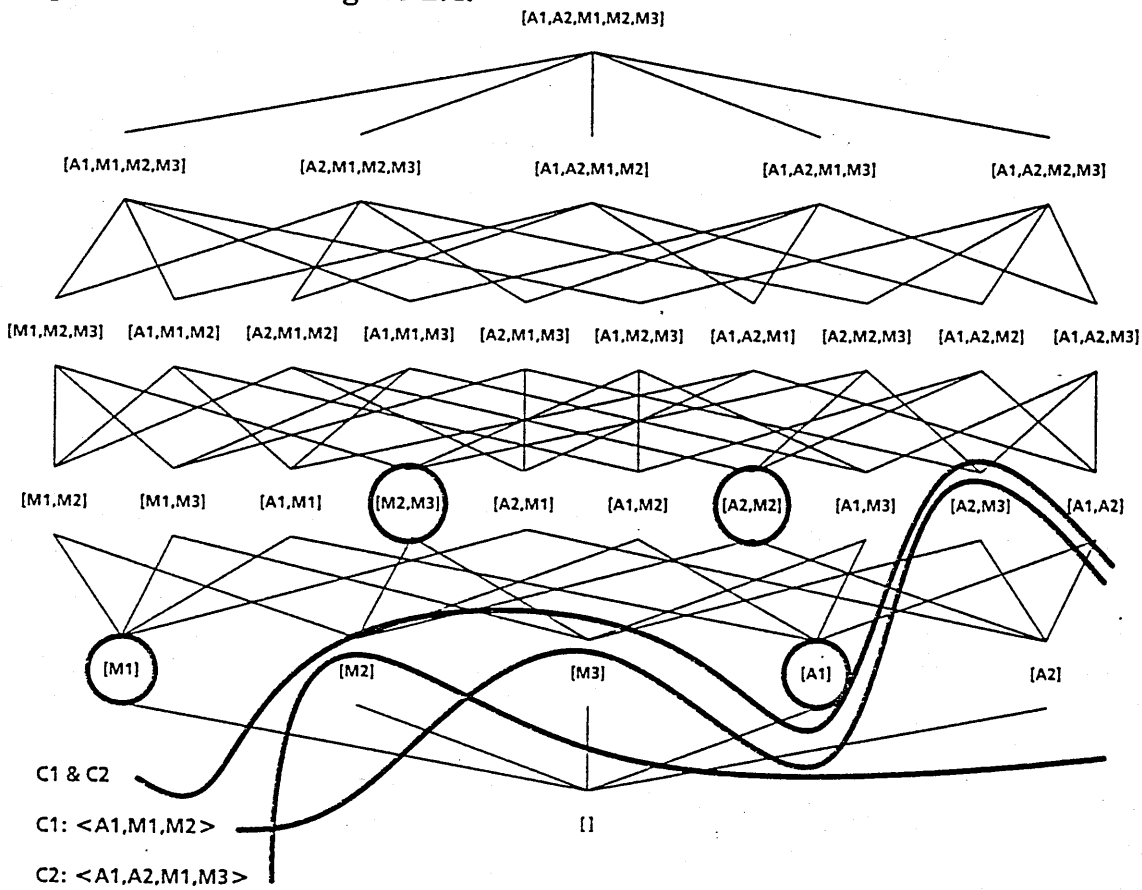


Figure 2.4. The candidate space after measurements. [DeKleer and Williams 1987 p105.]

G is then measured and 12 is observed. The circuit is shown again in Figure 2.5 after this value is asserted and constraint propagation has taken place.

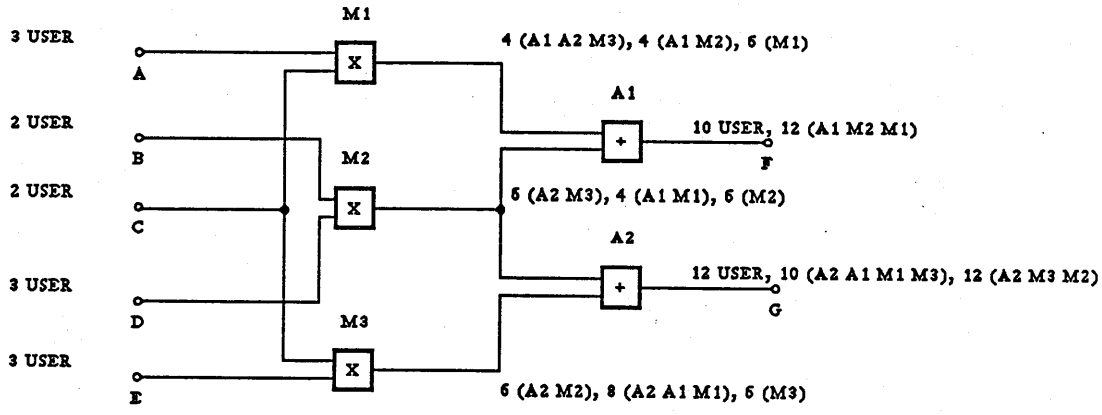


Figure 2.5. The circuit after G=12 is observed.

This produces the new minimum conflict set {A1 A2 M1 M3}. Candidate generation produces the new candidates {A1}, {M1}, {A2 M3} and {M2 M3}. Thus we have the situation in which the ATMS is predicting that multiple faults can occur; after all from the structure of the circuit if M2 is outputting an erroneous value then something else must be wrong to compensate for the difference between the values measured at F and G.

Diagnosis continues by making successive measurements and inserting them into the model until the candidates are reduced to one set. DeKleer and Williams have developed a 'minimum entropy' method for determining the best measurement to make next. However this method assumes that each measurement incurs equal cost which is not practicable in many situations where components are sometimes difficult to access or where special test equipment has to be used.

DeKleer and Williams claim that the candidate generation mechanism eliminates the need for faults models because

> "... if a component's behaviour is inconsistent with its model, then it must be faulty. This results in a domain independent diagnostic technique."
> [DeKleer and Williams page 112.]

Fault models are useful because they predict the presence of symptoms other than the ones observed and can be used to find

problems. Not all devices can simply be replaced. Take the case in which an electronic gate is actually functioning properly, but that candidate generation has pinpointed the gate as faulty because there is a short circuit where the chip has been soldered to the printed circuit board. Replacement wouldn't help. DeKleer and Williams would argue then that the wires would have to be modelled too. Ultimately every aspect of the physics of wires, and the structure of the circuit board would need a representation. No, diagnosis is no good unless a reason can be given for replacing a part. By having the reason we know how effective the action we take can be, and the problem the action is supposed to address, then if nothing happens at least the reason for the action can be examined to see if there was some way in which the action could have missed a probable cause of the problem. The ATMS is no doubt useful for narrowing down the search but it is clear that other techniques have to be used where tests cannot easily be made and where the device varies its behaviour over time.

It is interesting to note that the constraints do not imply any form of causality about the model. The act of back propagating from an erroneous observed value is permitted because the output value could only be achieved if the input was that predicted by the constraint propagation. Some devices are multi-directional, ie they can transmit signals either way through them so eg. an adder could work as a subtractor. This point illustrates an interesting deficiency among modelling with constraints, because often when a device breaks its output is merely a dead value, such that it is not possible to back propagate to say what the inputs are. Ie. it is multi-valued in reverse. Back propagation is then impossible unless you allow inputs to take values from a range.

2.1.2. Constraint Suspension

Randy Davis [1984] has built a system that reasons diagnostically from the structure and behaviour of a device. He sees that diagnosis can

only be done if there is knowledge of the causal pathways that make up the device. The most obvious causal pathways are the functional connections between modules of a device such as wires in an electronic circuit. However physical adjacency of components is an important consideration because components can interact thermodynamically, electromagnetically etc. in ways that haven't been anticipated by the designer. The structural organization of a device therefore has important implications for the ways in which the device can fail. There are other causal pathways through which modules can interact (eg. water spraying at high pressure onto other components) but representing them all is an impossible task. This raises the question of complexity versus completeness because some diagnoses won't be possible unless all causal pathways are known.

To solve this problem Davis categorizes the types of failure that can occur. Initially only faults in the functional description are considered. These include adders giving a wrong result and loose wires on inputs to components. If these types of fault don't produce a diagnosis then the next category of failures is considered. This second category allows for small changes in the structure of the device, such as accidental links between adjacent wires, commonly known as *bridge faults*. Again if no diagnosis is possible further categories are taken into account. The categories are taken in order from the following list:

- localized failure of function
- bridge faults
- unexpected direction
- multiple point of failure
- intermittent error
- assembly error
- design error

The order of the categories can be gleaned from an expert, and will vary between different circuits and devices. The contents of the list will vary between different types of devices.

The consequences of these deliberations are manifested in the choice of representation. A functional and structural hierarchy is chosen. Functionally devices are represented by interconnected constraints or black boxes. The representation is much like the series of commands one would issue in order to construct the device.

The constraints are used to simulate the values of inputs and outputs in a circuit. Davis is careful to distinguish between simulation and rules which allow us to model the inferences we make about a device. That is if there is an adder in a circuit the simulation rule predicts the value of the output from the inputs, but given an output and an input we can infer the other input by subtraction. The constraint network used has two independent components, one for each type of inference. The networks also keep track of dependencies, each slot keeping track of the rule that mentions that particular node. This helps in doing hypothetical reasoning and is fundamental to the discrepancy detection approach to troubleshooting.

The traditional approach to troubleshooting digital circuitry is *path-sensitization*. A suspected fault is forced to reveal itself by choosing a set of input values such that the output depends only on the suspected unit's functionality. If the output isn't what was anticipated, then the suspected unit is faulty (probably). It may require a few sets of inputs and output values to narrow the set of possible candidates down to the suspected unit before one is sure about the condition of the suspected unit. The algorithm that generates the tests is called the D-algorithm [Roth 1966].

Diagnosis though is not simply a matter of test generation - that activity is reserved for verification of the way devices work. Diagnosis means the generation of a set of candidate parts from some observed

symptoms. In response to these problems Davis's method uses Discrepancy Detection and Candidate Generation.

Any difference between measured values and values inferred by the simulation is a *discrepancy*. The dependency information carried by the constraints enables a trace back along the causal pathways to produce a list of candidates that can cause the discrepancy.

Candidate generation happens in three stages: i) First simulate the circuit and collect the discrepancies, ii) use dependency information to collect possible candidates and iii) suspend the constraint of each candidate in turn until a globally consistent state is achieved.

When the constraint of a candidate is suspended, its simulation rules are removed from the model. If the device was faulty the contradiction caused by the conflict between the simulation rules and the observed values will be removed yielding a consistent state. So if removing the simulation rules of a candidate produces a consistent state it is likely that the candidate was faulty.

Suspending the constraints of a component is tantamount to saying you have no idea what it is supposed to be doing, it could have failed anyhow and so the method can detect any type of device failure. But, what if the fault is not a break in a component, but a change in the structure of the device, as Davis puts it:

> "Put simply, the virtue of the technique (candidate generation) is that it reasons from the schematic; the serious flaw in the technique is that it reasons from the schematic *and the schematic might be wrong*."
>
> [Davis 1984 p.369]

Of course the reason for this is that the behavioural description allows only for one type of interaction pathway. There are many others, indeed problems never previously encountered often display previously unimagined causal pathways. Accounting for all causal pathways is an infinite task, yet it is the only way that all faults can be explained.

Humans cope extremely well with new problems and Davis takes heed from this.

The attempt to get round the problem involves sacrificing the assumptions that have been made. If an attempt at trouble shooting doesn't work with the behavioural description then the assumptions are systematically dropped. The first one to go may be the assumption about the schematic and the new model allows for bridge faults. If a contradiction is still obtained then other assumptions have to be dropped, such as the direction of signals through wires etc. The assumptions represent *categories of failure* and are linked to the engineer's knowledge of what kind of things can go wrong. The order in which the categories are embraced is dependent on the engineers notion of what is more likely to go wrong at any point.

To demonstrate the diagnostic strategy Davis chooses a bridge fault. A bridge fault is an unanticipated connection between two wires on a circuit board, that happens due to bad solder joins or bent pins on some chips. Consider again the multipliers and adders circuit shown in Figure 2.2. When built the circuit board might appear as shown at left in Figure 2.6.

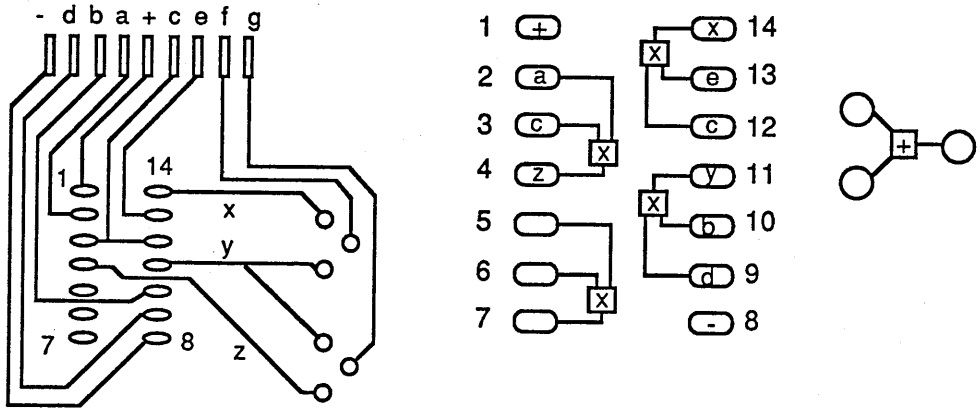


Figure 2.6. The circuit of Figure 2.2 as it may appear when constructed. The multipliers are all contained in an integrated circuit whose pin layout is reproduced on the right. The pin layout for each adder is shown on the right..

In this case the outputs at f and g are both giving the value 8 whereas a 12 is expected. Tracing back along the functional pathways produces

the fault candidates: {A1 A2},{M1 M3},{M2},{M3 A1},{M1 A2} (see section 2.1.1 to find out how these sets are generated). Suppose we know that M2 cannot be the culprit and we are reluctant to believe that more than one device can fail. How, then, can the results be explained? The fault could be due to a bridge between wires C and Y (at left of Figure 2.6), the lower value being dominant, and so Y would be 2 explaining the results.

Here the problem of a fault predicting simultaneous failures far apart in the functional representation is explained by a simple bridge between two wires adjacent in the structural representation.

Davis's own example represents more than one level of the structural and functional hierarchies and it turns out that the fault he describes can only be explained by looking for bridge faults in the second level description.

The work of the Davis team is thorough when applied to Digital diagnosis but I am sceptical if this is the way that experts diagnose that type of circuit. To generate the candidates various sets of inputs have to be chosen, much like the path-sensitization method. At present this method is done by hand and even if it could be made into an algorithm I doubt if it would have an application outside digital electronics. This is because the method requires a wide variation of allowed input values whereas in most other devices the input is fixed, - it is not useful to try running a car on water to test if the fuel line is blocked!

Knowing when to allow for a new category of failure is unclear. In the example I gave allowing multiple faults could explain the behaviour, but persistence with a single fault assumption and taking into account bridge faults also generates a feasible explanation. The diagnostic reasoning described by Davis does not take into account the consequences of the measurements. It forms no theories that suggest tests to be made, and any test results that are observed aren't used to change the current hypothesis, nor to generate a new category of failure.

2.1.3. Qualitative Diagnosis

Hunt [1989] and Price [1989] have built a computer system that diagnoses mechanical devices. They have developed an object oriented representation that models the components of a device and can simulate its behaviour. The representation lists the components of the devices and describes their initial positions. The behaviour of the components when subjected to a force are defined by simple rule-like relations. Their representation uses a qualitative model to describe the positions of buttons, levers etc. and the strength of forces. Simulation of the device is defined as propagation of an event through the device. Because of the qualitative nature of their representation there can be more than one successor to a state in accord with similar qualitative theories.

The basis of their diagnostic approach is a system of *assumptions*. The assumptions are classified as structural, functional and device-related. Structural assumptions refer to the components of the device, and provide such questions as: Are all the parts claimed to be present really there? Functional assumptions question whether a part is really doing what it is supposed to do, and has not changed its behaviour. Finally device assumptions are assumptions on the whole device rather than assumptions associated with components or connections within the device. A normal working device relies on all of the assumptions being true. Any of these assumptions can be retracted and the resulting changed device can still be simulated as a 'broken' device.

Diagnosis is in four stages. Firstly the diagnostician *Acquires the actual behaviour* of the device. This means the program (which does not appear to be named) obtains from the user what the device is supposed to do. In practice this is a list of the successive states of the device.

Secondly the program *Generates the causal chain*, this is a list of the assumptions associated with every component or connection which had some part to play in producing the actual behaviour. These are identified from the intended behaviour by comparing the actual and intended

behaviours until they diverge. The assumptions on the causal chain are added to a primary assumption list. The remaining assumptions are put on a secondary assumption list.

The third step of the diagnostician is *Candidate fault generation.* This step processes each assumption in the primary and secondary assumption lists in turn. This is accomplished by instantiating a model of the device with the assumption retracted and generating a state map of the behaviour of the possible fault model using the qualitative simulator. A *behaviour matcher* is used to compare the predicted and actual behaviour.

Finally the diagnostician *Lists all the candidate faults.* All behaviours that matched the observed or actual behaviours are displayed to the user.

One of the drawbacks of this approach is that it makes no attempt to study the described symptom. There is no mechanism in the representation for working back from a symptom description towards the cause of the fault. This in turns means that the qualitative diagnostician has difficulty in generating explanations for its search strategy.

The system restricts the numbers of assumptions that it considers could have caused the failure to only those assumptions related to the components that played a part in the intended behaviour. This attempt to narrow the search space will be very effective if the device breaks early on in its operation, eg after pressing the first lever. However sometimes a fault won't manifest in a device until the device has been running for a while, in which case nearly all the components of the device will have been called upon to perform their duty. This will leave an enormous amount of assumptions about the device each of which requires instantiating a model with the assumption retracted. This would be computationally expensive, and moreover is simply not the way a

human expert would tackle the problem. Some way of compartmentalizing the behaviour could help reduce this search space.

There doesn't appear to be any systematic way of generating the assumptions about the components of the device. Finally there is no mechanism for performing tests on the device, which would be another way of restricting the search space.

2.2. Qualitative Reasoning

The basic reason for the deficiencies of expert systems outlined in the introduction lies in the lack of a representation of the 'commonsense knowledge' about the real world that people have. People can appreciate that when a football is kicked it flies through the air, and how liquid pours from one container into another. They have to confront events like these every minute of the day, and so they have their own 'people physics'. An attempt to model some of this 'people physics' is called *qualitative reasoning* and takes the form of modelling parameters of a system such as velocity, volume, temperature with qualitative values. For instance the temperature of a substance could be modelled by regions on a number line:-

So the temperature of a substance could have the values, 0, (0,MP), MP, (MP,BP), BP, (BP, Infinity), infinity, where (x,y) means a value somewhere between x and y. These are the qualitatively distinct values of the parameter which identify the solid, liquid and gas phases of the substance. The entire range of values a parameter can take is called the *quantity space* of the parameter. To illustrate how qualitative values are combined to be useful in equations I show below how the simplest quantity space, {-,0,+} behaves under operations of addition and multiplication:-

+	-	0	+
-	-	-	?
0	-	0	+
+	?	+	+

*	-	0	+
-	+	0	-
0	0	0	0
+	-	0	+

Figure 2.7. Tables showing operations of addition and multiplication over the simple quantity space {-,0,+}. A ? indicates an ambiguous value which could be -,0 or +.

In these tables question marks indicate the ambiguity that arises from the addition of a positive, and a negative quantity. For instance this might occur when modelling a tug-of-war game. One team pulls with a force '+' and the other with a force '-'. Without extra information such as the strength or weight of the teams it is difficult to predict the winner. What is common in these instances is to carry on with all possible outcomes and offer these as solutions to the problems. Later other evidence may dismiss some of the proposed solutions.

Using this basic mathematics three principal approaches to modelling qualitative reasoning in physical systems have been defined. DeKleer and Brown use a confluence based approach, Forbus describes his Qualitative Process theory and Kuipers has devised the QSIM algorithm.

2.2.1. Confluence Based QR

The aim of DeKleer and Brown's [1984] work is to 'derive function from structure' for an arbitrary device. They adopt a reductionist approach in which the behaviour of a device is assembled from the known behaviour of subparts of that device. The example they use is a pressure regulator as shown in Figure 2.8. This device can be viewed as a pressure sensor and a valve working together (Figure 2.9).

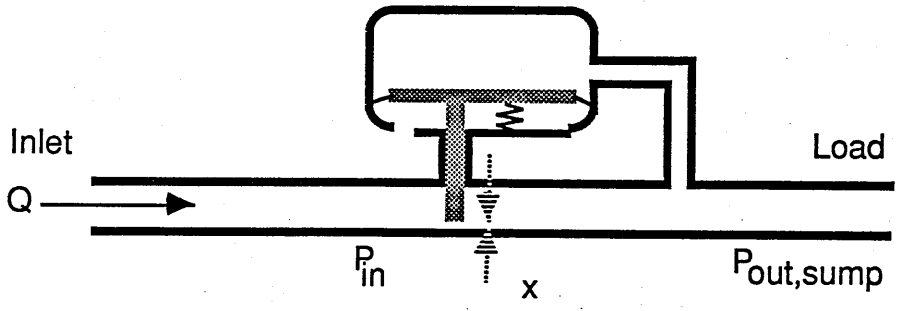


Figure 2.8. The pressure regulator as discussed in DeKleer and Browns paper. Q represents fluid flow, P is pressure and x is the width of the aperture open to flow.

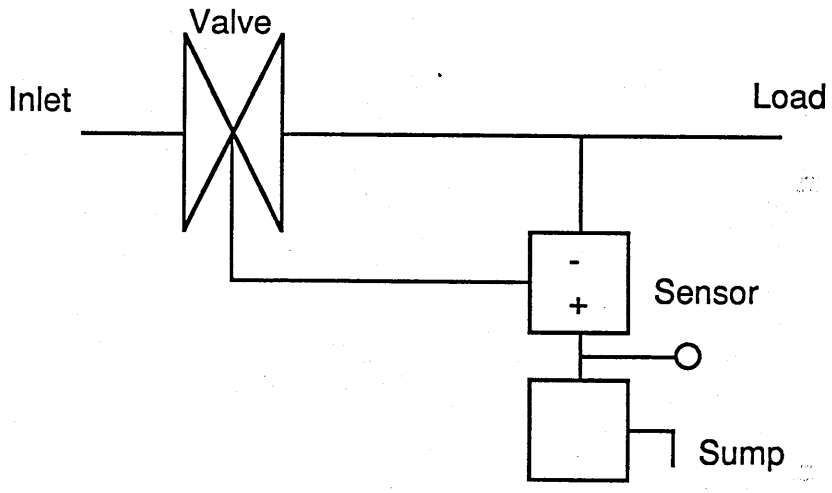


Figure 2.9 A model of the pressure regulator showing the subparts and their connections explicitly.

The behaviour of these subparts is defined by confluences. A confluence is a qualitative differential equation which constrains certain of the changing parameters of the device. For instance the confluence for the pressure sensor is:

$$\partial X + \partial P = 0.$$

This places a constraint between the pressure, P, and the position of the valve piston, X. If the pressure increases (ie. $\partial P = +$) then the position of the valve piston must close ($\partial X = -$) in order to keep the constraint satisfied, and this is what happens in a working valve.

Similarly the confluence for the valve is written:

$$\partial P - \partial Q + \partial X = 0,$$

where Q is the flow through the valve.

In their paper DeKleer and Brown suggest that it is possible to combine these confluences for the subparts to produce a confluence defining the behaviour of the whole device. To do this it is necessary to take account of the properties of the materials which link or interface the subparts. Two conditions are important for liquids: the *continuity condition* and the *compatibility condition*. The former is a version of Kirchhoff's law applicable to liquids and says that any liquid entering a conduit must be matched by an equal amount of liquid flowing out of the other end. This can be thought of as conservation of matter. The compatibility condition ensures that the pressure difference between two points in a network of pipes must be the same irrespective of the path between them. If it weren't, a flow would occur thereby evening out the difference.

Important too, is knowledge of the environment in which the device is operating. In this case the device is connected to a pressurized load, but the pressure regulator would work very differently if connected to a vacuum.

In their paper they don't show how to derive overall behaviour from the subparts, and in fact they admit most confluences come from an adaptation of a conventional physical model or from a formalization of one's commonsense understanding of the device. The model for the pressure regulator has three states:

Open:	[A=Amax],	[P]=0,	∂P=0,
Working:	[0<A<Amax],	[P]=[Q],	∂P+∂A-∂Q=0,
Closed:	[A]=0,	[Q]=0,	∂Q=0.

Figure 2.10. In this figure, A represents the area available to flow, P the pressure at the inlet side to the regulator, and Q the rate of flow of the liquid.

Parameter values determine which state is applicable at the time, and should the values change, (constrained by the applicable confluence), the device may change state and a new confluence will determine behaviour. Thus there are two phases of modelling the device with confluences: intrastate and interstate behaviour. Modelling intrastate behaviour involves seeing how all parameters change should one of them change. Modelling interstate behaviour determines the new state of the device if one of the parameters steps outside the range defined for that state. For instance when heating a container of liquid, intrastate behaviour can determine that the temperature of the liquid will rise (this may be called the heating state). When the liquid reaches boiling point the liquid will no longer heat up and the container will begin to boil dry. A new confluence appropriate to the boiling state would be needed to model this new state of affairs.

A state diagram represents the possible transitions which a device can go through during operation. The state diagram for the pressure regulator in which the mass of the piston has also been modelled is shown below (Figure 2.11). Modelling the mass of the piston introduces a delay in the feedback of the pressure sensor and so oscillations may ensue. This is seen as a circular route in the state diagram. Modelling friction would show how these oscillations can decay, with an eventual transition to a quiescent state.

DeKleer and Brown call their program ENVISION and the prediction of the various states of the system is called an *envisionment*.

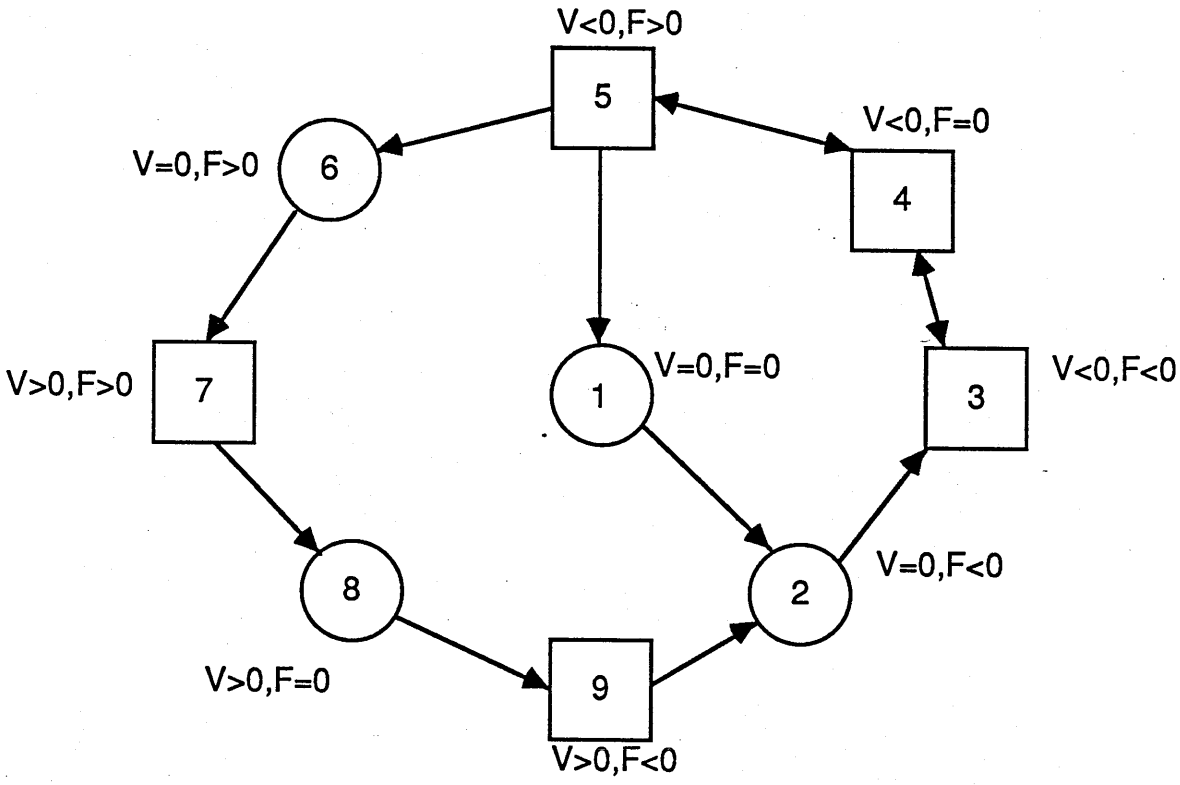


Figure 2.11. State diagram for the confluence model of the pressure regulator in which the behaviour of the spring and the mass of the moving piston have been accounted for. Circles represent momentary states and squares are states that last for a period. F represents the force on the piston by the spring, and v the velocity of the moving piston. The situation in which the piston is closing whilst still under influence of the spring is captured in state 3. Some of the states form a ring and this means that the piston could oscillate.

2.2.2. Qualitative Process Theory

Forbus [1984] centres his approach to supplying commonsense reasoning on the notion of *processes*. Processes are activities such as boiling, flying and collision which change the values of parameters of the objects involved. Here too parameters are described qualitatively and a process might stop when a parameter moves to the next value in its quantity space.

A ball thrown into the air would have a velocity parameter with quantity space `{down zero up}`. When the ball is thrown up the initial process is `climbing` in which `v` has the value `up`, but is decreasing

eventually becoming `zero`. The climbing process becomes inactive and the instantaneous process `turning` takes over to describe the motion of the ball at the peak of its flight. This process too becomes inactive as the ball begins to drop. A process called `falling` applies in which the velocity is `down` and increasing in magnitude. The sequence of processes, `climbing`, `turning`, `falling` is called a *history*.

Objects are represented in the QPT by individual-views which describe the situation of an object. A contained liquid is represented thus:

```
Individual View: Contained-Liquid(p)
   Individuals:
         con a container
         sub a liquid

   Preconditions:
         Can-Contain-Substance(con,sub)

   Quantity Conditions:
         A[in(sub,con)] > ZERO

   Relations:
         There is p, a piece-of-stuff
         amount-of(p) = amount-of-in(sub,con)
         made-of(p) = sub
         container(p) = con
```

Figure 2.12. An individual view of a contained liquid, such as tea in a mug. A[X] means 'amount of X'.

Each individual view (IV) has four elements. The individuals are a collection of objects involved in the model. Preconditions specify relations which must be satisfied for the IV to be relevant. Quantity conditions specify values the individuals must have in order for this view to be appropriate, and relations define various parameter aspects. It is important to note the difference between quantity conditions and preconditions. QPT can only determine the fate of the parameters specified in the quantity conditions. The preconditions are those that describe the structure of the situation and cannot be handled by QPT - for instance if someone breaks the mug then can-contain-substance(con)

would be violated and a different IV would be needed (if there was one). The QPT algorithm can only identify when processes start and end due to one of the parameters in the quantity conditions moving outside the specified range.

An important relation in QPT is *qualitative proportionality*. Denoted α_q this relation links two parameters such as the level and volume of liquid in a container. The relation merely ties the two quantities together - it doesn't say how they vary with respect to each other, just that if one changes then so must the other. Two refinements are α_{q+} and α_{q-}, which describe monotonically increasing and decreasing effects respectively. Such relations represent causality in the process - the observed effects of one variable changing due to another. Such relations are merely formalizations of observed effects.

To represent how things change Forbus describes processes which act on the individuals. Like individual views, processes have preconditions, quantity conditions and relations, but in addition each one has a list of influences - what objects in the process are doing to some of the parameters. Figure 2.13 is an example of the heat flow process suited to a description of heating a saucepan on a stove.

```
Process: Heat Flow
    Individuals:
        src an object, Has-Quantity(src, heat)          ;Source.
        dst an object, Has-Quantity(dst, heat)          ;Destination.
        path a Heat-Path, Heat-Connection(path, src, dst)

    Preconditions:
        Heat-Aligned(path)

    Quantity Conditions:
        A[temperature(src)] > A[temperature(dst)]

    Relations:
        Let flow-rate be a quantity
        A[flow-rate] > ZERO   ;Temperature at source
                              ;greater than at destination.
        flow-rate α_q+ (temperature(src) - temperature(dst))

    Influences:
        I-(heat(src), A[flow-rate])
        I+(heat(dst), A[flow-rate])
```

Figure 2.13. A description of the Heat-flow process. A[x] refers to the amount of x, and I+(x,y), I-(x,y) are influences which tend to increase or decrease parameter x with increasing y, and vice-versa.

Change in QPT arises out of direct or indirect influences. Direct influences are only specified in processes such as the tendency to increase the amount of heat at the destination in Figure 2.13. Indirect influences come from parameters related by qualitative proportionality to directly influenced quantities. For instance if I blow down a horn I can only directly influence the speed of the air - the volume increase results from the greater vibrating reed amplitude which is due to the faster air flow.

Because parameters can change either as a result of a direct or indirect influence they can move to a qualitatively distinct state (ie. move to a different value in the quantity space) in which case the current individual view is inappropriate or the current process instance doesn't accurately reflect the state of affairs. When this happens the individual views and process instances become inactive and new ones apply. This is the case, for instance, when a liquid stops increasing in temperature and begins to boil. Initially it is modelled by a process in which one of the

influences describes the temperature of the liquid as rising, and when it boils another process describes the influence as reducing the amount of liquid due to evaporation, but not increasing in temperature. Eventually the individual view of the contained liquid would be inappropriate because the container would boil dry. Every such change in QPT occurs as a result of a process, and new processes needed to describe a changed situation are drawn from a process vocabulary. This process vocabulary needs to be fully specified if complete modelling of a situation is to be performed. QPT can't generate its own process description to describe unanticipated states.

The way parameters change in QPT is via their Ds values. This is the sign of the representation of the qualitative derivative of the parameter. A value of 1, 0 or -1 means that the value is increasing, unchanging or decreasing respectively. Limit analysis is the technique used to find what happens when one of the parameters moves to a different value in the quantity space. It involves extrapolation of the behaviour of parameters by examining their Ds values. A parameter whose value is increasing will eventually move to the next point in its quantity space, and when this happens the current process instance may become inactive, and new ones apply. This is the source of ambiguity in QPT which can occur in 3 ways:

(i) Parameters can have more than one neighbour,
(ii) more than one process can become active, and
(iii) more than one parameter can change at any one time.

The active processes at any time make up the process structure, and process histories describe what is happening when. Processes interact via shared parameters and the set of processes which can affect a situation are called p-components.

Forbus would like QPT to become a 'language for behaviour' in which processes are the primitives and p-components and encapsulated histories are compound forms. P-components are two or more processes

that can interact because they all have an influence on the same parameter or parameters, as though all the processes in the p-component were behaving as one new process. Encapsulated histories summarize a behaviour over a period, in which there may have been many processes and individuals involved. An encapsulated history is thus knowledge of a sequence of behaviour which can be used again if the appropriate preconditions for it are met or the situation which led to its being exists again.

Encapsulated histories are used by humans to understand events like bouncing. When we picture a bounce we imagine objects moving towards each other, colliding, and finally moving away. As our experience of bounces grows we learn to distinguish between different types of bounce such as elastic, inelastic, absorption or collision. If we need to explain what happened in a bounce we can unwrap the encapsulated history for 'bounce' to see how the outcome depends on the individuals and preconditions involved. See [diSessa 1983 page 23] in which she discusses how people learn these *phenomenological primitives* or p-prims and how they are used to explain events.

Forbus's encapsulated histories are too specific to allow the flexible analyses just described but represent an important step in this direction. I think that similar techniques would be useful in describing commonly encountered units or mechanisms in devices. Eg. an electrical circuit requires a source of voltage, and a circuit loop, but does various things according to what lies on that loop.

2.2.3. Qualitative Simulation

Kuipers [1984] makes the same underlying algorithm, present in both ENVISION and QPT, more explicit in his qualitative simulation (QSIM). The structural description of a physical situation is defined by placing qualitative constraints between variables in the model to produce a qualitative differential equation (QDE). The qualitative constraints

include `add`, a qualitative version of addition which ties three variables together:

```
(add a b c)        ; c is the qualitative sum of a and b.
```

For example working with the simple quantity space {-,0,+} and assuming a and b are known then c can be calculated from this table:

a	-	-	-	0	0	0	+	+	+
b	-	0	+	-	0	+	-	0	+
c	-	-	?	-	0	+	?	+	+

Figure 2.14. Qualitative addition for simple quantity space. A ? indicates the value is ambiguous and could be -, 0 or +.

In the table a `?` indicates that the value could be –, `0` or + and in QSIM all three values are used and so the reasoning splits three ways. Similar tables exist for the cases in which `a` and `c`, and `b` and `c` are known allowing deduction of the third variable. In QSIM variable values are usually represented as a pair like `(+ inc)` if the variable is "plus and increasing". The second term refers to the way the variable is changing over time and can also have the values `std` (steady), `dec` (decreasing) and `nil` (direction unknown). Qualitative constraints which use the measure of change variable include `M+`, `M-` and `d/dt`. `M+` and `M-` place a monotonically increasing or decreasing constraint between two variables and `d/dt` links one variable to the rate of change of the other, eg. speed to distance.

In the QDE a minimum number of variables need be specified so that the constraints can be used to fill out the other values, this is called completing the state. Also specified in the QDE are the ranges of certain variables for which the QDE is applicable. Simulation is performed by seeing which variables are increasing or decreasing towards their next (landmark) value in the quantity space (also specified in the QDE). When

a variable does move to the next value in the quantity space then other variables may be affected due to the constraints within the QDE, or a transition may occur and a new QDE apply. If two or more variables are both heading towards a landmark value then there will be ambiguity over the next state as it will be impossible to tell which will reach its landmark first, and so reasoning has to be done with the three resulting cases - one case where a variable reaches its landmark value before the other, vice versa and the case in which both landmarks are reached simultaneously.

Kuipers algorithm, now at a sophisticated level of development, can automatically generate its own landmark values for parameters when the rate of change of a variable becomes zero. This has the advantage of facilitating testing for oscillations in a system - one of the ways in which QSIM can terminate.

Simulation is carried out until no more transitions can be found, a quiescent state is reached or a cyclic oscillation identified.

2.2.4. Helix - An example of the use of QSIM in diagnostics.

HELIX [Hamilton 1988] is a computer program that reasons about helicopter engines qualitatively. Modern helicopters require more technology because of the move to single-pilot operation, hence an 'Automated cockpit' that can perform some of the functions of a co-pilot is needed. Its function is to monitor the state of the helicopter's 2 engines from data about coolant pressure, engine torque, fuel consumption etc.

Hamilton complains that traditional algorithmic approaches are too complex, and can't handle unanticipated situations. He has chosen an AI approach to do robust status monitoring and diagnostics. Helix reasons from a qualitative model of the power-train of a helicopter, and avoids fault models. Helix merges the ideas of constraint suspension and Qualitative Reasoning.

The first step towards doing diagnostics with HELIX involves assembling a structural model of a device on a graphical interface. The

building blocks are drawn from a library of component models which include generic representations of some of the more common elements of devices, such as control levers, gearboxes, pumps etc. The connections between elements define confluences between variables. The library contains primitive and compound models which are sets of variables and confluences. Variables are 5-tuples of the form {material, attribute, direction, component, derivative}. Confluences define qualitative equations between the variables of the device. The resulting model can be made into a primitive component and used as part of new models. In this way a hierarchical decomposition of the qualitative functionality of the device is built.

A 'Model Instantiator' simplifies the confluences relating the parameters of the system, removing confluences which have no terminals or refer to two or less variables. Resulting models are heavily simplified and optimised enabling coarse-grained reasoning to be done. When a fault is detected down a branch of the model hierarchy the constraints are expanded in an attempt to isolate the fault still further, possibly going deeper inside the hierarchy.

When a fault occurs in a system modelled by HELIX all data about the fault is presented. The fault will manifest itself as a discrepancy among the highest level constraints (confluences). HELIX then systematically suspends the constraints of a second level model. Most of the suspensions still leave an inconsistent state. Those suspensions which leave a consistent state imply that the suspended unit is faulty, and that becomes the hypothesis. Next, a detailed model of the suspected unit is created along with high-level views of the devices with which it interacts. Constraint suspension is repeated at this level. This process can produce more than one suspected device, but this analysis combined with observations of the device over time will eliminate some of the candidates. HELIX displays the diagnosis and offers advice, such as

shutting down an engine or increasing throttle. Advice is assembled by production rules taking the diagnosis and flight conditions as data.

Hamilton illustrates Helix working on the power-train of a helicopter. Two engines supply the required torque to turn the helicopter blades. Each engine has fuel pumped to it by a compressor. The speed of the blades is monitored by a feedback system. More fuel is supplied if the speed drops, and vice versa. He doesn't write down the model explicitly so I produce a QSIM representation of the model below.

For each engine:

$$\text{Torque } \alpha_{q+} \text{ Fuel-flow} \qquad (1)$$

$$\omega \; \alpha_{q-} \; T_{load} \qquad (2)$$

$$\text{error} = \omega - \omega_{set} \qquad (3)$$

$$\partial/\partial t \text{ Fuel-flow} = -\text{error} \qquad (4)$$

ω is the angular velocity of the engine (revs), T_{load} is the torque the engine is trying to generate, ω_{set} is the desired angular velocity and $\partial/\partial t$ Fuel-flow represents whether more or less fuel is supplied to the engine.

The torque at the blades is the sum of the torques from each engine:

$$T_{blades} = T_{eng1} + T_{eng2} \qquad (5)$$

When the fuel pump in engine 1 fails the torque produced falls, burdening engine 2 with its load (5). The revs, fuel-flow and torque of engine 1 thus drop (1). The constraints for engine 1 say that the fuel flow should increase as the revs drop, (3) and (4), and a discrepancy from normal behaviour is detected. The discrepancy makes it impossible to complete the constraints as they are.

Helix now takes a second level model consisting of engine 1, engine 2, transmission and flight controls. Constraints representing behaviour are suspended in turn in each of these units. Only when the constraints

in engine 1 are removed can a consistent state be achieved. Therefore the failure of engine 1 is the hypothesis.

Finally a model view of another cross-section of components is performed, which isolates the suspected devices to 5 elements of the fuel system. Further data analysed four seconds later rejects some of these elements.

The workings of Helix are thus mainly a hierarchical modelling of the structure of a device and sets of constrains defining the normal behaviour. The diagnostic approach appears very naive. Simply suspending constraints one-by-one is random guess-work, and by no means can all faults be detected in this way. The method relies on constraints being broken. Sometimes however when a device breaks it can affect other parts of the device not necessarily associated with the same branch of the functional hierarchy. Simple examples of this include the occasions when pieces of metal break loose and fall onto a circuit board bridging a contact.

To be effective this method also relies on having lots of data available. Often in qualitative descriptions of devices parameters are related to each other via chains of constraints. If A α_{q+} B and B α_{q+} C then suspending either constraint will have the same effect on C, and so making it impossible to distinguish between a fault in A or B, if data for B is not available.

It doesn't make sense to suspend some constraints which exist to describe the physics of a situation. Examples include constraints between pressure and volume or the relation of heat-flow to temperature difference. These types of constraints are there to define the physics of the world and cannot 'go wrong'.

2.2.5. Summary of Qualitative Techniques

Many similarities can be drawn among the three Qualitative Reasoning algorithms, as summarised in Figure 2.15 below. However the differences are important. DeKleer and Brown try to build their

confluences from known behaviour of the subparts, Forbus is interested in the process going on and the individuals involved, whereas the Kuipers algorithm is used to forecast events.

Aspect	DeKleer and Brown	Forbus	Kuipers
Theory	Qualitative Physics based on Confluences	Qualitative Process Theory	Qualitative Simulation
Qualitative value	P = + ∂P = -	A[temperature] D[pressure]	(+ inc)
Qualitative Equation	Confluence eg. ∂P+∂X = 0	Influences in process description I+(level, flow)	Constraints (m+ level volume)
Sources of ambiguity	Constraints not uniquely resolvable	More than one successor in the quantity space, more than one process active. Competing variables.	Variables compete to reach landmark values. Constraints not uniquely resolvable.
Capturing Events	Envisionment	Process history	Qualitative Plots

Figure 2.15. Comparison of various aspects of three Qualitative Reasoning approaches.

2.3. Knowledge based Diagnostic Strategies

2.3.1. CRIB

Keravnou and Johnson [1986] have devised an approach to diagnosis based on the competence of an expert. Their important emphasis is that the difference between novice and expert diagnosticians is that although the novices have knowledge about the 'patients' or devices being fixed, this knowledge is poorly structured. Experience is the process of structuring, and building links to other parts of this knowledge [Feltovich et al 1984]. They dismiss most existing approaches to diagnosis as purely algorithmic in nature and hence not having the

competence associated with human experts. They also believe it is important to explicitly model the diagnostic strategy to produce better explanations.

Their exemplar is based on diagnosing a computer system down to the field replaceable parts, which are also the leaf nodes of the device hierarchy. The patient computer is described as a simple taxonomy of subunits. Associated with each machine unit are 3 groups or sets of symptoms. The T (total) group represents all the symptoms that can occur for the given subunit. The K (key) group are the symptoms whose presence is sufficient to establish a fault for that subunit. Finally the S (subgroup) is a subset of the K-group whose presence is necessary but not sufficient to establish a fault in the subunit.

Diagnosis is a cycle of "*test* the faulty system and *observe* its result; *analyse* the result and determine whether it is possible to: *split* the system into faulty and non-faulty components; *test* the faulty subsystem and repeat" - TOAST! [Keravnou and Johnson 1986 p.89-90]. Diagnosis begins from the initial hypothesis that the device is faulty (after an observation "indicating divergence from proper functioning") and involves successive refinements of this hypothesis down the device hierarchy until a field replaceable or repairable unit is encountered.

The process of hypothesis refinement is involved and is supposed to be the closest to a model of human competence. The observed symptoms (Os) of the faulty device will comprise an S-group of one or more of the subunits of the device taxonomy. If the Os constitute a K-group of some subunit then this is strong evidence that that subunit is faulty, making it the current hypothesis. However if the Os form only an S-group (a subset of the K-group) then this is less strong evidence for the hypothesis, and it becomes semi-activated. If there is no suggested refinement by this process, yet some hypotheses have been semi-activated then an attempt to acquire additional symptom information is made. A hypothesis

referring to a replaceable/repairable unit as being faulty can be directly refuted or concluded through the observation of *cure symptoms*.

So this process which begins at the highest level of the device taxonomy gradually works its way down, working with more and more specific symptoms until a replaceable/repairable unit is encountered. If a hypothesis refinement is dropped then diagnosis resumes from the immediately more general hypothesis, mimicking the reluctance of an engineer to completely abandon their current pursuit. This process of hypothesis refinement is summarized in Figure 2.16.

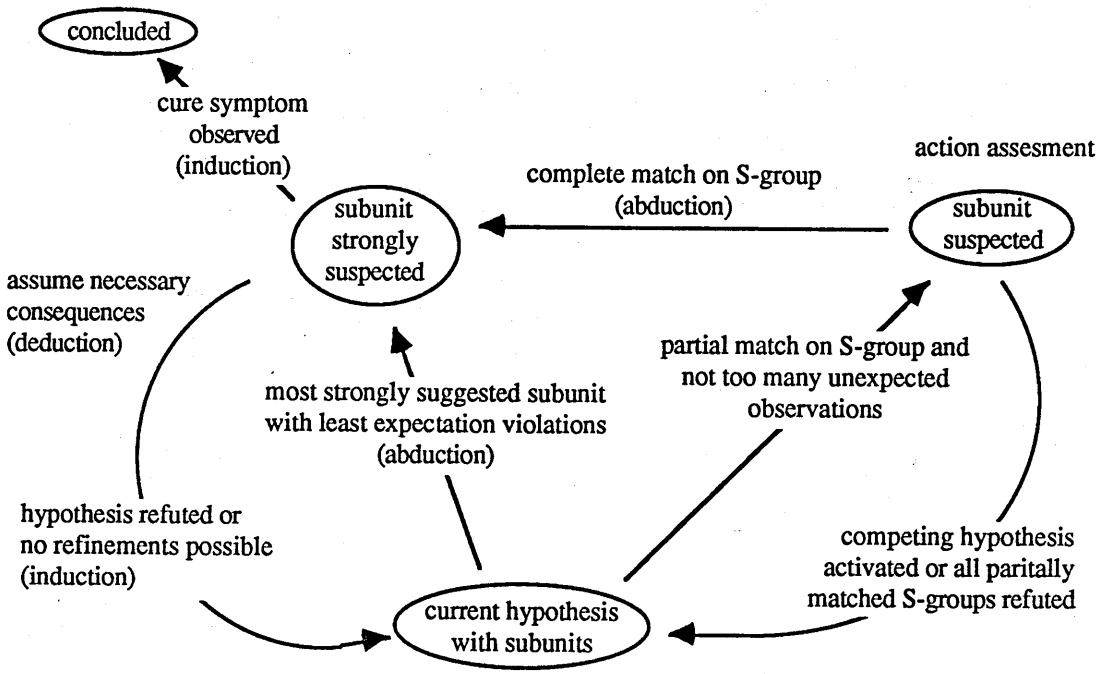


Figure 2.16. Hypothesis refinement in CRIB. [Adapted from Keravnou and Johnson 1986 page 94.]

Keravnou and Johnson complete their analysis of fault diagnosis by including in it a theory of what actions can be taken to cure the problem. Sometimes the actions are designed to repair the device, but often they are aimed at testing a hypothesis. The action chosen depends on its time to perform and its discriminatory power:

"Select the action that yields the most mentioned symptoms unless there is an S-group with one outstanding symptom only and the relevant action has a shorter total time of performance."

[Keravnou and Johnson 1986 page 100.]

In summary, the method is primarily a theory of competence. However it introduces some interesting ideas for diagnosis. Firstly the hierarchical representation is heavily used and serves to focus the diagnostic strategy. This is important in large devices where focussing on the relevant subsystems is essential for avoiding information overload. Rules are used to decide whether to 'unpack' another level of detail or search elsewhere in the device.

Hypotheses are entertained even when there is only a partial match on the symptoms involved. This is helpful in diagnosis because it draws to attention other things that may have gone wrong if the main hypothesis proves to be incorrect. These secondary hypotheses are put on hold and ranked by an ordering mechanism. This element provides for graceful degradation of the expert system.

Keravnou and Johnson also expand on an element of diagnosis previously overlooked or simplified by previous authors of diagnostic systems; that of the cost of a test. The cost of a test is built into their competence model and helps decide whether to pursue a hypothesis further or try a different one.

CRIB, however, remains a rule based system. Many rules are required to diagnose each fault that the device can have, and to distinguish between faults. Only known faults can be diagnosed.

The device hierarchy used in CRIB is a strict one. There is no room for allowing two subsystems to interact. This means that the choice of the representation of the device will be crucial. For instance Davis's work (section 2.1.2) showed how a fault easily diagnosed in one representation (structural hierarchy) requires more work in another (functional hierarchy).

2.3.2. Meta Level Reasoning

Genesereth and Smith [1981] demonstrate how production systems could perform much better if somehow they could reason about what strategies to use in solving a problem. This is called meta-level reasoning. Meta rules reason about the structure of the base-level or problem solving rules and choose which base-level rules to apply in solving a problem. For instance if a rule needs to find an MP whose telephone number is a prime number, the rule may contain premises which search through a list of MP's, telephone numbers and primes. Because the list of MP's is the shortest this is the better list to search through first. The base rules are the rules which do the looking up of facts. Meta-level rules arrange that the list of MP's telephone numbers are searched through first.

The nature of the base-level rules can be explicitly represented (ie. by saying `better(rule1, rule2).`), but it is better if meta-level rules can infer the appropriateness of a rule from the structure of the base level rules. This enables the meta-level reasoner to decide to use one rule before another, or even to re-order the premises in the base-level rules. This is like having knowledge about knowledge.

With respect to doing a diagnosis having some form of meta architecture would be very powerful. Often in diagnosis one reaches an impasse where a decision has to be made between taking something apart to examine its innards or making a few simple measurements. The measurements might not yield any fruitful information, whereas the other approach would tell you definitely whether the unit was faulty. Some algorithms wouldn't know how to decide and typically might choose the action that was highest in a stack of possibilities. A meta-level approach could sift information such as cost of test, discriminatory power, accessibility of units etc to decide what to do, and hence could naturally explain the decision much better.

2.4. Summary of Diagnostic Methods.

Some of the diagnostic systems described in this chapter have been typically algorithmic in nature and can only be applied to limited sets of examples or to examples with unusual constraints. The others have been too general in their approach leaving the computer to iterate through many fault models until a behaviour is generated that agrees with the observed behaviour of the real world device.

In the table of Figure 2.17 I comment on the applicability of each of the diagnostic methods described to five domains. The domains listed in the table have the following characteristics:

- Christmas tree lights. This is a serial circuit. Bulbs are easy to test, by visual inspection or with an Avometer™. If one element of the circuit breaks the whole circuit fails. Other faults include short circuits or the wrong bulbs being used.

- Digital circuit. This domain consists of large arrays of components. Each component has a strict logical behaviour. Tests involve measuring logical values, but can only be performed at selected points in the circuit because the logical gates are buried inside silicon chips. Some remedies include replacement of the integrated circuit or programming the chip to avoid the problem.

- Car. There are many independent subsystems that make up a car. The domain is rich with rules of thumb coming from enthusiastic DIY'ers. There are many tests available that range from the cheap and easy, to comprehensive engine condition tests.

- Medical domain. This too is a very rich domain in which there are lots of tests and remedies that can be given to a patient. Each test has its cost and requires trained staff to perform it. The subsytems of a human body are much more interdependent than those of a car.

- Doorlock. This is an enclosed mechanism. Faults include jammed components, broken springs etc. Not much point in opening up the lock for inspection because the whole unit is cheap to replace. User can get a

feel for the fault by the way the operating lever or key behaves and various noises produced.

Some of the problems associated with the systems described in this chapter stem from the fact that the approaches are geared towards certain domains. The coming chapter examines the process of diagnosis and suggests that a variety of techniques should be used when trying to fix broken devices.

Method \ Domain	Christmas Tree lights	Digital Circuit	Car	Medical Domain	Doorlock
ATMS	Can be used with a model of the components that produces the justifications for the observed measurements.	Appropriate in conjunction with a mechanism for producing the justifications. Elements have well defined behaviours and parameters can take on many different values.	Appropriate to electrical systems. The representation of the sub-systems within the car would have to be well structured to avoid too many variables being maintained at any one time.	The ATMS is a way of cutting down the search needed. Appropriate to use when there are several alternatives to be explored that have different consequences for the patient.	If the problem solver allows the user to try several options in the debugging process the ATMS can be used to arrange the search strategy.
Constraint Suspension	Categories of failure would include fused bulbs and short circuits. Sufficiently small domain for constraint suspension to work quickly.	Appropriate, but slow because domain is likely to contain many constraints and categories of failure. Design of a search strategy will be essential.	Can be used if you know which subsystem to start with. Only some of the systems of the car are easily represented with constraints, such as electrical or hydraulic components.	Not appropriate. Each of the systems of the human body would need their own categories of failure.	Not appropriate. The different components of a doorlock would each need their own categories of failure.
Qualitative Analysis	Inappropriate, diagnosis is likely to be tests on bulbs and measurements of voltages. No qualitative modelling is involved.	Inappropriate. The highly discrete nature of parameters and relations between them rule out qualitative techniques.	Can be used to model some of the causal relations involved in the operation of a car engine.	Use to describe some of the equilibrium processes in the human body. Enables following causal relations to find faults.	Appropriate. Components have simple states. Causal relations between components easily defined.
Helix	Not appropriate because the domain wouldn't be modelled qualitatively.	Not appropriate because the domain wouldn't be modelled qualitatively.	Can be used to diagnose parts modelled qualitatively. Subsystems must be well defined otherwise the technique will be computationally expensive and unwieldy.	Not appropriate. A qualitative model of the human body wold contain hundreds of relations. Suspending each one to do modelling is expensive.	Appropriate. The strategy would find the qualitative relation that ceases to hold in the presence of a fault. Rules are needed to explain how to cure the fault.
CRIB	Inappropriate. Can't use a hierarchy to model this domain because the elements of the circuit depend on each other to be working.	Not appropriate. Best to use algorithmic techniques rather than rules.	Appropriate. Information rich domain, many tests and cure actions available. Suitable for hierarchical decomposition.	Appropriate, but depends on how well the domain can be broken up into independent sub-systems.	Appropriate. The device is easily represented by a hierarchy and various tests can be tried on the components.

Figure 2.17. Applicability of a series of diagnostic methods to selected domains.

Chapter Three

Diagnosis: A Mixture of Skills

The discussion in the previous chapter has highlighted the problems with existing fault diagnosis systems and pointed to difficulties in attempts to represent the behaviour of devices. In this chapter I describe the skills essential for doing diagnosis of devices in a new situation and how those skills combine to form the diagnostic process in NOSTRUM.

Detailed literature describing the steps in fault diagnosis is rare. I describe in the next section the steps I have identified during my own experiences in repairing household gadgets, starting cars and in conversations with mechanics and marine engineers.

3.1. The Skills

Effective diagnosis is a mixture of skills. In Figure 3.1 I have categorized these skills into five phases with respect to fault diagnosis of devices. The diagnostic phases listed there are the 'fall back' steps that a diagnostician encountering a fault with that device would use. This is a 'failsafe' means of getting to the root of a problem. Someone who had worked on that type of device before would be acquainted with the weaknesses of the device, characteristic signs that point directly to a fault. This latter type of knowledge helps cut down the search tree severely and can often lead straight to the fault. Experiential knowledge about a device can help counter some strongly-held beliefs about the ways in which devices go wrong. For instance in most electrical devices wire is never usually considered as a fallible component, but my own experience with the type of wire used in headphones often leads me to check that first.

Below I now explore each of these diagnostic phases and look for the implications they have for a system trying to mimic the processes of a human diagnostician.

Skill	Description
Fault recognition	Discovery of a problem. To make this observation the diagnostician must have some idea of what the device is supposed to be doing and what it is not. Sometimes the fault is something extra that the device is doing, such as generating a grinding noise.
Symptom recognition	Once a problem has been discovered it is necessary to find where in the device the fault manifests. This can be done in some cases by following the expected sequence of operations of the device until a discrepancy is detected.
Tracing	After the fault has been located to some part of the device it is necessary to trace back to find its cause.
Hypothesize & Test	During tracing many causal pathways will be followed. At some points it will be necessary to distinguish between possible causes by making a test. The test chosen will depend on how viable the hypothesis is and what test equipment is available.
Repair Action	When a hypothesis is confirmed the diagnostician must choose the appropriate repair action. If this fails to solve the problem the assumptions underlying the tests that suggested the hypothesis will be brought into question.

Figure 3.1. The phases of a diagnosis.

3.1.1. Fault Recognition.

This stage occurs when the user of a piece of equipment first becomes aware of the breakdown of a device. Sometimes this will be signalled by a

warning light or buzzer, by the device simply not working, or doing something that it is not supposed to. In order to spot this fault the user has to be aware that the observed phenomenon is not expected, that is they have to have some idea of how the device is supposed to behave. This is the minimum amount of knowledge required to be aware of a problem.

3.1.2. Symptom recognition

After a problem has been detected in a device the diagnostician must find where in the device the fault manifests. If the fault has been recognized by a warning light the user will know to which part of the device the light refers. For instance in a car engine the oil pressure warning light will point to the amount of oil in the engine. If the fault has been discovered because of an unusual noise generated by the device the user may have to try and repeat the operation of the device until the source of the noise can be pinpointed. During this stage the user of a device is using simple knowledge of the structure of the device or associations between indicators and components to locate the fault.

3.1.3. Tracing

Tracing the fault from its manifestation to cause requires a deeper understanding of the causal pathways of the device. The diagnostician has to trace back, following wires, pipes or even routes of magnetic flux to look for an explanation for the fault. The diagnostician who is familiar with the device will be able to draw on past experience with it, possibly recalling similar situations and hence being able to make constructive guesses about how it may have failed. Ultimately if the guesses fail the diagnostician will have to go back to first principles and look for physical reasons for the observed device behaviour.

3.1.4. Hypothesize and Test

The process of tracing will eventually lead the diagnostician to what may be the cause of the fault, or to a point in the device where one of

many different changes in the structure of the device could account for the observed behaviour. To distinguish among them the diagnostician can either suggest a test that needs to be made or predict the consequences of hypothesized causes and match these against the observed state of the device.

Tests have costs and benefits. A test may vary from a simple observation, such as a humming sound, to removing a printed circuit board from the device and sending it away for examination. The benefits of a test are equally varied. They can eliminate whole sections of the device, or simply confirm a suspicion. In practice a diagnostician chooses a test on a brief evaluation of these costs and benefits. Sometimes an attempt to repair the device is made by replacement of an accessible and cheap component. Figure 3.2 shows the 'Testability Ladder' used by NOSTRUM to assess from a pool of possible tests which would be the simplest to perform. The ladder lists in increasing difficulty the types of test a user can perform.

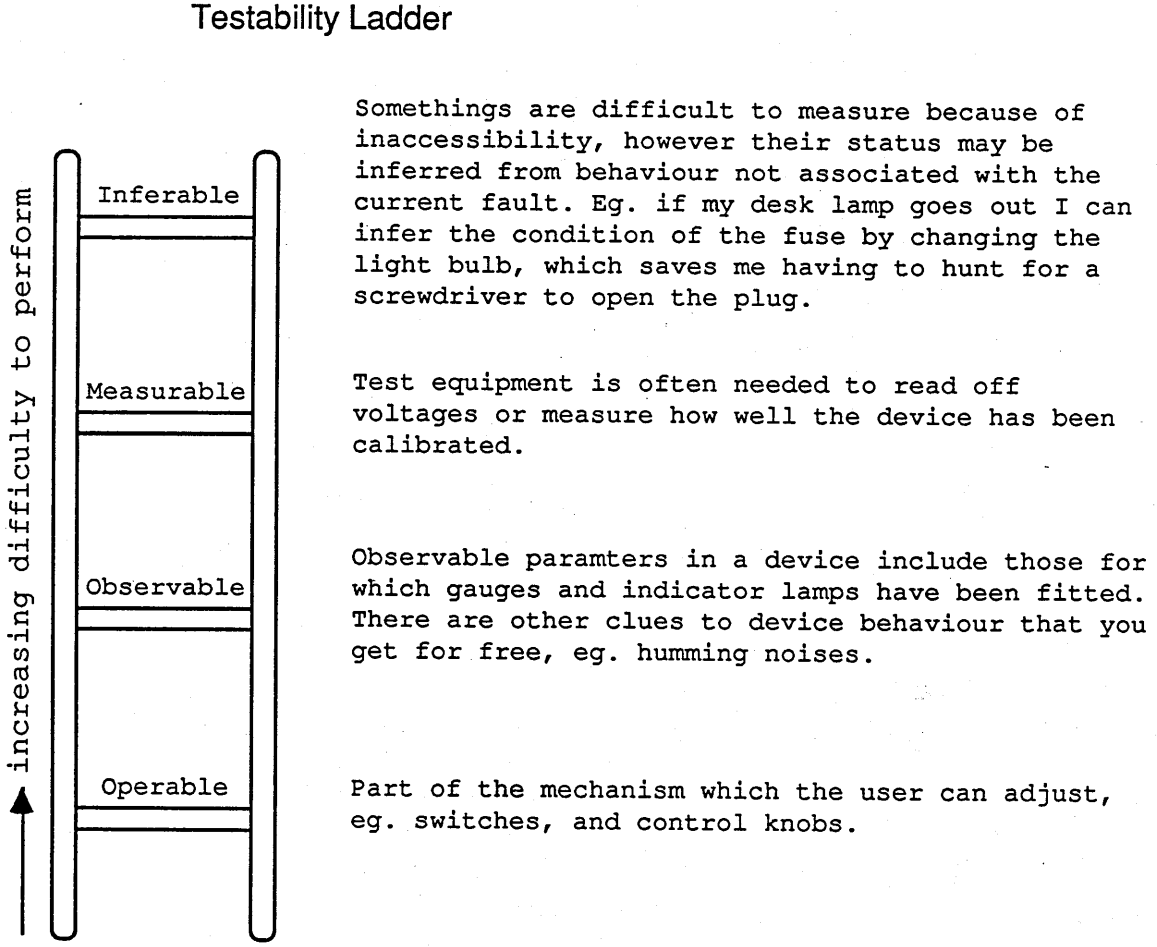


Figure 3.2. The Testability Ladder used by NOSTRUM to assess among competing tests which would be the simplest for a user to perform.

For a computer system to be able to devise these tests it must also contain knowledge about the components of the device at a 'black box' level. For instance if the structure of a machine can be modelled by a chain of 'black boxes' in which each box supposedly works correctly if it gets the right signal from the preceding box, then it is easy to do diagnosis by applying 'divide and conquer' to the chain.

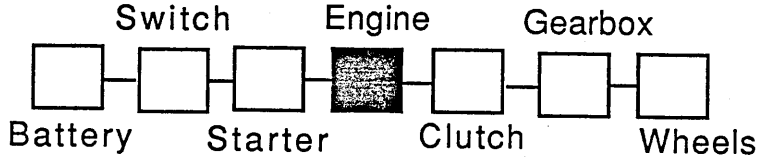


Figure 3.3. A chain of Black boxes with a faulty one somewhere along the line.

Some devices have redundant linkages which allow them to be operated from more than one source (eg. the 2 starting mechanisms of a car as shown in Figure 3.4). When a fault occurs in this type of device then components can be eliminated from diagnostic inquiry by operating the device through an alternative route.[1]

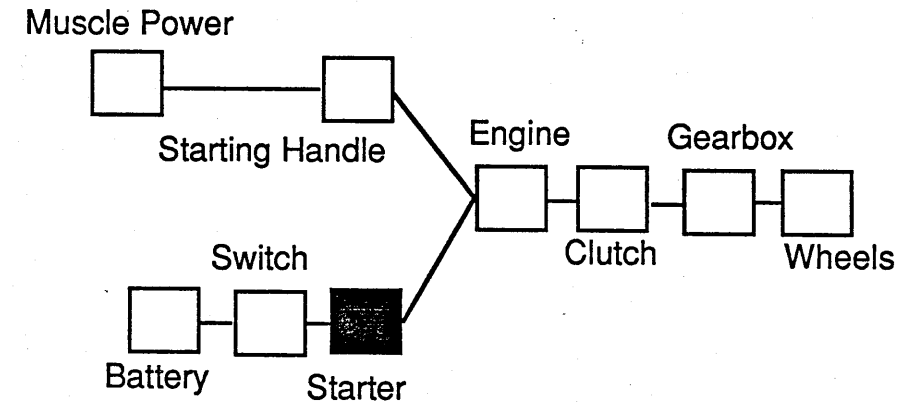


Figure 3.4. The starter motor is suspected, but can be by-passed using the starting handle.

This type of knowledge enables *commonsense reasoning* about the structure of the device. The diagnostician can make use of this type of knowledge to assist in generating tests and making them easier to perform.

---

[1]Note: if the starting handle is not available then another way to start the car is by passing turning motion through the wheels, gearbox and clutch direct to the crankshaft. A model of these parts will normally be designed only with transmission in mind and hence the model must be flexible enough to allow passing torque either way through the transmission. This could be done by making the clutch, gearbox and differential members of a class of bi-directional devices.

These skills require the diagnostician to have familiarity with the components of the device at both fine-grained and coarse-grained levels of detail. The ability to abstract the behaviour of subunits of the device will be essential if the diagnostician is able to focus on and manipulate the suspected parts.

3.1.5. Repair Action

When the diagnostician embarks on making a repair they are not always confident that it will fix the device. They might be making the repair as part of a test if the repair is typically easy to do, accessible or if the replacement part is cheap. Sometimes though this is not the case and the action taken was the result of an exhaustive investigation. When this happens the steps that led to the repair action must be re-examined. Were there any assumptions made about the device that should now be reconsidered in light of the new information? Were any steps missed out, could the action of replacement have damaged another part of the device through clumsiness and indeed could the replacement part be broken in the same way?

As an example of how an apparently infallible test steers the diagnostician away from the real cause of a fault I describe a test for the ignition system of a car:

A standard test to check whether a spark plug is firing is to remove the suspect plug and lay it on the engine block. The ignition lead is connected to it so that it is still in the circuit. The engine is turned over using the starter motor and if the plug is working properly sparks are seen. In one reported case however [The Courier 19??] this test was carried out and verified the spark plug to be working. But the symptoms remained when the plug was replaced. In desperation the owner fitted a brand new set of plugs and the problem went away. Closer examination of the original plug revealed a hairline fracture in the porcelain. Under the high pressure inside the engine cylinders this crack was opening up and allowing the spark to escape to earth without igniting the petrol.

In the above rather extreme example the diagnostician would have had to have a remarkable insight into the physics of sparks in order to revoke the results of the test. Knowledge of the limitations will help a diagnostician suggest better tests and be wary of when not to pay too much attention to them.

The amount of time spent in each of these five stages by a diagnostician will depend greatly on the intricacies of the application. Other sources of information can help sidestep some of these stages altogether and some examples of this are listed in Figure 3.5.

Batch information	If the device belongs to a batch that is known to suffer certain weaknesses it can often be quicker to check these first. Batches of a device can acquire a bad reputation due to design faults or bad manufacture. The diagnostician can go straight to the usually faulty component and inspect it. Replacement is often made in these cases without performing a full test on the repaired device because of the extreme confidence in the diagnosis.
Previous Experience	As a diagnostician gains experience with a class of devices they become aware of the more common ailments and optimize the diagnostic strategy to cover those cases first. The optimized strategy suggests short-cut tests that can discriminate between common cases. Sometimes they may have even designed a specialist tool or test equipment to aid the diagnosis.
Common symptoms	Sometimes an experienced diagnostician familiar with a device or class of devices can immediately suggest a hypothesis after the briefest description of a symptom. They often ask a highly specific question the answer to which apparently lifts all doubt in their own minds as to what has gone wrong. For example: a car that cannot be put into gear would immediately suggest a broken clutch, however if it is known that the car has been standing for some time the clutch plate will have become stuck to the flywheel. The simple solution is to rock the car until the plate jolts free. In this example the experienced mechanic would simply ask "How long has the car been standing?" or if the vehicle is covered in cob-webs this question need not be asked!

Figure 3.5. Some situations that help short cut diagnosis.

NOSTRUM is concerned with diagnosis in the absence of this information. It is concerned with doing diagnosis on those occasions

when the types of information in Figure 3.5 are not necessarily available, ie. when the things that can go wrong cannot necessarily be listed beforehand. This is not to say that the design of NOSTRUM shuns such information, in fact its design has allowed for this information to be incorporated or even learned in a naive way. NOSTRUM is interested in the skills a diagnostician has to fall back on when working with a new device and using principles about a device learned in the classroom.

3.2. The Diagnostic Process

Figure 3.6 shows how the skills described in the previous section are brought together to form the diagnostic method in NOSTRUM. As described in chapter 4, NOSTRUM uses a model of the device being diagnosed. The numbers in parentheses in the following paragraphs correspond to the circled numbers in Figure 3.6.

(1) The model represents the operating principles of the components involved and is able to simulate the behaviour of a device. When a difference between NOSTRUM's simulation and the behaviour of the real world device emerges a fault is recognized.
(2) NOSTRUM uses a description of the observed symptom and the model to trace the causal pathways of the device.
(3) The models of the operating principles are used to generate hypotheses about possible causes of the fault, suggesting new values for parameters of the device.
(4) NOSTRUM simulates the device with the newly proposed values to try and build a picture of the device in the faulty state.
(5) Each of the faulty states produced by the hypotheses will have values that can be checked.
(6) With reference to the 'Testability Ladder' the easiest test to perform is chosen. The user is asked to perform the test and return the result.

(7) The user response may prompt NOSTRUM to suggest a repair action. If not, the user's measurement is treated as another symptom and used to generate further hypotheses.
(8) If the proposed repair action still fails to produce a diagnosis then the assumptions used to generate the hypothesis are re-examined.

Figure 3.7 shows how the diagnostic process would look if experiential knowledge were incorporated into NOSTRUM.

In Chapter 4 I show how the skills and diagnostic process described here are implemented in NOSTRUM.

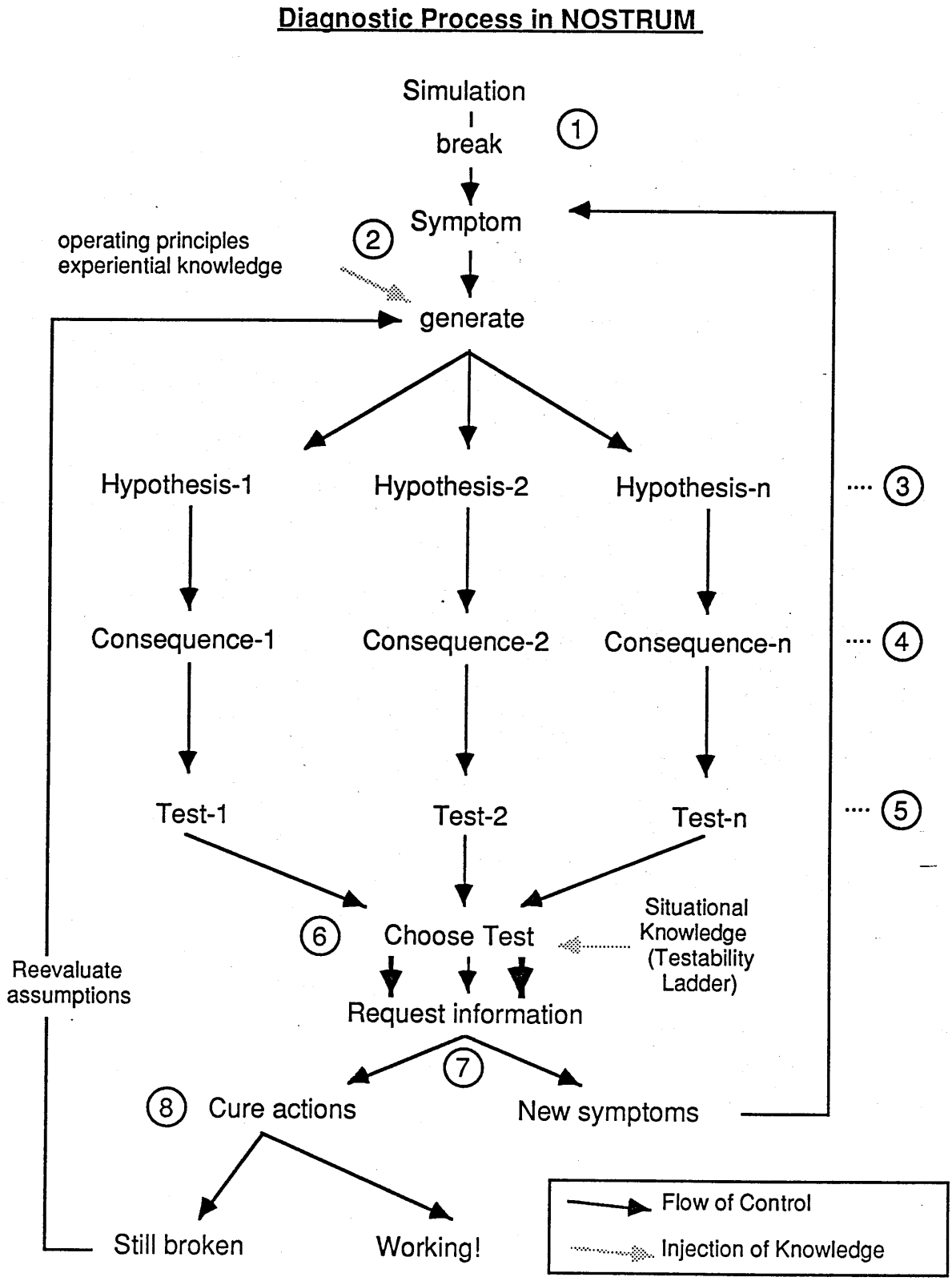


Figure 3.6. The diagnostic process in NOSTRUM. Circled figures refer to paragraphs in the text.

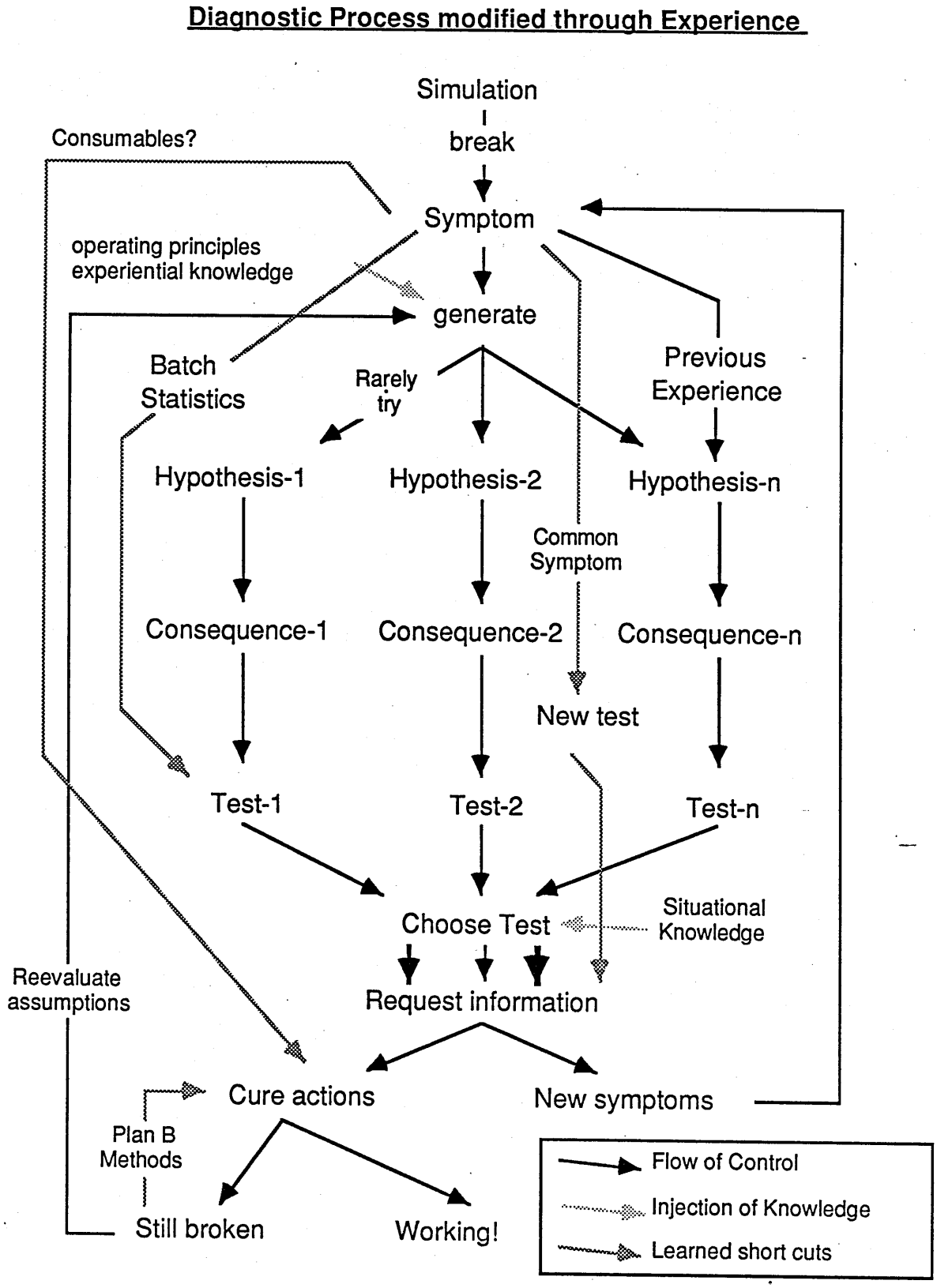


Figure 3.7. The diagnostic process modified by experience gained after working for some time with a device or class of devices.

Chapter Four

Roots of a Representation

In chapter 2 we saw various methods of representing the behaviour of devices. The techniques ranged from the limited usefulness of rule based representations to qualitative analyses of the physical processes in a device. In this chapter I propose a representation based on the *operating principles* of the components of a device. The representation succeeds in being able to mimic the behaviour of the device whilst enabling the tracing of fault conditions back through the device to previous states.

Chapter 3 classified diagnosis into 5 phases. The 5 phases suggest features that a computer program trying to reproduce the process would need. In the following table I relate the skills of chapter 3 to the features of a computer program that would need to represent them.

Skill	Required Feature
Fault recognition	A notion of what the device is supposed to do.
Symptom recognition	A familiarity with the structure of the device, where the parts are.
Tracing	A deeper understanding of the causal pathways within the device, and an understanding of how change in one part of the device causes change in another part.
Hypothesize & Test	Knowledge of how the causal pathways can breakdown or be interfered with. The ability to predict the consequences of a change in the structure of the device. Knowledge of what can be tested.
Repair Action	Knowledge about repairing/replacement parts.

Figure 4.1. The Five Phases of diagnosis.

4.1. Nostrum - Rationale

All devices are built from a set of components which can be treated as black-boxes at varying levels of abstraction. For instance washing machines and cars both use engines as a source of turning power so a designer of such devices might say "...we could use an engine here ...". There are many sorts of these *device units* which can be viewed at varying levels of abstraction and that can be connected together to make the device of your choice. Here are examples of how some units are used together:

Clock activates an alarm.
Clock turns the heating on.
Motor is used to drive a car.
Motor winds cassette tape.
Light activates a switch.
Light helps us to see.
Ballcock shuts water off.
Pendulum regulates clock timing.
Spring stores energy in a childs toy.
Spring softens the ride along a bumpy road.
Spring shuts a door behind us.
Optical sensor finds right track on a Compact Disc.
Feedback mechanism regulates fridge temperature.

Figure 4.2. Examples of the use of components in devices.

The list shows devices used in different ways. In one a clock is considered as a unit which activates something at a certain time. In another a lever controls the accuracy of a clock. The list also shows how the same component can be used in different ways in different devices; but that it is always the same aspect of the component that gets used. Take for instance the spring: it does indeed store energy but the energy can be absorbed quickly and let out slowly (childs toy), absorbed quickly and let out quickly (vehicle suspension) and absorbed slowly and let out slowly (fire door). In each case there is the presence of a compressor (turning key, bump, push) and a release (moving toy, vehicle jumping and door closing). Always it is the operating principle of a spring at

work (compress, store energy, release) but interacting with different things.

Sometimes a whole type of procedure can be abstracted out, such as the notion of feedback which contains driving and measuring components to keep another parameter under control.

If a device can be represented by connecting together known models of the device units then the behaviour of a device could be understood in terms of the behaviour of its components. Such a representation would facilitate diagnosis of a device through propagation of observed symptoms to faults in subdevices, and would enable the easy representation of a wide range of devices. Thus to develop a system that can do diagnosis it is important to build up a *device vocabulary* so that a new device can be understood in terms of the interconnections of known device units.

This argument is reinforced by books that try to explain how devices work.

4.1.1. How it works diagrams

Figure 4.3 is taken from a popular science book called "How it Works" [Graf and Whalen 1977]. The book explains how all sorts of everyday devices work. Each device has two or three figures showing its innards and some text explaining the sequence of events involved during the operation of the device. The diagrams have arrows indicating which way wheels turn and the direction of the flow through pipes and around circuits. By relating the images in the diagrams to one's everyday experiences of levers, flows and electricity it is possible to 'understand' how the device works. My long-term aim is to equip a system with sufficient understanding of these device units so that it can read these diagrams and hence understand the way the device works.

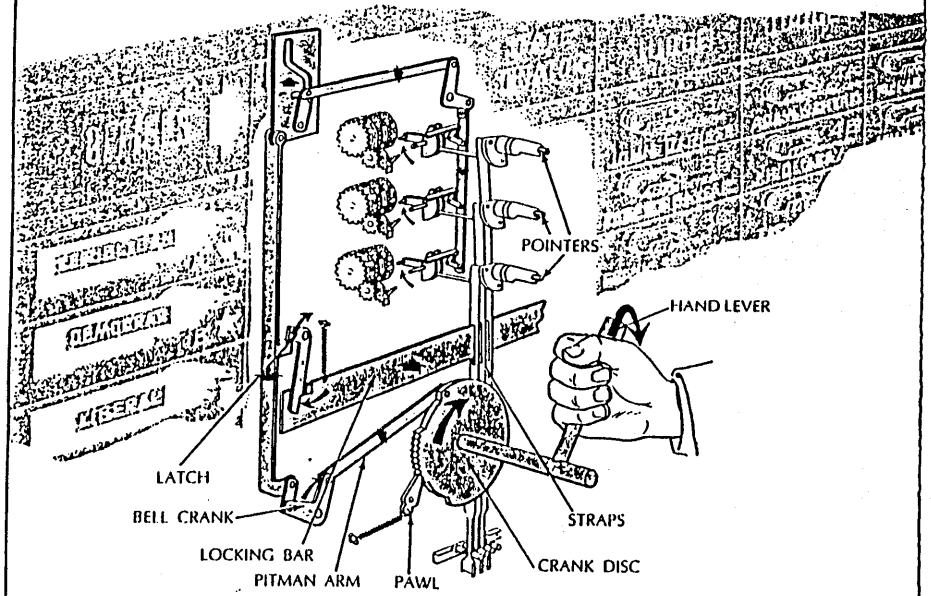
If a vocabulary for device behaviour can be established so that by reading these diagrams the functionality can be understood, then it is a short step to doing diagnosis. Fault models can be easily defined and

moreover the consequences of the fault will easily be visualized in the device. For instance if a wheel cannot turn sufficiently for a slot to enable a bar to drop then symptoms would be 'stuck wheel' and 'bar won't drop'.

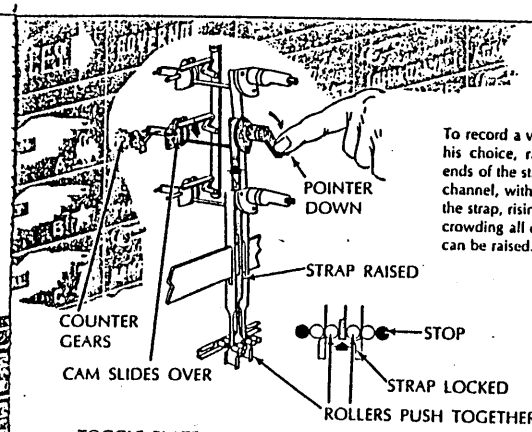
This paradigm calls for a very flexible representation that can handle all sorts of hypothetical faults and also a mechanism for assessing when hypotheses become too unfeasible.

Voting Machine

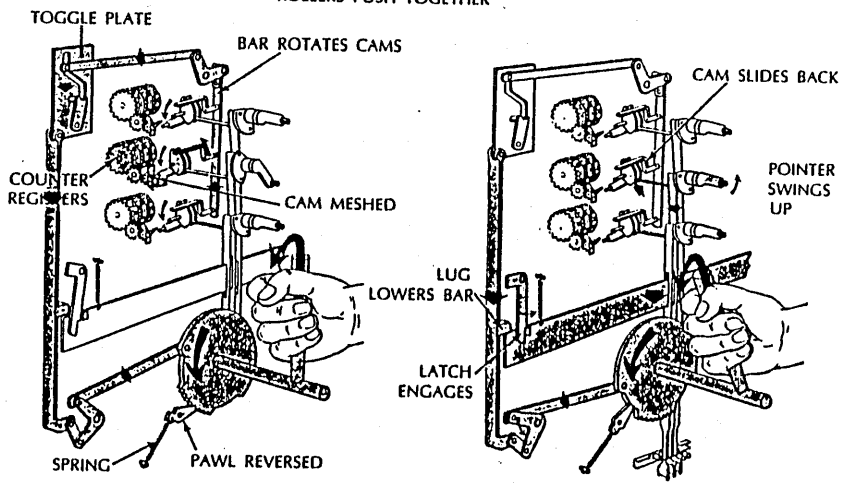
Though expensive, voting machines pay for themselves by reducing routine election costs and minimizing the number of recounts. Another advantage is that they give quick returns after the polls close. They require little maintenance, last a long time, and—according to some politicians—help draw voters to the polls. Interlocks prevent pulling the lever for too many candidates or voting both YES and NO on a question. A paper roll permits write-in votes. The first voting machine was used in Lockport, New York, in 1892. Today machines are used in nearly every state.



To cast a ballot in secret, a person entering the booth closes the curtain by swinging the hand lever to the right. This also turns a crank disc, working two pitman arms (only one is shown above). Through bell cranks, the arms trigger a latch on each side, releasing a spring-mounted locking bar, which snaps up. (The bar runs through a slot in each of the metal straps connected to the selection-lever pointers and locks them until it is released.) Now the pointers are free, ready for voting. Ratchet teeth on the crank disc, with a pivoted pawl, prevent reversing the lever once it has been moved. When the pawl slips past the last tooth, it is free to flip the other way for the return of the lever, as shown on the next page.



To record a vote, the voter swings down the pointer of his choice, raising the strap connected to it. Bottom ends of the straps hang between small rollers in a rigid channel, with stops at both sides. The enlarged end of the strap, rising between two rollers, forces them aside, crowding all others so that no other strap in that group can be raised.



To change his vote, the voter need only flip up the incorrectly selected pointer and swing down another. But stops and rollers, previously set up, let him vote only for the right number of candidates (one mayor, four councilmen, for example). Switching a pointer down slides the lever's registry cam into engagement with a three-digit counter that totals the votes for that candidate. When the voter has made all his selections, he swings back the hand lever, with the ratchet pawl now reversed. The crank disc and its linkage first pull down a toggle plate (top left). Through a bell crank this rotates all pointer cams, but only those in mesh with counters register votes. Next, lugs pull down the locking bar. The bar pulls down the straps, flipping up all down-swung pointers. Finally the mechanism opens the curtain.

Figure 4.3. A typical diagram taken from 'How it Works' [Graf and Whalen 1977].

4.2. Christmas Tree Lights

To illustrate the type of reasoning described in section 4.1 and chapter 3, I use the deceptively simple example of a frugal set of Christmas tree lights (Figure 4.4). At the highest level this can be thought of as a simple circuit with a switch which when closed the lights come on. However the number of ways in which this simple system can go wrong is considerable.

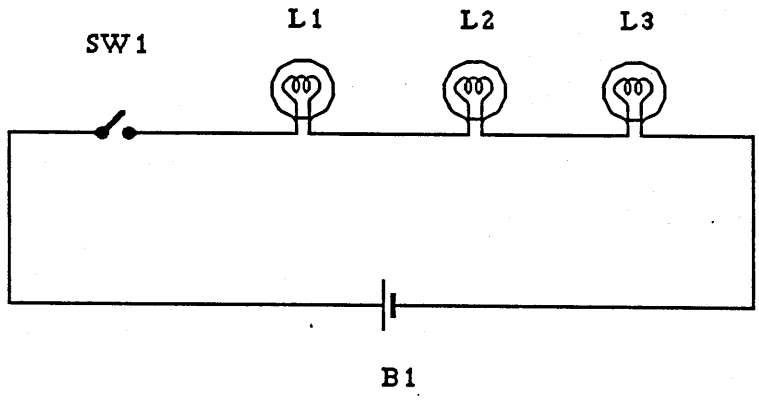


Figure 4.4. A Frugal Christmas tree light set.

The light set is composed of a switch, battery and three lights each screwed into a socket. To understand this device is to understand how each of these components interact. The behaviour of each component (or *device unit*) has to have some associated knowledge, a description of what it is supposed to do, how it does it and what it needs in order to do it. This associated knowledge represents the *operating principles* of the device unit. Figure 4.5 describes the operating principles of the components of the Christmas tree light set.

Device Unit	Operating Principles
Bulb	For a bulb to be working its filament must be intact and there must be a current flowing through it. The stronger this current the brighter the bulb and the greater the voltage drop across it. A bulb's resistance depends on its voltage rating and wattage.
Battery	A battery provides an electro-motive force for a circuit which decays as the battery loses its charge.
Switch	A switch has zero resistance when closed and infinite resistance when open. Dirty contacts present a high resistance when the switch is closed, and if high voltages are used sparks can jump the switch if it doesn't open enough.
Wires	Wires conduct electricity with negligible resistance.

Figure 4.5. Examples of Operating Principles for electrical components.

In NOSTRUM the operating principles of device units are modelled using networks of constraints. In the next sections I described the constraint models for a light bulb, a circuit and a switch.

4.2.1. Constraint model of a light bulb

The constraint model for a bulb is shown in Figure 4.6.

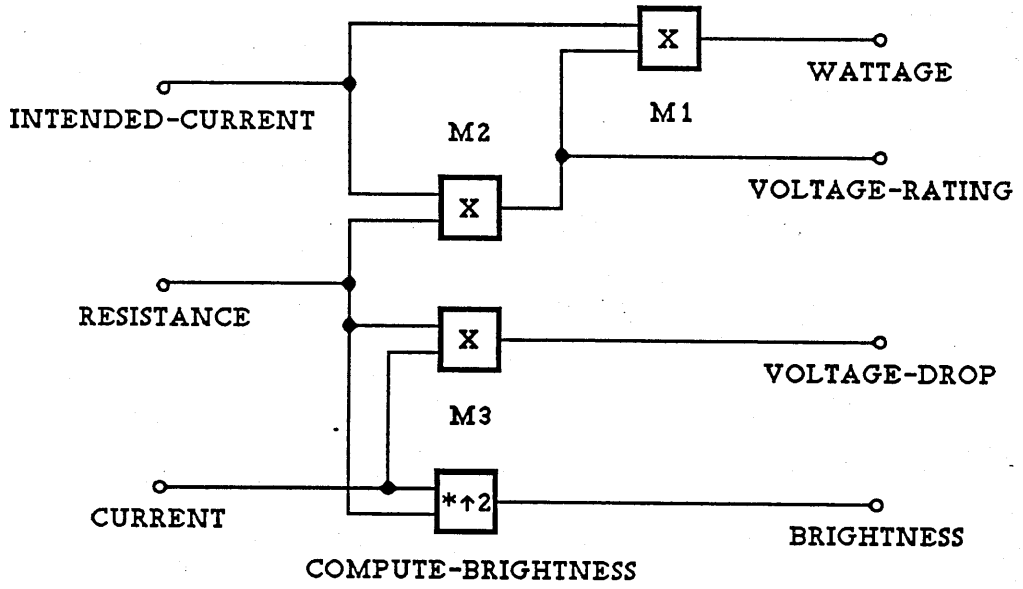


Figure 4.6. The constraint network representing the operating principles of a light bulb.

The constraint network contains three multipliers `M1-3` and a brightness constraint `Compute-Brightness`. The structure of the constraint network is derived from two laws of electrical currents, Ohm's Law and the Power law:

Ohm's Law: V=IR

Power Law: P=IV,

where P is the Power of the bulb, I is the current through it, V is the voltage across it and R is its resistance.

Given the Wattage and Voltage Rating of a bulb it is possible to derive the current that is intended to pass through the bulb using the Power law:

```
Wattage = Voltage-Rating * Intended-current.
```

This relation is effected by the `M1` constraint of Figure 4.6. The `M2` constraint in that figure is an application of Ohm's law relating the intended current to voltage rating and resistance of the bulb.

The `M3` constraint in Figure 4.6 relates the voltage drop across the bulb to its resistance and the actual current flowing through it. The final constraint, `Compute-brightness`, makes use of another version of the

power law: $P=I^2R$ to infer the brightness of the bulb from its resistance and the current flowing through it. Values are passed between the constraints according to the rules of *constraint propagation* as described in section 2.1.1.

Two important parameters of any light bulb are its Wattage and Voltage. These are the two parameters that are specified when buying a light bulb at a shop. In terms of how the bulb performs in a circuit though, two other parameters are important: Resistance and Current. Given the Wattage and Voltage rating of the bulb the constraint network of Figure 4.6 can deduce the bulb's resistance, and the less important parameter, `intended-current`. Figure 4.7 shows the constraint network for a light bulb with a power of 16W and voltage rating of 4V. Figures 4.6 and 4.7 are in fact screen snapshots of Circuit Buff which forms the graphical arm of NOSTRUM. Details of how Circuit Buff is used and how the constraints are implemented are described in sections 6.1 and 6.2.1 respectively.

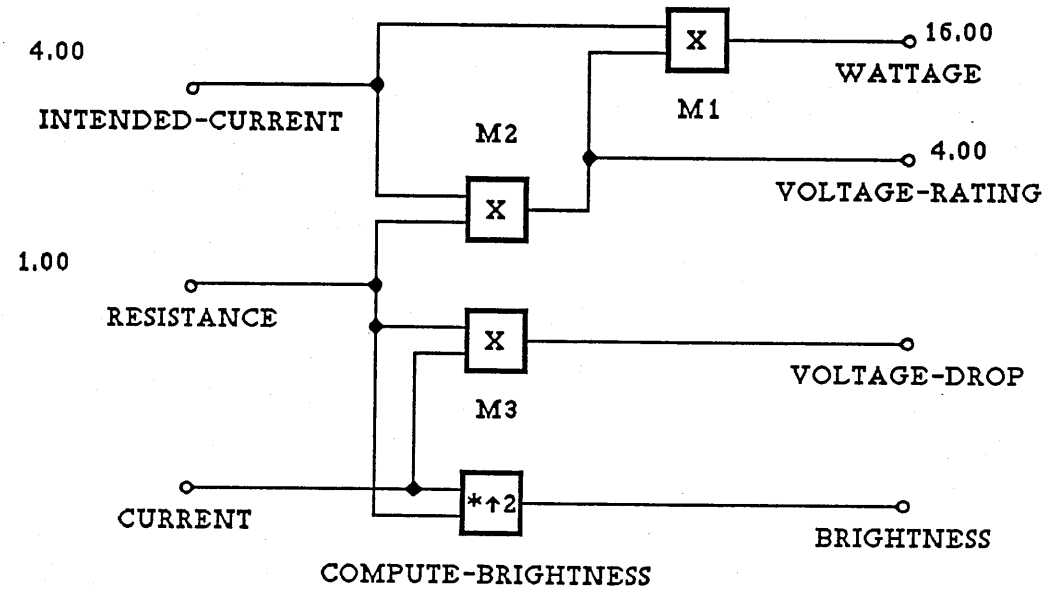


Figure 4.7. The constraint network for a 16 Watt, 4 Volt light bulb.

The constraint mechanism therefore defines the behaviour of the parameters of the light bulb according to the physics of electrical current

flow. The remaining parameters of the light bulb can only be determined when the current through it is known. The value of the current is in turn determined by the circuit of which the bulb forms part.

4.2.2. Circuit Constraint Model

A circuit is also an entity known to engineers, they understand that for a circuit to exist there must be a loop for electrons to flow around, a driving force, and that there will be something that needs driving (a load). Thus the circuit is also considered as a device unit with its own set of operating principles. The constraint network for a circuit is shown in Figure 4.8.

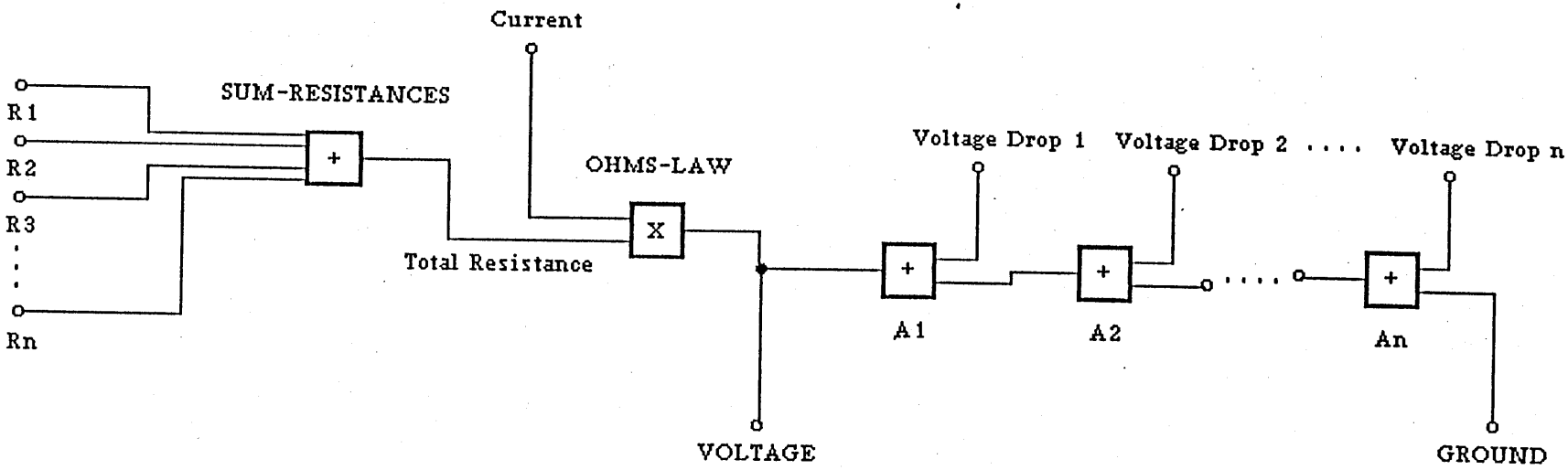


Figure 4.8. The constraint network representing the operating principles of a circuit that contains n components. This network behaves like a template; any number of components can be slotted in, and using their resistances and the voltage across them all the other parameters can be deduced.

The central element of Figure 4.8 is the constraint that represents Ohm's law. It is simply a multiplier that takes the `total-resistance` (deduced from the adder constraint `sum-resistances`) and `voltage` as inputs to establish a value for the current in the circuit. The network represents a template that can be used to describe any circuit that is comprised of n elements. The series of adders `A1, A2 ... An` compute the voltages on the wires between the components.

4.2.3. The Switch Constraint Model

To demonstrate that the constraint system in NOSTRUM is not purely restricted to numeric values I reproduce the constraint model for a switch in Figure 4.9.

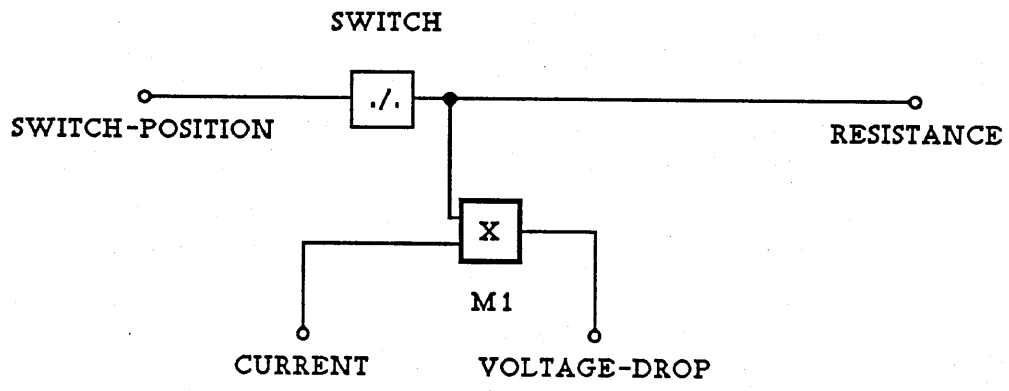


Figure 4.9. The constraint model of a switch.

The model features the switch constraint which has one input and one output. The input is the state of the switch which can be `OPEN` (off) or `CLOSED` (on). When the input is `OPEN` the output, which represents the resistance of the switch, has the value `INF`, meaning infinity. When the input is `CLOSED` the output value is `0`. The other constraint `M1` is a multiplier which computes the voltage drop across the switch.

The constraint models described in this section have been designed to behave as 'building blocks' so that they can be used in many combinations to define the structure of a range of devices. The following

section shows how they are combined to simulate the Christmas tree light set.

Chapter Five

Using Nostrum to Diagnose Christmas Tree Lights

In this section I show how NOSTRUM diagnoses a Christmas tree light set using the type of reasoning described in chapters 3 and 4. Details of how NOSTRUM is implemented are reserved for a later chapter.

The procedure for using NOSTRUM comprises several steps which are summarized below:

(1) The structure of the Christmas Tree light set is defined graphically.

(2) The structure is transferred to a frame language.

(3) Production rules identify the components of the circuit.

(4) Using known models of the operating principles of the components a constraint model of the circuit is built.

(5) The model is initialized to reflect the state of a working device.

(6) A symptom is given to the model, initiating diagnosis.

(7) The constraint model generates hypotheses.

(8) The user is prompted to make observations and measurements to try and establish or condemn the hypotheses.

(9) Repair actions are suggested.

These steps are described in greater detail in the following sections.

5.1. Defining the Device Structure

The structure of the Christmas tree light set is defined using the graphical toolkit, Circuit Buff. Circuit Buff is a subsystem of NOSTRUM that allows a user to define a series of icons that represent components

of devices. The icons can be positioned on the screen and connected together through a mouse-and-menu interface. Details of using Circuit Buff to construct diagrams of devices are described in section 6.1.

Figure 5.1 shows a screen snapshot of Circuit Buff after the structure of the Christmas Tree light set has been defined.

5.2. Transfer to a Frame Language

A menu option in Circuit Buff copies the device structure into the KEE™[1] knowledge base SOLDER. The Solder Knowledge base forms the second half of NOSTRUM[2].

The Solder knowledge base contains a hierarchy of objects or *frames* [Minsky 1975] that define the characteristics and behaviour of classes of device units. The structure of the Solder knowledge base is explained in detail in section 6.2. The main object class in the Solder knowledge base is called `device-unit`. This object has several offspring which include `Lamp`, `Battery`, `Switch` and `Wire`. These objects contain *slots* which can be used to model their operating principles.

Transferring the device from Circuit Buff instantiates its components as children of the corresponding sub-class of `device-unit`. Icons in Circuit Buff now correspond to KEE frames, or KEE units with

---

[1]KEE™ is a registered trademark of Intellicorp Ltd. Mountain View, Califorinia.

[2]Historical note: NOSTRUM is a computer program that has evolved during the completion of this Ph.D. Originally the constraint propagation system was built entirely using lisp `flavors` and still forms part of Circuit Buff. This version allows the user to draw constraint networks, define constraints and initiate propagation through a mouse and menu interface. However for diagnosis, access to a hypothetical worlds system was necessary. Such a system is provided by KEE. Consequently the behaviour of the constraint propagator was also implemented using the KEE object system. The differences in the representation between the two styles are described in detail in chapter 6.

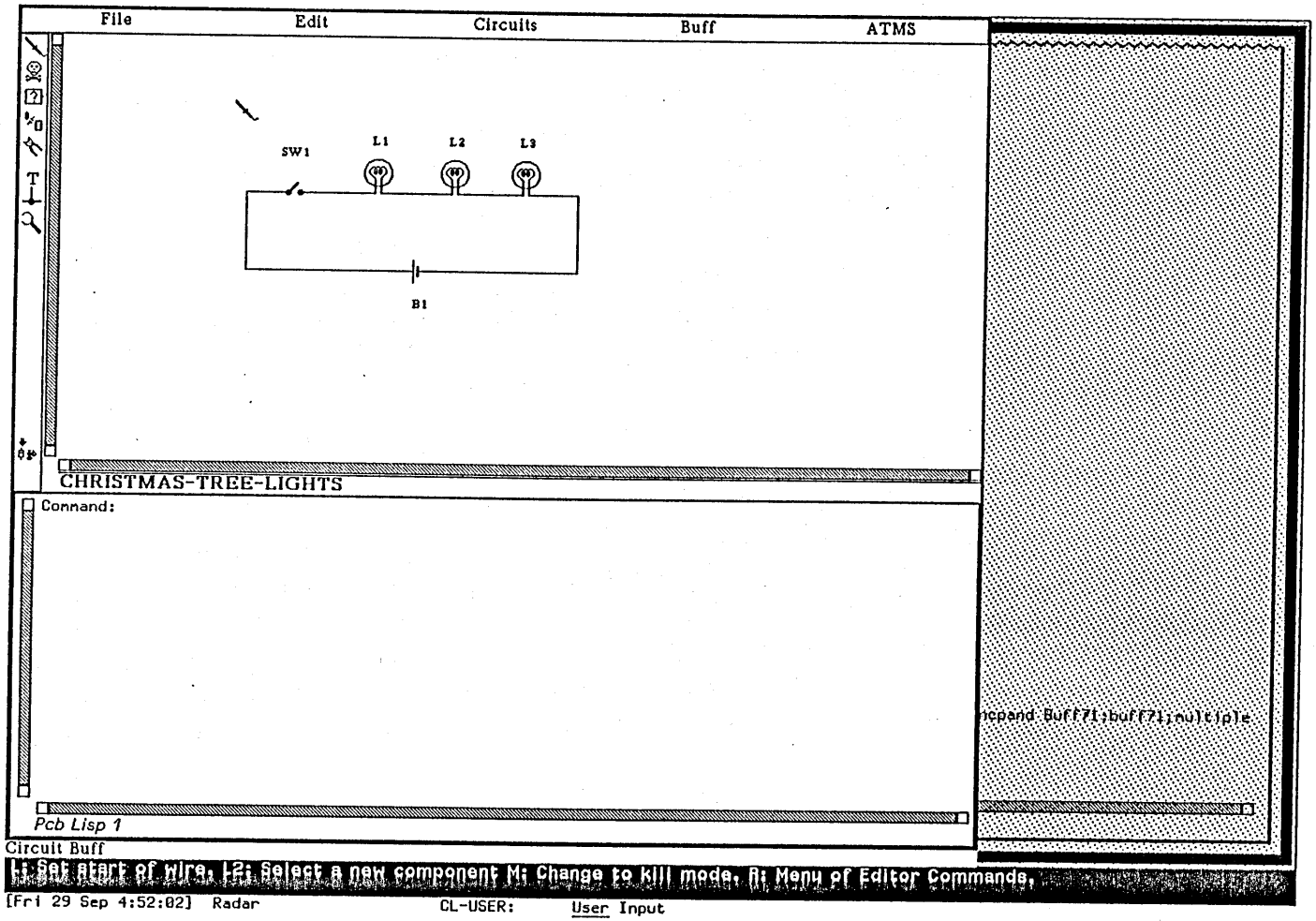


Figure 5.1. A screen snapshot of Circuit Buff showing the structure of the frugal Christmas tree light set. Details of the screen layout and menus of Circuit Buff are shown in section 6.1.

the same names. The transfer process also copies the connectivity of the components to the connections slot in the KEE frames.

5.3. Production rules interpret circuit Structure.

To interpret the structure of the circuit a set of pattern matching rules are then used. The purpose of the rules is to identify the circuit and its elements so that a constraint model can be built. The rules are shown in Figure 5.2. Four rules are used in forward chaining mode. At first only the START-LOOKING rule can fire. It picks out the battery B1 and finds the wire that is connected to its 'plus' terminal. The RHS (right hand side or consequent) of this rule creates a new object B1-ONWARDS as a child of the CIRCUIT-PART unit in the Solder knowledge base. This new object is used alternately by the CARRY-ON-LOOKING-FROM-WIRE and CARRY-ON-LOOKING-FROM-DEVICE rules. Each of these uses the CURRENT-END slot of the B1-ONWARDS unit and the connections slot of the wires and components to build up the route of the circuit.

So the rules move round the device starting from the battery B1 and constructing a representation of the circuit in the B1-ONWARDS unit. When the rules come full circle back to B1 the fourth rule IDENTIFY-LOOP fires. This rule establishes that the circuit forms a loop and changes the parent of the B1-ONWARDS unit from CIRCUIT-PART to CIRCUIT.

At this point the circuit and all its elements are indirect children of the DEVICE-UNIT class in the SOLDER knowledge base. Using the constraint models of the device units and their connectivity NOSTRUM proceeds to build the constraint model for the whole device.

5.4. The Constraint Model is built.

The final action of the production rules is to initiate construction of the constraint model of the circuit. This is done by instantiating the corresponding constraint model (as shown in section 4.2) for each element of the circuit. The constraint models of the components are then

```
;;; -*- Package: K; Mode: LISP; Syntax: Common-lisp -*-

;Rules for identifying circuit loops.

(defrule start-looking spot-circuits   ;The START-LOOKING rule in the SPOT-CIRCUITS ruleclass.
  "This rule gets the search off the ground by beginning at a battery."
  (if (?b is in battery)                          ;If there is a battery
      (the connections of ?b is ?c)               ;that is connected through its
      (lisp (equal (car ?c) 'plus))               ;'plus' terminal to
      (?w is in wire)                             ;a wire...
      (lisp (equal (unit.name ?w) (cadr ?c)))
      (?cp = (intern (format nil "~D-ONWARDS" (unit.name ?b))))
   then (?cp is in circuit-part)                  ;then identify a circuit part
        (the start of ?cp is ?b)                  ;that starts at the battery and
        (the current-end of ?cp is ?w)            ;whose current end is the wire.
        (the components of ?cp is ?b)             ;Record the components encountered
        (the components of ?cp is ?w)             ;so far.
        (lisp (put.value ?cp 'route (list ?w ?b)))))

(defrule carry-on-looking-from-wire spot-circuits
  "Given that part of a circuit has already been found look for more of it."
  (if (?cp is in circuit-part)                    ;If there is a circuit-part
      (the current-end of ?cp is ?ce)             ;which currently ends at
      (?ce is in wire)                            ;a wire.
      (the connections of ?ce is ?c)              ;And the wire is also connected
      (?cd = (unit (car ?c)))                     ;to another component that
      (cant.find (the components of ?cp is ?cd))        ;hasn't yet been encountered
   then (change.to (the current-end of ?cp is ?cd))     ;then make the new component the
        (the components of ?cp is ?cd)            ;current end, and record the encountered
        (lisp (put.value ?cp 'route (cons ?cd (get.value ?cp 'route))))))       ;device.

(defrule carry-on-looking-from-device spot-circuits
  "Given that part of a circuit has already been found look for more of it."
  (if (?cp is in circuit-part)                    ;If there is a circuit-part
      (the current-end of ?cp is ?ce)             ;which currently ends at
      (?ce is in electric-device)                 ;a component.
      (the connections of ?ce is ?c)              ;And the component is also connected to
      (?cd = (unit (cadr ?c)))                    ;a wire that has't yet been encountered.
      (cant.find (the components of ?cp is ?cd))
   then (change.to (the current-end of ?cp is ?cd))     ;Then add the new component to the
        (the components of ?cp is ?cd)            ;current end of the circuit-part and record
        (lisp (put.value ?cp 'route (cons ?cd (get.value ?cp 'route))))))       ;the wire.

(defrule identify-loop spot-circuits
  "When the last wire has been found claim that a circuit has been found."
  (if (?cp is in circuit-part)                    ;If there is a circuit-part
      (the current-end of ?cp is ?ce)             ;whose current end is a wire.
      (?ce is in wire)
      (the connections of ?ce is ?c)              ;And the wire is connected to another
      (?cd = (unit (car ?c)))                     ;component
      (?cd is in battery)                         ;that is a battery.
      (lisp (equal (cadr ?c) 'ground))            ;And the connection is to the ground terminal.
   then (change.to (?cp is in circuit))           ;Then elevate the circuit-part to a circuit,
        (lisp (unitmsg ?cp 'sort-out-components!))      ;and construct a constraint model
        (lisp (unitmsg ?cp 'model!))))            ;of the circuit components using lisp code.
```

Figure 5.2. Production rules used in forward chaining mode in Solder to recognize a Ciruit from the structural description defined by Circuit Buff.

connected together using the connectivity information in the B1-ONWARDS unit. This process results in a complex network of constraints. The constraint network is constructed within the SOLDER knowledge base as a set of KEE units. A graphical representation of this constraint network is reproduced in Circuit Buff as shown in Figure 5.4(a).

In Figure 5.4(a) the constraint models for the Switch (SW1) and the Light Bulbs (L1, L2 & L3) have been compacted into 'black boxes'. The black boxes have the same inner structure as the Switch and Bulb of section 4.2, and the same number of external connections. In Circuit Buff black boxes that represent more complex constraint networks are referred to as *concoctions*, see section 6.1.5.

5.5. Initializing the Model

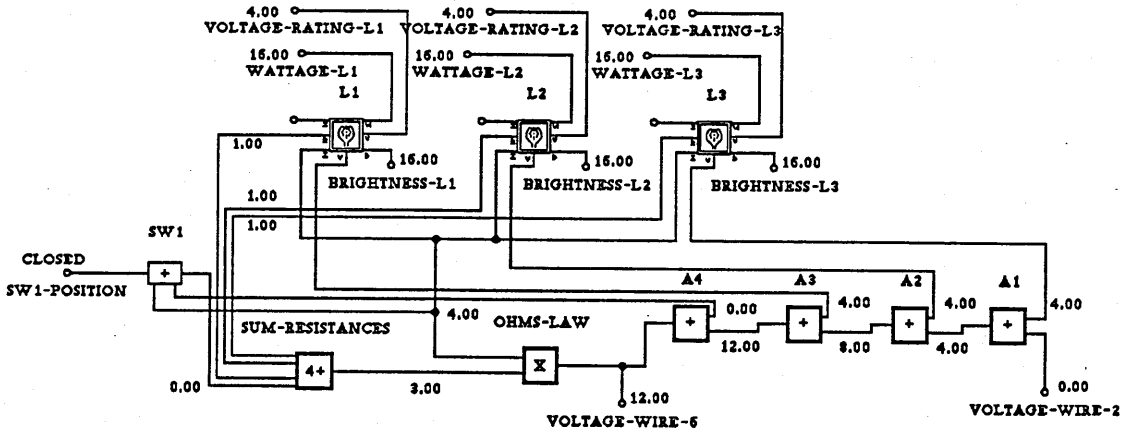
This step involves setting the known parameters of the circuit on the constraint model. For the Christmas tree light set each bulb is 16W and 4V. To set these values in the constraint model the KEE interface is used. The user clicks on the KEE unit representing the component (L1 in this example) and chooses NEW-INFO! from the resulting menu. The user is then prompted for the parameter and value (NOSTRUM's prompts `appear in this font`, the user's responses **`appear in this font`**):

```
Slot: Wattage
New-value: 16
```

Clicking on L1 again and choosing NEW-INFO! allows the user to specify the voltage:

```
Slot: Voltage
New-value: 4
```

This process is repeated for each light bulb. Finally the voltage of the battery and the state of the switch are entered using a similar procedure.



(a)

(Output) Facts in ALL-SYSTEMS-GO World

Primitive Facts:

(A WATTAGE OF L1 IS 16.00)
(A VOLTAGE-RATING OF L1 IS 4.00)
(AN INTENDED-CURRENT OF L1 IS 4.00)
(A RESISTANCE OF L1 IS 1.00)
(A WATTAGE OF L2 IS 16.00)
(A VOLTAGE-RATING OF L2 IS 4.00)
(AN INTENDED-CURRENT OF L2 IS 4.00)
(A RESISTANCE OF L2 IS 1.00)
(A WATTAGE OF L3 IS 16.00)
(A VOLTAGE-RATING OF L3 IS 4.00)
(AN INTENDED-CURRENT OF L3 IS 4.00)
(A RESISTANCE OF L3 IS 1.00)
(A SWITCH-STATE OF SW1 IS CLOSED)
(A RESISTANCE OF SW1 IS 0.00)
(A TOTAL-RESISTANCE OF B1-ONWARDS IS 3.00)
(A VOLTAGE OF WIRE-6 IS 12.00)
(A CURRENT OF B1-ONWARDS IS 4.00)
(A VOLTAGE-DROP OF SW1 IS 0.00)
(A VOLTAGE OF WIRE-5 IS 12.00)
(A BRIGHTNESS OF L1 IS 16.00)
(A VOLTAGE-DROP OF L1 IS 4.00)
(A VOLTAGE OF WIRE-4 IS 8.00)
(A BRIGHTNESS OF L2 IS 16.00)
(A VOLTAGE-DROP OF L2 IS 4.00)
(A VOLTAGE OF WIRE-3 IS 4.00)
(A BRIGHTNESS OF L3 IS 16.00)
(A VOLTAGE-DROP OF L3 IS 4.00)
(A VOLTAGE OF WIRE-2 IS 0.00)

Deduced Facts:
None

(b)

Figure 5.4. (a) The constraint network for the Christmas tree light set showing initialized values. (b) The same facts in (a) as they appear in the `all-systems-go` hypothetical world of the Solder knowledge base.

The last value entered enables all the constraints to complete, leaving the network fully specified. This completed network is depicted in Figure 5.4 (a).

At this point the model represents the state of an actual working Christmas tree light set. The values deduced by the constraint propagation correspond to a user's expectation of what the device should be doing. So far all the values set by the user and derived by the constraints are established as facts within the `all-systems-go` hypothetical world. These facts are shown in Figure 5.4 (b).

This stage corresponds to the circled 1 in Figure 3.6 on page 55.

5.6. Describing the Symptom

The initialization procedure of the previous section left all the lights shining with a brightness of 16Watts (refer to Figure 5.4). In a broken Christmas tree light set this may not be so. For this example I assume that bulb L1 is observed not to be shining, ie. it has a brightness of zero. To make this observation the user clicks on the L1 unit in the KEE knowledge base. From the resulting menu the `OBSERVE!` item is chosen. The user is then prompted for the parameter and observed value:

```
Slot: BRIGHTNESS
New Value: 0
```

Figure 5.5 shows an actual screen snapshot of this process. This point corresponds to the circled 2 in Figure 3.6 on page 55.

5.7. The constraint model generates hypotheses

The new observation initiates more constraint propagation to propose new values for parameters in the constraint network. The proposed values will have different consequences for the circuit. To manage the different scenarios NOSTRUM makes use of the KEE Worlds facility. KEE worlds represent a series of hypothetical scenarios in which different facts are assumed to hold. The worlds form a hierarchy: in NOSTRUM the root of this hierarchy is called the `ALL-SYSTEMS-GO`

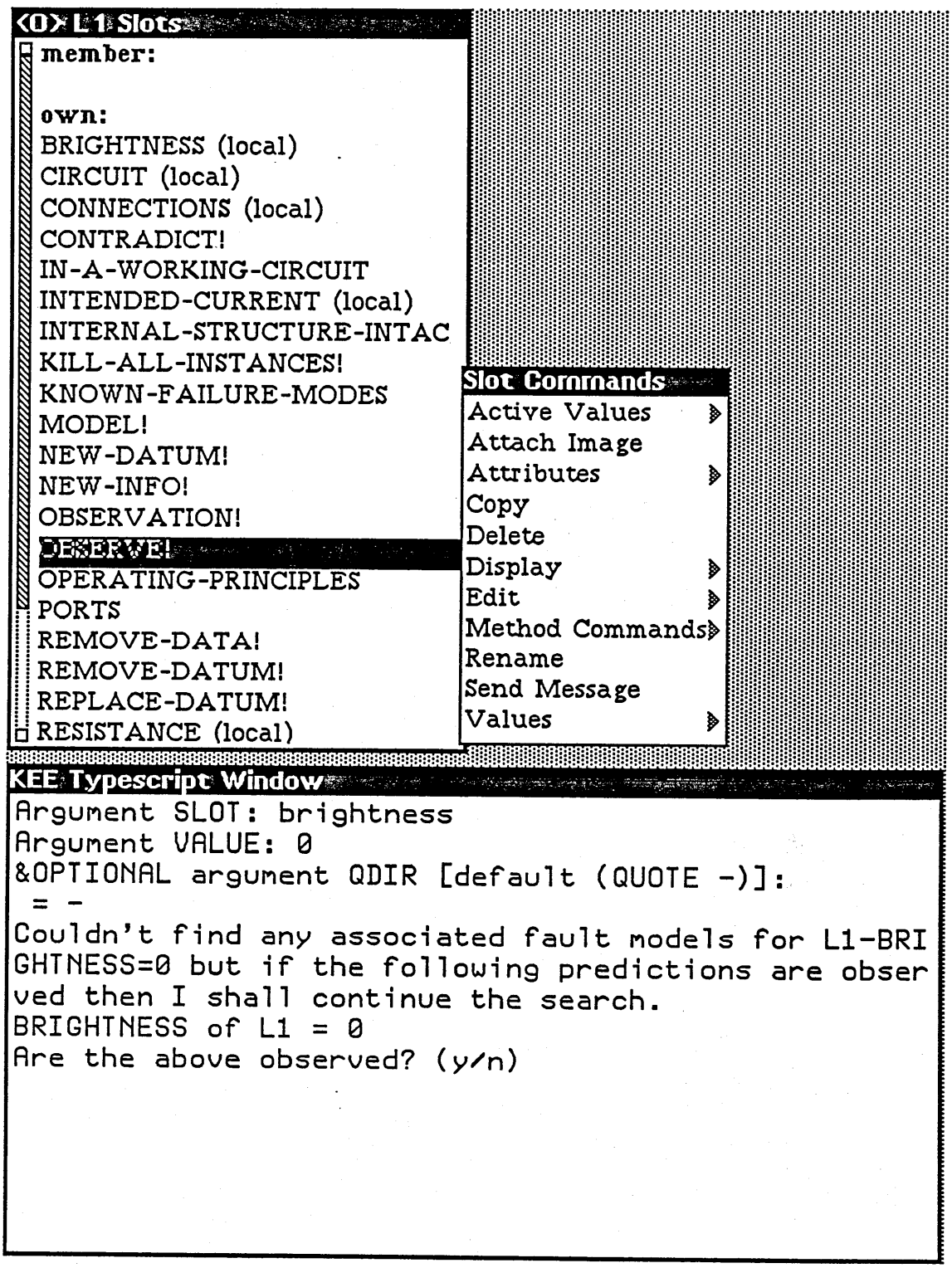


Figure 5.5. Showing how the symptom observation that L1 is off is described to Nostrum, through the KEE interface to the Solder Knowledgebase.

world. In this world all devices are assumed to be working perfectly and the facts in the world represent the parameters and values of a fully functional device. Offspring of the root world inherit all the facts of the

parent world except for those that are explicitly overridden. Background facts are true in all worlds. Background facts are facts that haven't been given a world dependency, such as the facts representing the structure[1] of the device and class - subclass links.

When an observation is made about a value in the constraint model that is different to that expected NOSTRUM creates a child of the ALL-SYSTEMS-GO world. The child world inherits all of the facts of the ALL-SYSTEMS-GO and contains the observation. This world is called OBSERVATION-*nnn*, where the *nnn* is an arbitrary number generated[2] to make sure the world has a unique name. The new world represents the device in the observed state.

NOSTRUM then creates a child of the OBSERVATION-nnn world in which it asserts the observation onto the relevant node in the constraint network - in this case the Brightness slot of L1. The new world is named after this assertion: L1-Brightness=0-nnn. Again the world inherits all the facts of its parent, but the changed value and any consequences of it derived from constraint propagation override the inherited facts.

There are no consequences for the observation that the brightness of L1 was zero because the original value (16W) wasn't used to derive any other values. So NOSTRUM simply produces the L1-Brightness=0-nnn world as a child of the OBSERVATION-nnn world with the new fact that the brightness of L1 is zero.

---

[1]The implication here is that NOSTRUM cannot deal with faults arising from a break in the structure of a device. This is not strictly true, for instance if a lever breaks then force will not be propagated through it. Nostrum is capable of tracing that, isolating the parts of the device still working. Further discussion of this issue is reserved for chapter 8, but the interested reader is referred to Davis 1984.

[2]Hackers note: Gensymed in lisp jargon.

Referring back to Figure 4.6, on page 64, which shows the constraint network for a light bulb, it can be seen that the `brightness` parameter is related to the `resistance` and `current` parameters through the `*^2` constraint. This constraint represents the equation $B=I^2R$, so in order that the brightness B is zero either the current I or the resistance R must be zero. To deduce these values the constraint fires backwards to propose that either I=zero or R=0. Exactly how this is implemented is explained in section 6.2.1. This point corresponds to the circled 3 of Figure 3.6.

The values predicted for current and resistance are treated as observations, and one world is created for each, this time as a child of the `L1-Brightness=0-nnn` world (see Figure 5.6).

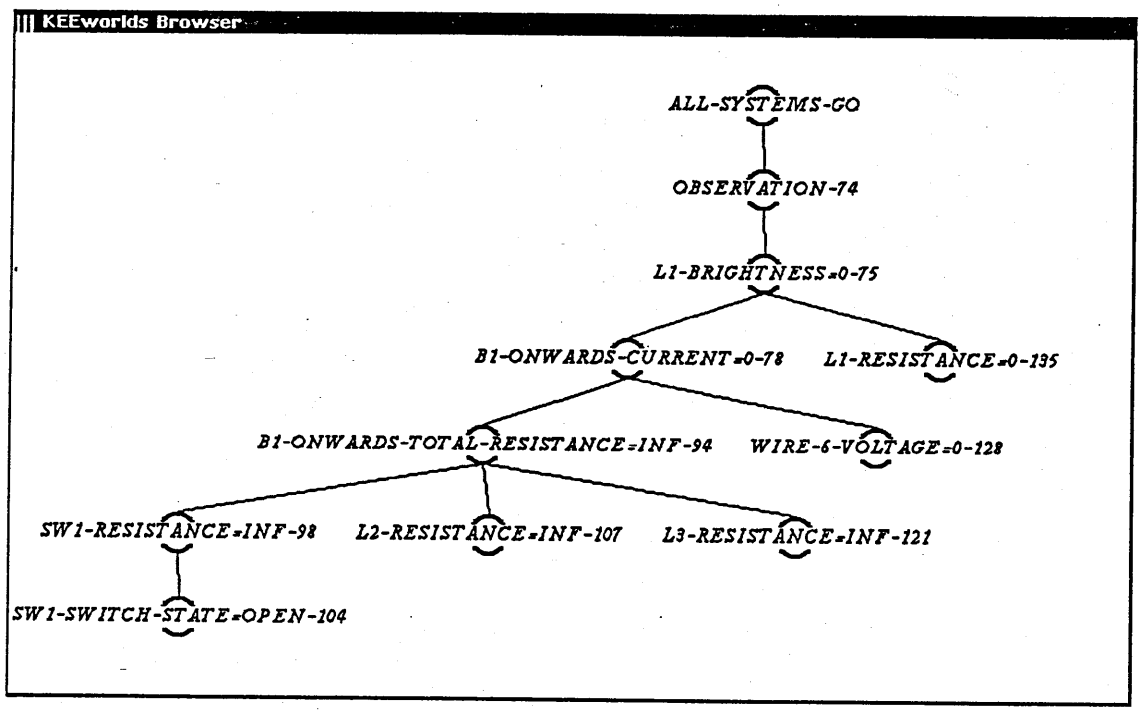


Figure 5.6 . Hypothetical worlds generated by the constraint mechanism to explain why the brightness of `L1` is zero. The name of each world represents the current hypothesis in that world.

In fact the constraint mechanism behaves like a depth first search and all the consequences of the current being zero are explored before those of the resistance being zero. So NOSTRUM first creates the `B1-ONWARDS-CURRENT=0-nnn` world, representing a scenario in which the

circuit has no current flow. The consequences of this hypothesis are computed via causal constraint propagation within the world (for details see section 6.2.1.1). The constraint propagation uses data dependencies to see which values were deduced from the previous value of the current.

This point corresponds to the circled 4 of Figure 3.6.

5.8. NOSTRUM requests the user to perform tests.

The names of the worlds generated by the search strategy represent the shift of the current hypothesis. Currently this is implemented as a depth first search through the constraint network. At each node NOSTRUM calculates the consequences that the hypothesis would imply. If there are any implications for observable or operable nodes then NOSTRUM and asks the user whether the predicted values are observed. Figure 5.7 shows an example of this.in which the current hypothesis is that the resistance of the circuit is infinite. The consequences of the hypothesis are displayed in the KEE Typescript window, and the user is asked if the predictions are observed. This point corresponds to the circled 5 of Figure 3.6.

An answer of YES continues the search, because the observation backs up the current hypothesis. An answer of NO stops the current hypothesis and forces NOSTRUM to continue its depth first search from a step higher in the tree, in search of a new hypothesis.

This questioning strategy is very 'dumb', but originates from one of the original goals of NOSTRUM which was to make a diagnostician that could reason in a new situation, ie in the absence of experiential knowledge.

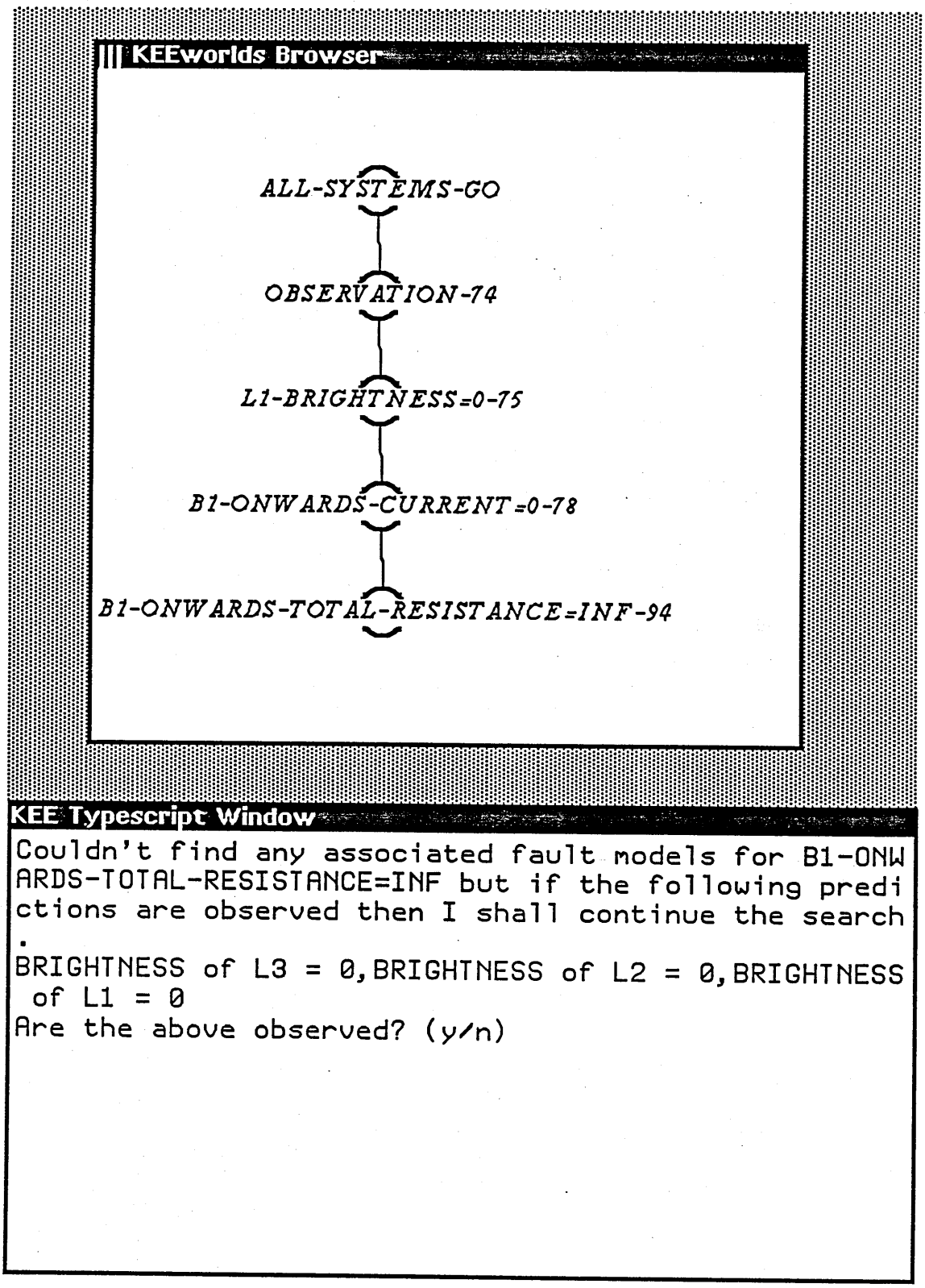


Figure 5.7. Screen snapshot morsel of the KEE interface to the Solder knowledge base. The KEE worlds browser (top) shows the shift of the current hypothesis, the most recent being at the bottom. Below this is the KEE typescript window in which the user is being asked if the predicted consequences of the current hypothesis are being observed.

To make the questions asked by NOSTRUM more 'intelligent' expert knowledge about failure modes can be associated with nodes of the constraint network. By doing this it is possible to take greater advantage of the symptom description to lead more directly to a fault.

To give an example: the symptom is 'flickering bulbs'. 'Expert' knowledge, that flickering is associated with loose connections, is available. In its search through the constraint network NOSTRUM would come across hypotheses that suggested flickering battery voltage, and flickering switch contacts. The expert would immediately dismiss the flickering-battery idea because batteries are well-known for their steady-state behaviour. The much more familiar symptom of 'dirty switch' contacts would attract the expert to the latter hypothesis.

A second example is that the battery could have the associated knowledge that its charge decays slowly. If, when the lights go out, the symptom describes a gradual dimming then the first hypothesis to come to mind is likely to be 'flat battery' rather than 'blown filament'.

In this way experiential knowledge can be used to make different parts of the network sensitive to different symptoms. Detailed examples of how the expert knowledge improves the search strategy are described in chapter 7.

Some heuristic knowledge has been incorporated into the Christmas tree light example. This knowledge is represented as simple associations between known fault cases and faulty parameter values. A simple example is that a fused bulb has an infinite resistance. This fact is represented by the simple association list: `(Resistance INF "Fused bulb")` which is stored in a slot that represents the class of light bulbs (for implementation details see section 6.2.2). Such knowledge enables NOSTRUM to put forward fault hypotheses to the user, and a screen snapshot of the present example is shown in Figure 5.8.

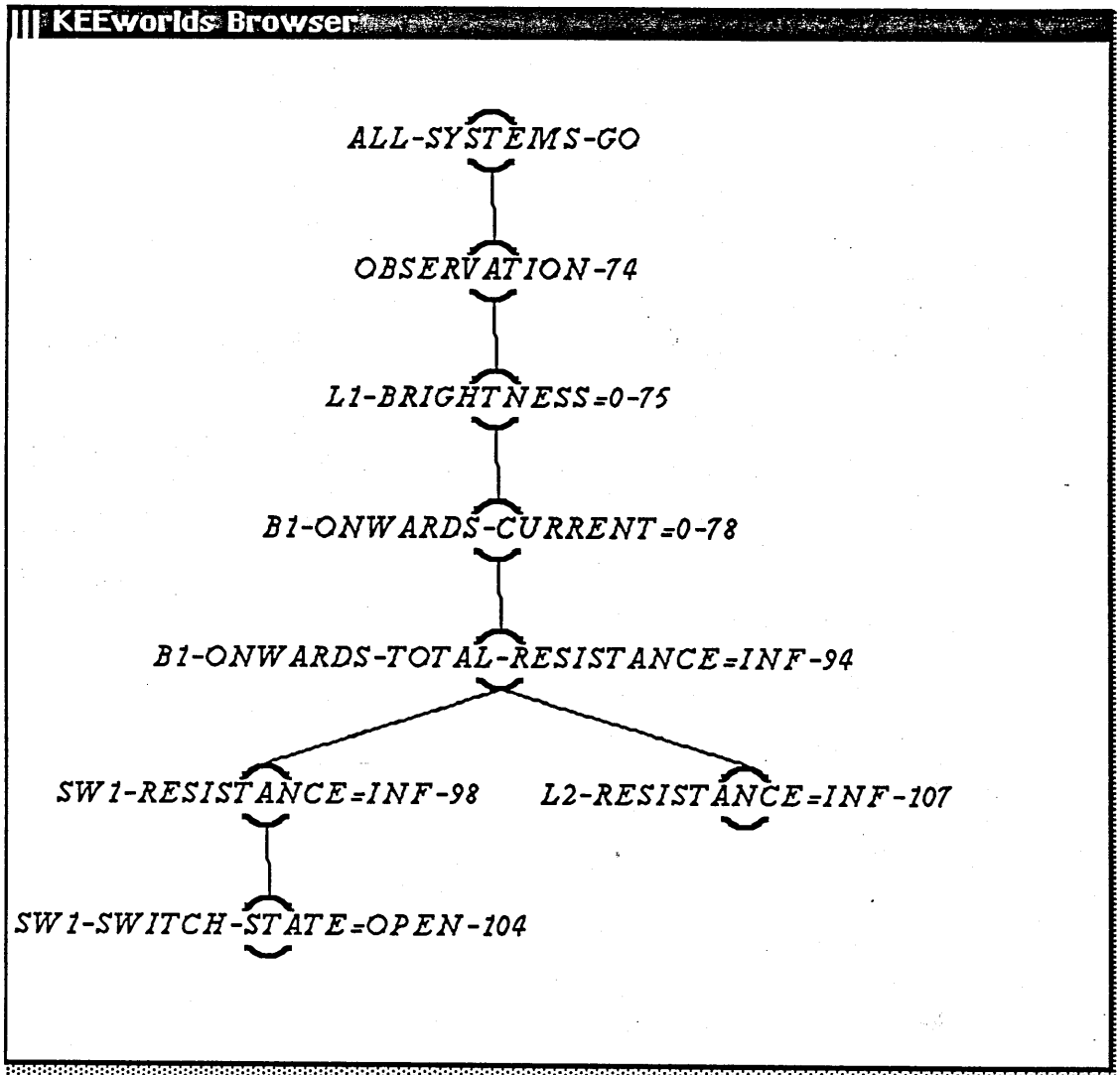


Figure 5.8. Nostrum suggests that the Light L2 has fused using experiential knowledge. The typescript window shows a morsel of user dialogue with NOSTRUM. The user has just confirmed that they are sure that the switch is closed, after which NOSTRUM suggests a fused bulb.

5.9. A Repair action is suggested.

When finally the user replies YES to one of the questions about the predicted value for a node NOSTRUM has made a diagnosis. If the node value can be restored by replacing a part, or by resetting a value, then hopefully the device is fixed. If not it is as though the user had replied NO to the question regarding the value of the node, and so the constraint propagation continues. This point corresponds to the circled 8 of Figure 3.6.

The process of diagnosing the Christmas tree light set as described in this chapter is summarized in Figure 5.9.

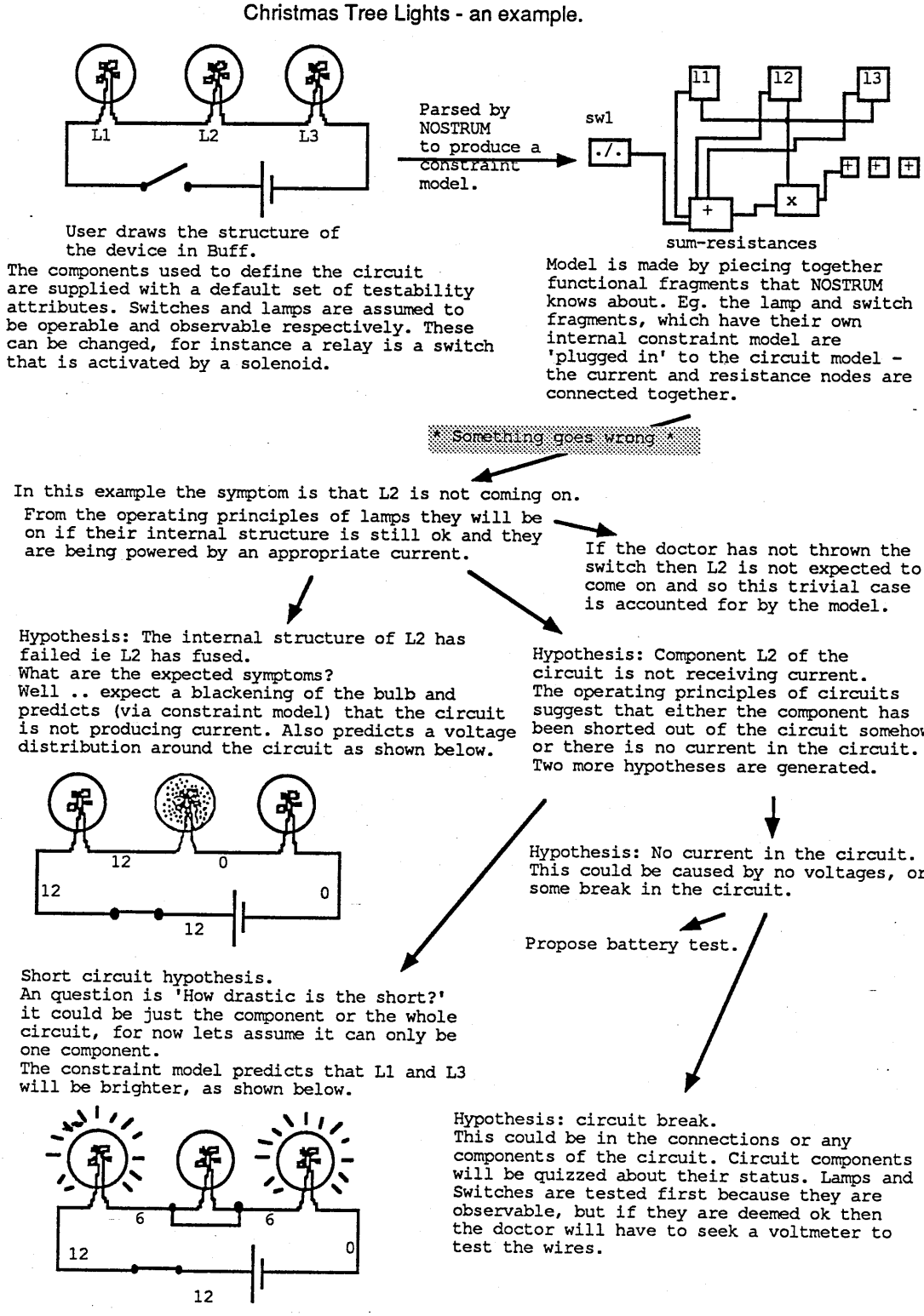


Figure 5.9. A summary of the diagnosis of the Christmas Tree light set.

Chapter Six

Implementing NOSTRUM

The previous chapter has described how NOSTRUM takes a description of a device, models it and performs a diagnosis. This chapter explains the software implementation that facilitates this.

NOSTRUM is implemented on a Symbolics 3600 Lisp machine running KEE™[1] (Knowledge Engineering Environment) but an early version was constructed on a Unisys Explorer. NOSTRUM is comprised of several components which are summarized below:

(1) A Graphical interface for defining device structure. This is called Circuit Buff.

(2) A hierarchy of objects representing device units. This is implemented as the Solder knowledge base in KEE.

(3) Pattern recognition rules for identifying circuit components (This was described in section 5.3).

(4) A constraint propagation mechanism that drives the simulation and hypothesis generation procedures.

[Note: Circuit Buff is not simply a front-end to KEE. The constraint propagation mechanism was originally implemented within Circuit Buff. The need for a hypothetical worlds system necessitated duplicating the constraint propagator using the KEE object system.]

These systems are discussed in the following sections.

---

[1]KEE is a registered trademark of Intellicorp Ltd, Mountain View, California.

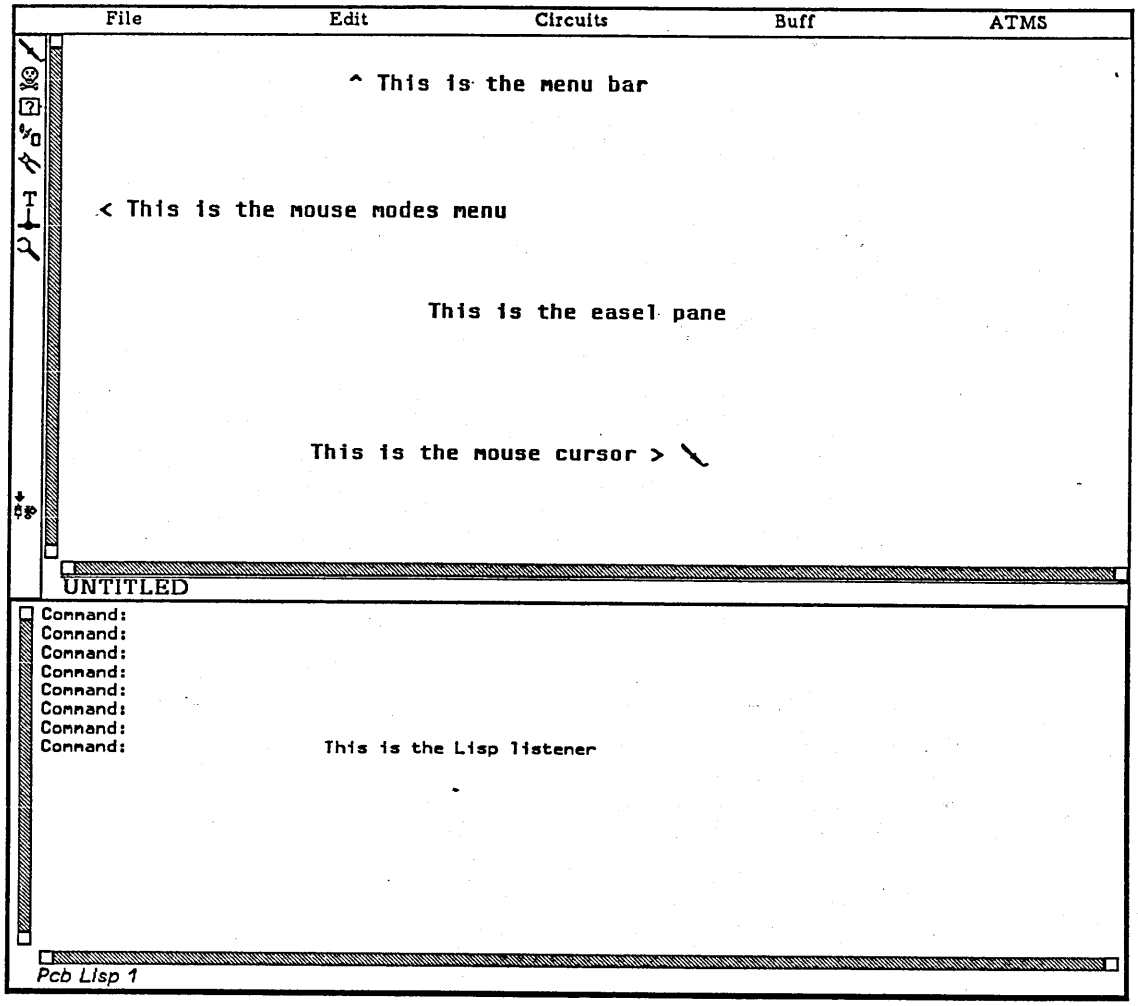


Figure 6.1. The various panes and menus of Circuit Buff.

6.1. Circuit Buff

Circuit Buff has two main functions in NOSTRUM. Firstly it is a graphical tool for defining the structure of a device. Secondly it is used to illustrate the constraint propagation mechanism that NOSTRUM uses to do diagnosis.

In the following description of Circuit Buff I show how the user constructs the Christmas tree light set using the mouse and menu interface. Circuit Buff is driven by a 'three fingered' mouse and in the next section left, middle and right clicks refer to a press and release of the corresponding mouse button.

6.1.1. Using Circuit Buff to Draw the Christmas Tree light set

Figure 6.1 shows the screen layout of Circuit Buff. The screen of Circuit Buff is split into several panes. At the top is the menu bar which provides a series of menus for performing file operations and editing commands. The centre of the screen is dominated by the *Easel pane*. This is where the structure of the device is drawn. To the left of the easel pane is the *Mouse Modes* menu. This menu contains a series of icons which are used to set the current mode of Circuit Buff. The available modes are summarized in Figure 6.2. Beneath the easel pane is a *lisp listener* whose presence is to aid in the debugging and development of the tool.

To begin drawing the device the user chooses the 'New Component' icon from the *Mouse Modes* menu. A menu of currently defined icons appears as shown in Figure 6.3.

The user selects a component from the menu and positions it on the easel pane by clicking on the left button of the mouse. The user is then prompted for the name of the component in a pop-up window (Figure 6.4).

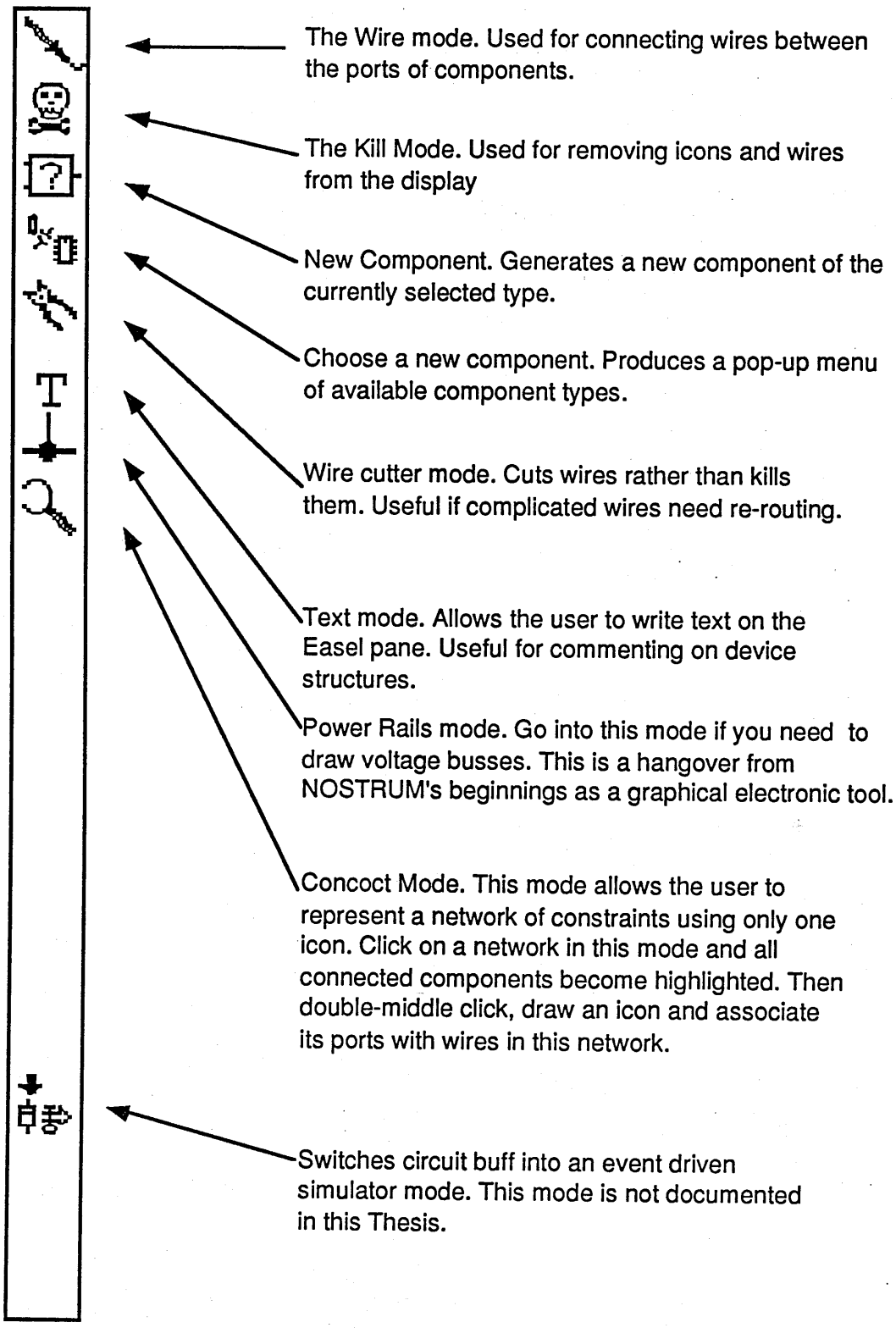


Figure 6.2. Expanded view of the Mouse Modes menu.

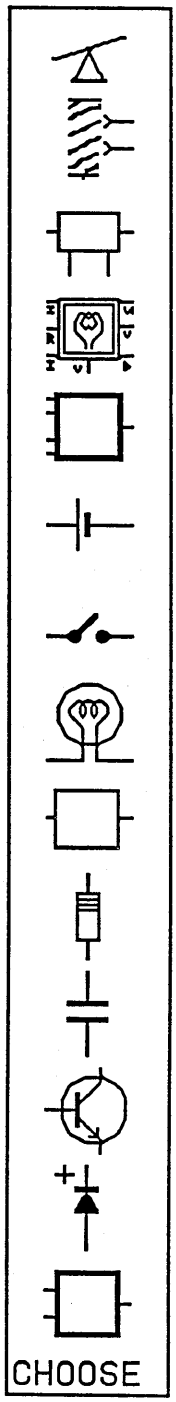


Figure 6.3. The pop up menu of currently defined component types. From the top these are Beam, Spring, Switch Constraint, Light Bulb Concoction, Multi input adder, Battery, Switch, Lamp, Two input converter, Resistor, Capacitor, NPN transistor, Diode and Simple Constraint.

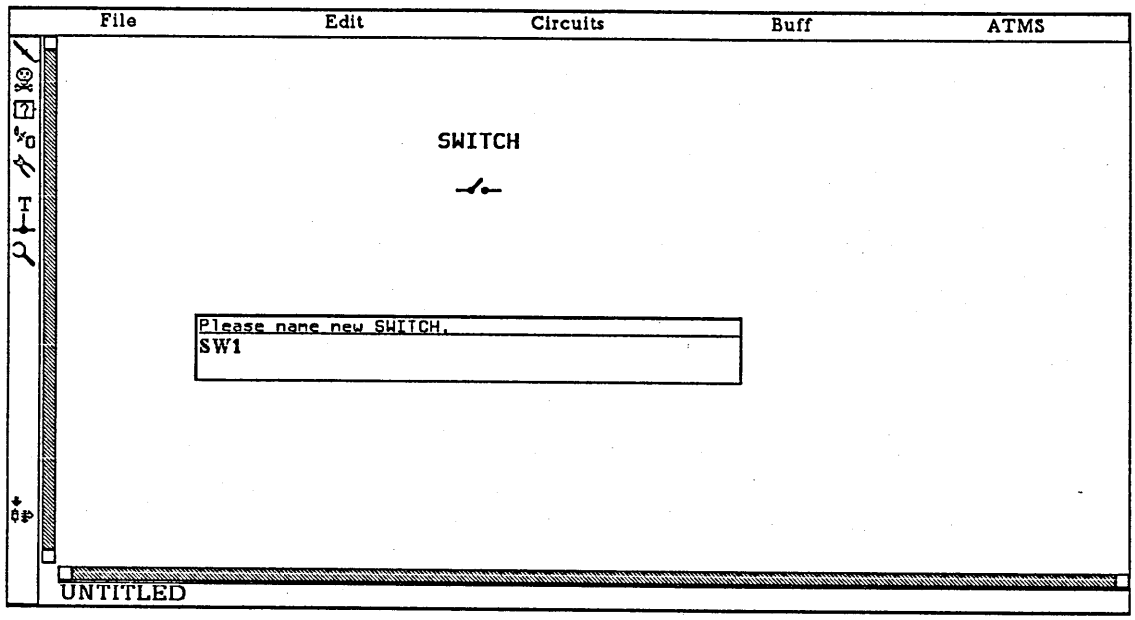


Figure 6.4. Setting the new components' name.

The name is printed above the icon. While the mouse is over the icon it becomes highlighted indicating that mouse clicks perform operations on that icon. The orientation of the icon can be rotated through 90 degrees with a middle click, and repositioned with a double middle click. The position of the icon's name is moved with a double left click.

The user repeats this process until the desired icons appear on the easel pane. Each icon has an associated set of *ports*. It is through these ports that the components communicate to each other. In Circuit Buff the ports are usually at the edges of the icon at the end of lines that emanate from the graphic. Components are connected to each other with wires which begin and end at the ports. To start a wire the user must first be in wire mode, which is accessed through the 'Wire Mode' icon in the mouse modes menu. The mouse cursor changes to the image of a soldering iron and the mouse who line displays the commands available in this mode. The tip of the soldering iron is moved over one of the ports of an icon which becomes highlighted. Left clicking then creates a wire connected to the port. As the user moves the mouse the end of the wire follows the tip of the mouse cursor with two orthogonal lines. The wire can be made to 'bend around' other icons on the screen

by fixing 'kinks' in it using the middle button. These kinks can be removed by a double-left click. To fix the end of the wire the tip of the mouse cursor is moved over the desired port and the user clicks on the left mouse button. See Figure 6.5.

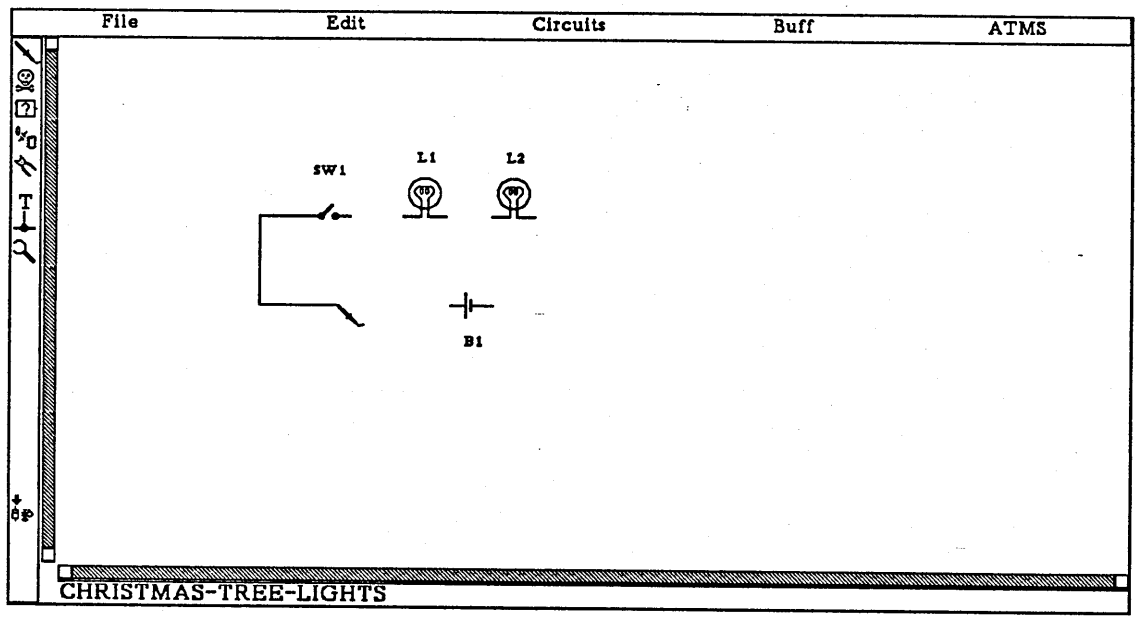


Figure 6.5. Drawing Wires in Circuit Buff.

Users' mistakes are well accounted for in Circuit Buff. All operations have a menu on the right button which gives the user the chance to abort the current operation. There is also a 'kill mode' in which the mouse cursor changes to a skull and crossbones. In this mode any highlighted object under the mouse is erased when the user left clicks on the mouse button.

In this way the structure of the device is very quickly built up. The next section describes how the user can define their own icons and how constraint networks can be constructed with Circuit Buff.

6.1.2. Defining your own icons

If the icon representing the desired class of device component is not present in the component types menu then new ones can be defined. To do this the user chooses 'Create Component types' from the 'Buff' menu in the menu bar. Buff then prompts the user for the device name and the

type of device to create. There are two types of devices: `electronic-devices` and `black-boxes`. The electronic device type will simply generate an icon with a set of ports. The black-box type is a more powerful version of electronic-device that contains extra properties for performing constraint propagation. If the user chooses the black-box type they are also prompted for a list of the constraint completion functions, these are explained in section 6.2.1.

When the name and type of component have been specified an icon editing window appears. An example of this window is shown in Figure 6.6. The window presents a 32 by 32 grid in which each element corresponds to a pixel of the icon. Pixels can be set or cleared using mouse clicks. An actual size version of the icon appears to the right of the grid.

The user can create ports for the icon by double left clicking on a grid square and entering the port name. Ports appear as crosses within the grid squares on the icon editor window.

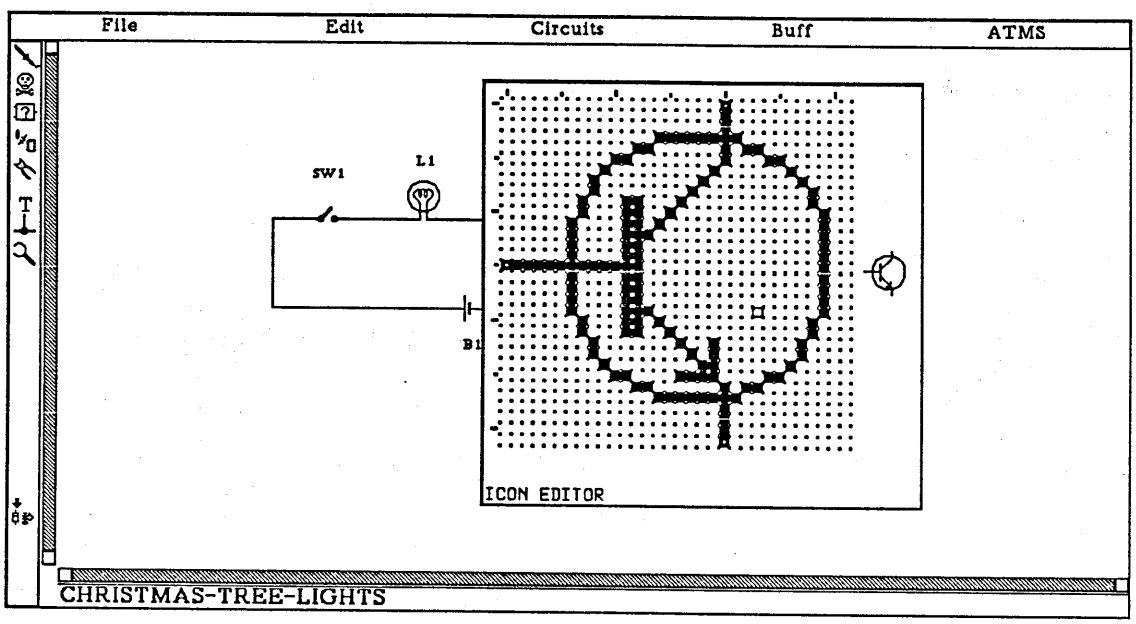


Figure 6.6. Showing how the icon editor is used to define a new icon in Circuit Buff.

To finish editing the icon the user right clicks and chooses exit from the menu. Circuit Buff then saves the newly defined icon and adds it to

the component types menu. The user can now proceed to use instances of the icon in their circuits.

6.1.3. Behind the Scenes

As the user constructs a diagram in Circuit Buff they are really building up a set of *lisp objects*. Circuit Buff uses the Symbolics Flavors system [Weinreb and Moon 1981] to represent these objects. A Flavor defines a class of objects. Each object has a set of *instance variables* and *methods* that define its character. The values of the instance variables define the current state of the object and the state can be changed by *sending messages* to the object. Sending a message to the object causes one of the methods to be run. The methods are pieces of lisp code that can change the values of the instance variables, but also can run arbritrary lisp code.

In Circuit Buff all the component types inherit properties of the `electronic-device` flavor. This flavor provides instance variables for the name of the device, the ports of the device, the connections to wires, the array that defines the icon and so on. The flavor also provides methods for moving the device around the screen, naming the device, connecting it to wires etc.

Each of the component types in Circuit Buff is a subclass of the `electronic-device` flavor. The component types inherit all the instance variables of the parent, but have their own unique icons and ports. When the user defines a new component type in Circuit Buff (section 6.1.2) they are creating a new subclass of `electronic-device` and setting its icon and ports instance variables.

Wires have their own set of instance variables and methods because they represent a quite distinct type of object. The hierarchy of the object or flavor classes in Circuit Buff is shown in Figure 6.7. Each icon and wire created in Circuit Buff becomes an instance at the right hand side of this hierarchy.

Some of the component types that have a constraint behaviour inherit extra instance variables and methods from the BB, and Black-Box flavors. A description of these flavors and the constraint propagation mechanism in Circuit Buff is described in section 6.2.1.

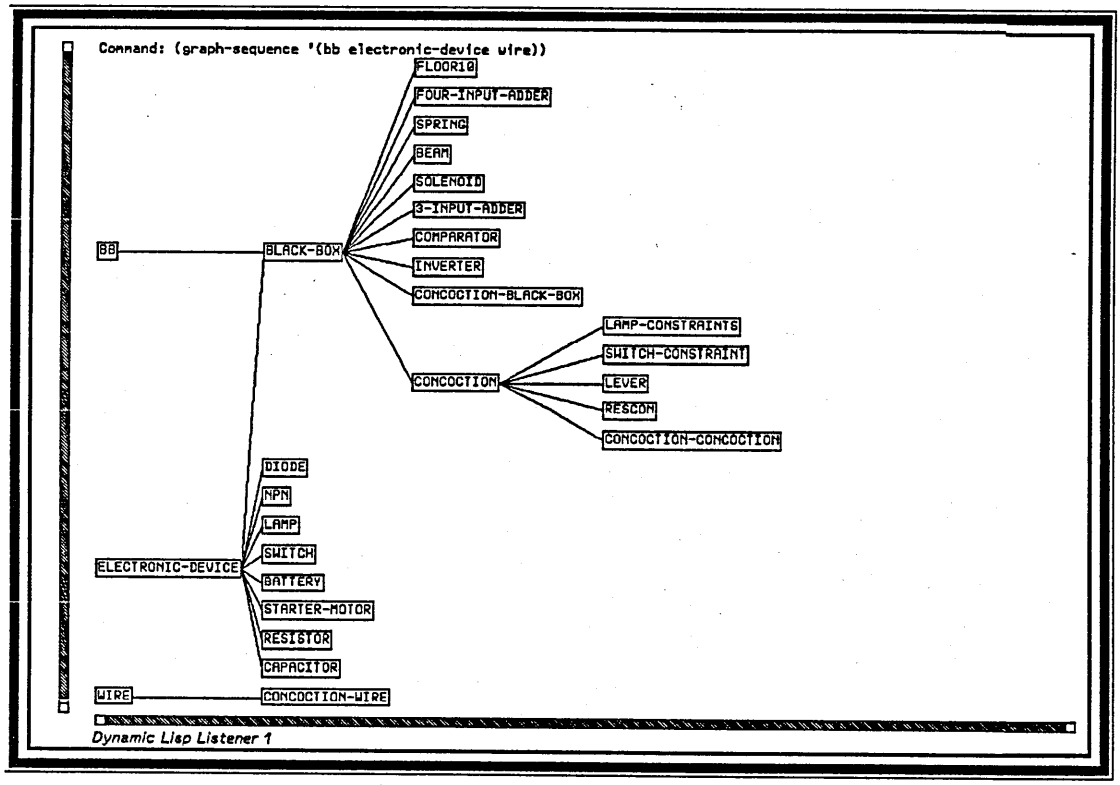


Figure 6.7. The hierarchy of Flavors in Circuit Buff. At the left of the figure are three objects which form roots of the hierarchy. BB provides messages for constraint propagation, Electronic-Device supplies methods for drawing, and moving the icons as well as instance variables that list the connections of the device and its rating. The Wire flavor supplies messages for drawing wires, connecting wires with electronic-devices and propagating constraint values.

6.1.4. Defining Constraints

The component types that are subclasses of the black-box flavor inherit methods for performing constraint propagation. In Circuit Buff the icons represent *constraints* and the wires represent *nodes* of a constraint network.

Node values are stored on the data instance variable of wires and the functions for deriving new node values are stored with the icon in the function-map instance variable. To illustrate the constraint

propagation mechanism I use the simple example of an adder as shown in Figure 6.8.

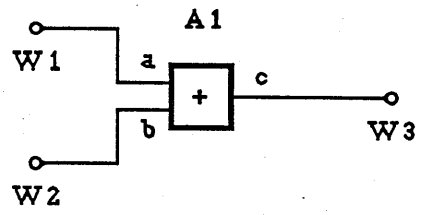


Figure 6.8. A simple adder constraint.

The icon `A1` is an instance of the `Black-Box` flavor (see Figure 6.7), which is connected to the three wires `W1-3` through the ports `a`, `b` and `c` respectively. The `function-map` instance variable of A1 contains three *function templates* for finding the value at one of the ports when values are known at the other ports. These functions are assigned to lisp variables by the following expressions:

```
(defvar bb-c=a+b '(+ a b))
(defvar bb-a=c-b '(- c b))
(defvar bb-b=c-a '(- c a))
```

The function templates are given names that reflect what they do, and are prefixed with 'bb-' which stands for black-box. The user can define any lisp functions to be used by the constraint propagator, they are not restricted to just number functions. For instance the switch constraint in the Christmas Tree light set uses symbol functions to map between the switch state and the resistance of the switch:

```
(defvar bb-switch-resistance-from-switch-state
  '(case switch-state
       (closed 0)
       (open 'inf)))
```

The constraint functions of a black box can be edited using the Circuit Buff interface. To do this the user right clicks on the icon and chooses Edit Properties from the pop-up menu. The resulting menu is shown in Figure 6.9.

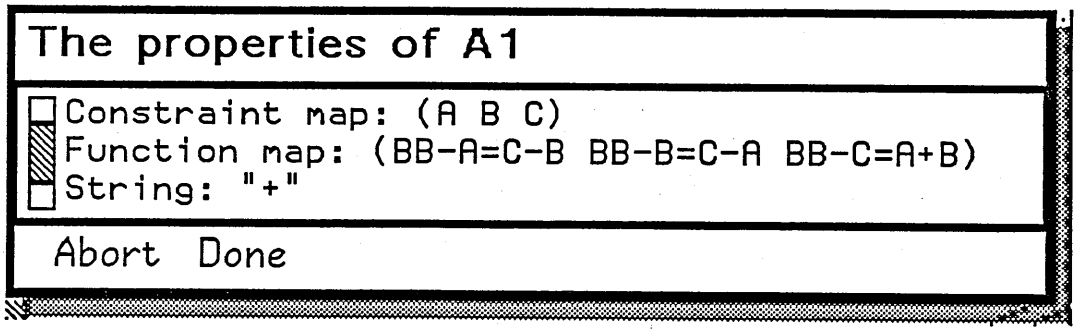


Figure 6.9. Pop up menu allowing the user to edit the functions of a black box.

The only limit on the number of ports a black box may have is the physical limit that will fit around the icon, and there is no distinction between input and output ports.

To set values on the wires in Circuit Buff the user middle clicks on the wire. and enters a value into the ensuing pop-up menu (Figure 6.10). In this example the user has clicks on the wire `W1` and enters the value 3. This procedure sends a message to the wire object telling it that there is a new value and that the value is 3. The message forces the `new-datum-on-wire` method of the wire to run. This method informs each `black-box` that is connected to the wire that a new value has been received. This is implemented as another message: `new-datum-at-port`. In this case only the adder `A1` receives the message.

The `new-datum-at-port` message forces the adder to check if there is enough information at the ports to complete the constraint. There is enough information if all but one of the wires connected to a constraint have values. At this point only the value for `W1` is specified and so constraint propagation halts.

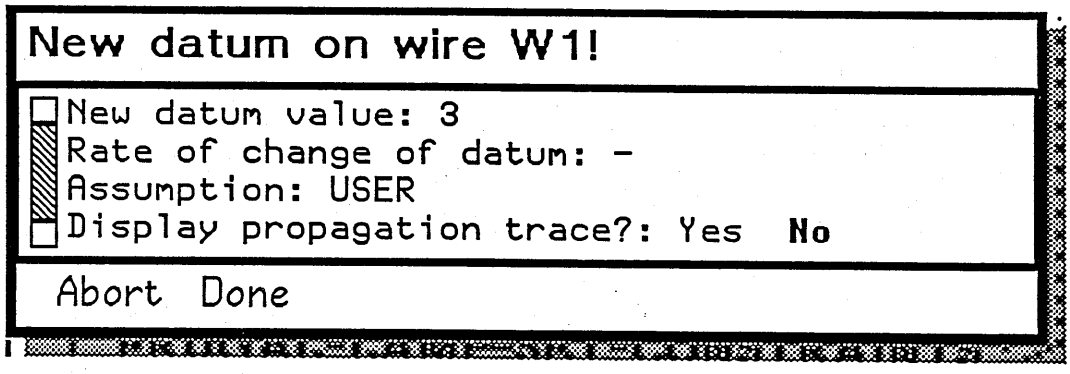


Figure 6.10. The pop-up menu allowing the user to specify a value on a wire. The second item in the menu prompts for the rate of change of the observed value, this is an experimental feature of Circuit Buff which is not discussed in this thesis. The value of the assumption option is used by the data dependency records to guide constraint propagation (see sections 6.2.1.). The final option allows the user to request a printed trace of the constraint propagation, this is used for debugging.

The user then middle clicks on `W2` and enters the value 4. As before this procedure sends a message to the `W2` object, which passes the value on to `A1`. This time the constraint in `A1` can complete to derive a value for `W3` at port `c`. To do this `A1` looks up the function for deducing the value at port `c` from the values at `a` and `b` in the `function-map`. The values for `a` and `b` are substituted in the function template and the resulting form is evaluated. The result is then forwarded to `W3` using the `new-datum-on-wire` message. As there are no other constraints attached to `W3` constraint propagation comes to an end.

To summarize: Constraint propagation is implemented by message passing. On receiving a value, a wire forwards the value to connected black-boxes. For each port of the black-box the constraint tries to complete. If it can, the new value is sent to the wire connected to the port. In turn the wire forwards the new value to other black boxes. The constraint propagation behaves in a depth-first manner.

6.1.5. Concoctions

Using the Circuit Buff interface networks of constraints can be simply constructed. The behaviour of each element of the network is

specified by defining the `function-map` instance variable and the function templates.

Networks of constraints can be condensed so that their behaviour is represented by a single icon. In Circuit Buff such icons are referred to as *concoctions*.

To make a concoction in Circuit Buff the desired constraint network is drawn and the function templates defined. Using the concoction tool the required elements of the network are selected and the icon editor window appears. The user draws an icon that represents the network and sets the ports. Finally the user has to associate the ports with some of the wires of the constraint network.

When this is done the concoction definition is saved to disk and the icon is added to the component types menu. Now each time the user uses the new icon they are really instantiating another copy of the original constraint network. The concocted icon can be connected to other components and constraints in the usual way, and concocted icons can themselves be concocted. In this way the concoctions provide a method of abstracting the details of component interactions away from the operation of the device. Circuit Buff allows the user to 'Zoom in' to focus on the inner structure of a constraint network and the current values on nodes.

Concoctions are implemented as flavors and appear in the object hierarchy of Figure 6.7. The constraint networks which define the operating principles of lamps and switches have been defined as concoctions and also appear in Figure 5.4(a) on page 76.

6.2. The Solder Knowledge Base

The Solder Knowledge Base contains a similar hierarchy of objects to the flavors of Circuit Buff, but by implementing it in KEE there is access to a rule interpreter and a hypothetical worlds system. Figure 6.11 shows a graph of the solder knowledge base. Each element of the graph is a KEE frame or object containing various slots. The slots of each object are inherited (from left to right) to their offspring via the class - subclass (solid lines) or class - instance (dashed lines) links.

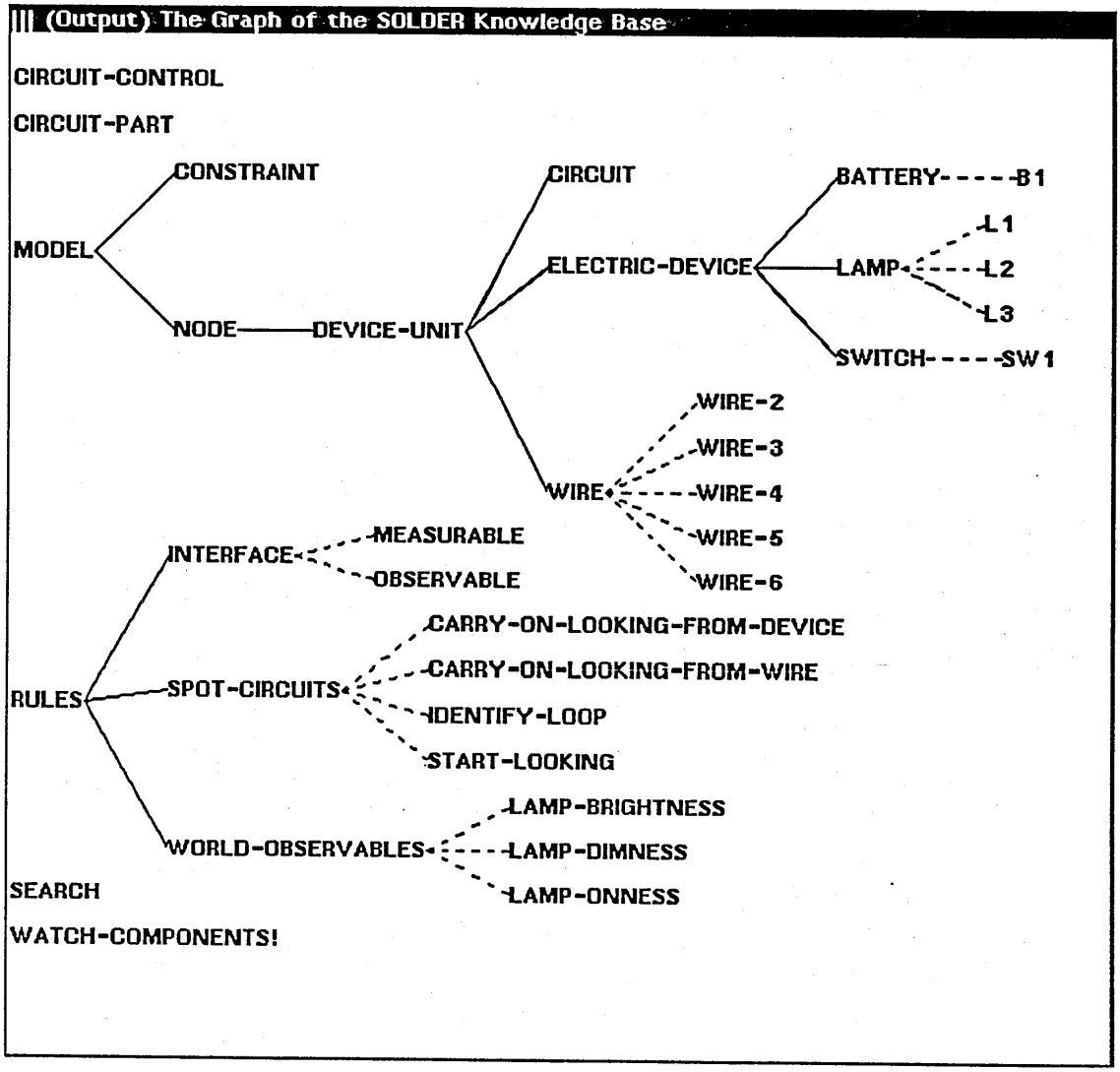


Figure 6.11. A graph of the Solder knowledge base in KEE. Solid lines represent class-subclass links, dashed lines represent class-instance links. The components of the Christmas Tree Light set are shown as leaf nodes beneath `device-unit`.

When the structure of a device has been completely defined in Circuit Buff a menu in that system allows the user to copy the lisp structures created there into the Solder Knowledge Base. The objects are copied to the corresponding classes, thus wires are instantiated as wire units, lamps as lamp units etc. Figure 6.11 shows the solder knowledge base with the components of the Christmas tree light set (Figure 4.4) transferred in this manner.

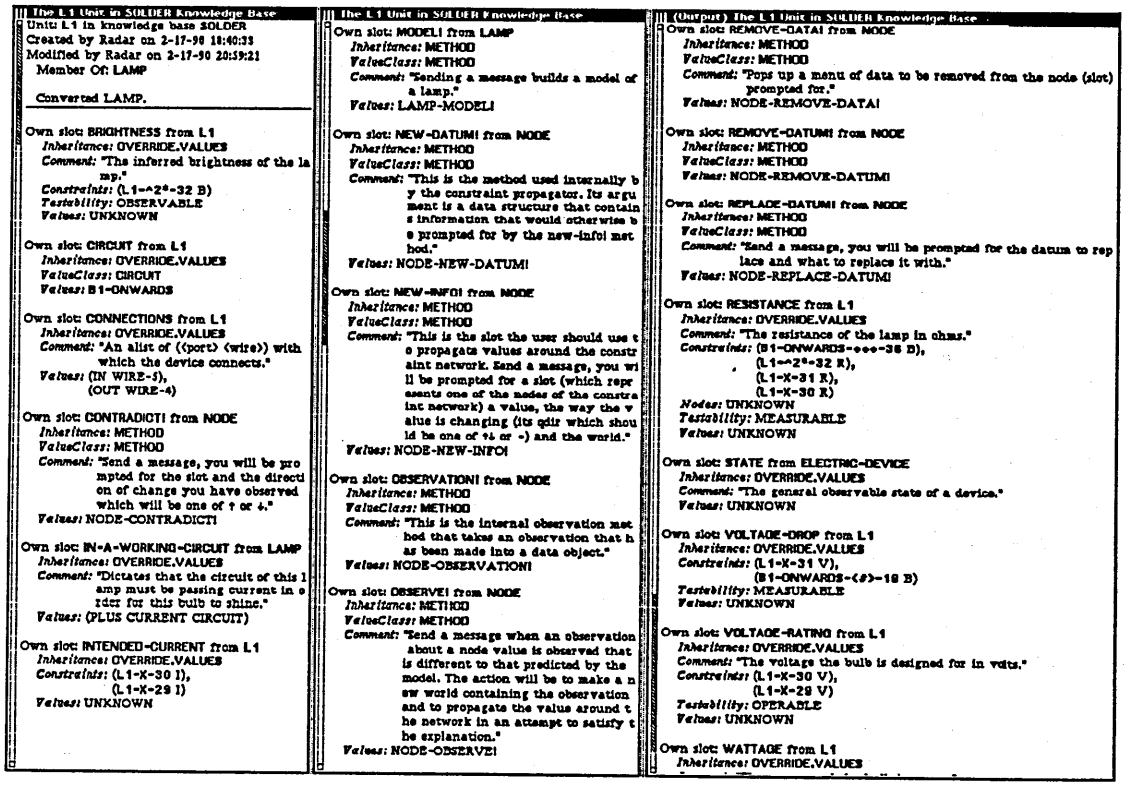


Figure 6.12. A detailed look at the slots in the KEE frame representing the lamp unit L1. Each slot represents an attribute of the object and each slot has various *facets* such as Comment, Inheritance and Value. The values of the `Wattage` and `Voltage-Rating` slots can be specified by the user. The constraint model of each lamp places the constraints shown in Figure 5.3 between the corresponding slots of this unit and the current slot of the frame representing the circuit. Each of the slots in this unit that forms a node in the constraint network has a `CONSTRAINTS` facet showing the constraint KEE unit to which it connects.

Components of the Christmas tree light set become indirect children of the `device-unit` class. These units contain *slots* that represent the parameters of the components. Figure 6.12 shows the `device-unit` for the lamp instance `L1`. The slots `Brightness`, `Intended-Current`, `Resistance`, `Voltage-Drop`, `Voltage-Rating` and `Wattage` are the

parameters of the lamp that are used to represent its operating principles. Notice that `current` is absent from this list because it is a parameter of the `circuit` device unit. The `L1` unit also has a `circuit` slot which represents the circuits to which it belongs. The value of this slot is unknown when the device is copied from Circuit Buff, but is filled out when the production rules identify a circuit.

Figure 6.12 also shows the `connections` slot of L1. The value of this slot is set when the device is copied from Circuit Buff. It contains an association list of the ports and wires to which the component connects. This slot is used by the production rules to identify a circuit loop.

Section 5.3 described how the rules identify the circuit. The final action of the rules is to build a constraint model of the circuit. This is done by sending each component of the circuit a `model!`[1] message. Each device unit inherits this message from its class definition, and appears as a slot in the frames.

---

[1]Within KEE I have adopted the convention that all message names are terminated with an exclamation mark. This distinguishes method slots from other slots in the KEE units.

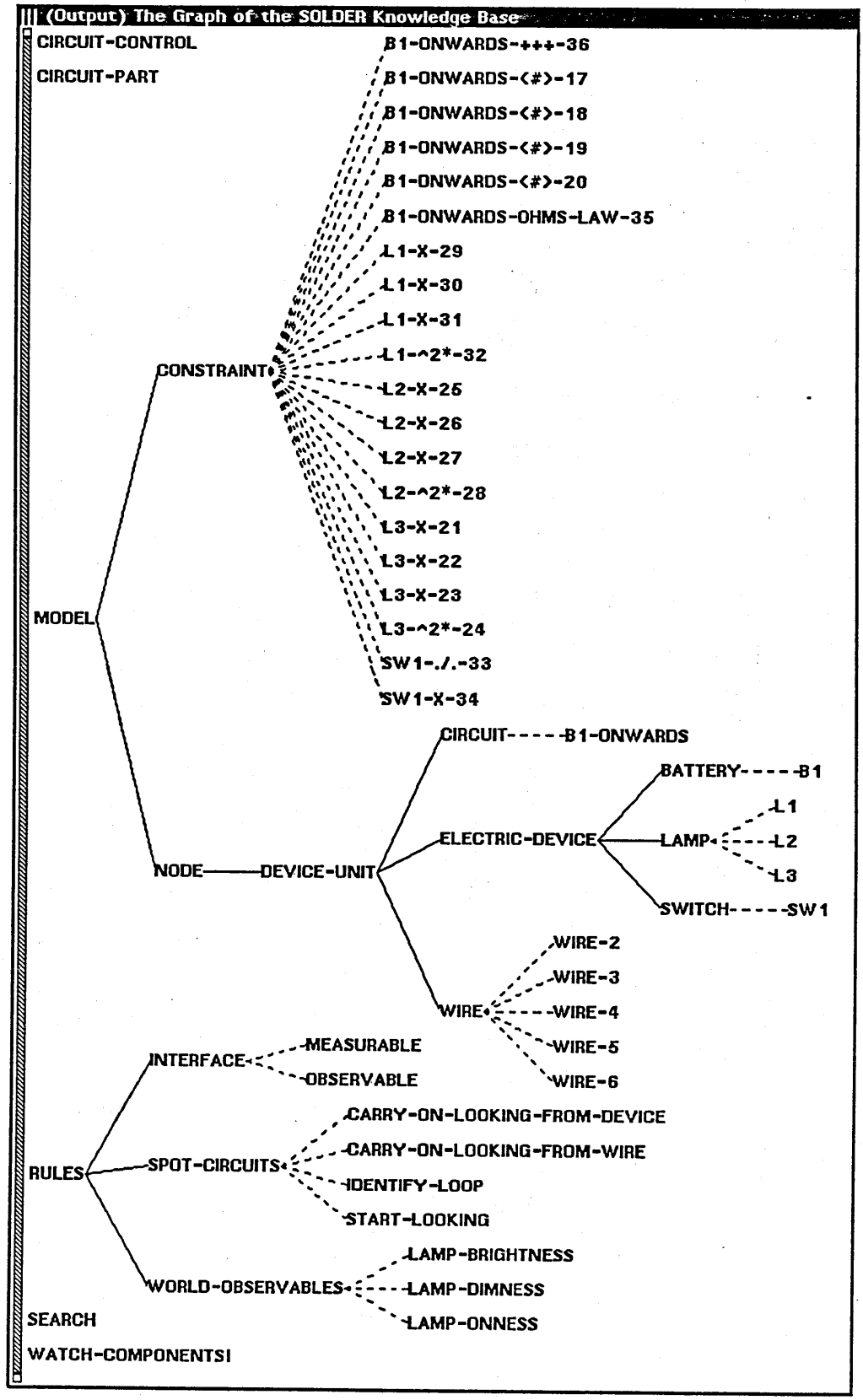


Figure 6.13. The Solder Knowledge base after the recognition rules Spot-Circuits have identified the circuit device unit `B1-Onwards`, and modelled the device units with the constraints. The constraints are shown as children of the constraint KEE unit and correspond to the graphical representation of the network shown in Figure 5.4(a).

The modelling process builds networks of constraints representing the operating principles of the device components. The constraints are built within the Solder knowledge base as children of the CONSTRAINT class (see Figure 6.13). The *constraints*, which appeared as 'black-box' icons in Circuit Buff, are KEE units in Solder. The *nodes* of the constraint network, which appeared as wires in Circuit Buff, are now slots of the device units, and values on the wires become values in these slots. The device units inherit the node behaviour from the node class. Referring back to Figure 6.12, which showed the Solder representation of L1, the parameters of the lamp have an extra facet called constraints. This facet contains an association list of the constraints and ports connected to the parameter.

The constraints are given names which indicate where they are and what they do. For instance L2-^2*-21 is the brightness constraint of lamp L2. The number at the end is gensymed to make it into a distinct symbol.

6.2.1. Constraint Propagation in Solder

There are two types of constraint propagation in Solder. One for initializing the constraint network and the other for performing diagnosis, these are referred to as *causal* and *diagnostic propagation*.

6.2.1.1. Causal Propagation.

Causal constraint propagation behaves in a very similar way to the constraint propagation mechanism of Circuit Buff which was described in section 6.1.4. Values are propagated through the constraints in a way that mimics causal behaviour. For instance in the Ohm's law constraint of the circuit the current can be deduced from the resistance and voltage. This is because the voltage is a force which drives the current against a resistance: the current can't cause the voltage of a battery. However, the voltage drop across a component may be causally deduced

if the resistance and current are known, because the resistance does cause a voltage drop.

No special mechanism has to be defined to allow the constraints to work in this way. One only has to define the constraint networks so that they represent causal change and then confine oneself to setting the parameters of the network that can only be causally set in the real world. Referring back to the Ohm's law example: its OK to set voltages and resistances because batteries and components can be changed. But a current can't be directly changed at will.

6.2.1.1.1. Implementation

The causal constraint propagation procedure within Solder is the same as that in Circuit Buff. However the values that nodes can take are represented differently as described below.

The values of the nodes of the constraint network are simple lisp structures. These structures are called `data`. Each datum has several slots and the four most important of these are listed in Figure 6.14.

Slot	Description
Name	This slot holds the name of the datum. Its value is usually a machine generated symbol.
Value	The value here is the value of the datum.
Constraint	The value of this slot represents the constraint that was used to derive the current value of the datum. If the user set it then the value here is `USER`.
Args	The value of this slot is a list of other data that the constraint used to derive the value.

Figure 6.14. The slots of the `data` lisp structure.

These slots define data dependencies that enable the derivation of a datum to be traced.

Data on the nodes are set through the user interface by sending the relevant KEE unit a `NEW-INFO!` message. The method requires a slot, a

value and an *informant*. The informant is a description of who is setting the value. If the value is being set by the user then this final argument is simply USER. These arguments are used to create a datum, and the datum is sent to the node using the NEW-DATUM! message.

If the datum is acceptable to the node then the method forwards it to the constraints that are attached to the slot. (Data are acceptable at a node if there are no other data already present.) The value is forwarded by sending the relevant constraints the new-datum-at-port! message.

The new-datum-at-port! message takes the port and datum as arguments and tries to complete the constraint for each of the other ports. For each port that there is a completion then the wire connected at that port is sent the new-datum! message.

These procedures are summarized in boxes 6.1 and 6.2.

Procedure:	New-Datum! method of NODE
Arguments:	Slot, and Datum
Do:	If the value of the datum is acceptable Then (1) Set the value of the node to be the datum (2) For each connected constraint, other than the informant, send a new-datum-at-port! message.

Box 6.1. Summary of the new-datum! procedure of nodes. This forms part of the causal constraint propagation algorithm.

Procedure:	New-datum-at-port! method of CONSTRAINT
Arguments:	Port, Datum
Do:	For each of the other ports of the constraint look up the appropriate function template and try to complete. If a value can be derived then create a datum and send it to the node connected to the port using the New-Datum! message.

Box 6.2. Summary of the new-datum-at-port! procedure of constraints. This forms part of the causal constraint propagation algorithm.

The causal constraint propagation mechanism is used in NOSTRUM to initialize the state of the device. The initialization is done within the `all-systems-go` hypothetical world. Diagnostic constraint propagation begins when a datum in the actual device is different to that predicted by the causal constraint propagation. The next section describes this procedure.

6.2.1.2. Diagnostic Constraint Propagation

Causal constraint propagation is used to initialize the state of the device. This initialization process puts data on the nodes of the constraint network within the `all-systems-go` hypothetical world. Each of the data contains a record of the constraints and data that were used to derive it. These records are used in the process of diagnostic constraint propagation to work back through a network to find the cause of a fault.

6.2.1.2.1. Implementation

In Solder, diagnostic constraint propagation is initiated by a user through the `OBSERVE!` message. The user clicks on the relevant `device-unit` and chooses `OBSERVE!` from the menu. Nostrum then prompts for the parameter (slot) and the observed value. When these have been specified Nostrum creates a child world of the `all-systems-go` world. The child world is called `OBSERVATION-nnn` and inherits all the facts of the `all-systems-go` world, but with an extra fact describing the observation. The information describing the observation is made into a datum and sent to the node using the `OBSERVATION!` message.

In chapter 5, this is how NOSTRUM was told that the bulb `L1` had zero brightness. I use that example again here to illustrate the constraint propagation.

When a node receives the `OBSERVATION!` message it creates a child of the current world. The new world is given a name describing the new value of the node. The child world inherits all the facts of the parent

world, except the fact describing the previous value of the node. This fact is overridden by a fact representing the observation.

To find the consequences that the new value implies causal propagation is run within the new world. The causal propagation looks for data that were deduced from the old value of the node and overrides them with data calculated from the observed value. When this is done the world represents the state of the device as it expected to appear if the node had that value.

In the example the brightness had been deduced from the values of resistance and current, but no inferences had been deduced from the value of the brightness. So in this case there are no immediate consequences of establishing that the brightness is zero.

Unless the node can provide an explanation for the observed value constraint propagation continues. There is no immediate explanation for why the brightness of `L1` is zero and so the value is passed on to the constraint that derived the previous value. This is implemented by sending the constraint a `new-observation-at-port!` message.

The brightness constraint is represented by the three function templates:

```
(defvar bb-B=I^2R '(* I I R))
(defvar bb-I=√B/R '(sqrt (/ B R))
(defvar bb-R=B/I2 '(/ B (* I I))
```

where B is the Brightness, R the resistance and I the current through the bulb. Using the new observation and previous values of I and R the constraints can complete to suggest values of I=0 and R=0. To suggest that I is zero the constraint must assume the original value of R holds, and vice versa.

When the constraint can complete it forwards the predicted values (which are made into data) to the relevant nodes. This is done using the `OBSERVATION!` message again. The message will create worlds for each

of the new data and predict their consequences within those worlds using the causal propagation mechanism.

Only if the predicted value for the node can be explained by some known failure mode will the constraint propagation come to a halt. This might happen for instance when the mechanism suggests that R=0, implying that the bulb has shorted out.

Boxes 6.3 and 6.4 summarize the `OBSERVATION!` and `NEW-OBSERVATION-AT-PORT!` messages.

Procedure:	`Observation!` method of `NODE`
Arguments:	`Slot`, and `Datum`
Do:	Create a world as a child of the current world. Give the world a name that describes the observation. Predict the consequences of the observation within the world by initiating causal constraint propagation. Unless the observation constitutes a valid diagnosis then forward the observed value to the constraint that derived the previous value, using the `new-observation-at-port!` message.

Box 6.3. Summary of the `observation!` procedure of `nodes`. This forms part of the diagnostic constraint propagation algorithm.

Procedure:	`New-observation-at-port!` method of `CONSTRAINT`
Arguments:	`Port`, `Datum`
Do:	For each of the other ports of the constraint look up the appropriate function template and try to complete. If a value can be derived then create a datum and send it to the wire connected to the port, using the `OBSERVATION!` message

Box 6.4. Summary of the `new-observation-at-port!` procedure of `constraints`. This forms part of the diagnostic constraint propagation algorithm.

The diagnostic constraint propagation procedure generates a series of worlds whose names reflect the shift of the current hypothesis. How this propagation comes to an end is described in the following section.

6.2.2. Using Heuristic Knowledge

As the propagator works its way through the constraint network it suggests various fault hypotheses. The hypotheses refer to values of various parameters of the device. In the Solder Knowledge base it is possible to associate heuristic knowledge with parameters of the device units. At the end of chapter 5 (see Figure 5.8) an example was given of how Nostrum put forward the hypothesis of a fused bulb. This is implemented as a list by associating the known fault model with a corresponding fault value in the `known-fault-models` slot of the device unit class. The structure of the association list is defined by:

```
(<node-name> <fault-value> <text-string>)
```

For the class of light bulbs these fault models are displayed in Figure 6.15.

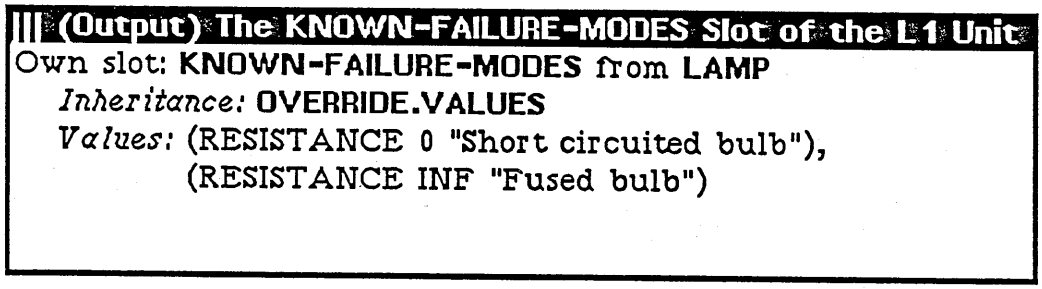


Figure 6.15. The known fault models of the class of light bulbs.

If the constraint propagator tries to suggest a value for a node that corresponds to a known fault model, then the associated text string is displayed to the user. The user can refute or accept the hypothesis, as was shown in Figure 5.8.

Currently the mechanism for representing this type of heuristic knowledge is stricly limited to equating the hypothesised values with the known fault values. However in future upgrades of NOSTRUM it is

anticipated that comparisons between these values become possible in order to cover a wider range of fault models.

As an extra extension to this system a set of keywords could be associated with the node names and values. Such a facility would allow NOSTRUM to ask even more discriminating questions in search of a diagnosis. For example the keyworkd `decays` could be associated with a `voltage` attribute of the class of batterys. In this case when NOSTRUM seeks a reason for a dim bulb from the battery device unit it can ask if the bulb *has been dimming*. This would discriminate a wrong-valued bulb from a failing battery.

Currently the presence of fault models within the constraint network does not affect the search strategy. Nostrum simply performs a depth-first search and suggests fault models if they are found. This is a focus of future work.

The following chapter shows how NOSTRUM is applied to other domains and how heuristic knowledge could be used to improve the search strategy.

Chapter Seven

Application of Nostrum to other domains

In this chapter I describe how NOSTRUM's methodology can be applied to two other domains: the Starting mechanism of a car and a simple doorlock. Most of the effort required to model each domain lies in the definition of the operating principles and writing the function templates. When these are done NOSTRUM's diagnostic strategy does the rest!

7.1. Diagnosing a Car Starting Mechanism.

In this section I describe the methodology that NOSTRUM uses to diagnose a car that has difficulty starting. In the example I show how the circuitry is modelled by constraints and that this leads to an initial diagnostic network. Then it is shown how addition of experiential knowledge can sensitize the network to speed up diagnosis.

To begin with NOSTRUM takes a schematic of the relevant parts of the car. This is input to NOSTRUM via the graphical interface, Circuit Buff, and is shown here in Figure 7.1.

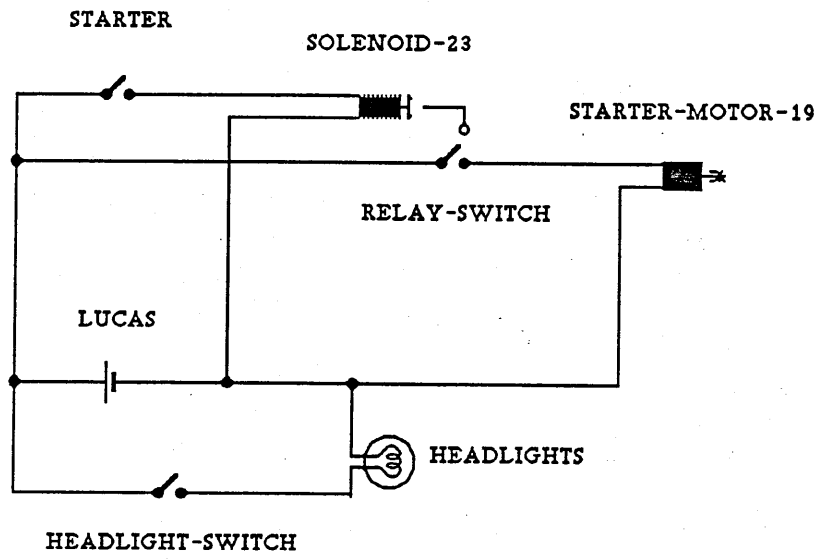


Figure 7.1. Schematic of the starting components of a car engine.

As for the Chistmas tree lights example a constraint network of the Circuit is derived from this input. The constraint network is constructed from library models of the components of the circuit and knowledge of the connectivity. Figure 7.2 shows part of the resulting constraint model, and the equivalent KEE knowledgebase is shown in Figure 7.3.

The constraint network shown is a model of the headlight circuit and the starter motor circuit. At the left of the diagram is the constraint node representing the battery's voltage, in this case 12. Hanging off this node are two adder constraints, `A1-SM` and `A1-H`, these compute the actual voltage driving the Starter-Motor and Headlight circuits by subtracting the voltage drop through the battery (caused by its internal resistance). The voltage drop across the battery is calculated for the starter motor circuit and headlight circuit by the `M1-SM` and the `M1-H` constraints respectively. These constraints use the value of the internal resistance of the battery, and the current in their respective circuits to make this calculation. From here the headlight circuit is very similar to the christmas tree lights circuit, but with only one bulb. The starter motor circuit requires further explanation.

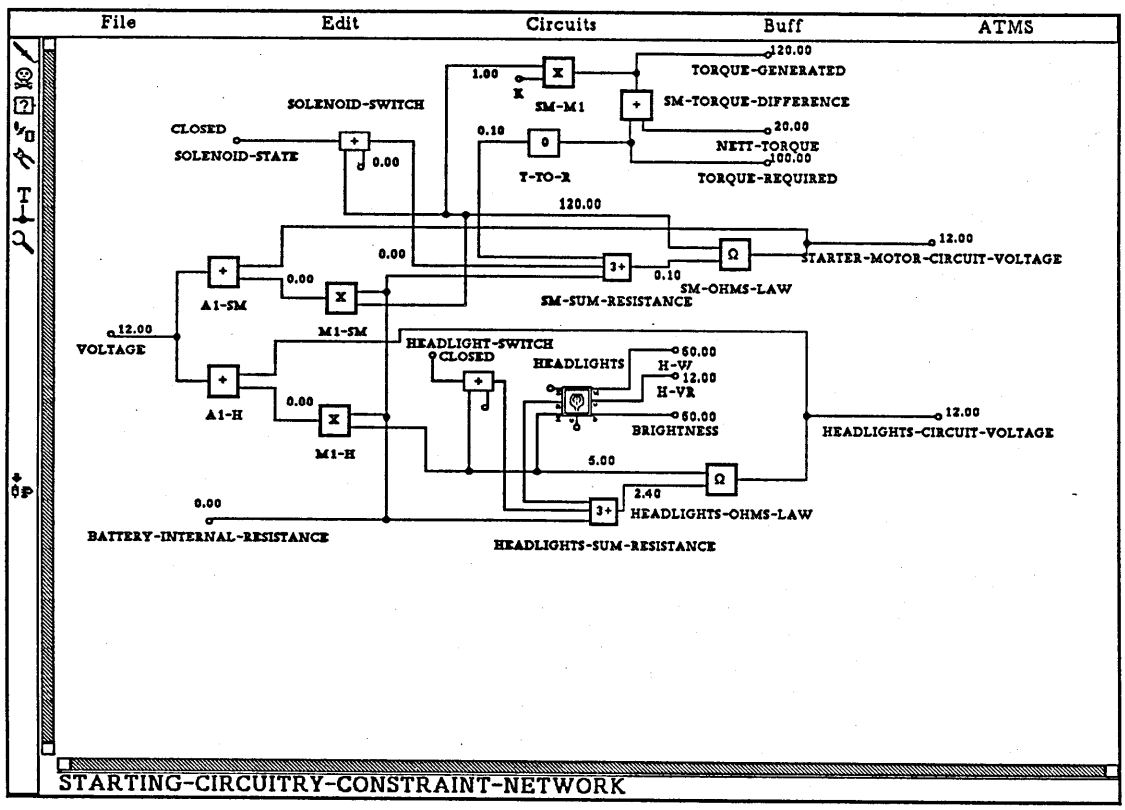


Figure 7.2. The constraint model of the starting components of a car engine.

The starter motor circuit contains the two constraints SM-SUM-RESISTANCE and SM-OHMS-LAW. They derive the total resistance and the current in the starter motor circuit from the resistance of its components and the voltage coming from the battery. The solenoid is modelled as a simple switch, but instead of being activated by a user [as it was in the Christmas tree lights circuit] it is activated by another circuit [the key switch circuit, which is not modelled here]. When the solenoid operates it 'Clicks' and closes the switch so that it offers no resistance to the Starter motor circuit.

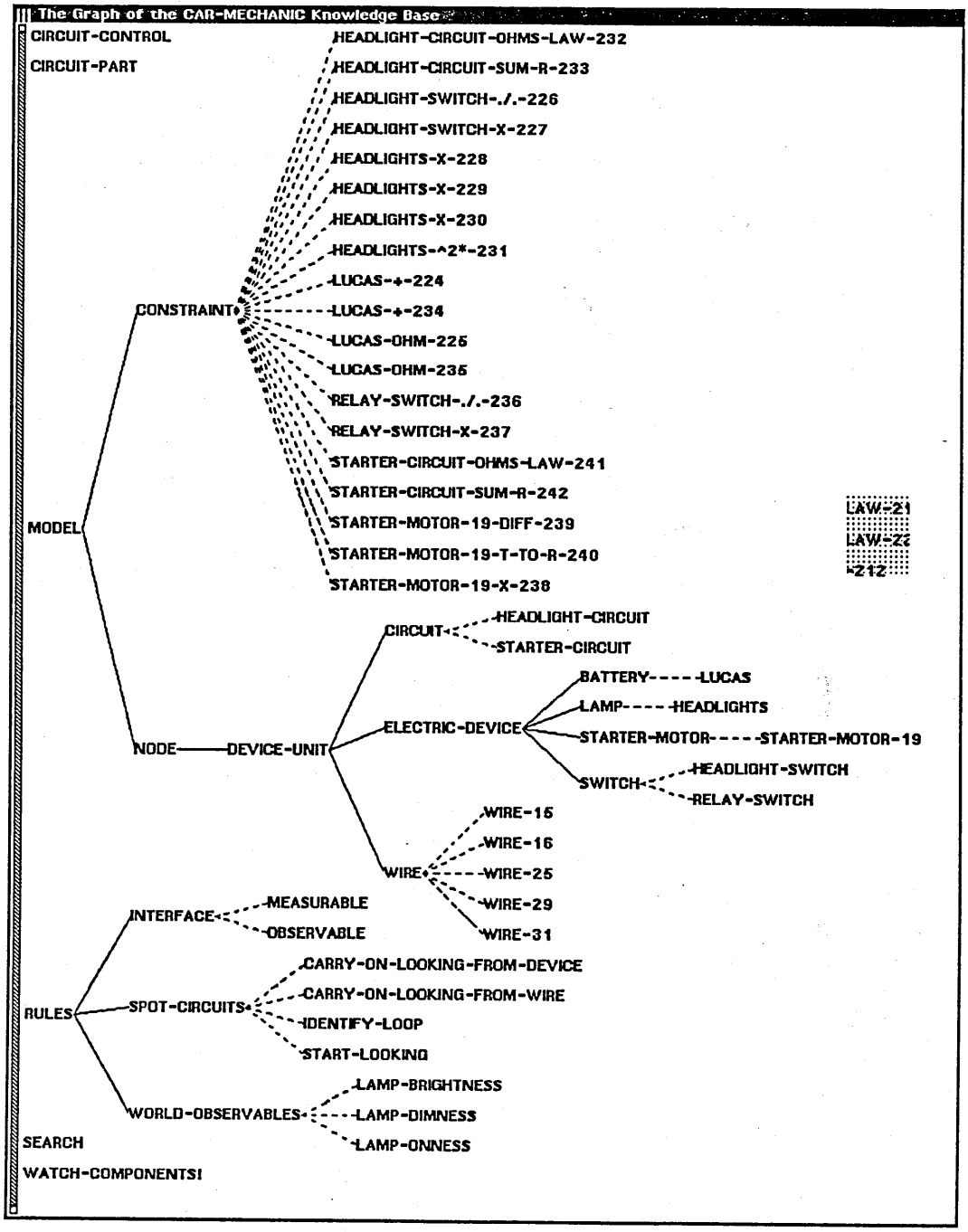


Figure 7.3. The KEE knowledgebase representing the constraint model of the Car Starting Circuit.

The starter motor itself is modelled by three constraints: `SM-M1`, `T-TO-R` and `SM-TORQUE-DIFFERENCE`. The `SM-M1` constraint is a multiplier that simply relates the current to the torque-generated by the starter motor by multiplying factor `K`. `K` is an arbitrary constant which I have chosen to be 1. In a working starter motor more current is drawn from the battery when the motor has to do more work. This happens because

the electrical impedance of the motor falls if resisted from turning due to an increased load. In fact the mechanism is really a complex sequence of magnetic and electrical interactions that take a few moments to come to equilibrium. I have approximated this mechanism into the single constraint T-TO-R [Torque to Resistance] and the behaviour of the constraint is represented by two simple lisp functions, which are shown below. The values used in the functions are the values used by the currrent example, and it is defined as an error if other values are used.

Function to calculate resistance from torque-required:

```
(defvar bb-r=f[tr]
   '(case tr
         (0 12)
         (10 1)
         (100 0.1)
         (otherwise
               (error "Unknown value for required torque."))))
```

Function to calculate torque-required from resistance.

```
(defvar bb-tr=g[r]
   '(case r
         (12 0)
         (1 10)
         (0.1 100)
         (otherwise
               (error "Unknown value for r in bb-tr=g[r]."))))
```

These functions simply return (where they can) the corresponding value from the following table:

Torque Required	Resistance
0	12
10	1
100	0.1

Figure 7.4. The functions used in the T-to-R constraint representing the relation between the torque load of the starter motor and its internal resistance. (Arbitrary units).

The SM-TORQUE-DIFFERENCE constraint derives the difference between the torque produced by the starter motor and that required. This value represents how vigourously the engine turns over when started. In this example a value of 20 represents a healthy start and a value of 10 represents a weak start.

The model is an approximation in many ways. Firstly the constraint mechanism as it stands can only model instantaneous changes, but the torque developed by a starter motor is the result of a complex sequence of causal interactions. Secondly the starting torque of the engine will be greater than when it is already moving because of the inertia of the flywheel. Here the constraints are only used to model one-time situations. Thirdly the relations between current and torque are made up. I have just used the multiplier constraint to illustrate that the torque generated increases as the current through the starter motor increases, the actual relation will depend on the number of windings in the coil, the shape of the armature and the material from which it is made.

To demonstrate the diagnostic capabilities of this constraint network two examples are given. The first case deals with a car engine that only turns over slowly when the key is turned, and the second addresses the case in which there is no engine turn over.

7.1.1. Case 1: Engine turns over weakly.

In order to demonstrate this case the constraint network must be initialized to reflect the initial state of the working device. Figure 7.5 shows this initialization. The circuit is initialized by sending `new-info!` messages to the relevant nodes. Below is a list of the nodes and assumptions used to set these values.

Node	Value	Assumption
Voltage	12	user
Battery-internal resistance	0	user (good battery)
Solenoid-state	closed	user (solenoid working)
headlight-switch	closed	user
H-W (headlight wattage)	60	user
H-VR (voltage rating)	12	user
torque-required	100	user
motor-K	1	user (arbitrary parameter)

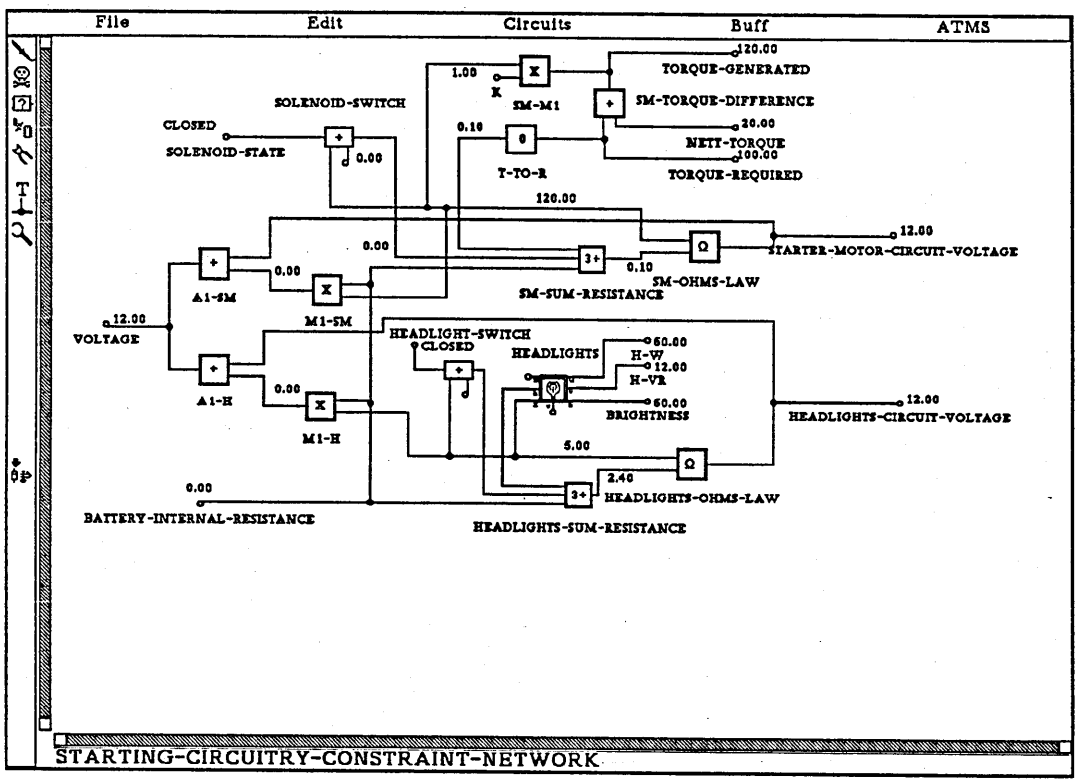


Figure 7.5. The Starting Circuit constraint network initialized for starting the car.

As usual diagnosis begins with a symptom. In this case the car isn't starting, and the would-be driver has noticed that the engine is only turning over weakly under the starter motor. To initiate diagnosis the user makes the observation that the `nett-torque` of the engine is 10. This is done by sending the `observe!` message to the `nett-torque` node of the starter motor and results in the dialogue that begins in Figure 7.6.

In Figure 7.6 the user is firstly prompted for the `node` (slot) and `observed value`, which are entered. This is followd by 5 requests for the optional arguments that the `observe!` method takes. The first of these, `QDIR` (Qualitative DIRection), is a future extension to NOSTRUM that is designed to represent the direction in which the value is observed to be changing. A minus sign indicates a steady or unchanging value. The qualitative rate of change system in NOSTRUM is still experimental and is not discussed further here.

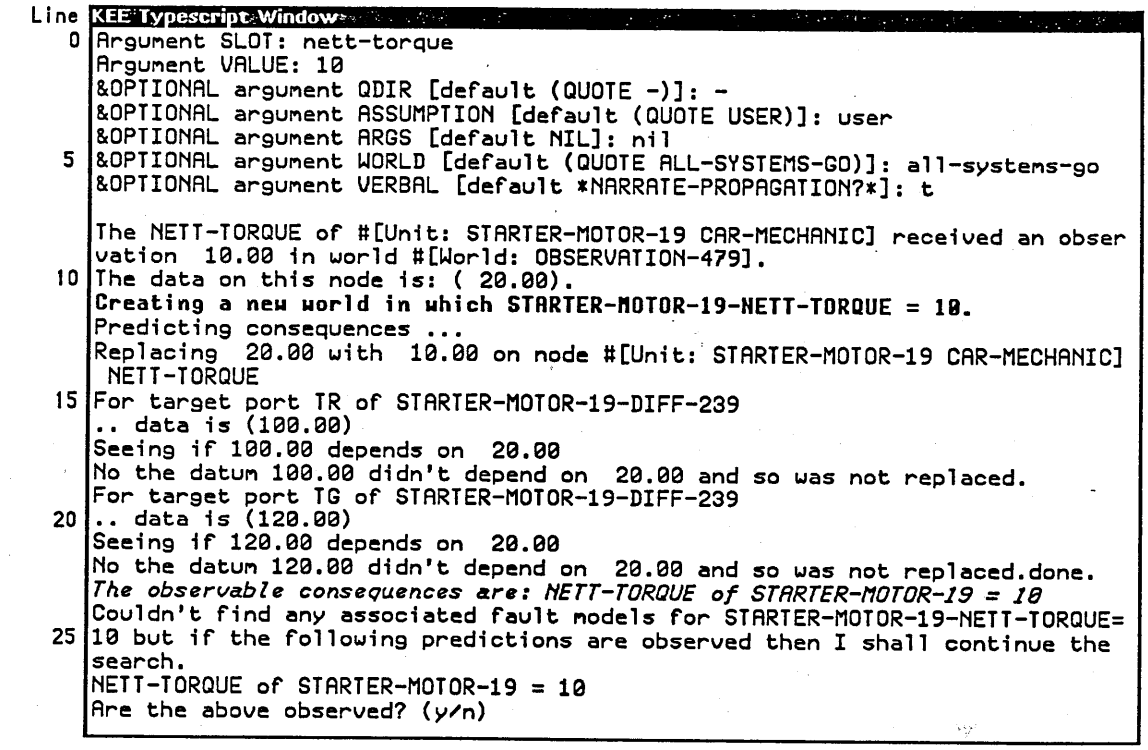


Figure 7.6. Specifying the symptom of a weakly turning engine.

The `assumption` argument represents the justification for the observed value, often this is a constraint name, but here the value is set by the user. The next argument, `args`, requests which data in conjunction with the assumpton have been used to drive the observed value. The world argument requests the hypothetical world from which to begin diagnosis. Finally the user is requested if they want trace information of the kind shown in the rest of Figure 7.6.

After all the arguments have been entered Nostrum begins to print details of its constraint propagation. Firstly (lines 8 & 9 of Figure 7.6) an observation world is created as a child of the `all-systems-go` world, in which the observaton is placed. Using diagnostic constraint propagation the observation is fed onto the node causing the creation of the `starter-motor-19-nett-torque=10-nnn` world. Then using causal propagation the consequences of the observation are predicted.

But since no data are derived from the old value there are no consequences. (The old value of 20 was derived from the user-set value of

100 for the torque-required and the value of 120 computed by the circuit initialization.)

The observable consequences are printed (line 23) and the user is requested if they are observed (lines 24-28). As there are no observable consequences the user must choose "y" in order to continue the search.

```
Line  KEE Typescript Window
   0  Mooting observation  10.00 for port NT of STARTER-MOTOR-19-DIFF-239.
      Trying completion of (?  10.00 120.00).
      Dispatching completion of comb (?  10.00 120.00) to target port TRof STARTE
      R-MOTOR-19-DIFF-239 for acceptance.
      The TORQUE-REQUIRED of #[Unit: STARTER-MOTOR-19 CAR-MECHANIC] received an o
   5  bservation 110.00 in world #[World: STARTER-MOTOR-19-NETT-TORQUE=10-480].
      The data on this node is: (100.00).
      Creating a new world in which STARTER-MOTOR-19-TORQUE-REQUIRED = 110.0.
      CAN'T MAKE (TORQUE-GENERATED #[Unit: STARTER-MOTOR-19 CAR-MECHANIC]), 120.0
       an assumption cus depends on 110.00
  10  Predicting consequences ...
      Replacing 100.00 with 110.00 on node #[Unit: STARTER-MOTOR-19 CAR-MECHANIC]
       TORQUE-REQUIRED
      For target port R of STARTER-MOTOR-19-T-TO-R-240
      .. data is (  0.10)
  15  Seeing if   0.10 depends on 100.00
      Data   0.10 depends on the datum being replaced: 100.00
      The state is (   NEW 110.00)
      Couldn't complete (NEW 110.0) = (R TR) for function (CASE TR (0 12) (10 1)
      (100 0.1) (OTHERWISE (ERROR Unknown value for required torque.))), please s
  20  upply value: ()
      ... which completes to    NILdone.
      The observable consequences are: NETT-TORQUE of STARTER-MOTOR-19 = 10
      Node is user set, stopping propagation.
      Trying completion of (100.00  10.00 ?).
  25  Dispatching completion of comb (100.00  10.00 ?) to target port TGof STARTE
      R-MOTOR-19-DIFF-239 for acceptance.
      The TORQUE-GENERATED of #[Unit: STARTER-MOTOR-19 CAR-MECHANIC] received an
      observation 110.00 in world #[World: STARTER-MOTOR-19-NETT-TORQUE=10-480].
      The data on this node is: (120.00).
  30  Creating a new world in which STARTER-MOTOR-19-TORQUE-GENERATED = 110.
      Making the value on (TORQUE-REQUIRED #[Unit: STARTER-MOTOR-19 CAR-MECHANIC]
      ), 100 an assumption.
      Predicting consequences ...
      Replacing 120.00 with 110.00 on node #[Unit: STARTER-MOTOR-19 CAR-MECHANIC]
  35  TORQUE-GENERATED
      For target port I of STARTER-MOTOR-19-X-238
      .. data is (120.00)
      Seeing if 120.00 depends on 120.00
      No the datum 120.00 didn't depend on 120.00 and so was not replaced.
  40  For target port K of STARTER-MOTOR-19-X-238
      .. data is (  1.00)
      Seeing if   1.00 depends on 120.00
      No the datum   1.00 didn't depend on 120.00 and so was not replaced.done.
      The observable consequences are: TORQUE-GENERATED of STARTER-MOTOR-19 = 110
  45  ,NETT-TORQUE of STARTER-MOTOR-19 = 10
      Couldn't find any associated fault models for STARTER-MOTOR-19-TORQUE-GENER
      ATED=110 but if the following predictions are observed then I shall continu
      e the search.
      TORQUE-GENERATED of STARTER-MOTOR-19 = 110,NETT-TORQUE of STARTER-MOTOR-19
  50  = 10
      Are the above observed? (y/n)
```

Figure 7.7. Continuation of the diagnostic trace initiated in Figure 7.6.

Diagnostic propagation begins by trying to assert that the torque generated by the starter motor is 110 and creates a new world in which consequences of the hypothesis can be predicted (refer to Fig 7.7 in the

following discussion). As part of the predicting consequences process the data connected to the constraint must be made into assumptions, since they have been used to derive the hypothesized value. This is done by setting a slot in the structure that represents the datum. At lines 8 & 9 of Figure 7.7 Nostrum explains that the value of 120 for the `torque-generated` cannot be made into an assumption because it depended (was derived from) the value of 110 for the node being replaced, `torque required`.

To predict consequences Nostrum replaces the previous values with the hypothesized values within the new world according to the rules of causal propagation. This is done in lines 11 & 12 of Figure 7.7 where the hypothesized value of 110 replaces the old value of 100 for the torque-required of starter-motor-19. (Even though Nostrum knows that this is a user assrted value it firstly predicts the conseqeunces to see if there are changes that the user may not be aware of.) This value is then propagated to the T-to-R constraint at line 13 where the data for the `resistance R` (currently 0.1 Ohms) has been derived from the old value of 100 for the `torque required`. The constraint tries to complete, but it doesn't have a value for R that corresponds to a required torque of 110 (see Figure 7.3). As it can't complete the user is explained as to why at lines 18 &19 and asked to supply a value at line 20. For this example I have entered the emply list or `NIL` value which stops causal propagation along this arm. This also concludes Nostrum's prediction of consequences for `torque-required` and propagation stops. Diagnostic propagation now takes over for the other node of the `SM-torque-difference` constraint in which a value of 110 is suggested for the `torque-generated` by the starter motor.

Again the values that were used to derive the hypothesized value are made into assumptions (lines 31 & 32), and the consequences predicted (lines 33-43). During the prediction of consequences any nodes whose value are replaced are tested to see if they are observable parameters. If

they are, then the replacement value is noted and printed back to the user after the rest of the consequences have been predicted (lines 44-45 & 49-50). If it is not known whether a node has observable consequences the user is requested for this information via a pop-up menu (see Figure 7.8].

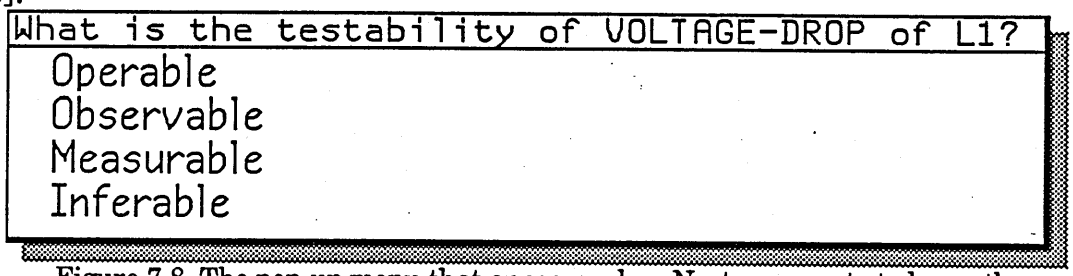


Figure 7.8. The pop-up menu that appears when Nostrum wants to know the testability of a node.

The prediction of consequences goes on until no more values are found that depend on the newly hypothesized value (line 43). This done the user is asked to confrim the absence or presence of the observed consequences, (line 51) and any hypothesized fault models. Here the user chooses "y" to continue the diagnostic propagation, and the trace continues in Figure 7.9.

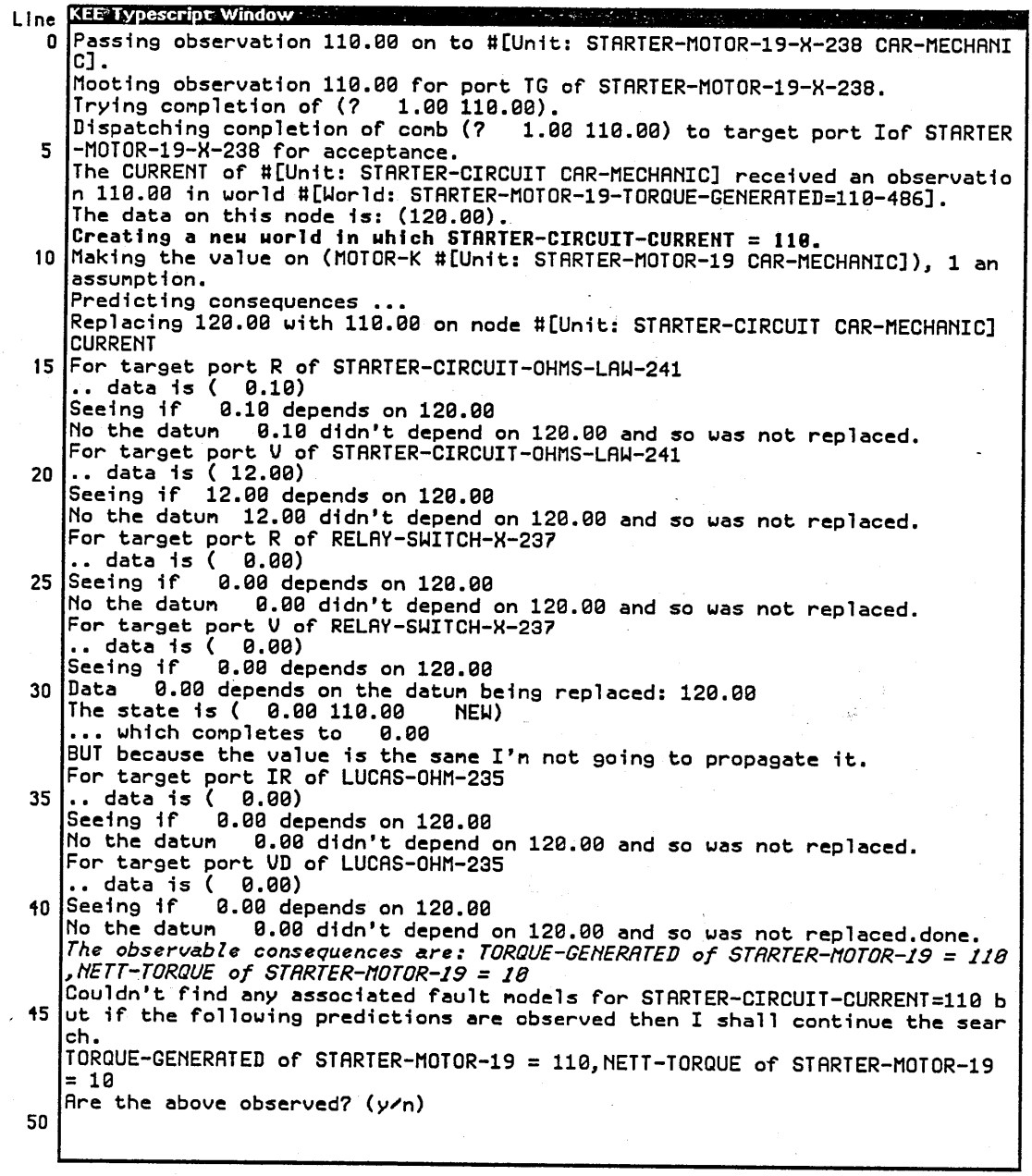


Figure 7.9. The diagnostic trace continued from Figure 7.7.

Figure 7.9 continues the trace output from Figure 7.7. At line 0 it begins by accepting the hypothesis that the value of the torque generated is 110 and passes it on to the connected constraint `starter-motor-19-x-238`. This causes a completion of the constraint to hypothesize a value of 110 Amps for the current in the starter motor circuit. (line 9 of Figure 7.9). The new value prompts prediction of consequences (lines 12-41) but there are no new ones in this case. The already deduced consequences

are the repeated in lines 42 & 43 (and again in lines 47 & 48) and the user is prompted whether to continue the search.

The reduced current can be explained by an increased resistive input to the circuit which is where the trace is rejoined in Figure 7.10.

```
Line  KEE Typescript Window
   0  Passing observation 110.00 on to #[Unit: STARTER-CIRCUIT-OHMS-LAW-241 CAR-M
      ECHANIC].
      Mooting observation 110.00 for port I of STARTER-CIRCUIT-OHMS-LAW-241.
      Trying completion of (110.00 ?  12.00).
      Dispatching completion of comb (110.00 ?  12.00) to target port Rof STARTER
   5  -CIRCUIT-OHMS-LAW-241 for acceptance.
      The TOTAL-RESISTANCE of #[Unit: STARTER-CIRCUIT CAR-MECHANIC] received an o
      bservation   0.11 in world #[World: STARTER-CIRCUIT-CURRENT=110-490].
      The data on this node is: (  0.10).
      Creating a new world in which STARTER-CIRCUIT-TOTAL-RESISTANCE = 6/55.
  10  Making the value on (VOLTAGE #[Unit: STARTER-CIRCUIT CAR-MECHANIC]), 12 an
      assumption.
      Predicting consequences ...
      Replacing   0.10 with   0.11 on node #[Unit: STARTER-CIRCUIT CAR-MECHANIC]
      TOTAL-RESISTANCE
  15  For target port A of STARTER-CIRCUIT-SUM-R-242
      .. data is (  0.10)
      Seeing if   0.10 depends on   0.10
      No the datum   0.10 didn't depend on   0.10 and so was not replaced.
      For target port B of STARTER-CIRCUIT-SUM-R-242
  20  .. data is (  0.00)
      Seeing if   0.00 depends on   0.10
      No the datum   0.00 didn't depend on   0.10 and so was not replaced.
      For target port C of STARTER-CIRCUIT-SUM-R-242
      .. data is (  0.00)
  25  Seeing if   0.00 depends on   0.10
      No the datum   0.00 didn't depend on   0.10 and so was not replaced.done.
      The observable consequences are: TORQUE-GENERATED of STARTER-MOTOR-19 = 110
      ,NETT-TORQUE of STARTER-MOTOR-19 = 10
      Couldn't find any associated fault models for STARTER-CIRCUIT-TOTAL-RESISTA
  30  NCE=6/55 but if the following predictions are observed then I shall continu
      e the search.
      TORQUE-GENERATED of STARTER-MOTOR-19 = 110,NETT-TORQUE of STARTER-MOTOR-19
      = 10
      Are the above observed? (y/n)
```

Figure 7.10. The diagnostic trace continued from Figure 7.9.

Lines 0-4 of Figure 7.10 explain that the new current of 110 amps is passed to the `SM-Ohms-law` constraint (refer to Figure 7.5). This generates a new hypothesis that the total resistance has the reduced value of 6/55 (=0.10909...) as opposed to 0.1. No new observable consequences are deduced from this, but the search is continued in Figure 7.11.

In order to explain the increased resistance diagnostic propagation 'poles' each of the nodes feeding the `sm-sum-resistance` constraint. The first of these is the node that represents the resistance of the starter motor. Therefore a new value is hypothesized for it, as displayed at line 9 of Figure 7.11. As usual the consequences are predicted. However note

that at line 20-21 the prediction comes prematurely to a halt because the causal propagation tries to assert a value for a node that contains an assumed datum, ie. a loop has been detected - and broken. Diagnostic propagation still persists in trying to assert a value on the node, but because the constraint can't complete the user is allowed to enter a null value, stopping the propagation.

Again no new consequences are predicted from this hypothesis, and the propagation down this arm comes to a halt. The trace is resumed again in Figure 7.12 in which the second feeder to `sm-sum-resistance` is poled.

```
Line  KEE Typescript Window
   0  Passing observation   0.11 on to #[Unit: STARTER-CIRCUIT-SUM-R-242 CAR-MECH
      ANIC].
      Mooting observation   0.11 for port D of STARTER-CIRCUIT-SUM-R-242.
      Trying completion of (?   0.00   0.00   0.11).
      Dispatching completion of comb (?   0.00   0.00   0.11) to target port Aof
   5  STARTER-CIRCUIT-SUM-R-242 for acceptance.
      The RESISTANCE of #[Unit: STARTER-MOTOR-19 CAR-MECHANIC] received an observ
      ation   0.11 in world #[World: STARTER-CIRCUIT-TOTAL-RESISTANCE=6/55-495].
      The data on this node is: (  0.10).
      Creating a new world in which STARTER-MOTOR-19-RESISTANCE = 6/55.
  10  Making the value on (RESISTANCE #[Unit: RELAY-SWITCH CAR-MECHANIC]), 0 an a
      ssumption.
      Making the value on (INTERNAL-RESISTANCE #[Unit: LUCAS CAR-MECHANIC]), 0 an
       assumption.
      Predicting consequences ...
  15  Replacing   0.10 with   0.11 on node #[Unit: STARTER-MOTOR-19 CAR-MECHANIC]
       RESISTANCE
      For target port TR of STARTER-MOTOR-19-T-TO-R-240
      .. data is (100.00)
      Seeing if 100.00 depends on   0.10
  20  Cannot replace the value at TR of STARTER-MOTOR-19-T-TO-R-240 because it is
       an assumption.done.
      The observable consequences are: TORQUE-GENERATED of STARTER-MOTOR-19 = 110
      ,NETT-TORQUE of STARTER-MOTOR-19 = 10
      Couldn't find any associated fault models for STARTER-MOTOR-19-RESISTANCE=6
  25  /55 but if the following predictions are observed then I shall continue the
       search.
      TORQUE-GENERATED of STARTER-MOTOR-19 = 110,NETT-TORQUE of STARTER-MOTOR-19
      = 10
      Are the above observed? (y/n)
  30  Passing observation   0.11 on to #[Unit: STARTER-MOTOR-19-T-TO-R-240 CAR-ME
      CHANIC].
      Mooting observation   0.11 for port R of STARTER-MOTOR-19-T-TO-R-240.
      Trying completion of (  0.11 ?).
      Dispatching completion of comb (  0.11 ?) to target port TRof STARTER-MOTOR
  35  -19-T-TO-R-240 for acceptance.
      Couldn't complete (6/55 NEW) = (R TR) for function (CASE R (12 0) (1 10) (0
      .1 100) (OTHERWISE (ERROR Unknown value for r in bb-tr=g[tr].))), please su
      pply value: ()
```

Figure 7.11. Continuation of the Diagnostic Trace.

```
Line KEE Typescript Window
   0 Trying completion of (  0.10 ?   0.00   0.11).
     Dispatching completion of comb (  0.10 ?   0.00   0.11) to target port Bof
     STARTER-CIRCUIT-SUM-R-242 for acceptance.
     The RESISTANCE of #[Unit: RELAY-SWITCH CAR-MECHANIC] received an observatio
     n   0.01 in world #[World: STARTER-CIRCUIT-TOTAL-RESISTANCE=6/55-495].
   5 The data on this node is: (  0.00).
     Creating a new world in which RELAY-SWITCH-RESISTANCE = 0.009090908.
     Making the value on (RESISTANCE #[Unit: STARTER-MOTOR-19 CAR-MECHANIC]), 0.
     1 an assumption.
     Making the value on (INTERNAL-RESISTANCE #[Unit: LUCAS CAR-MECHANIC]), 0 an
  10  assumption.
     Predicting consequences ...
     Replacing   0.00 with   0.01 on node #[Unit: RELAY-SWITCH CAR-MECHANIC] RES
     ISTANCE
     For target port I of RELAY-SWITCH-X-237
  15 .. data is (110.00)
     Seeing if 110.00 depends on   0.00
     I ofRELAY-SWITCH-X-237 is the HYPOTHESIS
     For target port V of RELAY-SWITCH-X-237
     .. data is (  0.00)
  20 Seeing if   0.00 depends on   0.00
     Data   0.00 depends on the datum being replaced:   0.00
     The state is (  0.01 110.00    NEW)
     ... which completes to   1.00
     Replacing   0.00 with   1.00 on node #[Unit: RELAY-SWITCH CAR-MECHANIC] VOL
  25 TAGE-DROP
     For target port SWITCH-STATE of RELAY-SWITCH-./.-236
     .. data is (CLOSED)
     Seeing if CLOSED depends on   0.00
     No the datum CLOSED didn't depend on   0.00 and so was not replaced.done.
  30 The observable consequences are: TORQUE-GENERATED of STARTER-MOTOR-19 = 110
     ,NETT-TORQUE of STARTER-MOTOR-19 = 10
     Couldn't find any associated fault models for RELAY-SWITCH-RESISTANCE=0.009
     090908 but if the following predictions are observed then I shall continue
     the search.
  35 TORQUE-GENERATED of STARTER-MOTOR-19 = 110,NETT-TORQUE of STARTER-MOTOR-19
     = 10
     Are the above observed? (y/n)

     Passing observation   0.01 on to #[Unit: RELAY-SWITCH-./.-236 CAR-MECHANIC]
     .
  40 Mooting observation   0.01 for port RESISTANCE of RELAY-SWITCH-./.-236.
     Trying completion of (?   0.01).
     Dispatching completion of comb (?   0.01) to target port SWITCH-STATEof REL
     AY-SWITCH-./.-236 for acceptance.
  45 Couldn't complete (NEW 0.009090908) = (SWITCH-STATE RESISTANCE) for functio
     n (CASE RESISTANCE (0 'CLOSED) (INF 'OPEN)), please supply value: ()
```

Figure 7.12. The diagnostic trace continued.

Next the solenoid switch (connected to port B of sm-sum-resistance (Figure 7.5 and line 1-2 of Figure 7.12) is poled as a potential cause of the extra resistance. The diagnostic propagation tries to suggest that the relay switch has faulted to give a resistance of 0.00909... Ohms (line 6 of Figure 7.12). No new predictions are derived from this hypothesis, and the user artificially brings the constraint propagation to a halt when Nostrum tries to complete the constraint for the switch position (lines 41-46 of Figure 7.12).

Figure 7.13 picks up the search as Nostrum returns to pole the final port of the `sm-sum-reistance` constraint.

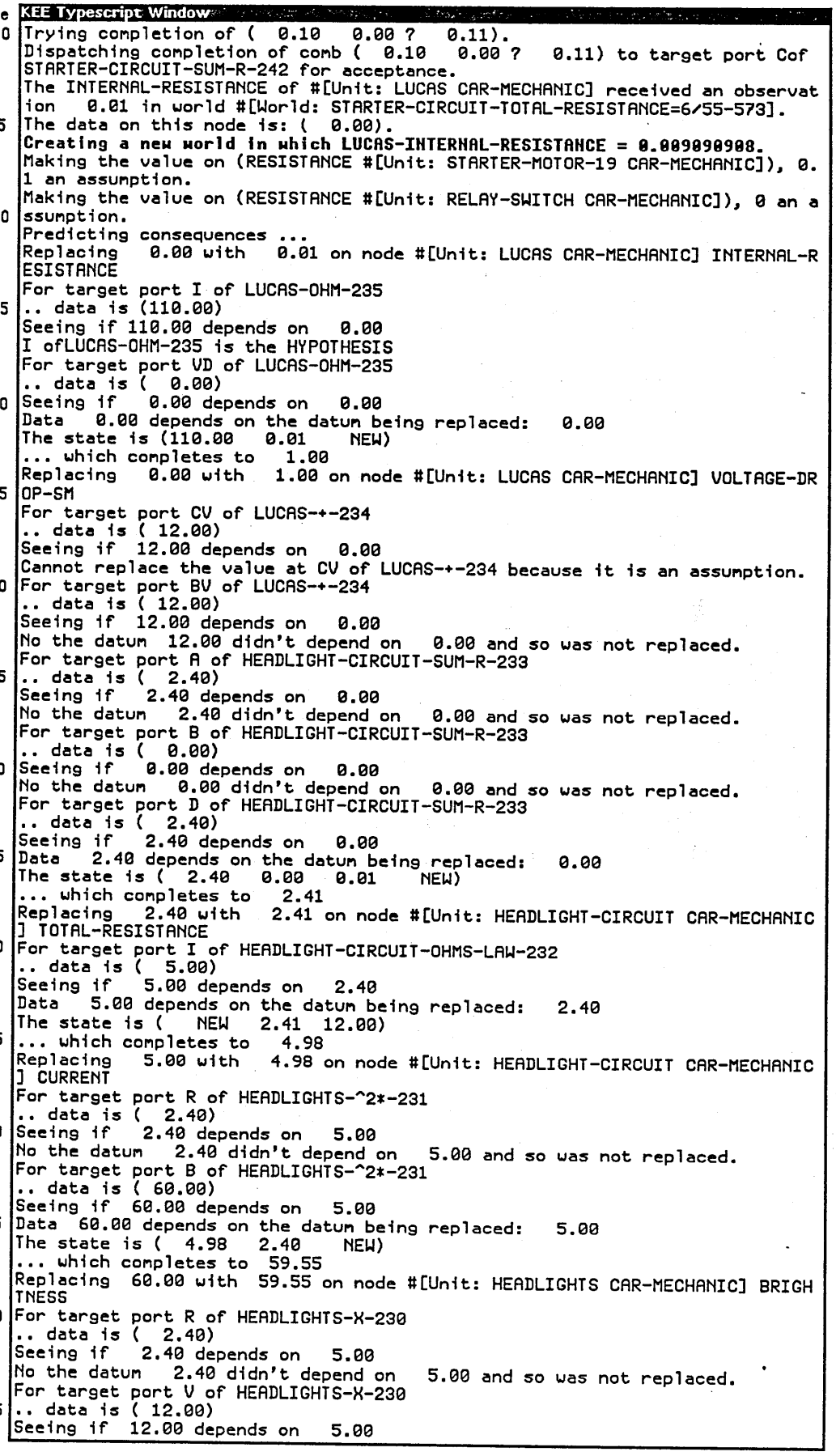


Figure 7.13(a). The trace continued.

```
Line  KEE Typescript Window
      Data  12.00 depends on the datum being replaced:   5.00
      The state is (  2.40   4.98    NEW)
      ... which completes to  11.95
   80 Replacing  12.00 with  11.95 on node #[Unit: HEADLIGHTS CAR-MECHANIC] VOLTA
      GE-DROP
      For target port R of HEADLIGHT-SWITCH-X-227
      .. data is (  0.00)
      Seeing if   0.00 depends on   5.00
   85 No the datum   0.00 didn't depend on   5.00 and so was not replaced.
      For target port V of HEADLIGHT-SWITCH-X-227
      .. data is (  0.00)
      Seeing if   0.00 depends on   5.00
      Data   0.00 depends on the datum being replaced:   5.00
   90 The state is (  0.00   4.98    NEW)
      ... which completes to   0.00
      BUT because the value is the same I'm not going to propagate it.
      For target port IR of LUCAS-OHM-225
      .. data is (  0.01)
   95 Seeing if   0.01 depends on   5.00
      IR ofLUCAS-OHM-225 is the HYPOTHESIS
      For target port VD of LUCAS-OHM-225
      .. data is (  0.00)
      Seeing if   0.00 depends on   5.00
  100 No the datum   0.00 didn't depend on   5.00 and so was not replaced.
      For target port V of HEADLIGHT-CIRCUIT-OHMS-LAW-232
      .. data is ( 12.00)
      Seeing if  12.00 depends on   2.40
      No the datum  12.00 didn't depend on   2.40 and so was not replaced.
  105 For target port I of LUCAS-OHM-225
      .. data is (  4.98)
      Seeing if   4.98 depends on   0.00
      No the datum   4.98 didn't depend on   0.00 and so was not replaced.
      For target port VD of LUCAS-OHM-225
  110 .. data is (  0.00)
      Seeing if   0.00 depends on   0.00
      Data   0.00 depends on the datum being replaced:   0.00
      The state is (NIL   0.01    NEW)done.
      The observable consequences are: BRIGHTNESS of HEADLIGHTS = 59.548023,TORQU
  115 E-GENERATED of STARTER-MOTOR-19 = 110,NETT-TORQUE of STARTER-MOTOR-19 = 10
      Are the above observed? (y/n)
```

Figure 7.13(b). The Trace continued.

Port C of SM-Sum-Resistance is then poled (lines 0-2) of Figure 7.13. This node represents the internal resistance of the battery. The value on this node was initialized to zero on the assumption that it was a good battery. A new world is created (line 6) in which the battery has positive resistance and the consequences predicted as before. Many node values depend on this assumption as a detailed examination of the trace shows. During the trace many pop-up menus like that in Figure 7.8 appear to ask about the testability of nodes. The only new nodes that are observable are the headlights' brightness and the position of the headlight switch, the other nodes are either measurable or inferable. Figure 7.14 shows the constraint network at this stage, after all the consequences have been predicted within the `lucas-internal-resistance=0.009009090908-nnn` world. A

summary of the worlds generated during this diagnosis is displayed in Figure 7.15.

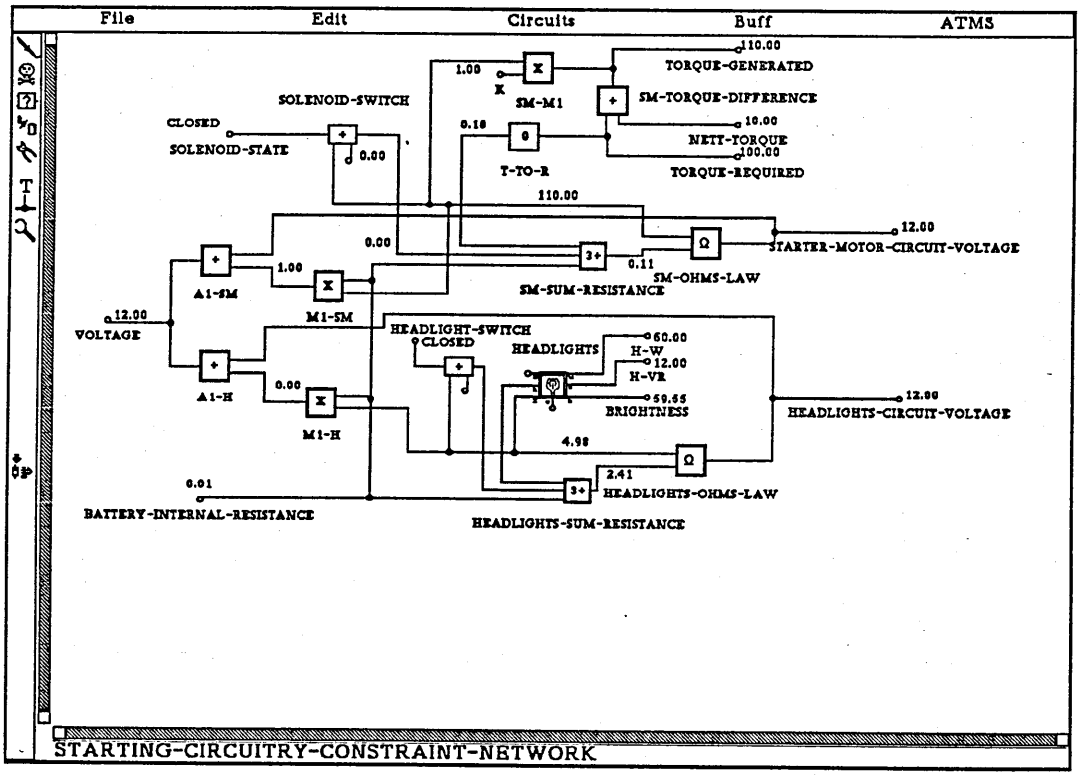


Figure 7.14. Constraint network for the starting circuitry of a car, showing the predicted values in the `lucas-internal-resistance=0.009009090908-nnn` world.

It is lines 114-116 of Figure 7.13 (b) that really show Nostrum's power. They are suggesting that if the hypothesis for a weakly starting car is a discharged battery then this can be tested by seeing if the headlights are shining normally. This test is one commonly used by car mechanics, but has been derived by Nostrum using models of operating principles, and knowledge of what parts can easily be tested.

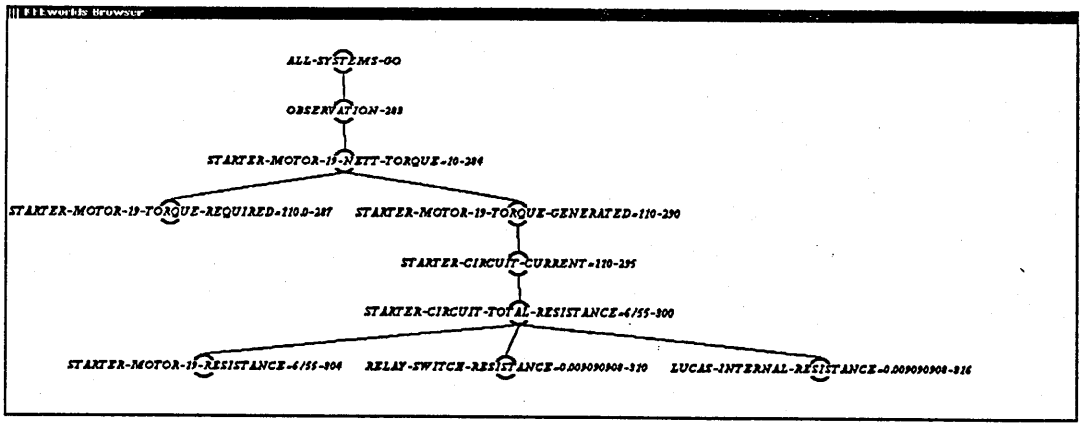


Figure 7.15. The hypothetical worlds generated by Nostrum to try and explain a weakly starting engine.

7.1.2. Case 2: Engine Does not start.

In this example I show how a different fault in the same constraint network leads Nostrum to blame another part of the car engine. The constraint network is initialized as before (Figure 7.5) to represent the expected behaviour of the car engine. In this case the symptom is that the starter motor fails to do anything when the ignition switch is turned. On the constraint network this symptom is transformed into an observation that there is no torque generated by the starter motor.

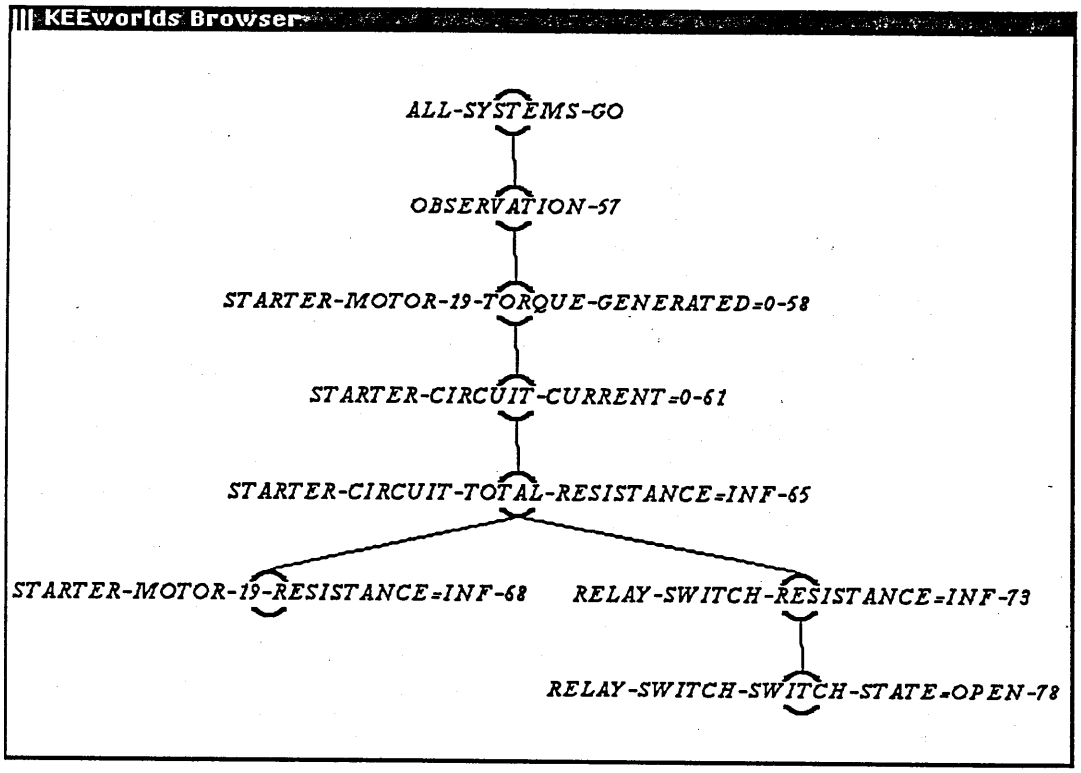


Figure 7.16. The Hypothetical worlds generated in order to diagnose an imobile starter motor.

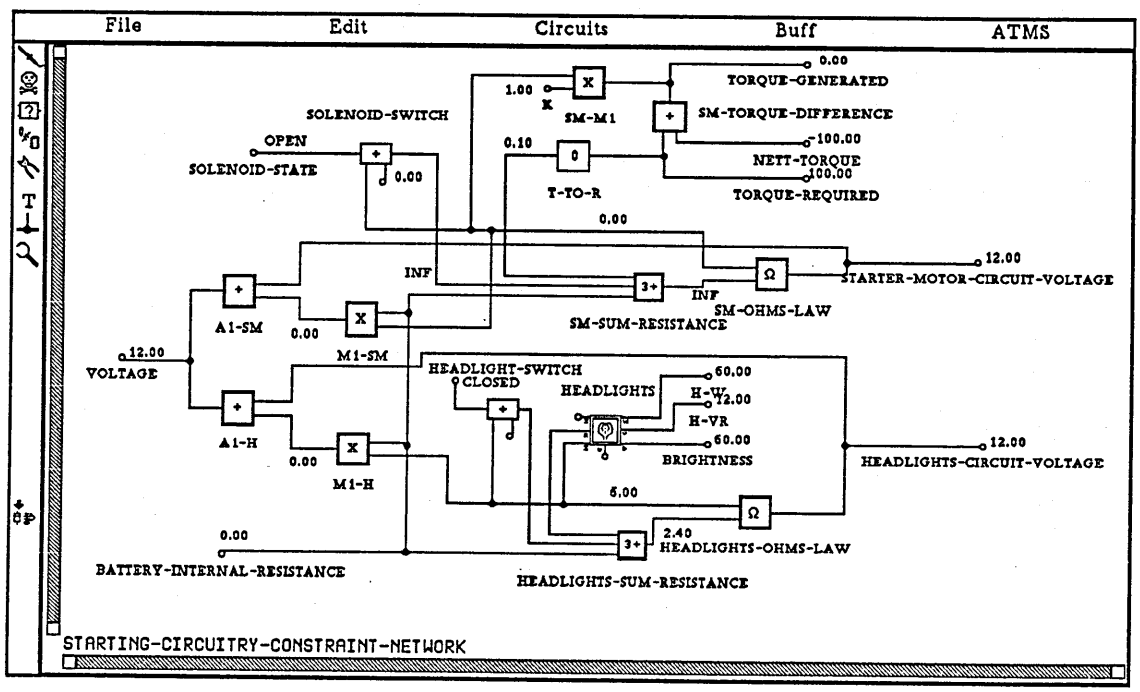


Figure 7.17. The constraint network of the car starter circut showing the values in the `relay-switch-state=open-78` world.

The user therefore sends an `observe!` message to the starter-motor unit with arguments slot: `torque-generated`, and value: `0`. This

results in the generation of the hypothetical worlds shown in Figure 7.16. Figure 7.17 shows the constraint network as it appears in the `relay-switch-state=open-78` world.

To generate this hypothesis Nostrum asks fewer questions about the testability of nodes, because those questions were already answered in Case 1. However because of the depth first nature of Nostrum's search strategy there are still many questions asked of the user similar to those described in the figures of Case 1. A better strategy for Nostrum would be to wade into several hypothetical worlds first, looking for predicted values that correspond to known fault models. In this way the constraint network would become 'sensitized' to different symptoms. Ie. a torque generated = 0 symptom would map directly to an open switch, and a weak start would map directly to a flat battery.

Future work with Nostrum will aim to link the deep (model bahviour) and shallow (experiential) knowledge.

7.2. Doorlock

I use the same doorlock example previously used by [Hunt and Price 1989] to show how NOSTRUM can be applied to a mechanical domain. Figure 7.18 shows the structure of the doorlock.

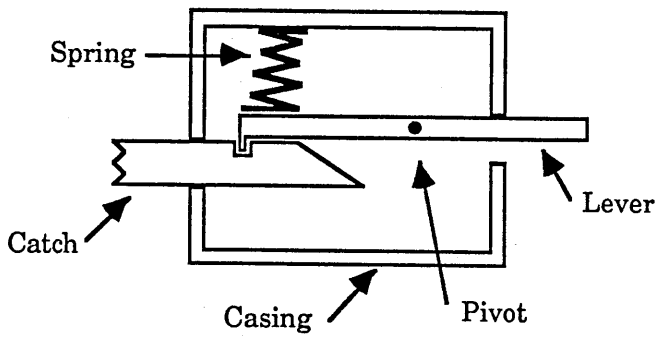


Figure 7.18. An x-ray view of a simple doorlock. The catch is released by pressing on the lever.

Central to the operation of the lock is the lever. The constraint model of the type of lever used in the doorlock is shown in Figure 7.19. There

are three types or orders of lever depending on the relative positions of the pivot, applied force and load. For a description of these the reader is referred to Meccano Mechanisms Set Instructions 1971.

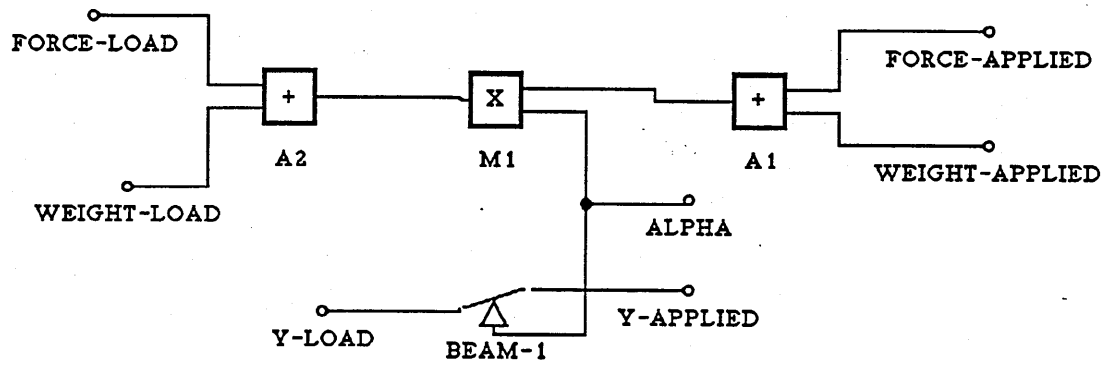


Figure 7.19. The constraint model of a lever.

To emphasize the symbolic capabilities of the constraint propagator within Nostrum I use in this example qualitative values to represent the operation of the device. The example also shows the difficulties of modelling devices which do not come to an instantaneous equilibrium.

The lever constraint model contains 4 constraints, and can be considered as two parts: one part propagates the force in the lever from one side to the other (the top half of Figure 7.19), and the other part propagates the relative height of either side of the lever (the single constraint at the bottom of Figure 7.19). Each side of a lever can have a force applied to it and a weight attached, this results in a resultant force on the arm of the lever. The adder constraints A1, and A2 are used to sum these forces. The multiplier constrains the resultant forces on the arms, and the ratio of the lengths of the arms `alpha`. `Alpha` is also used by the inverse-multiplier (`beam-1` in Figure 7.19) to constrain the heights of either side of the lever. For `A1`, which constrains `force-applied`, `weight-applied` and `resultant-applied` the behaviour of the constraint is defined by:

```
(setf bb-resultant-applied=force-applied+weight-applied
    '(case weight-applied
         (0 force-applied)))
```

This function simply lets the value of `force-applied` pass through the constraint to the `resultant-applied` node, it can do this because the weights applied in this example are zero. There are similar inverse functions, and an equivalent set for `A2`.

The multiplier M1 constrains the two resultant forces and the ratio of the two arm lengths. In this example the ratio is assumed to be 1.

```
(setf bb-resultant-load=ratio*resultant-applied
        '(case ratio
               (1 (case resultant-applied
                        (push-down 'push-up)
                        (push-up 'push-down)))))
```

The constraint `beam-1` behaves like a multiplier and one of its functions is given below:

```
(setf bb-y-load=-y-applied/ratio
   '(case ratio
          (1 (case y-applied
                   (down 'up)
                   (up 'down)))))
```

The full constraint network for the doorlock is given in Figure 7.20, with the lever compressed into a concoction.

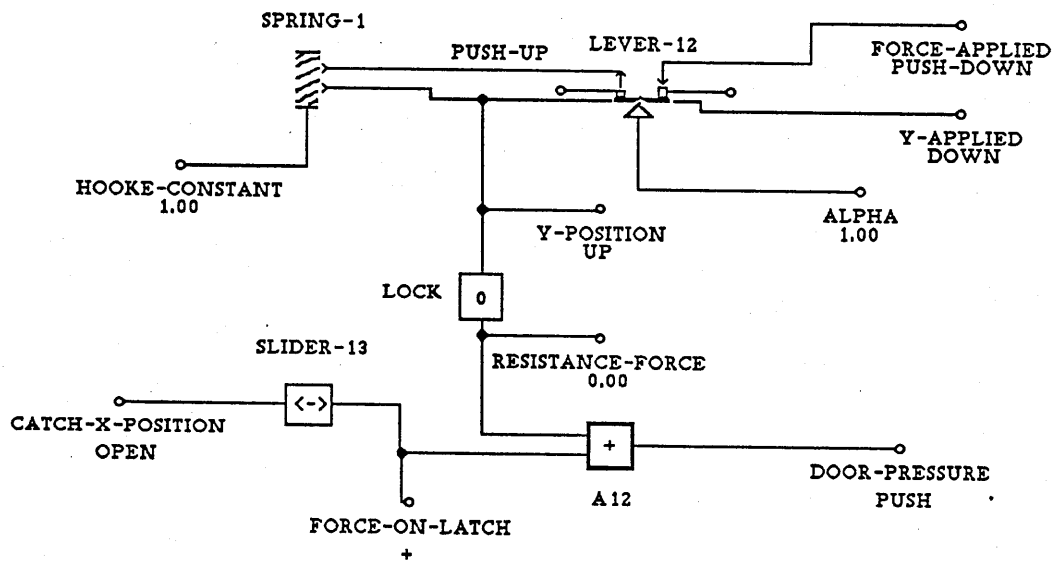


Figure 7.20. The complete constraint network for the doorlock.

The constraint for the spring is a simple version of hookes law:

```
(setf bb-f=k*x
    '(case k (1 (case x (up 'push-up)
                        (down 'push-down))))).
```

The catch mechanism is implemented as three simple constraints. The first of these is the lock constraint. When the load side of the lever is up the lock provides no resistance to the catch, and when the load side is down it provides a resistance R to the movement of the catch.

The catch itself is implemented as a simple slider, and is a constraint between two nodes: catch-x-position and force-on-latch. If the force-on-latch is + the catch-x-position is open, and if it is zero the catch remains closed. The adder A12 constrains the resistive force provided by the lock, the force on the latch and the pressure on the door. The behaviour of this adder is defined by its constraint function:

```
(setf force-on-latch=door-pressure-resistance-force
  '(case (list resistance-force door-pressure)
       ((0 0) 0)
       ((0 push) '+)
       ((0 shove) '+)
       ((r 0) 0)
       ((r push) 0)
       ((r shove) '+))))
```

This constraint essentially says the door will open if the lock is open, given the slightest push, but if the lock is closed then it requires a shove to open the door.

Figure 7.20 shows the constraint network initialized in the case where the lever has been pressed, the catch released, and the door opens. Figure 7.21 shows the worlds generated by Nostrum when the observed symptom is that the door is still closed.

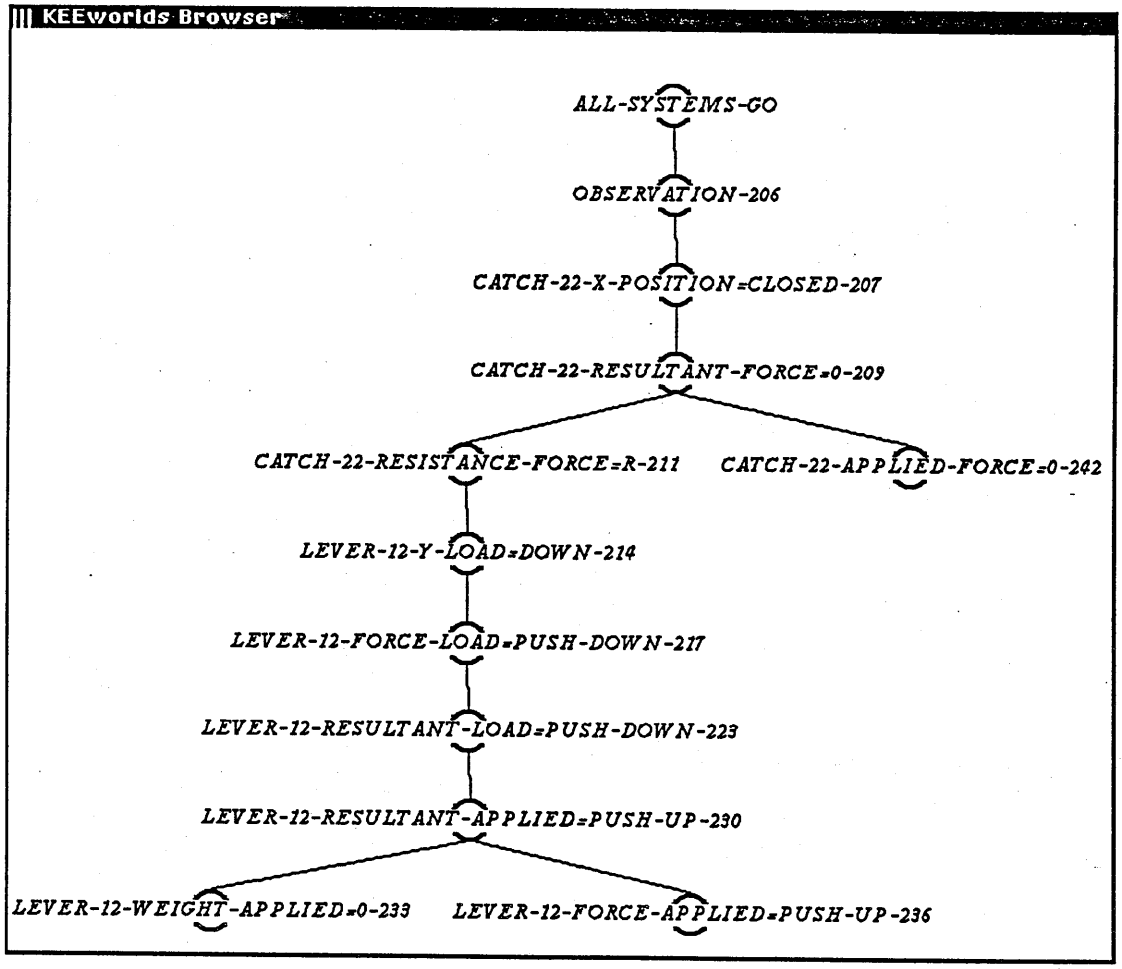


Figure 7.21. Showing worlds generated to explain why the door won't open when the catch is pressed.

Figure 7.22 shows the constraint network as it appears in the `lever-12-y-load=down-214` world. In this world the hypothesis is that the load side of the lever is still down, and so the catch is still locked. This can be tested by looking at the observable value of the `y-applied` node, the model predicts that it should be `up`.

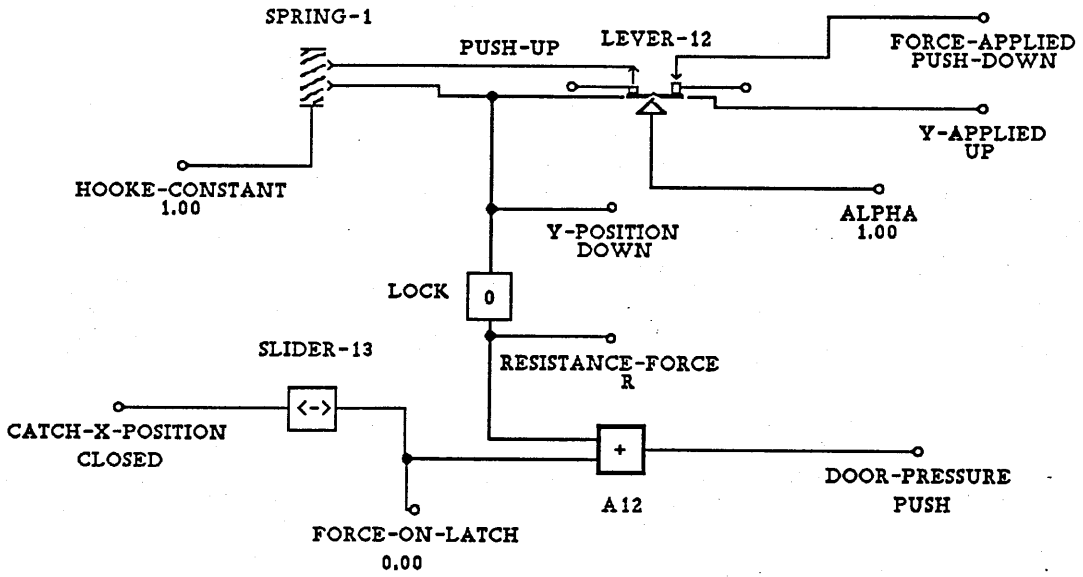


Figure 7.22. Doorlock model in the `lever-12-y-load=down-214` world.

This example demonstrates how the constraint system can behave in an almost rule-like way. The lisp functions used in the constraints mostly comprised `case` statements that can be likened to if-then rules.

The diagnostic tree produced by the example is not very satisfactory when compared to how an expert would work with a doorlock. The model requires instantaneous changes in the values of parameters, but the use of a doorlock requires feedback: as you press on the catch you increase the pressure until the doorlock 'gives', then you push the door. The trace produced by Nostrum in this example is simply a set of questions asking "are you sure that node x has value y" and as such does not constitute intelligent diagnosis. A useful extension to Nostrum would be to combine the ideas of qualitative reasoning with the constraint propagation mechanism, this would allow Nostrum to suggest things like "Press the catch harder". To do this extension would involve adding a *quantity space* to each node, and allowing the constraints to produce more than one result when ambiguities arise.

Chapter Eight

Conclusion

Chapter 3 identified 5 skills needed to do diagnosis in the absence of experiential knowledge for a particular device. This thesis has shown that the demands such skills make are met if constraints are used to model the operating principles of the components of the device. This mapping is summarized in Figure 8.1

Skill	Nostrum's solution
Fault recognition	Corresponds to the difference between behaviours expected (defined by initialization of the constraints) and observed.
Symptom recognition	This is done using the connectivity of the components defined by the structural description of Circuit Buff.
Tracing	Provided for by the causal pathways defined by the models of the operating principles.
Hypothesize and test.	The constraints can be used in a diagnostic propagation to proffer values for other nodes in the constraint network. Causal propagation of these new values can lead to easy to perform tests.
Repair Action	NOSTRUM identifies nodes of the constraint network that can explain the observed behaviour and suggests replacement or repair.

Figure 8.1. Aspects of NOSTRUM that cater for the required skills of fault diagnosis.

8.1. Summary

In this thesis I have showed how existing approaches to diagnosis have failed to use an effective model or understanding of their domain to find the causes of faults. NOSTRUM is the only system that uses a device model in conjunction with a symptom description to work backwards towards a fault, at all times predicting the consequences of proposed failures and comparing them with the real world device. The failure of other diagnostic systems is in not having a representation of devices that models the behaviour of the components. Using constraints provides NOSTRUM with such a model that allows propagation back from a symptom, and forward from proposed faults.

The example describing faults occuring within a car engine showed how Nostrum can use its knowledge of the structure of a device to suggest easy-to-perform tests that can confirm or deny its hypotheses. Also shown in that example was how the addition of experiential knowledge might be used by a search algorithm to improve the order in which Nostrum asks its questions.

8.2. Implication for Explanations

Much of the defence for the weak implementations of some early rule-based diagnostic systems has been the claim of a need for explanations. These systems have often tagged an explanation system on to the diagnostic system as a separate module. I believe that a true explanation of a diagnosis should be related to the actual inference steps that were used in making that diagnosis. Because NOSTRUM diagnoses from the symptom and directs its hypotheses according to comparisons between the predictions and the real world device it can justify its hypotheses as genuine attempts to find a cause. In fact the diagnostic strategy in NOSTRUM is can be viewed as finding an explanation for the observed symptom.

8.3. Limitations

NOSTRUM has used a system of constraints to model the way devices work. The power of the constraints lies in the definition of the functions and their inverses. The current implementation of the constraints system only allows single values to be propagated, and doesn't make any attempt to restrict the range of values that a node can take. For instance, in many applications the resistance of a device cannot be a negative number, so it would be useful to restrict the value at such a node to be a positive real. By doing this any hypothesis that tried to suggest a negative value would be rejected.

It is not easy to define inverse functions for constraints in some cases. Some functions may be what mathematicians call 'one to many' in which many input values can give the same output value. Inverting such functions causes problems because how do you know which is the real input value? One possible option is to take all values and generate hypothetical worlds for each.

The constraint system can currently simulate the operation of a device over successive states, however each of those states must be equilibrium or unchanging states. This presents NOSTRUM with difficulties in modelling concepts such as Alternating Current[1] and motion. Modelling forces in mechanical devices is a particular problem because they imply acceleration, and as levers etc give way under the force the force lessens. These are familiar problems in discussions of causality and the reader is referred to [Iwasaki and Simon 1986 (a & b), DeKleer and Brown 1986, and Morgan 1988].

---

[1]Should the user choose an appropriate representation and mathematical model, symbols representing the amplitude and phase of alternating current could be propagated through the constraint network

8.4. Future Work

NOSTRUM's strength lies in the representation and variety of its device units. Currently only a handful of device units have been represented and implemented. Future work will seek to expand this number.

As the discussion of chapter 3 highlighted, an expert's knowledge becomes rapidly modified and reorganized with experience. NOSTRUM has only made modest attempts to incorporate this learning procedure by remembering which tests it asked the user to make, but that couldn't be performed. The learning procedure would require NOSTRUM to diagnose many devices and build up some kind of picture of how often various hypotheses were successful.

When an engineer uses a piece of test equipment on a device they are changing the structure of that device. NOSTRUM should be able to adapt its model of the device to take account of the test equipment and consequently realize that it is just as prone to failure as any other parts of the equipment.

The examples of this thesis have been small domains. In larger systems the constraint network for a device would be truly enormous. To be useful in such a domain NOSTRUM would need an effective way of abstracting the behaviour of subunits. The concoctions of Circuit Buff were an original attempt to do this. In a real abstraction the behaviour of the sub-unit is viewed at a meta-level from the behaviour of the parts. Even the language used to describe the lower level components is different, for instance one considers the power-output and frequency response characteristics when buying an amplifier, and yet at the component level the designers will have considered concepts such as voltage gain, feedback and phase etc.

By definition abstraction loses some of the details of a device. When abstracting away the workings of a car engine one usually forgets about the effects of timing and carburation on engine performance, whilst

retaining the knowledge of how pressing the accelerator can make the engine turn faster. Thus an abstraction mechanism only retains the features of a device subpart that are relevant to interaction with other subparts.

Development of such an abstraction mechanism isn't too many steps away, and as already mentioned the concoction mechanism was an early attempt to do this. However the concoctions still use the constraint network used to define them to predict node values. I believe that when an expert abstracts a subpart of a device they use a new model of the behaviour that mimics at a more general level the behaviour of the detailed constraint model.

References

Bobrow, D.G. 1984 *Qualitative Reasoning about Physical Systems* Noth-Holland, also as Artificial Intelligence **24** 1984.

Breuker, J.A. and Weilinga, B.J. (1985) KADS: structured knowledge acquisition for expert systems. *Proceedings of Fifth International workshop on expert systems and their applications*, Avignon, France.

Clancey, W.J. (1985) Heuristic Classification *Artificial Intelligence* **27**, pp289-350.

Davis R. Diagnostic Reasoning Based on Structure and Behaviour, *Artificial Intelligence* **24** (1984) pp.347-410.

DeKleer, J and Brown J.S.,Qualitative Physics Based on Confluences, *Artificial Intelligence* **24** (1984) pp.7-84.

DeKleer, K. and Brown J.S., Theories of Causal Ordering, *Artificial Intelligence* **29** (1986) pp. 33-61.

DeKleer, J.; Williams, B.C., Diagnosing Multiple Faults, *Artificial Intelligence* **32** (1987) pp.97-130.

diSessa, Andrea A., Phenomenology and the Evolution of Intuition in *Mental Models*, Dedre Gentner and Albert L. Stevens Eds. Lawrence Erlbaum Associates, 1983.

Feltovich, P.J.; Johnson, P.E.; Moller, J.H.; Swanson, D.B.; (1984) LCS: the role and develpment of medical knowledge in diagnostic

expertise. In *Readings in Medical Artificial Intelligence: the first decade*(Clancey and Shortliffe, eds), pp.275-319. Addison-Wesley.

Florentin J.J., KEE Software Review. *Expert Systems* 4(2), May 1987.

Forbus K.D. Qualitative Process Theory, *Artificial Intelligence* **24** (1984) pp.85-168.

Genesereth Michael R., and Smith David E., Meta-Level Architecture, Stanford Heuristic Programming Project Memo HPP-81-6, Stanford University, May 1981.

Graf, Rudolf F., Whalen, George J. and the editors of Popular Science. (1977) How it Works: Illustrated. Sphere Books Ltd. London.

Hamilton, T.P., HELIX: A helicopter diagnostic system based on qualitative physics, *Artificial Intelligence in Engineering* **3**(3) (1988) pp.141-150.

Hunt, J. (1989) A Qualitative Diagnostician for Mechanical Devices. in Engineering Applications of Artificial Intelligence (in press).

Iwasaki, Y. and Simon, H.A., Causality in Device Behaviour, *Artificial Intelligence* **29** (1986a) pp. 3-32.

Iwasaki, Y. and Simon, H.A., Theories of Causal Ordering: Reply to DeKleer and Brown, *Artificial Intelligence* **29** (1986b) pp. 63-72.

Keravnou, E.T. and Johnson L. Competent Expert Systems: A case study in Fault Diagnosis (1986) Kogan Page Ltd, London.

Kuipers B. Commonsense Reasoning about Causality: Deriving Behaviour from Structure, *Artificial Intelligence* **24** (1984) pp.169-204.

Kuipers, Benjamin and Chiu, Charles. Taming Intractible Branching In Qualitative Simulation. (1987) in IJCAI 87 Proceedings of the Tenth Internation Joint Conference on Artificial Intelligence, pp1079-1085.

Meccano Mechanisms Set Instructions, (1971) Meccano Tri-ang Ltd., Binns Road, Liverpool L13 1DA England.

Minsky, M.L., (1975). A Framework for Representing Knowledge, pp 211-280 in *The Psychology of Computer Vision*, ed P.H.Winston, McGraw Hill.

Morgan, A.J. 1988 The Qualitative Behvaiour of Dynamic Physical Systems. PhD Dissertation Wolfson College, Cambridge.

Price, C. J. (1989) Developing a Qualtitative Representation of Mechanical Devices for use in Diagnosis in Engineering Applications of Artificial Intelligence **1**(2).

Raiman O. (1986) Order of Magnitude Reasoning. Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86), pp.100-4.

Roth, J.P., Diagnosis of automata failures: A calculus and a method, *IBM J. Res. Develop.* **10** (1966) 278-291.

Schreiber, G., Bredeweg, B., Davoodi, M., Wielinga, B. 1987 *Towards a Design Methodology for KBS*. VF Mmo 97 November 1987, Department of Social Science Informatics, University of Amsterdam.

Shortliffe, E. H. (1976) *Computer-Based Medical Consultations: MYCIN*. New York: Elsevier.

Stefik, M.J., Planning with Constraints (MOLGEN: Part 1), *Artificial Intelligence* **16**(2) (1981) 111-140.

Struss, P. Mathematical Aspects of Qualitative Reasoning in *International Journal of Artificial Intelligence in Engineering* (1988)

Weinreb, D. and Moon, D. The Lisp Machine Manual, 1981.

Weld, D.S. (1988) Choices for Comparative Analysis: DQ Analysis or Exaggeration? *Artificial Intelligence in Engingeering* **3**(3) p.174.

Willoughby A. A., Making Qualitative Reasoning more Quantitative. in Proce 9th International workshop on Expert Systems and their Applications, Specialized Conference on 2nd Generation Expert Systems, Avignon, France. June 1989 pp117-127.

REPAIRED