Diplomarbeit

AD^2L

An Abstract Modelling Language For Diagnosis Systems

ausgeführt am Institut für
Informationssysteme,
Abteilung Datenbanken und Expertensysteme,
der Technischen Universität Wien

unter Anleitung von
Ao. Univ. Prof. Dipl. -Ing. Dr. techn. Markus Stumptner
und
Dipl. -Ing. Dr. techn. Franz Wotawa
als verantwortlich mitwirkendem
Universitätsassistenten

durch
Christian Piccardi
1100 Wien, Stockholmerplatz 7/2

Wien, am

for my mother,
in love and gratitude

for my grandmother,
who is always with me in my heart and my memories

Preface

During the last two decades many researchers in the field of Artificial Intelligence have been concerned with the development of powerful diagnosis systems. This is not an accident since diagnostic reasoning is a very common task in everyday life. Consider technicians trying to identify the broken parts of a malfunctioning device or physicians trying to determine a patient's disease given the symptoms. In a more general way we can say that given a set of observations describing how the system under consideration actually behaves, the main task of diagnostic reasoning is to determine which parts of the system are not working properly and are thus responsible for the system to show incorrect behaviour. Two major approaches to automated diagnosis can be distinguished. Heuristic diagnosis is based on the knowledge and experience of human domain experts, whereas model-based diagnosis uses an explicit model of the system to be analysed, thus allowing to predict the system's behaviour.

In model-based diagnosis and in other areas, a well structured design methodology is generally preferred. Instead of immediately implementing the accumulated and analysed knowledge in a given formalism according to the rapid-prototyping approach, a system and implementation language independent formalism should be found. This would enable the knowledge engineer to concentrate on the design of the knowledge base, without having to consider the influence of implementational details on aspects of the gathered knowledge.

Another consideration that favours the development of a more abstract formalism is the application of mobile diagnosis. Since not everyone can afford an expensive diagnosis system many technicians have to rely on the services provided by dedicated servers. It would be very inefficient for the clients having to use different modelling languages depending on the server they use, since adapting to new programming paradigms can be a very time consuming and thus costly

additional expenditure. As a consequent the introduction of a commonly agreed modelling language would make the application of mobile diagnosis systems more attractive allowing the clients to choose the server that offers best service and conditions, without having to invest additional time and money.

The first part of this thesis is concerned with the definition of an implementation independent language called AD^{2}L (**A**bstract **D**omain **D**efinition **L**anguage) that allows for efficient system modelling and that captures the semantics of most of the currently used modelling languages for diagnosis systems. In a second step we will show how models defined in this language can be translated to equivalent models for constraint systems. Finally we will show that for a subset of AD^{2}L a mapping to a set of corresponding Horn-clauses can be found.

Contents

1 Introduction **1**

1.1 Diagnosing Digital Circuits 2

1.2 Bayesian Networks 3

2 Basic Definitions **6**

2.1 The System 7

2.1.1 The Components 8

2.1.2 The System Description 8

2.1.3 Extensions 9

2.2 The Observation 10

2.3 The Diagnosis 11

2.3.1 Consistency-Based Diagnosis 11

2.3.2 Abduction-Based Diagnosis 12

2.4 Complexity Considerations 13

2.5 The DIAGNOSE algorithm 14

2.6 Alternative Algorithms 15

2.7 Probing 17

2.8 Summary 18

2.9 Context-free Grammars 20

2.10 Alphabet 21

2.11 Fundamental Categories 21

2.12 Terms 22

2.13 Sentences 23

2.14 Rules 23

2.15 Definitions 24

2.15.1 Type Definitions 24

2.15.2 Function Definitions 25
2.15.3 Port Definitions 26
2.15.4 Variable Definitions 26
2.15.5 Axiom Definitions 27
2.15.6 Behaviour Definitions 27
2.15.7 Subcomponent Definitions 28
2.15.8 Connection Definitions 28
2.15.9 Component Definitions 29
2.15.10 Observation Definitions and Diagnoses 32
2.16 Summary 33

3 Conceptualisation **34**
3.1 Domain and Objects 34
3.2 Functions and Relations 35

4 Semantics **37**
4.1 Interpretation 38
4.2 Term semantics 38
4.3 Truth Value 39
4.4 Rules 40

5 From AD^{2}L to Constraint Systems **41**
5.1 Structure of a Constraint System 42
5.1.1 Flat Systems 42
5.1.2 Hierarchical Systems 43
5.1.3 Basic Operations 43
5.1.4 Behaviour Definition 45
5.1.5 The Propagation Algorithm 46
5.2 Translation on the Structural Level 46
5.2.1 Atomic Constraints 47
5.2.2 Hierarchical Constraints 48
5.3 Translation on the Behavioural Level 49
5.3.1 Predicate Sentences 50
5.3.2 Existentially Quantified Sentences 50
5.3.3 Universally Quantified Sentences 51

5.3.4 Conjunctive Sentences 52
5.3.5 Implication Sentences 53
5.3.6 Rules 55
5.4 Creating an Example Constraint 57
5.4.1 Logical Conjunction 57
5.4.2 Logical Disjunction 58
5.4.3 Logical Negation 60
5.4.4 Half Adder 61

6 Complexity **63**
6.1 Predicate Sentences 63
6.2 Existentially Quantified Sentences 64
6.3 Universally Quantified Sentences 64
6.4 Conjunctive Sentences 65
6.5 Implication Sentences 65
6.6 Rules 65

7 Translation to Horn Clauses **67**
7.1 Complexity 69
7.2 Summary 70

8 Discussion and Conclusion **71**
8.1 Comparison to Related Work 72
8.2 Future Work 74

List of Figures

1.1 full adder 3
1.2 A small subtree of a medical Bayesian network 4

2.1 A possible HS-DAG for the full adder example. 16

5.1 Logical nand-gate 42
5.2 Constraint system representation of a logical nand-gate 43
5.3 a composed constraint for a logical nand-gate 44
5.4 a composed constraint for a logical half adder 62

8.1 The four layer model of expertise 73

List of Tables

2.1 Alphabet 21

5.1 The propagation algorithm PROPAGATE 46

7.1 The resulting set of sentences for the translation of a rule formulated in AD^2L_{hc} to the abstract constraint system ACS. 69

7.2 The resulting set of sentences for the translation of a rule formulated in AD^2L_{hc} to Horn clauses. 70

Chapter 1

Introduction

> Come, let us go down, and there confound their language, that they may not understand one another's speech!
>
> — Genesis, 11:7 —

The great success of diagnostic reasoning in various domains has led to a multitude of sophisticated and powerful diagnosing systems in the past. Unfortunately most of these systems use languages for specifying the domain knowledge, which are influenced by the implementation language of the system, making the interchange of information a costly and time consuming task. The languages used in current systems cover the whole spectrum of programming paradigms having object-oriented and procedural semantics like Smalltalk and C or functional and declarative semantics like LISP and Prolog. Beside the time consuming translation from one representation to another the main drawback is the influence of a certain language on the way the domain knowledge has to be formulated. During the construction of a system model, the knowledge engineer always has to consider the influence of implementational details implied by the chosen language thus not allowing him to concentrate solely on the design of the knowledge base.

The primary concern of this thesis is the definition of an implementation independent language called AD^{2}L (**A**bstract **D**omain **D**efinition **L**anguage) that allows for efficient system modelling and avoids the above mentioned drawbacks. In a second step we will show how models defined in this language can be translated to equivalent models for constraint systems. In a last step we will show

that an *implication-free* subset of AD^2L can be mapped to a corresponding set of Horn-clauses.

To get a better understanding of the context of this thesis and to provide an intuitive approach to the field of automated diagnosis, some common scenarios are chosen and discussed from a knowledge engineer's point of view.

1.1 Diagnosing Digital Circuits

One of the most important example domains of diagnostic reasoning in the past, was finding faulty parts in digital circuits.

Let us take the full adder shown in figure 1.1 which is taken from [1] as an example. Our model is supposed to contain structural and behavioural knowledge, determining which components are connected and what function they implement. The dark squares represent the digital components which interact through connecting arcs. Their behaviour is defined as giving the result of the logical And, Or and Xor as determined by the label of each component.

Consider now the following observation of output and input ports of the system that obviously leads to a contradiction compared to the way in which we expect the system to behave:

$$I_1 = 1, I_2 = 0, I_3 = 1, O_1 = 1, O_2 = 0$$

There are five components that could be responsible for the faulty behaviour. A very simple approach to diagnosis would be to consider every component as faulty, which would of course explain the incorrect behaviour. As a result in a real world situation we would have to substitute each component with a new one, which obviously is not very desirable. In real applications experience shows that the malfunction of few components is far more probable than the malfunction of many or even all, indicating only a very small correlation between component failures. Therefore we are interested in finding only minimal sets of components which, when considered faulty, explain the discrepancy between expected and observed behaviour of the system.

In our full adder example we would therefore start with assuming a single component as faulty and check whether the misbehaviour can now be explained. The only single component which could explain the wrong values at both output

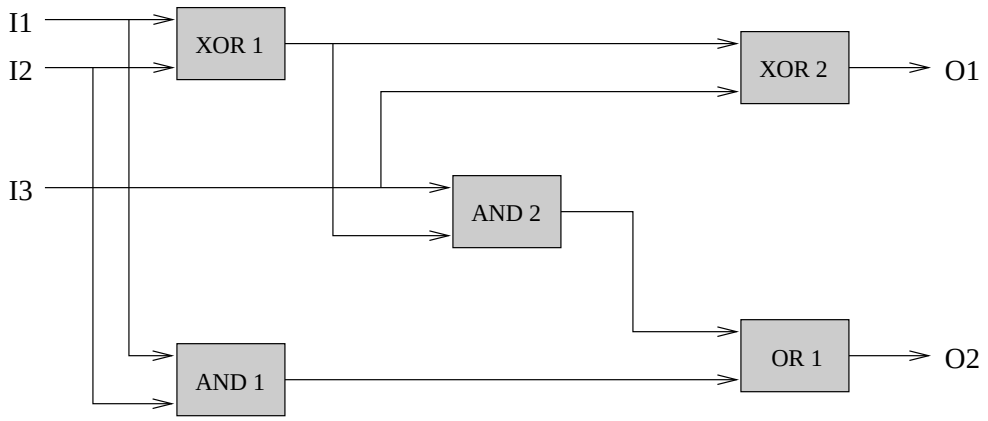


Figure 1.1: full adder

ports is the gate Xor1. If we are also interested in diagnoses with two faulty components we would get the sets {Xor2, Or1} and {Xor2, And2}. All further diagnoses would contain as a subset one of the above found diagnoses and thus are not minimal. In order to reduce the number of diagnoses found we could measure the values along other connections, such as the connection between Xor1 and Xor2 and finally arrive at the true cause of the malfunction.

We will discuss this example in more detail in chapter 2 which elaborates on the basic definitions of model-based diagnosis.

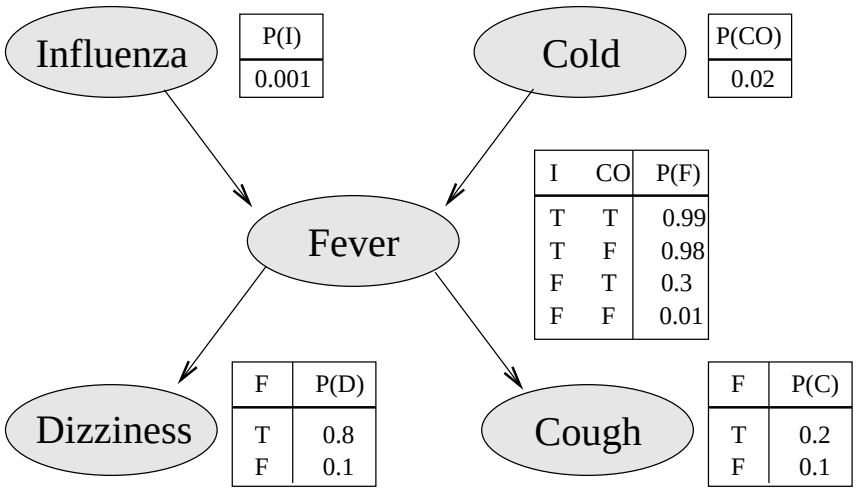
1.2 Bayesian Networks

Another way to model knowledge are Bayesian Networks [1], which are used to capture uncertain knowledge in a natural and efficient way. They represent the dependence between variables and give a concise specification of the joint probability distribution[2]. The main advantage of this approach over logical reasoning is that rational decisions can be made even when there is not enough information to prove that this decision is correct in the current context.

A Bayesian network is a graph with the following properties:

[1] In the literature Bayesian Networks are also often referred to as **belief networks, causal networks** or **probabilistic networks**

[2] The **joint probability distribution** determines the probability of each complete assignment of values to random variables



The abbreviations I, CO, F, D, C stand for Influenza, Cold, Fever, Dizziness and Cough, respectively.

Figure 1.2: A small subtree of a medical Bayesian network

- The nodes of the network are a set of random variables.
- Nodes are connected by a set of directed links. The intuitive meaning of a directed link from node X to node Y is that X has direct influence on Y.
- A conditional probability table is associated with each node, determining the effects the parents have on the node. The parents of a node are all those nodes that have a directed link pointing to it.
- The graph is acyclic and thus has no directed cycles.

Figure 1.2 illustrates a typical Bayesian network for medical purposes. Each node is augmented with a conditional probability table. We see, for example, that a person who has a cold and no influenza will also have fever with a probability of 0.3.

The basic task for the probabilistic inference system is to compute the posterior probability distribution for a set of **query variables**, given exact values for some **evidence variables**. That is, the system has to compute $\mathbf{P}(\text{Query}|\text{Evidence})$.

To calculate $P(X_1 = x_1 \wedge X_2 = x_2 \wedge \ldots \wedge X_n = x_n)$ we use the following

formula:

$$P(X_1 = x_1 \wedge X_2 = x_2 \wedge \ldots \wedge X_n = x_n) = \prod_{i=1}^{n} P(x_i | Parents(X_i)$$

Example 1.2.1 If we were interested in the probability of someone having influenza and fever, but no cough, we would calculate:

$$\begin{aligned} & P(I, \neg CO, F, \neg C)^3 \\ & = P(I) * P(\neg CO) * P(F | I \wedge \neg CO) * P(C) \\ & = 0.001 * 0.98 * 0.98 * 0.8 = 0.00076832 \end{aligned}$$

Bayesian networks are a very powerful formalism and are flexible enough so that any node in a network can serve as either a query or an evidence variable.

For a detailed discussion on uncertain knowledge and reasoning see [12].

[3] Abbreviation for P(I=True, CO=False, F=True, C=False)

Chapter 2

Basic Definitions

> A theory has only the alternative of being right or wrong. A model has a third possibility: it may be right, but irrelevant.
>
> — Manfred Eigen —

In the previous chapter, the importance of diagnostic reasoning in the field of AI was mentioned.

Two major approaches to diagnosis can be distinguished in the literature: Heuristic diagnosis and model-based diagnosis. The first uses heuristic information to determine the faulty components in a system. The information is based on statistical explorations and expert knowledge which is coded into rules that usually associate symptoms with the corresponding causes. The main disadvantage of this approach is the time consuming effort to gather expert knowledge and the effort to maintain large rule databases. Typical application fields of heuristic diagnosis are medical domains, such as the diagnosis and therapy of blood infection diseases by the system MYCIN [3].

The model-based approach, which is also often referred to as "diagnosis from first principles" emphasises the use of a structural and behavioural description of the system under consideration. The fundamental assumption in model-based diagnosis states that the model explicitly describes the intended behaviour of the system under consideration. As can easily be seen this is wrong in all cases, since the knowledge engineer has to restrict himself to relevant aspects of the system in order to keep the model complexity small. If for example we were interested

in diagnosing digital circuits, we would definitely not need to know about the quantum mechanical effects in the single gates at this level of abstraction. In general, if the granularity of the homomorphism from systems to models is high enough and the relevant aspects are captured, we get a good approximation of the system and the model-based approach proves to be very powerful. The difficulties that arise lie in finding out which aspects of the system have to be modelled and which can be neglected in order to still get an adequate system description. There are two approaches in model-based diagnosis concerning the behavioural description of a system. In the first approach, the correct behaviour at a certain level of abstraction is specified, determining how the system normally works, whereas in the second approach the faulty behaviour is specified, determining how the system works when failures occur.

In the remainder of this chapter we will introduce the fundamental concepts of model-based diagnosis using the terminology and notation developed by Reiter [1] in his excellent work about "diagnosis from first principles". We use first-order language to define the structure and correct behaviour of the system under consideration, where predicates, functions and constants begin with upper-case letters and variables begin with lower-case letters. All Variables are assumed to be universally quantified. To illustrate the discussed concepts we will use the full adder shown in figure 1.1.

2.1 The System

As we saw in chapter 1, there are many possible domains for the application of model-based reasoning like medicine and electrical engineering. Possible systems under consideration could be a human, a part of a human such as his liver or lungs, digital or analog circuits, or even a whole nuclear plant.

The core of a model-based diagnosis system is an adequate description of the domain under consideration. This description is a composable model of the system, where systems characteristics can be derived by the constituting components characteristics, thus obeying the law "no function in structure".

The model allows us to make predictions about the expected behaviour and to compare it to the observed behaviour. Any discrepancies that arise indicate that the system is not working properly. The model can then be used to find an

explanation for this behaviour by identifying the malfunctioning parts.

Since model-based diagnosis is used for many different domains, we give a very general definition of what constitutes a system.

Definition 2.1.1 A system is a pair (SD, COMPS), where:
(1) SD, the system description, is a set of first-order sentences
(2) COMPS, containing the system components, is a finite set of constants

2.1.1 The Components

The first step in defining a model is to identify the relevant components at a certain level of abstraction. Referring to our full adder example, we will model the system at gate level, thus neglecting the transistors the gates are made of, and instead use the logical And, Or and Xor gates as our atomic components.

Example 2.1.1 Components of the full adder:

$$COMPS = \{Xor_1, Xor_2, And_1, And_2, Or_1\}$$

2.1.2 The System Description

The more demanding part during the process of defining a model is to specify how the identified components behave and interact. The system description contains axioms describing the structure and component behaviour. We can separate this description into two parts: the *structural model* and the *behavioural model*.

The first is concerned with the topology of the system, describing which components are connected to each other.

Example 2.1.2 Let us take a look at our full adder again and identify the set of axioms determining the structural model. The binary predicate =[1] is used to describe the topology of the adder, which is defined by the components and the connections between their ports.

$AndGate(And_1), AndGate(And_2), XorGate(Xor_1), XorGate(Xor_2), OrGate(Or_1),$
$Out(Xor_1) = In2(And_2),$

[1]Some authors use the predicate $Conn/2$ in place of the predicate "$= /2$" in infix notation and write $Conn(Out_n(X), In_m(Y))$ instead of $Out_n(X) = In_m(Y)$

$Out(Xor_1) = In1(Xor_2)$,
$Out(And_2) = In1(Or_1)$,
$In1(And_2) = In2(Xor_2)$,
$In1(Xor_1) = In1(And_1)$,
$In2(Xor_1) = In2(And_1)$,
$Out(And_1) = In2(Or_1)$,
$In1(Xor_1) = 0 \vee In1(Xor_1) = 1$,
$In2(Xor_1) = 0 \vee In2(Xor_1) = 1$,
$In1(And_2) = 0 \vee In1(And_2) = 1$.

The behaviour a component exhibits in a certain state or mode is specified by its corresponding behavioural model. This is usually a collection of rules of the form "if ϕ then ψ" where ϕ and ψ are logical formulas. The behaviour of the whole system is defined by the behaviour of all its components.

Example 2.1.3 Reconsidering our full adder from this point of view, we get the following first order formulas specifying the behavioural model of each component:

$AndGate(x) \wedge \neg AB(x) \rightarrow out(x) = and(in1(x), in2(x))$,
$XorGate(x) \wedge \neg AB(x) \rightarrow out(x) = xor(in1(x), in2(x))$,
$OrGate(x) \wedge \neg AB(x) \rightarrow out(x) = or(in1(x), in2(x))$,

The negation of the predicate $AB/1$,which stands for abnormal, is used to determine how the components behave during normal operation.

2.1.3 Extensions

There are also some possible extensions to the system description that deviate from Reiters basic approach. The first one is to include **probabilistic information** for each possible mode of a component. This is very similar to humans taking advantage of knowledge about how likely it is for a component to exhibit incorrect behaviour.

A second approach as proposed by Gottlob et al. in [4] is to formulate **physical impossibilities**, which enables the reasoning system to remove candidate diagnoses that are judged physically impossible.

Example 2.1.4 For example, we could formulate the following sentence:

$$\neg(Pipe(p) \wedge Closed(p) \wedge Val(Out(p), 5l/sec))$$

expressing that it is impossible to transport 5 litres of a fluid per second through a closed pipe.

Should we, during our search, arrive at a candidate diagnosis that includes the formula in the above example we can drop it right away and do not need to pursue it anymore.

A third approach as proposed by Dressler et al. in [5] is to include **Fault Models** which represents knowledge about how components behave when malfunctioning. To specify the abnormal behaviour, we would introduce new predicates (e.g. StuckAtZero, StuckAtOne, ...) and formulate the corresponding formulas.

Example 2.1.5 For the StuckAtZero mode of any gate the behaviour model would be:

$$Gate(x) \wedge StuckAtZero(x) \rightarrow out(x) = 0$$

2.2 The Observation

During operation, a system can deviate from the expected behaviour. In order to determine where the origin of the failure lies, we need an observation of the system that illustrates how the failure manifests itself. Given this observation we can then use the system model to infer which components are responsible for the failure.

Definition 2.2.1 An observation OBS of a system is a finite set of first-order sentences. For a system (SD, COMPS) with corresponding observation OBS we write (SD, COMPS, OBS) and call it a diagnostic problem.

Example 2.2.1 An example for an observation leading to a discrepancy with the system model of the full adder is the following:

$$in1(Xor_1) = 1,$$
$$in2(Xor_1) = 0,$$

$in1(And_2) = 1,$
$out(Xor_2) = 1,$
$out(Or_1) = 0.$

2.3 The Diagnosis

Once we have found a discrepancy between the system model and an observation, we are interested in the source of the failure. The purpose of the diagnostic process is to find out all components that, when considered as malfunctioning, can explain the incorrect behaviour. In order to achieve this we have to retract the normal mode behaviour assumption of these components until we arrive at an explanation for the system behaviour. In the following part we will discuss the two approaches for finding a diagnosis.

2.3.1 Consistency-Based Diagnosis

Definition 2.3.1 A consistency-based diagnosis[2] for a diagnostic problem (SD, COMPS, OBS) is a minimal set $\Delta \subseteq COMPS$ such that

$$SD \cup OBS \cup \{AB(c)|c \in \Delta\} \cup \{\neg AB(c)|c \in COMPONENTS - \Delta\}$$

is consistent.

This definition states, that in order to find a diagnosis we have to find a minimal set of components which we assume as faulty, such that together with the set of components for which the correct behaviour mode assumption holds the system description is consistent with the observation. If one abnormal component suffices to explain the behaviour, we call it a *single-fault* diagnosis otherwise a *multiple-fault diagnosis*. The need for a minimal set Δ has its origin in the use of the principle of *parsimony* introduced by Reiter: the correlation of component failures has been proven to be very small in practice, thus only a minimal set of components should be considered faulty, since components are assumed to fail independently.

Example 2.3.1 Remembering the observation in example 2.2.1, we get the following minimal diagnosis for our full adder:

[2]Reiter used this definition in his paper about diagnosis from first principles

$\Delta_1 = \{XOR_1\}$
$\Delta_2 = \{XOR_2, OR_1\}$
$\Delta_3 = \{XOR_2, AND_2\}$

As can easily be seen, each superset of the above diagnosis Δ_i is consistent as well.

A consistency-based diagnosis provides a *weak form of explanation*. That is, a diagnosis is an explanation for an observation if it does not lead to a contradiction. We do not know how the faulty components behave, we just know that they are not working properly.

2.3.2 Abduction-Based Diagnosis

In the abduction based approach, each component $c \in COMPS$ has associated a set of behavioural modes $B_1, \ldots, B_n$ describing the correct as well as the faulty behaviour. Given an observation leading to a discrepancy the aim is to find a minimal set of behavioural mode assumptions, such that these assumptions together with the system description logically entail the observation and are consistent with the system description and the observations.

Definition 2.3.2 An abduction-based diagnosis for a diagnostic problem (SD, COMPS, OBS) is a minimal set Δ of behavioural mode assumptions such that

1. $SD \cup \Delta \models OBS$
2. $SD \cup \Delta \cup OBS$ is consistent.

An abduction based diagnosis provides a *strong form of explanation*, since it directly supports the observation. The drawback of this approach is the large domain knowledge required. If not all failure models are known, the system could fail in generating all diagnoses. In domains like medicine where the faulty behaviour is simpler to describe than the correct behaviour this approach may still be very useful. Several efforts have been made to overcome some of the difficulties that arise with this approach such as integrating consistency-based and abduction-based diagnosis. For more information on this topic the interested reader should refer to [6].

Example 2.3.2 Referring again to our full adder we would get as an abduction-based diagnosis the following mode assumptions:

$$\Delta = \{StuckAtOne(Xor_2), StuckAtZero(Or_1)\}$$

2.4 Complexity Considerations

Now that we have given a definition for a diagnosis of a diagnostic problem (SD, COMPS, OBS) we can start to test a set of components for which we assume incorrect behaviour and check whether the inconsistency due to the observation can be explained. If it cannot be explained, we try the same for another set of components. For the consistency approach, we would check for each $\Delta \subseteq COMPS$, if

$$SD \cup OBS \cup \{AB(c)|c \in \Delta\} \cup \{\neg AB(c)|c \in COMPS - \Delta\}$$

is consistent.

Obviously, there are $2^{|COMPS|}$ different possible sets of components in the context of consistency-based diagnosis and even more if we introduce additional failure models, which renders computing all diagnoses an NP-complete problem.

One way to reduce the complexity is to allow only diagnoses with small cardinality. Given n components and restricting ourselves to diagnoses with cardinality m we get the following number C of different candidates:

$$C = \sum_{k=1}^{m} \binom{n}{k}$$

stating that there are for example n single-fault diagnoses.

Another way to reduce the time complexity would be to "reverse" the diagnostic process and not to start by assuming all components as working correctly, but by considering them as being faulty. In the following steps we would then remove a component from the set Δ and consider it as working correctly. We repeat this procedure as long as no inconsistency occurs and finally arrive at a diagnosis for our diagnostic problem. Given n components we would have to remove $n-1$ components in the worst case to reach a correct diagnosis. Obviously the time complexity of this approach is bounded by $O(nC_{check})$ where C_{check} is the time

needed to check the consistency of the set of components assumed as faulty, the set of components assumed as working correctly and the system description.

A more sophisticated approach to reduce the average time complexity was developed by Reiter. Due to its importance for the field of model-based diagnosis the next section is dedicated to the introduction of this algorithm in more detail.

2.5 The DIAGNOSE algorithm

In his article Reiter presented an approach based on the concept of hitting sets in order to find correct diagnoses and to reduce time complexity. To explain the idea underlying a hitting set we first have to introduce the definition of a conflict set.

Definition 2.5.1 A conflict set for a diagnostic problem (SD, COMPONENTS, OBS) is a set $\{c_1, ..., c_k\} \subseteq COMPONENTS$ such that

$$SD \cup OBS \cup \{\neg AB(c_1), ..., \neg AB(c_n)\}$$

is inconsistent. If C is a conflict set for (SD, COMPONENTS, OBS) and no proper subset C^* of C is a conflict set for (SD, COMPONENTS, OBS) then C is called a minimal conflict set.

In other words, a conflict set contains at least one component that leads to an inconsistency if we assume all to work correctly.

Definition 2.5.2 Let C be a collection of sets. A hitting set H for C is a set such that: $H \subseteq \bigcup_{S \in C} S$ and $H \cap S \neq \{\}$ for each $S \in C$. If no proper subset H^* of H is a hitting set of C then H is called a minimal hitting set.

Put in words, a hitting set for a collection C of sets is defined as a set that contains at least one element from each set in C.

Now that we have defined the basic concepts we can proceed to the fundamental theorem underlying the algorithm.

Theorem 2.5.1 A set $\Delta \subseteq COMPONENTS$ is a diagnosis for a diagnostic problem (SD, COMPONENTS, OBS) iff Δ is a minimal hitting set of all (minimal) conflict sets for (SD, COMPONENTS, OBS).

Example 2.5.1 Considering our full adder and the corresponding observation we get the following collection C^* of conflict sets:

$$C^* = \{(Xor_1, Xor_2), (Xor_1, And_2, Or_1)\}$$

resulting in the collection D^* of minimal hitting sets which as stated by the above theorem are our diagnoses.

$$D^* = \{(Xor_1), (Xor_2, And_2), (Xor_2, Or_1)\}$$

Intuitively the algorithm does the following: Let CS be the ordered set of all minimal conflict sets and HS the empty set. Now put the first element of the first conflict set into HS, and check for all following conflict sets if they contain at least one element of HS, in other words if HS is already a hitting set. If this is not the case for a certain conflict set, then take its first element and add it to HS. This procedure is repeated until HS contains at least one element of each conflict set. Obviously HS is now a hitting set and if it is minimal, then it is also a diagnosis. Instead of adding always the first element of a conflict set to HS, we could also choose an arbitrary element thus producing a different hitting set.

For the computation of conflict sets Reiter uses a general theorem prover to find refutations for formulas like $\neg AB(x_i) \wedge \ldots \wedge \neg AB(x_j)$ by means of resolution.

In order to find the corresponding hitting sets Reiter introduced the concept of a HS-DAG (see figure 2.5.1 for an example) and applied several pruning rules to keep the branching complexity as small as possible. For a detailed introduction to HS-DAGs see [1]. For a correction to Reiters algorithm and a proof that every minimal hitting set can be found in a HS-DAG see [7].

2.6 Alternative Algorithms

Some other diagnosis algorithms worth mentioning are the system MOMO by Friedrich and Nejdl [8] developed at the technical university of Vienna, the DP framework developed by Struss [9] and the GDE (General Diagnostic Engine) system by de Kleer and William [10].

The core of MOMO (MOdel-based diagnosis by MOdel-generation) for example is implemented by a small hyperresolution-based procedure using prolog. A diagnosis corresponds to a consistent logical model of the theory consisting of the

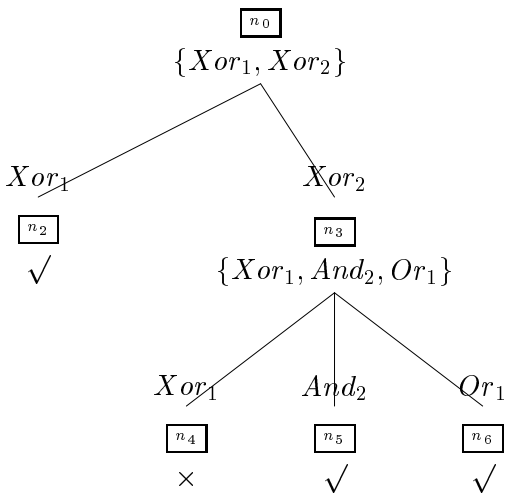


Figure 2.1: A possible HS-DAG for the full adder example.
Collecting the elements at the arcs when going from the root n_0 of the DAG to a leaf labelled $\surd$ results in a minimal hitting set and thus a diagnosis

system description and the observations. In contrast to Reiter who uses conflict sets to generate the diagnosis candidates as hitting sets, diagnoses in MOMO are generated in a direct fashion as subset of the consistent logical models. The full MOMO system includes measurement selection functions and user interface extensions and has mainly been used to develop and test several ideas in the context of model-based reasoning.

The system GDE is a more sophisticated method which exploits the behaviour of the components as well as failure probabilities using an assumption-based truth maintenance system (ATMS). GDE is also based on the computation of minimal conflict sets and hitting sets as proposed by Reiter, even if a different terminology is used.

Still, there are some differences.

- Failure candidates are represented in terms of minimal sets of violated assumptions.
- The diagnostic process is incremental, taking advantage of the iterative nature of diagnosis.
- A clear separation is drawn between diagnosis and behaviour prediction,

resulting in a domain independent diagnostic procedure.

- The system combines model-based prediction with sequential diagnosis to propose measurements to localise the faults.

The basic principle underlying the DP (Diagnosis as a Process) Framework is to explicitly represent assumptions and choices that drive the diagnostic process in the same way as behavioural mode assumptions of components. Struss uses the term "working hypotheses" to denote diagnostic assumptions. Examples for such working hypotheses are the single fault hypothesis, exhaustive knowledge about fault modes, or other assumptions about a certain model currently used. During the diagnosis process, additional strategies are required to guide the revisions of the working hypothesis. One of these strategies is to first consider single-fault diagnosis and then move to diagnosis with higher cardinality. In other words, DP is a framework where diagnostic and behavioural assumptions are revised uniformly during the diagnostic process.

For more and detailed information on certain algorithms and systems the reader should refer to [11].

2.7 Probing

Remembering the full adder example we identified three different diagnoses. Which should we now consider as being the true cause for the incorrect behaviour of our system? Obviously we cannot start any repair actions until we know exactly which components are responsible for the failure. What we have to know in order to be able to discriminate among the different diagnoses is to gather additional knowledge.

This process of looking for discriminating information is called *probing* and is widely applied in diagnostic reasoning systems. Since additional measurements are expensive in terms of time and money, we are interested in a minimum number of subsequent measurements needed to identify the faulty components.

Example 2.7.1 Given the following observation for the full adder

$in1(Xor_1) = 1,$
$in2(Xor_1) = 0,$

$in1(And_2) = 1,$
$out(Xor_2) = 1,$
$out(Or_1) = 0.$

we got the corresponding set D^* containing the three possible diagnoses.

$$D^* = \{(Xor_1), (Xor_2, And_2), (Xor_2, Or_1)\}$$

One of the diagnoses assumes the single gate Xor_1 to be responsible for the incorrect behaviour. In order to support or refute this assumption, we need an additional measurement at the output port of this gate. Should the value at the output port be 1, then the single failure assumption is not justified, since the component does what it is expected to do. Should on the other hand the value be 0 then obviously the component shows incorrect behaviour and is identified as being faulty. The same procedure is applied to the other diagnoses as well until the true cause of the malfunction is found.

Several strategies for the selection of measurements can be found in the literature. One of the simpler techniques is called *guided probing*, starting at the incorrect output of the system and moving back along causal connections as specified in the system description, until a faulty component is identified.

2.8 Summary

Traditionally the core of a model-based diagnosis approach are models of the structure and the behaviour of the system under consideration. Based on these models, predictions can be made about how the system should work. Discrepancies between the predicted behaviour and the actual behaviour indicate that some parts are faulty providing information for the identification of the faulty components. One of the key concerns is to find good models which contain appropriate simplifying assumptions without omitting necessary facts. A main distinction can be drawn between models describing the correct behaviour and models describing the faulty behaviour. Two different logical characterisations providing a different form of explanation have been worked out: consistency-based diagnosis and abduction-based diagnosis. In the first approach a diagnosis explains an observation if it does not contradict it, in the the second approach a diagnosis explains

an observation if it directly supports it. Computing diagnosis can be considered to be a non-monotonic process, where assumptions about the functioning of components are revised as new observations are available. Several techniques have been developed to reduce the exponential complexity of the diagnostic process. Methods to distinguish among competing hypotheses are very important. We discussed the probing strategy which uses additional measurements about the actual system behaviour, to reduce the number of possible hypotheses.

As the main advantages of the model-based diagnosis approach we can state the following:

- a clear logical foundation for this approach has been worked out
- no empirical knowledge about the modelled domain is needed
- the reasoning process is not restricted to a priori known possible failure modes, being able to diagnose unknown and unforeseen faults.
- the model is composable, since once specified component characteristics are not falsified by adding new components and thus can easily be extended.
- often the modelling effort is reduced by referring to an already existing model provided by the constructor of the device.

All these advantages made the concept of model-based diagnosis a highly accepted approach for diagnostic reasoning in many domains.

> The human mind has first to construct forms, independently, before we can find them in things.
>
> — Albert Einstein (1879-1955) —

In order to define a programming language the first step is to describe how programmes can be formulated, thus determining its syntax. For the specification of the syntax we will rely on a widely used notation, the so called *context-free grammars* or, for short, BNF (abbreviation for Backus-Naur Form).

2.9 Context-free Grammars

This section is denoted to the short introduction of context-free grammars and their use in defining languages.

A context-free grammar consists of four components:

- a set V of *nonterminals*
- a set T of *terminals*
- a set P of *productions* of the form $A \to \alpha$ where $A \in V$ and $\alpha \in (V \cup T)^*$.
- a *start symbol* S

For notational convenience, productions with the same nonterminal on the left side, are combined to a single production with the possible right hand sides separated by the symbol |, read as "or".

Using the grammar, strings can be derived by beginning with the start symbol S and repeatedly replacing a nonterminal with its corresponding right-hand side as determined by the productions in P. The set of all strings that can be derived in this way is the language defined by the grammar.

Example 2.9.1 Some of the more intelligent apes and less intelligent human egocentrics have a restricted linguistic repertoire which can partly be described by the following grammar.

Starting with the set of possible productions P we get:

P = { S → Pronoun VP,
VP → V NP,
NP → Det N,
Pronoun → I,
Det → the | a,
N → banana | ball | diamond | ...,
V → like | want | see | ...}

The productions state that a sentence S consist of a pronoun followed by a verbal phrase VP. A verbal phrase consists of a verb V followed by a nominal phrase

NP. A nominal phrase is a determiner Det followed by a noun N. The last two productions state which words are nouns or verbs, respectively.

For the sets of nonterminals V and terminals T we get:

$$V = \{\text{Pronoun, VP, NP, Det, N, V}\}$$

$$T = \{\text{I, the , a, banana, ball, money, like, want, see}\}$$

Obviously, the starting symbol is S.

The following sentences could now be derived using this grammar:

I like the ball
I see a banana
I want the diamond
etc.

Some related topics like parse trees and syntax directed translation will be introduced in a later chapter, when discussing compiler specific aspects.

2.10 Alphabet

The alphabet of our language consists of the 93 ASCII characters that have standard print representations as shown in table 2.1.

! " # $ % & ‘ ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = >
? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\ ]
_ ’ a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ˜

Table 2.1: Alphabet

2.11 Fundamental Categories

The following fundamental categories are used for building up more complex expressions:

- **Constants:** constants are used to denote domain obje cts, functions, relations and truth values for expressions. We therefore call them *object constants, function constants, relation constants* and *logical constants*, respectively.
- **Variables:** Variables are used to quantify over individual objects in a subset of the domain. For this reason they are associated with a set of objects.
- **Operators:** Operators are used to form complex expressions. Four types of operators can be distinguished:
  - *definition operators* This category of operators is used to add new objects to the domain and to define new functions and relations.
  - *rule operators* The rule operators are used to formulate the causal dependencies and legal inference steps within the system.
  - *term and sentence operators* These operators are used to build complex terms and complex sentences,

As the above mentioned operators indicate, there are four major types of expressions in the language — *terms, sentences, rules* and *definitions*.

2.12 Terms

Terms are used to denote objects in the modelled system. We can distinguish constants, variables and functional terms. The set of legal terms is defined below.

<term> → <const> | <var> | <functerm>
<functerm> → <funconst> (<termlist>)
<termlist> → <term> | <term>, <termlist>

Example 2.12.1 Examples for valid terms are:

c
X
f(c,X)
f(g(X,Y),d)
etc.

2.13 Sentences

Sentences are used to express relationships between objects in the world. We distinguish *predicate sentences, conjunctive sentences, implication sentences,* and *quantified sentences.* The latter can be further divided into *universally* and *existantially quantified sentences.*

<sentence> → <predsent> | <quantsent> |
<implsent> | <conjsent> |
(<sentence>)

<predsent> → <predconst> (<termlist>)

<conjsent> → <sentence>,<sentence>

<implsent> → <sentence> -: <sentence>

<quantsent> → forall <varlist> : <sentence>|
exists <varlist> : <sentence>

<varlist> → <var> | <var>, <varlist>

Example 2.13.1 Examples for valid sentences are:

VAL(n,100)
EQ(X,Y,f(Z)), LESS(X,a)
P(a,b), P(b,c) -: P(a,b)
forall X,Y: P(X,Y)
forall X: (P(X)-: (exists Y: Q(Y)))

2.14 Rules

Rules are used to formulate the causal dependencies and legal inference steps within the system. When using the inference operator "=:" we call the left

hand side of a rule the *prerequisite* and the right hand side of the rule the *consequent*. When using the inference operator "=:=" the left and right hand side are *prerequisite* and *consequent* at the same time.

<rule> →[<sentence>] =: <sentence>. |
<sentence> =:= <sentence>. |

Example 2.14.1 Possible rules are:

VAL(i1,true),VAL(i2,true) =:= VAL(out,true).
VAL(out,true) =:= forall X: (INPUT(X) -: VAL(X,true)).
VAL(in,true) =:= VAL(out,false).
exists X: (INPUT(X) -: VAL(X,true) := VAL(out,true).

The first two rules could be used to define the behaviour of a logical *conjunction*, with possible inference steps from left to right and right to left. The third describes part of the logical *negation* and the last the logical *disjunction*.

2.15 Definitions

Definitions are used to add new objects to the domain and to introduce new types, functions and relations. We can distinguish nine types of definitions — *type definitions, function definitions, component definitions* and within a component *port definitions, variable definitions, subcomponent definitions, connection definitions, axiom definitions* and *behaviour definitions*.

2.15.1 Type Definitions

Apart from already predefined types we need the flexibility to define new types to model a system in an adequate way. Types are associated with sets, so we can define a new type by stating all set elements or by using the set-theoretic union of sets and already existing types.

<typedef> → **type** <typenamelist> : <unionlist>.
<typenamelist> → <typename> | <typename>, <typenamelist>
<unionlist> → <set> | <typename> | <unionlist> + <unionlist>

<set> → { <elementlist> }
<elementlist> → <element> | <element>, <elementlist>
<element> → <word> | <number> | <predconst>(<number>)

Example 2.15.1 Possible type definitions are:

type bool : {true, false}.
type analog : {high, low, connected(1)}.
type predicates : { prime(1), less(1) }.
type mynumber : **number** + predicates.

The type **number** is already predefined in the language and is equivalent to the set of all natural numbers.

2.15.2 Function Definitions

Functions are used to define mappings from a tuple of objects to a single object, called the function value. Using functions allows for very compact rule definitions and enhances the flexibility of the language.

<funcdef> →**function** <fhead> <rulelist> **end function**
<fhead> → <funconst>(<varlist>) : <typename>
<rulelist> → <rule> | <rule> <rulelist>

Example 2.15.2 Possible function definitions are:

function not(in: bool) : bool
    val(in,true) =: val(not,false).
    val(in,false) =: val(not,true).
end function

function fact(in: number) : number
    val(in,1) =: val(not,1).
    less(1,in) =: val(not, *(in, fact(dec(in)))).
end function

2.15.3 Port Definitions

Each component has associated input and output ports which are used to interact with the environment. For each port the category and the value domain have to be specified. Possible categories are *input, output* or the bidirectional *inout* ports. The value domain is specified by an already defined typename.

<portdef> → {**input** | **output** | **inout**} <portlist> : <typename>.
<portlist> → <port> | <port>, <portlist>
<port> → <portname> | <portname>[<range>]
<range> → <number>–<number>

Example 2.15.3 Possible port definitions are:

input i1, i2: bool.
output o1, o2: number.
inout io1, io2, a[1-100]: bool.

To allow for compact port definitions we introduced *range expression* as illustrated in the last example. The expression a[1-100] results in the definition of the ports a[1] ... a[100] of type bool.

2.15.4 Variable Definitions

Variables are used to quantify over a set of objects. The use of variables often makes the definition of rules very compact and the number of rules used for describing functions very small. Variables are associated with a set of ports which can also be specified via *range expressions* as described above.

<vardef> → **var** <varlist> : <portlist>.

Example 2.15.4 Possible variable definitions are:

var X: i1, i2.
var Y: o1, a[5-17].

2.15.5 Axiom Definitions

Axioms are used to formulate the relationship between objects. We restrict our language to expressions consisting of a single predicate, stating for which objects the predicate is true. Like the introduction of variables, the use of axioms allows for more efficient and compact rule definitions.

<axiomdef> → **axiom** <axiomlist>.
<axiomlist> → <axiom> | <axiom>, <axiomlist>
<axiom> → <predconst>(<arglist>)
<arglist> → <predarg> | <predarg>, <arglist>
<predarg> → <port> | <var>

Example 2.15.5 Possible axiom definitions are:

axiom P(in1,o1), Q(a[1-5],a[8-12]).

The predicate Q containing the *range expressions* states that for each combination of the given ports, Q is true.

2.15.6 Behaviour Definitions

The heart of each component is its behavioural model, stating how it behaves under certain circumstances. Each component in the system knows one or more different behavioural modes based on different assumptions. For example the behaviour of a charged battery differs from the behaviour of an empty battery. The definition of a behavioural mode is very similar to the function definition discussed above. Differences are the optional keyword *default*, stating which behavioural mode should be assumed at the beginning for the component, and a optional probability and repair cost value for each mode, stating how probable the occurrence of the behaviour is and how expensive it would be to repair the component once it shows this behaviour.

<behaviourdef> →[**default**] **behaviour** <bhead> <rulelist> **end behaviour**
<bhead> → <behaviourconst> [**prob**:<probability>] [**cost**:<repaircost>]
<probability> → <realnumber>

<repaircost> → <word>

Example 2.15.6 Possible behaviour definitions are:

default behaviour nab **prob**:0.99
(exists X: val(X,true)) =: val(out, true).
(forall X: val(X,false)) =:= val(out,false).
end behaviour

behaviour stuckAtZero **prob**:0.01 **cost**:stuckCost
=: val(out, false).
end behaviour

2.15.7 Subcomponent Definitions

When defining components it is often necessary to refer to the functionality of prior defined components. For this reason a mechanism is needed to instantiate components within the current component definition.

<subcompdef> → **subcomponent** <scnamelist> : <compconst>.
<scnamelist> ::= <scname> | <scname> <scnamelist>
<scname> ::= <var> | <var>[<range>]

Example 2.15.7 Possible subcomponent definitions are:

subcomponent
b1,b2: battery.
ha[1-5]: halfAdder.

2.15.8 Connection Definitions

Given components and the corresponding ports, we need a mechanism to define connections between them. Furthermore we need a way to specify certain

attributes for a given connection, such as its being observable and the costs associated with observing it.

<connectdef> → **connection** <connlist>
<connlist> → <connection> | <connection> <connlist>
<connection> → <conname> [**obs**:<obs>] [**cost**:<obscost>] :
<portexpr> -> <portexprlist>.
<conname> → <connconst> | <connconst>[<range>]
<portexprlist> → <portexpr> | <portexpr>,<portexprlist>
<portexpr> → <port> | <compconst>(<port>)
<obs>: **yes** | **no**
<obscost>: <word>

Example 2.15.8 Possible connection definitions are:

connection
c : in1 -> o1.
d **obs:no** : comp(out1) -> o1, o2.
f[1-3] **obs:yes cost**:obsCost :
comp1(out[2-4]) -> comp2(in[5-7]), comp3(in[9-11].

2.15.9 Component Definitions

Components are the building blocks for each system model. We can distinguish two kinds of components. *Atomic components* are defined without reference to any existing component and its functionality. They are defined by specifying their behaviour models under certain assumptions. The second category are *composed components* which are mainly defined by referring to already existing subcomponents. The behaviour of this type of component is determined by the connections between its subcomponents.

- **Atomic Component:**

<acompdef> → **component** <compconst> <a_deflist> **end component**

<a_deflist> → <a_definition> | <a_definition> <a_deflist>
<a_definition> → <portdef> | <vardef> | <axiomdef> | <behaviourdef>

- **Composed Component:**

<c_compdef> → **component** <compconst> <c_deflist> **end component**
<c_deflist> → <c_definition> | <c_definition> <c_deflist>
<c_definition> → <portdef> | <subcompdef> | <connectiondef>

Example 2.15.9 Let us illustrate the use of component definitions by specifying the behaviour of a digital half adder. In order to construct a digital half adder we first need the definition for three *atomic components*, namely the logical conjunction, disjunction and negation.

1. **logical conjunction**

component AndGate
input in1, in2: bool.
output out: bool.

default behaviour nab **prob**:0.98
VAL(in1, true), VAL(in2, true) =:= VAL(out, true).
VAL(in1, false) =: VAL(out, false).
VAL(in2, false) =: VAL(out, false).
end behaviour

behaviour stuckAtZero **prob**:0.01 **cost**:andStuckCost
=:VAL(out, false).
end behaviour

behaviour stuckAtOne **prob**:0.01 **cost**:andStuckCost
=:VAL(out, true).
end behaviour
end component

2. **logical disjunction**

```
component OrGate
    input in1, in2: bool.
    output out: bool.

    default behaviour nab prob:0.98
        VAL(in1, false), VAL(in2, false) =:= VAL(out, false).
        VAL(in1, true) =: VAL(out, true).
        VAL(in2, true) =: VAL(out, true).
    end behaviour

    behaviour stuckAtZero prob:0.01 cost:orStuckCost
        =:VAL(out, false).
    end behaviour

    behaviour stuckAtOne prob:0.01 cost:orStuckCost
        =:VAL(out, true).
    end behaviour
end component
```

3. **logical negation**

```
component NotGate
    input in : bool.
    output out: bool.

    default behaviour nab prob:0.98
        VAL(i, true) =:= VAL(out, false).
        VAL(i, false) =:= VAL(out, true).
    end behaviour

    behaviour stuckAtZero prob:0.01 cost:notStuckCost
        =:VAL(out, false).
    end behaviour
```

behaviour stuckAtOne **prob**:0.01 **cost**:notStuckCost
=:VAL(out, true).
end behaviour
end component

Given the *atomic components* defined above we can now define the behaviour for the *composed component* half adder.

component Halfadder
input a,b : bool.
output c,z: bool.

subcomponent
and1: AndGate.
or1,or2: OrGate.
neg1,neg2: NotGate.

connection
s1: a -> or1(in1), and1(in1).
s2: b -> or1(in2), and1(in2).
s3: or1(out) -> neg1(in).
s4: neg1(out) ->or2(in1).
s5: and1(out) -> or2(in2), c.
s6: or2(out) -> neg2(in).
s7: neg2(out) -> z.
end component

2.15.10 Observation Definitions and Diagnoses

In order to provide the diagnosis tool with reference values for certain ports, we need a way to specify these observations.

<obsdef> ::= **observation** <obsconst> <assignmentlist>.
<assignmentlist>::= <assignment> | <assignment>, <assignmentlist>
<assignment>::= <port>=<element>

Example 2.15.10 Possible observations are:

observation obs1
a=1, b=0, z=1, c=1.

observation obs2
x=P(5), b="robot", z[1-5]=-5, c[5-7]=e1.

To associate a component with a defined observation we use the key function *diagnose*.

<diag> ::= **diagnose**(<compconst>, <obsconst>)

Example 2.15.11 To start the diagnose for the component Halfadder with observation obs1 we would write:

diagnose(Halfadder, obs1).

2.16 Summary

AD^{2}L enables the knowledge engineer to define components with associated ports and to specify the corresponding behaviour with rules which may contain existentially and universally quantified sentences. Two types of components are distinguished: atomic components and composed components. Atomic components are defined by specifying their ports and possible behaviour modes. Composed components are defined by specifying connections between the constituting subcomponents. No behaviour modes are given for this type of components, since the component behaviour is derived by the behaviour of its subcomponents.

Observations have a name and consist of port-value assignments. Finally an observation can be associated with the observed component.

Chapter 3

Conceptualisation

> The Universe is a grand book which cannot be read until one first learns to comprehend the language and become familiar with the characters in which it is composed. It is written in the language of mathematics.
>
> — Galileo Galilei (1564 -1642) —

The formalisation of knowledge requires a *conceptualisation* of the system under consideration in terms of objects, functions and relations.

3.1 Domain and Objects

A *domain* is the set of all objects assumed to exist in the system to be modelled. The notion of an *object* covers a wide spectrum. Objects can be concrete physical entities like a transistor, a neuron or a battery, or they can be abstract like a natural number, a set of words or Gödel's incompleteness theorem. Objects can be atomic or composed like a digital circuit consisting of many subcomponents, and they can even be fictional like the USS Enterprise, Gandalf the magician or the proof that NP=P.

Every domain contains at least the following basic objects:

- All words expressible using the alphabet of the language
- All natural numbers

- the boolean *true* and *false*
- *self*- an object used within a component to refer to the component's identity.

3.2 Functions and Relations

There are two kinds of possible relationship.

- A *function* associates an unique object, the value, with a finite sequence of objects, its arguments. In other words, a function is a finite set of n-tuples where the first n-1 objects are the arguments and the last object is the corresponding value. For each possible combination of the arguments the function contains exactly one tuple, since there can only be one associated value. A function that requires n arguments is called a n-ary function. For example the logical *and* contains the triple (true,true,true), indicating that all arguments being true results in the value of the function being true as well. The following predefined functions representing arithmetic operations on entities of type **number** are known: $+^2, -^2, *^2, /^2$.
- A *relation* is a finite set of arbitrary n-tuples. Each of these tuples is said to satisfy the relation. A relation consisting of n-tuples is called a n-ary relation. For example the relation $\subset$ contains the tuples ($\{a\}$,$\{a,b\}$) and ($\{a\}$,$\{a,c,e\}$) indicating that $\{a\}$ is subset of $\{a,b\}$ and $\{a,c,e\}$). The above example illustrates the difference between a relation and a function. A relation can contain any n-tuple, whereas in a function the last object in a tuple is always the same for a certain combination of the first n-1 tuples.

  The following predefined predicates are known: $=^2, <^2, >^2$ having the usual meaning and VAL^2 associating the object given as first argument, with its value.

In fact every function is a relation as well, whereas the opposite is not true.

The language definition requires all functions and relations to have fixed arity, that is, each tuple in the corresponding set contains the same number of objects. For example, it is not possible to define a function + that contains the tuples (1,2,3) and (1,2,3,6) since they do not have the same number of objects. Although it would be convenient to have this kind of flexibility in defining functions and

relations, it is not essential and could possibly interfere with our wish for strong type checking, thus introducing a new source of errors.

Chapter 4

Semantics

> "When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean - neither more nor less."
> "The question is," said Alice, "whether you can make words mean so many different things."
> "The question is," said Humpty Dumpty, "which is to be master - that's all."
>
> — Lewis Carroll —

The semantics determine the facts in the world to which the sentences refer. Without semantics a sentence is just a collection of electrons within a computer system or ink-spots on a white page. With semantics each sentence makes a claim about the world.

One of the major goals when defining the syntax of the language was to provide a very intuitive approach. Unfortunately the definition of the semantics of the rules expressible in AD^2L is a bit more complicated, since we have to use formal methods to avoid ambiguities easily introduced by natural language descriptions.

Given a model M, we use $CS(M), PS_n(M)$ and $FS_n(M)$ to denote the set of all constants, all n-ary predicates and n-ary functions used in M. For $\bigcup_{i=0}^{\infty} PS_i(M)$ and $\bigcup_{i=0}^{\infty} FS_i(M)$ we write $PS(M)$ and $FS(M)$, respectively. The set $\Sigma M = CS(M) \cup PS(M) \cup FS(M)$ is called the signature of model M.

4.1 Interpretation

An interpretation of a model M is the tuple $m = (D, \phi, I)$ where

1. D is a non empty set, the domain of m
2. ϕ is a function defined on ΣM where the following holds
   (a) $\phi(c) \in D$ for each $c \in CS(M)$.
   (b) for each $f \in FS_n(M)$, $\phi(f)$ is a function of type $D^n \to D$.
   (c) for each $P \in PS_n(M)$, $\phi(P)$ is a function of type $D^n \to \{T, F\}$.
3. I, the *variable assignment function*, is of type
   $$V \to D$$
   for each variable V.

The variable assignment function I is essential for defining the semantics of quantified sentences.

4.2 Term semantics

Given the interpretation m of a model M, we can assign a *semantic value* to all terms in the language. For this reason we define a *semantic function* s_m which maps each element from the set T of terms to an object in the domain D.

$$s_m : T \to D$$

Given an object constant ρ, the semantic value is defined as the object assigned to that constant by the given interpretation.

$$s_m(\rho) = \phi(\rho)$$

Given an individual variable v, the semantic value is defined as the object assigned to that variable by the variable assignment function I.

$$s_m(v) = I(v)$$

Given a functional term, the semantic value is obtained by applying the function denoted by the function constant to the objects denoted by its arguments.

$$s_m(f(t_1, \ldots, t_n)) = \phi(f)(s_m(t_1), \ldots, s_m(t_n))$$

4.3 Truth Value

Given an interpretation m for a model M we can now determine the truth value for a certain sentence. For this reason we define a function s_t which assigns to each element of the set S of sentences the corresponding truth value.

$$s_t : S \rightarrow \{true, false\}$$

Given a sentence consisting of a single predicate, the truth value is obtained by applying the predicate denoted by the predicate constant to the objects denoted by its arguments. When viewing the predicate as a set of tuples, we define the truth value as *true* if and only of the tuple of objects determined by the arguments is an element of this set, otherwise as *false*.

$$s_t(P(t_1, \ldots, t_n)) = \phi(P)(s_m(t_1), \ldots, s_m(t_n)) = \begin{cases} true & \text{if } < s_m(t_1), \ldots, s_m(t_n) > \in \phi(P) \\ false & \text{otherwise} \end{cases}$$

Given a conjunction the truth value is defined as true if and only if the truth value of every conjunct is true.

$$s_t(\psi_1, \ldots, \psi_n) = \begin{cases} true & \text{if } s_t(\psi_i) \text{ is true for all } i, 1 \leq i \leq n \\ false & \text{otherwise} \end{cases}$$

Given an implication $\psi_1 \rightarrow \psi_2$, the truth value is defined as follows: If ψ_1 is true then the truth-value of the implication is true if and only if ψ_2 is true. If ψ_1 is false then the implication is true, regardless of the truth-value of ψ_2 (ex falso quod libet).

$$s_t(\psi_1 \rightarrow \psi_2) = \begin{cases} true & \text{if } s_t(\psi_1) = false \text{ or } s_t(\psi_1, \psi_2) = true \\ false & \text{otherwise} \end{cases}$$

Given an universally quantified sentence, the truth value is defined as true if and only if for all possible variable assignment functions I the truth value of the sentence following the colon is true.

$$s_t(forall\ v_1, \ldots, v_n : \psi) = \begin{cases} true & \text{if } \forall I\ s_t(\psi) = true \\ false & \text{otherwise} \end{cases}$$

Given an existentially quantified sentence, the truth value is defined as true if and only if for at least one possible variable assignment function I the truth value of the sentence following the colon is true.

$$s_t(exists\ v_1, \ldots, v_n : \psi) = \begin{cases} true & \text{if } \exists I\ s_t(\psi) = true \\ false & \text{otherwise} \end{cases}$$

4.4 Rules

We have seen in the previous sections how the truth values of sentences are defined. Since rules can be seen as a binary relation between sentences, we are now able to define the semantics of rules.

Given a rule Γ with

$$\Gamma = \Sigma_L{:=}\Sigma_R$$

we can infer the consequence Σ_R if and only if the prerequisite Σ_L is true. In other words an inference system can infer Σ_R iff the following holds:

$$s_t(\Sigma_L) = true.$$

With the above given formal semantic definitions we have introduced a way to describe the meaning of terms, sentences and rules in a precise and unambiguous form.

Chapter 5

From AD^{2}L to Constraint Systems

> In order to translate a sentence from English into French two things are necessary. First, we must understand thoroughly the English sentence. Second, we must be familiar with the forms of expression peculiar to the French language. The situation is very similar when we attempt to express in mathematical symbols a condition proposed in words. First, we must understand thoroughly the condition. Second, we must be familiar with the forms of mathematical expression.
>
> — George Polyá (1887 - 1985) —

Many modern systems for model based diagnosis use constraint systems to describe the domain under consideration. In order for AD^{2}L to be a useful language usable for a large number of currently available diagnosis systems, there has to be a translation for each model described in this language to an equivalent model for a constraint system.

We will show in this chapter that under certain assumptions concerning the constraint system such a translation always exists and then compare the complexity of the resulting model for the constraint system with the complexity of the model defined in AD^{2}L in terms of rules and predicates used. An important restriction has to be made on the variables used in AD^{2}L rules for this and the

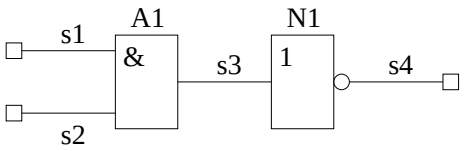


Figure 5.1: Logical nand-gate

following chapters – we only allow for variables quantifying over **finite** sets of objects. Without this restriction we would not be able to find an adequate translation to the simple constraint system, since more sophisticated programming constructs than simple *if-then* statements were necessary in this case.

5.1 Structure of a Constraint System

This section is dedicated to the introduction of the characteristics of constraint systems and a typical propagation algorithm used in such systems.

5.1.1 Flat Systems

The basic elements of a constraint system, as the name already indicates, are constraints and variables. In general, constraints define relations between variables. For our purpose constraints can be seen as objects having a set of input, output and bidirectional ports mapping an arbitrary number of input values to an arbitrary number of output values and which are associated with a set of behaviour definitions determining how this mapping is done. Each port of a constraint is connected to a single variable and each variable can be connected to an arbitrary number of ports. Each variable can be assigned a value. For variables that have no value assigned a special value is assumed, e.g. *nil* in some systems. Usually there are no restrictions on the values a variable may have. The only requirement is the ability of the constraints to process these values.

Figure 5.1 shows the structure of a *nand*-gate, consisting of two gates implementing the logical conjunction and the logical negation, respectively. Figure 5.2 illustrates how the *nand*-gate can be modelled using a constraint system.

For each gate a constraint with the same number of input and output ports is created. The constraints are associated with a set of behavioural modes deter-

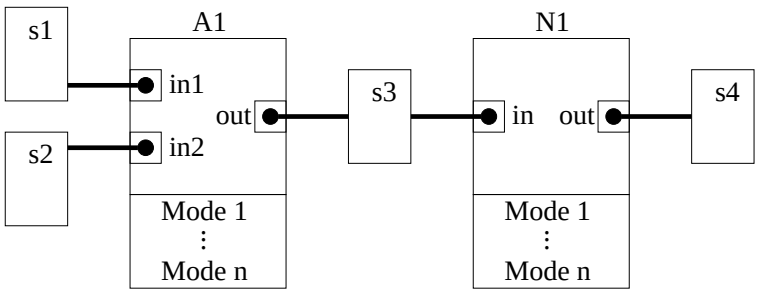


Figure 5.2: Constraint system representation of a logical nand-gate

mining how input values are mapped to output values. Each of the connections s1 ... s4 is modelled by a variable with the same name. In the context of digital circuits, constraints are equivalent to gates and variables correspond to connections between them.

5.1.2 Hierarchical Systems

We have seen in the previous section how flat systems, consisting solely of atomic components, can be translated to equivalent constraint systems. Many constraint systems also allow the construction of hierarchical models. This way, components can be combined to make up new composed components which can then be used in the same way as atomic components. That is, we can define connections between them, thus constructing more complex models.

Figure 5.3 shows the conversion of the model for a *nand*-gate to a composed component, which can then be treated as a single constraint. Composed components can again be combined to create even more complex constraints.

5.1.3 Basic Operations

We have seen how a domain can be modelled using constraint systems. Now we are interested in the kind of operations a constraint system has to understand in order to construct a model for a given domain.

The main building blocks for each model in a constraint system are constraints and variables. Therefore we need a mechanism to add them to a model. For constraints we need to be able to specify the input and output ports as well as the associated behavioural models. Some constraint systems allow a default

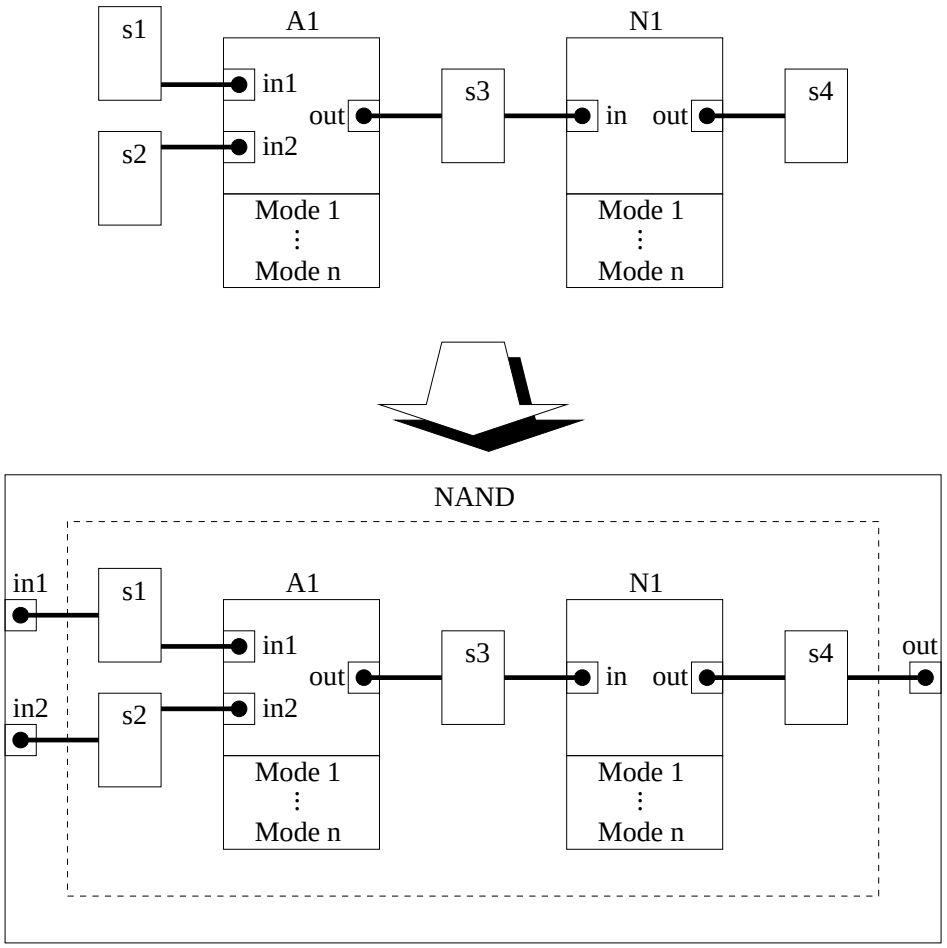


Figure 5.3: a composed constraint for a logical nand-gate

behaviour mode to be set, denoting how a constraint is assumed to behave in the general case. For each of the behavioural models a probability can be specified, determining how probable it is for the constraint to show this behaviour. When modelling digital circuits it is often necessary to associate behaviour modes of constraints with a value denoting the repair costs for the modelled component showing this behaviour. Furthermore we need a mechanism to combine constraints to construct hierarchical systems. For this reason we need an operation that allows to convert a model made up of a number of constraints into a single new constraint. For this new constraint, we need the additional information how its ports are connected to the ports of the constituting constraints. Thus we need to be able to associate with a composed constraint a data structure describing

these port connections.

Finally, we need a mechanism to specify the connections between ports and variables.

As a result, we get the following list of basic operations necessary for constructing a model which all constraint systems are assumed to provide:

1. define a new constraint with a given name
2. specify input, output and bidirectional ports for a constraint
3. associate a constraint with a set of behaviour definitions
4. associate a constraint behaviour mode with a value indicating its probability
5. associate a behaviour mode with a cost value (e.g. to denote the repair costs)
6. instantiate an already defined constraint
7. declare a new variable
8. connect a variable with the port of a constraint
9. convert a model to a single constraint
10. associate a data structure with a composed constraint describing how the ports are connected to the ports of its subconstraints.

5.1.4 Behaviour Definition

Various systems provide different languages for defining behavioural models. Often the system's implementation language is used as the language for the behaviour definition as well. What all these languages have in common, is the possibility to express simple conditional statements like *if* α *then* β, where α is a conjunction of negated or nonnegated conditions. Should there be no conjunction then it can be simulated by grouping *if-then* statements. If we can show that each rule in AD^{2}L can be translated to these simple conditional statements, then it can be translated to an arbitrary language used by a constraint system.

5.1.5 The Propagation Algorithm

A propagation algorithm is used in a constraint system to calculate new variable values, based on already known values and the specified behaviour of the constraints. The propagation can be seen as a process consisting of iterating steps. First all constraints and variables with known values are used to calculate a new value for a variable. Each variable whose value has changed and each constraint that is connected to such a variable are processed in the next iteration. Has no new value for a variable been derived then the propagation stops. Table 5.1.5 gives a detailed description of the algorithm PROPAGATE taken from [14].

Table 5.1: The propagation algorithm PROPAGATE

1. Every variable and every constraint that have been assigned a new value is put into the set *schedule*.
2. For all elements $x \in$ *schedule* do:
   - if x is a constraint, then try to calculate a new value for a variable and put the variable into the set *newSchedule*. If the calculated value leads to a contradiction with another variable value then raise a flag to signal an inconsistency.
   - if x is a variable, then put every constraint that is connected to x into the set *newSchedule*. These constraints will be used in the next step to calculate new variable values.
3. Let *schedule* := *newSchedule* and goto 2, until *schedule* is the empty set.

5.2 Translation on the Structural Level

The structure of a system is determined by the constraints used and the connections between them.

In AD²L we have two ways to define a constraint. First we can assume it to be an atomic constraint, thus defining its ports and associated behavioural models. Second we can assume it to be a composed constraint, specifying its ports, the

constituting constraints and the connections between them. We will now show for each of these possibilities the necessary steps for creating an equivalent constraint within a constraint system.

5.2.1 Atomic Constraints

Consider the following atomic constraint definition in AD²L.

component constraintName
  input $i_1 \ldots i_n$: <typename>.
  output $o_1 \ldots o_m$: <typename>.
  inout $io_1 \ldots io_l$: <typename>.

  default behaviour behaviourName$_1$ **prob**: p_1 **cost**: c_1
    ...
  end behaviour

  behaviour behaviourName$_2$ **prob**: p_2 **cost**: c_2
    ...
  end behaviour

  behaviour behaviourName$_k$ **prob**: p_k **cost**: c_k
    ...
  end behaviour
end component

The necessary steps to translate this definition to a constraint system are:

1. define a new constraint with name *constraintName*
2. add the input ports $i_1 \ldots i_n$ to constraint *constraintName*
3. add the output ports $o_1 \ldots o_m$ to constraint *constraintName*
4. add the bidirectional ports $io_1 \ldots io_l$ to constraint *constraintName*
5. associate the behaviour definitions for $behaviourName_1 \ldots behaviourName_k$ with the constraint *constraintName*

6. associate behaviour modes $behaviourName_1 \ldots behaviourName_k$ of constraint *constraintName* with probabilities $p_1 \ldots p_k$

7. associate behaviour modes $behaviourName_1 \ldots behaviourName_k$ of constraint *constraintName* with values $c_1 \ldots c_k$

5.2.2 Hierarchical Constraints

Given the following composed component definition in AD^{2}L

component constraintName
input $i_1 \ldots i_n$: <typename>.
output $o_1 \ldots o_m$: <typename>.
inout $io_1 \ldots io_l$: <typename>.

subcomponent
$c_{11} \ldots c_{1n}$: component$_1$.
...
$c_{m1} \ldots c_{mk}$: component$_m$.

connection
...
s_i: c_{i0}(port$_{i0}$) -> c_{i1}(port$_{i1}$), ..., c_{in}(port$_{in}$)
...
end component

The necessary steps to translate this definition to a constraint system are:

1. create variables s_i

2. connect the variables s_i with the ports port$_{i0}$ ... port$_{in}$ of constraints c_{i0} ... c_{in}.

3. convert the connected constraints to a composed constraint with name *constraintName*

4. associate a data structure with constraint *constraintName*, describing how the ports are connected to the ports of the subconstraints.

5. add the input ports $i_1 \ldots i_n$ to constraint *constraintName*

6. add the output ports $o_1 \ldots o_m$ to constraint *constraintName*

7. add the bidirectional ports $io_1 \ldots io_l$ to constraint *constraintName*

5.3 Translation on the Behavioural Level

Let us consider the abstract constraint system ACS which restricts itself to a very simple mechanism for behaviour definitions, using only *if-then* statements (we write $\alpha \models \beta$ instead of *if* α *then* β), the logical conjunction ($\wedge$) and negation ($\neg$). We have already argued that if we can find a translation for AD^{2}L rules to these simple conditional statements, then we can find a translation to an arbitrary language used by a constraint system.

In the following we will define a function *acs* which maps an AD^{2}L expression to a corresponding set of expressions for the abstract constraint system.

$$acs: \text{ AD}^2\text{L } \rightarrow 2^{L(ACS)}$$

where L(ACS) is the language used by the abstract constraint system.

Given a sentence Σ and an interpretation I we write Σ^{I} to denote the resulting sentence when substituting each variable in Σ with the value determined by I.

Example 5.3.1 Consider the following interpretation I where each tuple determines the value that has to be substituted for a variable:

$$I = \{(X, 1), (Y, 2), (Z, 3)\}.$$

The following tabular shows some sentences and the results when applying the interpretation I.

Σ	Σ^{I}
true	true
P(c)	P(c)
P(X)	P(1)
P(X,Y), Q(X,Z)	P(1,2), Q(1,3)
P(W,Z,c)	P(W,3,c)
Q(f(g(X),Z))	Q(f(g(1),3))

We write $\Omega_{X_1,\ldots,X_n}$ to denote the set consisting of all possible interpretations for variables $X_1, \ldots, X_n$.

We will use the following variable definitions for the explaining examples:

var X: p1, p2.
var Y: p3, p4.

resulting in the set of possible interpretations

$$\begin{aligned}\Omega_{X,Y} = \{ \quad & \{(\text{X,p1}),(\text{Y,p3})\} \\ & \{(\text{X,p1}),(\text{Y,p4})\} \\ & \{(\text{X,p2}),(\text{Y,p3})\} \\ & \{(\text{X,p2}),(\text{Y,p4})\}\}\end{aligned}$$

5.3.1 Predicate Sentences

For all predicate sentences Σ that do not include variables, the function *acs* is defined as the set containing the unchanged sentence.

$$acs(\Sigma) = \{\Sigma\}$$

Example 5.3.2 The function *acs* maps a predicate sentence to a set containing this sentence.

Σ	$acs(\Sigma)$
P(c)	{ P(c) }
P(f(d,4),c)	{ P(f(d,4),c) }

Note that we need not define a mapping for literals containing variables since they are substituted according to a given interpretation before the function *acs* is applied. In other words the function *acs* is only defined on sentences consisting of ground literals.

5.3.2 Existentially Quantified Sentences

Given an existentially quantified sentence Σ, we use the following mapping:

$$\Sigma = \text{exists } X_1, \ldots X_n : \sigma_1, \ldots, \sigma_n$$

$$acs(\Sigma) = \{\tau_1 \wedge \ldots \wedge \tau_n | \tau_i \in acs(\sigma_i^I), I \in \Omega_{X_1,\ldots,X_n}\}$$

Example 5.3.3 Consider the sentence

$$\Sigma = \text{ exists X, Y: P(X,Y)}$$

Applying the function *acs* to this sentence we get the set

$$\begin{aligned} acs(\Sigma) = \{ & \\ & \text{P(p1,p3)}, \\ & \text{P(p1,p4)}, \\ & \text{P(p2,p3)}, \\ & \text{P(p2,p4)}\}. \end{aligned}$$

Example 5.3.4 Considering the sentence

$$\Sigma = \text{ exists X, Y: P(X), Q(Y)}$$

we get the set

$$\begin{aligned} acs(\Sigma) = \{ & \\ & \text{P(p1)} \wedge \text{Q(p3)}, \\ & \text{P(p1)} \wedge \text{Q(p4)}, \\ & \text{P(p2)} \wedge \text{Q(p3)}, \\ & \text{P(p2)} \wedge \text{Q(p4)}\}. \end{aligned}$$

We can think of the resulting sentences as possible ways to make the existentially quantified expression true. That is, we reformulate the expression in terms of simple conjunctions, considering every possible interpretation for the embedded variables. If there is at least one conjunction whose truth-value is true, then the existentially quantified expression is true as well.

5.3.3 Universally Quantified Sentences

Given an universally quantified sentence Σ we have the following mapping:

$$\Sigma = \text{forall } X_1, \ldots X_n : \sigma_1, \ldots, \sigma_m$$

$$acs(\Sigma) = \tau_1^{I_1} \wedge \ldots \wedge \tau_n^{I_1} \wedge \ldots \wedge \tau_1^{I_m} \wedge \ldots \wedge \tau_n^{I_m} | I_k \in \Omega_{X_1,\ldots,X_n}, \tau_i^{I_k} \in acs(\sigma_i^{I_k})\}$$

Since we want a sentence to be true for all possible values for the given variables we need a conjunction extending over all possible interpretations. Only if this conjunction is true, then the universally quantified sentence is true as well.

Example 5.3.5 Considering the following sentence

$$\Sigma = \text{ forall X, Y: P(X), Q(Y)}$$

we get the set consisting of the single conjunction

$$acs(\Sigma) = \{\text{P(p1)} \wedge \text{Q(p3)} \wedge \text{P(p1)} \wedge \text{Q(p4)} \wedge \text{P(p2)} \wedge \text{Q(p3)} \wedge \text{P(p2)} \wedge \text{Q(p4)}\}$$

Example 5.3.6 A more complex example is the sentence

$$\Sigma = \text{ forall X: (exist Y: P(X,Y)}$$

As a first step we have to calculate acs(exist Y: P(X,Y)). The resulting set is {P(X,p3), P(X,p4)}. Referring to our definition we now have to construct a conjunction for each possible combination with repetitions of elements in the above set with cardinality two. Extending over all possible interpretations, we get the following conjunctions

$$\begin{aligned} acs(\Sigma) = \{ & \\ & \text{P(p1,p3)} \wedge \text{P(p2,p3)}, \\ & \text{P(p1,p3)} \wedge \text{P(p2,p4)}, \\ & \text{P(p1,p4)} \wedge \text{P(p2,p3)}, \\ & \text{P(p1,p4)} \wedge \text{P(p2,p4)} \ \} \end{aligned}$$

Obviously, if at least one of the above conjunctions is true, then the universally quantified sentence is true as well.

5.3.4 Conjunctive Sentences

Given a conjunctive sentence Σ we get the following mapping:

$$\Sigma = \sigma_1, \ldots, \sigma_n$$

$$acs(\Sigma) = \{\tau_1 \wedge \ldots \wedge \tau_n | \tau_i \in acs(\sigma_i)\}$$

Example 5.3.7 Consider the following sentence

$$\Sigma = \text{ P(p6), forall X: (exist Y: P(X,Y))}$$

Σ is a conjunction of two sentences σ_1 and σ_2 where

$$\sigma_1 = \text{ P(p6)}$$

and

$$\sigma_2 = \text{ forall X: (exist Y: P(X,Y))}$$

Referring to example 5.3.6 we already know that

$$\begin{aligned} acs(\sigma_2) = \{ & \\ & \text{P(p1,p3)} \wedge \text{P(p2,p3)}, \\ & \text{P(p1,p3)} \wedge \text{P(p2,p4)}, \\ & \text{P(p1,p4)} \wedge \text{P(p2,p3)}, \\ & \text{P(p1,p4)} \wedge \text{P(p2,p4)} \ \} \end{aligned}$$

Since σ_1 is a ground literal we get:

$$acs(\sigma_1) = \ \{\text{P(p6)}\}$$

Referring to the definition of *acs* for conjunctive sentences, we have to find the set consisting of all sentences $\tau_1 \wedge \tau_2$, where $\tau_i \in acs(\sigma_i)$.

The resulting set therefore is:

$$\begin{aligned} acs(\Sigma) = \{ & \\ & \text{P(p6)} \wedge \text{P(p1,p3)} \wedge \text{P(p2,p3)}, \\ & \text{P(p6)} \wedge \text{P(p1,p3)} \wedge \text{P(p2,p4)}, \\ & \text{P(p6)} \wedge \text{P(p1,p4)} \wedge \text{P(p2,p3)}, \\ & \text{P(p6)} \wedge \text{P(p1,p4)} \wedge \text{P(p2,p4)} \ \} \end{aligned}$$

5.3.5 Implication Sentences

Given an implication sentence Σ, we get the following mapping:

$$\Sigma = \sigma_1 \text{ :- } \sigma_2$$

$$acs(\Sigma) = \{\neg(\tau_1 \wedge \neg\tau_2) | \tau_i \in acs(\sigma_i)\}$$

Example 5.3.8 A simple example is the sentence

$$\Sigma = \text{ P(c) :- Q(c)}$$

Obviously the result is

$$acs(\Sigma) = \{\neg(\text{P(c)} \wedge \neg\text{Q(c)})\}$$

Example 5.3.9 A more interesting example is the following sentence:

$$\Sigma = \text{ forall X: P(X) :- Q(X)}$$

We have to calculate acs(P(X) :- Q(X)) for each possible interpretation of X and to combine the results in a conjunction. The resulting set is

$$acs(\Sigma) = \{\neg(\text{P(p1)} \wedge \neg\text{Q(p1)}) \wedge \neg(\text{P(p2)} \wedge \neg\text{Q(p2)})\}$$

Example 5.3.10 As a last example, let us consider the sentence

$$\Sigma = \text{ forall X: (P(X) :- (exists Y: Q(X,Y)))}$$

We will show how to translate this sentence step by step. The structure of this sentence can be seen as

$$\Sigma = \text{ forall X:}\sigma$$

where

$$\sigma = \text{P(X) :- (exists Y: Q(X,Y))}$$

The resulting set is of the form:

$$acs(\Sigma) = \{\tau^{I_1} \wedge \tau^{I_2} | \tau^{I_i} \in acs(\sigma^{I_i})\}$$

where $I_1 = \{(X, p1)\}$ and $I_2 = \{(X, p2)\}$ are the possible interpretations for variable X. Obviously we have to find $acs(\sigma)$ for each possible interpretation in order to construct the resulting set for the universally quantified sentence. For the interpretation I_1 we get the sentence

$$\sigma^{I_1} = \text{P(1) :- (exists Y: Q(1,Y))}$$

We can write this sentence as

$$\sigma^{I_1} = \sigma_1 \text{ :- } \sigma_2$$

where

$$\sigma_1 = \text{P(p1)}$$

and

$$\sigma_2 = \text{exists Y: Q(X,Y)}$$

To find the resulting set for the implication sentence, we first have to find $acs(\sigma_1)$ and $acs(\sigma_2)$. Referring to the definition of the function acs for predicate sentences and existentially quantified sentences we get:

$$acs(\sigma_1) = \{\ \mathrm{P(p1)}\}$$

and

$$acs(\sigma_2) = \{\ \mathrm{Q(p1,p3),\ Q(p1,p4)}\}$$

Using these result we find

$$acs(\sigma^{I_1}) = \{\neg(\mathrm{P(p1)}\ \wedge \neg\mathrm{Q(p1,p3)}),\ \neg(\mathrm{P(p1)} \wedge \neg\mathrm{Q(p1,p4)})\}$$

Analogous we get for the interpretation I_2

$$acs(\sigma^{I_2}) = \{\neg(\mathrm{P(p2)}\ \wedge \neg\mathrm{Q(p2,p3)}),\ \neg(\mathrm{P(p2)} \wedge \neg\mathrm{Q(p2,p4)})\}$$

Now we are able to calculate the function value for the universally quantified sentence Σ resulting in the following set:

$$\begin{aligned} acs(\Sigma) = \{\quad & \neg(\mathrm{P(p1)}\ \wedge \neg\mathrm{Q(p1,p3)}) \wedge \neg(\mathrm{P(p2)}\ \wedge \neg\mathrm{Q(p2,p3)}),\\ & \neg(\mathrm{P(p1)}\ \wedge \neg\mathrm{Q(p1,p3)}) \wedge \neg(\mathrm{P(p2)}\ \wedge \neg\mathrm{Q(p2,p4)}),\\ & \neg(\mathrm{P(p1)}\ \wedge \neg\mathrm{Q(p1,p4)}) \wedge \neg(\mathrm{P(p2)}\ \wedge \neg\mathrm{Q(p2,p3)}),\\ & \neg(\mathrm{P(p1)}\ \wedge \neg\mathrm{Q(p1,p4)}) \wedge \neg(\mathrm{P(p2)}\ \wedge \neg\mathrm{Q(p2,p4)})\} \end{aligned}$$

Whenever one of the sentences in the above set is true, then the sentence Σ is true as well.

5.3.6 Rules

Behavioural models are defined using rules, stating how the values of certain ports are determined by the values of other ports. Since we have defined the mapping from sentences to corresponding sets of conjunctions, we are now able to define the function acs on rules.

- Given the rule Γ with

  $$\Gamma = \Sigma_L := \Sigma_R$$

  we get the following mapping:

  $$acs(\Gamma) = \{\tau_L \models \tau_R | \tau_L \in acs(\Sigma_L), \tau_R \in acs(\sigma_R)\}$$

Note that $|acs(\Sigma_R)| = 1$, since no existentially quantified sentences are allowed on the right-hand side of a rule.

- For a rule Γ with

$$\Gamma = \Sigma_L =:= \Sigma_R$$

we get the following mapping:

$$acs(\Gamma) = acs(\Sigma_L := \Sigma_R) \cup acs(\Sigma_R := \Sigma_L)$$

Example 5.3.11 We will take as an example part of the behaviour definition for a digital *and*-gate, assuming that variable X extends over all input ports and the constant *out* is associated with the gate's output port.

$$\Gamma = \text{ forall X: VAL(X,true) =:= VAL(out,true).}$$

Γ can be written as

$$\Gamma = \Sigma_L =:= \Sigma_R$$

where

$$\Sigma_L = \text{ forall X: VAL(X,true)}$$

and

$$\Sigma_R = \text{ VAL(out,true)}$$

To find the translation for Γ, we first have to determine the translation for Σ_L and Σ_R, resulting in:

$$acs(\Sigma_L) = \{\text{VAL(p1,true)} \wedge \text{ VAL(p2,true)}\}$$

and

$$acs(\Sigma_R) = \{ \text{ VAL(out,true)}\}$$

The final translation for Γ can now easily be found, resulting in:

$$\begin{aligned} acs(\Gamma) = \quad & \{ \text{ VAL(p1,true)} \wedge \text{ VAL(p2,true)} \models \{ \text{ VAL(out,true) } \} \\ & \cup \\ & \{ \text{ VAL(out,true)} \models \{ \text{ VAL(p1,true)} \wedge \text{ VAL(p2,true) } \} \end{aligned}$$

5.4 Creating an Example Constraint

Let us illustrate these concepts by translating the half adder defined in chapter 2.8 to the abstract constraint system ACS.

We will show the necessary steps for each component definition. For atomic components we will first translate all behaviour definitions to the simple conditional statements understood by ACS and then create an equivalent constraint. For composed components the main work lies in mapping the structure to a corresponding ACS model.

5.4.1 Logical Conjunction

component AndGate
  input in1, in2: bool.
  output out: bool.

  default behaviour nab **prob**:0.98
    VAL(in1, true), VAL(in2, true) =:= VAL(out, true).
    VAL(in1, false) =: VAL(out, false).
    VAL(in2, false) =: VAL(out, false).
  end behaviour

  behaviour stuckAtZero **prob**:0.01 **cost**:andStuckCost
    =:VAL(out, false).
  end behaviour

  behaviour stuckAtOne **prob**:0.01 **cost**:andStuckCost
    =:VAL(out, true).
  end behaviour
end component

Applying the function *acs* to each rule of the behavioural model named *nab* we get the following set of conditionals equivalent to this definition:

$$\begin{aligned}\text{nab}_{acs} = \{ \quad & \text{VAL(in1, true)} \wedge \text{VAL(in2, true)} \models \text{VAL(out, true)}, \\ & \text{VAL(out, true)} \models \text{VAL(in1, true)} \wedge \text{VAL(in2, true)}, \\ & \text{VAL(in1, false)} \models \text{VAL(out, false)}, \\ & \text{VAL(in2, false)} \models \text{VAL(out, false)} \ \}\end{aligned}$$

For the behavioural models named *stuckAtZero* and *stuckAtOne* we get

$$\text{stuckAtZero}_{acs} = \{ \quad \models \text{VAL(out, false)} \ \}$$

and

$$\text{stuckAtOne}_{acs} = \{ \quad \models \text{VAL(out, true)} \ \}$$

respectively.

To create an equivalent constraint for ACS we take the following steps:

1. create a new constraint with name *AndGate*
2. add the input ports in1 and in2 to constraint *AndGate*
3. add the output port out to constraint *AndGate*
4. associate the behaviour definitions for modes *nab, stuckAtZero* and *stuck-AtOne* with constraint *AndGate*
5. associate behaviour modes *nab, stuckAtZero* and *stuckAtOne* of constraint *AndGate* with probabilities 0.98, 0.01 and 0.01 respectively.
6. associate behaviour modes *stuckAtZero* and *stuckAtOne* of constraint *AndGate* with cost value *andStuckCost*

5.4.2 Logical Disjunction

component OrGate

input in1, in2: bool.

output out: bool.

default behaviour nab **prob**:0.98

VAL(in1, false), VAL(in2, false) =:= VAL(out, false).

VAL(in1, true) =: VAL(out, true).
VAL(in2, true) =: VAL(out, true).
end behaviour

behaviour stuckAtZero **prob**:0.01 **cost**:orStuckCost
=:VAL(out, false).
end behaviour

behaviour stuckAtOne **prob**:0.01 **cost**:orStuckCost
=:VAL(out, true).
end behaviour
end component

Applying the function *acs* to each rule of the behavioural model named *nab* we get the following set of conditionals equivalent to this definition:

$$\begin{aligned} \text{nab}_{acs} = \{ \quad & \text{VAL(in1, false)} \wedge \text{VAL(in2, flase)} \models \text{VAL(out, false)}, \\ & \text{VAL(out, false)} \models \text{VAL(in1, false)} \wedge \text{VAL(in2, false)}, \\ & \text{VAL(in1, true)} \models \text{VAL(out, true)}, \\ & \text{VAL(in2, true)} \models \text{VAL(out, true)} \} \end{aligned}$$

For the modes *stuckAtZero* and *stuckAtOne* we get the same translation as for the logical conjunction.

To create an equivalent constraint for ACS we take the following steps:

1. create a new constraint with name *OrGate*
2. add the input ports in1 and in2 to constraint *OrGate*
3. add the output port out to constraint *OrGate*
4. associate the behaviour definitions for modes *nab, stuckAtZero* and *stuckAtOne* with constraint *OrGate*
5. associate behaviour modes *nab, stuckAtZero* and *stuckAtOne* of constraint *OrGate* with probabilities 0.98, 0.01 and 0.01 respectively.
6. associate behaviour modes *stuckAtZero* and *stuckAtOne* of constraint *OrGate* with cost value *orStuckCost*

5.4.3 Logical Negation

component NotGate
input in : bool.
output out: bool.

default behaviour nab **prob**:0.98
VAL(i, true) =:= VAL(out, false).
VAL(i, false) =:= VAL(out, true).
end behaviour

behaviour stuckAtZero **prob**:0.01 **cost**:notStuckCost
=:VAL(out, false).
end behaviour

behaviour stuckAtOne **prob**:0.01 **cost**:notStuckCost
=:VAL(out, true).
end behaviour
end component

Applying the function *acs* to each rule of the behavioural model named *nab* we get the following set of conditionals equivalent to this definition:

$$\begin{aligned} \text{nab}_{acs} = \{ \quad & \text{VAL(in, true)} \models \text{VAL(out, false)} \\ & \text{VAL(in, false)} \models \text{VAL(out, true)} \ \} \end{aligned}$$

The translation for *stuckAtZero* and *stuckAtOne* remains unchanged with regard to to the logical conjunction and disjunction.

To create an equivalent constraint for ACS we take the following steps:

1. create a new constraint with name *NotGate*
2. add the input ports in1 and in2 to constraint *NotGate*
3. add the output port out to constraint *NotGate*
4. associate the behaviour definitions for modes *nab, stuckAtZero* and *stuckAtOne* with constraint *NotGate*

5. associate behaviour modes *nab, stuckAtZero* and *stuckAtOne* of constraint *NotGate* with probabilities 0.98, 0.01 and 0.01 respectively.

6. associate behaviour modes *stuckAtZero* and *stuckAtOne* of constraint *NotGate* with cost value *notStuckCost*

5.4.4 Half Adder

Given the above defined *atomic components* we can now define the behaviour for the *composed component* Halfadder.

component Halfadder
input a,b : bool.
output c,z: bool.

subcomponent
and1: AndGate.
or1,or2: OrGate.
neg1,neg2: NotGate.

connection
s1: a -> or1(in1), and1(in1).
s2: b -> or1(in2), and1(in2).
s3: or1(out) -> neg1(in).
s4: neg1(out) ->or2(in1).
s5: and1(out) -> or2(in2), c.
s6: or2(out) -> neg2(in).
s7: neg2(out) -> z.
end component

1. declare variables s1 ... s7

2. connect variable s1 with ports in1 of OrGate or1 and in1 of AndGate and1

3. connect variable s2 with ports in2 of OrGate or1 and in2 of AndGate and1

4. connect variable s3 with ports out of OrGate or1 and in of NegGate neg1

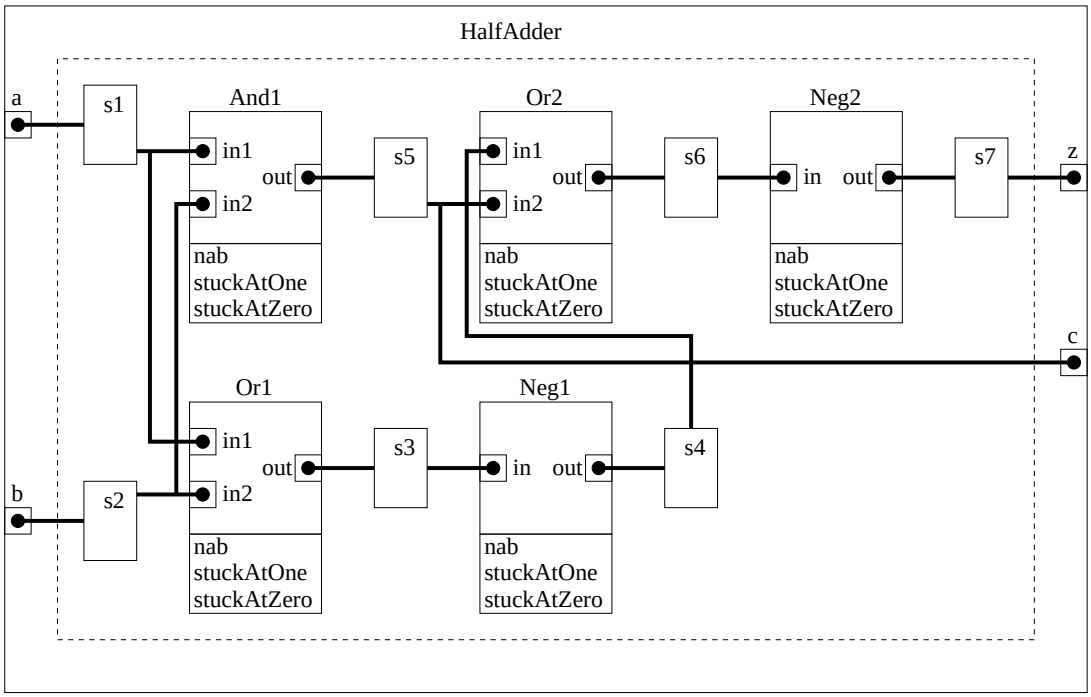


Figure 5.4: a composed constraint for a logical half adder

5. connect variable s4 with ports out of NegGate neg1 and in1 of OrGate or2
6. connect variable s5 with ports out of AndGate and1 and in2 of OrGate or2
7. connect variable s6 with ports out of OrGate or2 and in of NegGate neg2
8. connect variable s7 with port out of NegGate neg2
9. convert the connected constraints to a composed constraint with name *HalfAdder*
10. associate a data structure with constraint *HalfAdder*, describing how the ports are connected to the ports of the subconstraints, e.g. the association list ((a,s1), (b,s2), (c,s5), (z, s7)).

Figure 6.4 shows the resulting hierarchical constraint for the digital half adder.

Chapter 6

Complexity

> The Reader may here observe the Force of Numbers, which can be successfully applied, even to those things, which one would imagine are subject to no Rules. There are very few things which we know, which are not capable of being reduc'd to a Mathematical Reasoning; and when they cannot it's a sign our knowledge of them is very small and confus'd; and when a Mathematical Reasoning can be had it's as great a folly to make use of any other, as to grope for a thing in the dark, when you have a Candle standing by you.
>
> — John Arbuthnot —

In the previous chapter we showed how AD^2L rules are translated to corresponding conditionals within a constraint systems. Now we are interested in the complexity of this translation, that is the number of conditionals which have to be generated for a given rule and of how many conjunct each resulting conditional consists. We will start by defining the complexity of possible left-hand sides of a rule and finally arrive at the complexity definition for a complete rule.

6.1 Predicate Sentences

For all predicate sentences Σ not including variables the sentence-complexity $s(\Sigma)$ is defined as

$$s(\Sigma) = |acs(\Sigma)| = 1$$

Since predicate sentences consist of a single predicate expression we find the number $c(\Sigma)$ of conjuncts as

$$c(\Sigma) = 1$$

6.2 Existentially Quantified Sentences

Given the sentence

$$\Sigma = \text{exists } X_1, \ldots X_n : \sigma_1, \ldots, \sigma_n$$

the corresponding sentence-complexity $s(\Sigma)$ is

$$s(\Sigma) = \sum_{I \in \Omega_{X_1,\ldots,X_n}} \prod_{i=1}^{n} s(\sigma_i^I)$$

since for every $I \in \Omega_{X_1,\ldots,X_n}$ the cardinality of the set $acs(\sigma_i^I)$ is the same we can rewrite the formula to:

$$s(\Sigma) = |\Omega_{X_1,\ldots,X_n}| * \prod_{i=1}^{n} s(\sigma_i^I), I \in \Omega_{X_1,\ldots,X_n}$$

where the number of conjuncts in each resulting sentence is given as:

$$\sum_{i=1}^{n} c(\sigma_i^I), I \in \Omega_{X_1,\ldots,X_n}$$

6.3 Universally Quantified Sentences

Given the sentence

$$\Sigma = \text{forall } X_1, \ldots X_n : \sigma_1, \ldots, \sigma_m$$

the corresponding sentence-complexity $s(\Sigma)$ is

$$s(\Sigma) = \prod_{i=1}^{n} s(\sigma_i^I)$$

where the number of conjuncts in each resulting sentence is given as:

$$c(\Sigma) = \sum_{I \in \Omega_{X_1,\ldots,X_n}} \sum_{i=1}^{n} c(\sigma_i^I)$$

Since for every $I \in \Omega_{X_1,\ldots,X_n}$ the cardinality of the set $acs(\sigma_i^I)$ is the same we can rewrite the formula to:

$$c(\Sigma) = |\Omega_{X_1,\ldots,X_n}| * \sum_{i=1}^{n} c(\sigma_i^I), I \in \Omega_{X_1,\ldots,X_n}$$

6.4 Conjunctive Sentences

Given a conjunctive sentence Σ with

$$\Sigma = \sigma_1, \ldots, \sigma_n$$

the sentence-complexity $s(\Sigma)$ is defined as

$$s(\Sigma) = \prod_{i=1}^{n} s(\sigma_i)$$

and the number of conjuncts is given as

$$c(\Sigma) = \sum_{i=1}^{n} c(\sigma_i)$$

6.5 Implication Sentences

Given an implication sentence Σ with

$$\Sigma = \sigma_1 \text{ :- } \sigma_2$$

the sentence-complexity $s(\Sigma)$ is defined as

$$s(\Sigma) = s(\sigma_1) * s(\sigma_2)$$

For the number of predicates we get:

$$c(\Sigma) = c(\sigma_1) + c(\sigma_2)$$

6.6 Rules

- Given the rule Γ with

$$\Gamma = \Sigma_L := \sigma_R$$

the corresponding sentence-complexity $s(\Gamma)$ is defined as

$$s(\Gamma) = s(\Sigma_L)$$

and the number of predicates $c(\Gamma)$ is given as

$$c(\Gamma) = c(\Sigma_L) + c(\Sigma_R)$$

- Given the rule Γ with

  $$\Gamma = \Sigma_L =:= \sigma_R$$

  we find $s(\Gamma)$ as

  $$s(\Gamma) = s(\Sigma_L) + s(\Sigma_R)$$

  As stated before, $acs(\Sigma)$ may only contain one element, if Σ is used as the right-hand side of a rule, thus resulting in $s(\Sigma) = 1$. Since Σ_L and Σ_R both are left and right-hand side of a rule we can simplify the definition of $s(\Gamma)$ resulting in

  $$s(\Gamma) = 2$$

  The number of predicates $c(\Gamma)$ is given as

  $$c(\Gamma) = (c(\Sigma_L) + c(\Sigma_R)) * 2$$

Now that we have defined the functions s and c, we can tell for each rule expressible in AD^{2}L how many sentences will be generated for the simple constraint system and of how many conjuncts they consist. In particular, when using rules containing quantified sentences, a large number of sentences for the constraint system has to be generated. This indicates that behaviour modes can be described in a more compact manner using AD^{2}L.

Chapter 7

Translation to Horn Clauses

> Everything should be made as simple as possible, but not simpler.
>
> — Einstein, Albert (1879-1955) —

A very useful class of sentences for which a polynomial-time inference procedure exists are called **Horn clauses**[1]. A Horn clause is of the form

$$P_1 \wedge P_2 \wedge \ldots \wedge P_n \rightarrow Q$$

for non negated atoms P_i and Q.

A very important as well as simple inference rule associated with the notion of Horn clauses is called **Modus Ponens**[2] which has the following form:

$$\frac{\mathrm{P} \rightarrow \mathrm{Q}, \quad \mathrm{P}}{\mathrm{Q}}$$

In other words, from an implication and the premise of the implication, we can infer the conclusion.

Not every knowledge base can be expressed in terms of Horn clauses, but for those that can, we can use a simple inference procedure: apply Modus Ponens wherever possible until no new inferences can be made.

Because of the importance of the class of Horn clauses in inference systems an important question is whether we can find a mapping from rules in AD^{2}L to a

[1] Also known as **Horn sentences**, in honour of the logician Alfred Horn.

[2] Some authors write **Implication-Elimination** instead of **Modus Ponens**

set of such sentences. We have seen above that the sentences derived by applying the function *acs* are already very similar to Horn clauses, with two exception:

1. the resulting sentences when applying the function *acs* may have the form:

$$P_1 \wedge P_2 \wedge \ldots \wedge P_n \rightarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n$$

   Since Horn clauses allow only for a single literal as the consequence of the implication, we have to associate the above sentence with a set of sentences having the same left-hand side and a single consequence literal as the right-hand side.

   For this reason we define a function *hc* which maps each sentence of form

$$\Sigma = P_1 \wedge P_2 \wedge \ldots \wedge P_n \rightarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n$$

   to

$$hc(\Sigma) = \{P_1 \wedge P_2 \wedge \ldots \wedge P_n \rightarrow Q_i | 1 \leq i \leq n\}$$

   In other words, the function *hc* replicates the sentence Σ for each possible consequence literal.

2. As we stated above Horn clauses may not contain negated atoms, thus making it impossible to find a mapping for rules in $\mathrm{AD}^2\mathrm{L}$ containing implication sentences. If we restrict ourself to a subset of $\mathrm{AD}^2\mathrm{L}$ containing no implication sentences, we are always able to find a corresponding set of Horn clauses for each rule in this subset. We will denote this restricted subset by $\mathrm{AD}^2\mathrm{L}_{hc}$

For each rule $\Gamma \in \mathrm{AD}^2\mathrm{L}_{hc}$ the set of corresponding Horn clauses can be found by applying the function *hc* to the result of $acs(\Gamma)$. The set $HC(\Gamma)$ of Horn clauses for a given rule Γ is thus defined as:

$$HC(\Gamma) = \bigcup_{\tau \in acs(\Gamma)} hc(\tau)$$

7.1 Complexity

We have already defined the complexity of AD^{2}L rules concerning the translation to the abstract constraint system described in chapter 5. Another very interesting question is how this complexity changes when translating rules from the subset of AD^{2}L containing no implication sentences to sets of Horn clauses. Specifically we are interested in how many Horn clauses we obtain during the translation and of how many literals each of they clauses consists. Fortunately, we have already done most of the work in the previous chapters and only have to slightly adapt the complexity definitions for the new target language.

When applying the function acs to a rule $\Gamma \in \text{AD}^2\text{L}_{hc}$ the resulting set of sentences has the form illustrated in table 7.1

$$\Gamma = \Sigma_R := \Sigma_L$$

$$\begin{aligned} acs(\Gamma) = \{ \quad & \sigma_{L_{11}} \wedge \sigma_{L_{12}} \wedge \ldots \wedge \sigma_{L_{1n}} \models \sigma_{R_1} \wedge \sigma_{R_2} \wedge \ldots \wedge \sigma_{R_n}, \\ & \ldots \\ & \sigma_{L_{m1}} \wedge \sigma_{L_{m2}} \wedge \ldots \wedge \sigma_{L_{mn}} \models \sigma_{R_1} \wedge \sigma_{R_2} \wedge \ldots \wedge \sigma_{R_n} \ \} \end{aligned}$$

Table 7.1: The resulting set of sentences for the translation of a rule formulated in AD^{2}L$_{hc}$ to the abstract constraint system ACS.

In order to find a corresponding representation of this set in terms of Horn clauses we have to apply the function hc to each sentence within this set. Table 7.1 shows the form of the resulting set.

We see that for each sentence $\sigma \in acs(\Gamma)$ we get $c(\Sigma_R)$ new sentences, where the left-hand side remains unchanged and the right-hand side is a single predicate expression. Note that all conjuncts in table 7.1 and table 7.1 are simple predicate expression, since we only examine rules $\Gamma \in \text{AD}^2\text{L}_{hc}$, thus avoiding the introduction of negation by implication sentences and the corresponding grouping of predicate expressions in form of negated conjunctive sentences.

Since every sentence is replicated according to the number of consequence predicates, we get the following number $s_{hc}(\Gamma)$ of resulting Horn clauses:

$$s_{hc}(\Gamma) = s(\Sigma_L) * c(\Sigma_R)$$

The number of literals $c_{hc}(\Gamma)$ in each Horn clause is easily found as

$$\Gamma = \Sigma_R := \Sigma_L$$

$$\begin{aligned} hc(acs(\Gamma)) = \{ \quad & \sigma_{L_{11}} \wedge \sigma_{L_{12}} \wedge \ldots \wedge \sigma_{L_{1n}} \models \sigma_{R_1}, \\ & \ldots \\ & \sigma_{L_{11}} \wedge \sigma_{L_{12}} \wedge \ldots \wedge \sigma_{L_{1n}} \models \sigma_{R_n}, \\ & \ldots \\ & \sigma_{L_{m1}} \wedge \sigma_{L_{m2}} \wedge \ldots \wedge \sigma_{L_{mn}} \models \sigma_{R_1}, \\ & \ldots \\ & \sigma_{L_{m1}} \wedge \sigma_{L_{m2}} \wedge \ldots \wedge \sigma_{L_{mn}} \models \sigma_{R_n} \ \} \end{aligned}$$

Table 7.2: The resulting set of sentences for the translation of a rule formulated in $\mathrm{AD^2L}_{hc}$ to Horn clauses.

$$c_{hc}(\Gamma) = c(\Sigma_l) + 1$$

7.2 Summary

We have shown that for behavioural models defined in the language $\mathrm{AD^2L}_{hc}$, the implication-free subset of $\mathrm{AD^2L}$, a mapping can be defined to an equivalent set of Horn-clauses. Based on the results of chapter 6, we have defined the functions s_{hc} and c_{hc} determining the number of Horn-clauses that will be generated for a given rule, and the number of literals in a clause, respectively.

Chapter 8

Discussion and Conclusion

> We have a habit in writing articles published in scientific journals to make the work as finished as possible, to cover up all the tracks, to not worry about the blind alleys or describe how you had the wrong idea first, and so on. So there isn't any place to publish, in a dignified manner, what you actually did in order to get to do the work.
>
> — Richard Philips Feynman (1918 - 1988) —

The primary concern of this thesis was to develop an abstract language, called AD^2L, that allows for efficient and comfortable system modelling. AD^2L is meant to enable knowledge engineers to model the domain under consideration independently of the way this knowledge is used in following steps. We have shown that every model defined in AD^2L can be translated to a corresponding model for a constraint system, supporting simple conditional statements like *if* α *then* β, where α is a conjunction of negated or nonnegated conditions. Furthermore we have shown that for the implication-free subset AD^2L_{hc} of AD^2L, each model can be translated to a corresponding set of Horn-clauses. Finally we have defined the complexity for the translation of rules in terms of the resulting number of constraint system rules or Horn clauses and the number of corresponding literals.

8.1 Comparison to Related Work

This section is dedicated to the discussion of a very influential methodology in research on KE, underlying many modelling languages. We will discuss what parallels can be drawn to the concept underlying AD^2L.

A well known fact in the field of Software Engineering is that two fundamental premises should underlie the construction of complex software systems:

- A clear distinction should be made between the functional specification of a software system and its design and implementation
- Functional specifications should as far as possible be given in a precise, unambiguous language that abstracts from implementational details.

Knowledge Engineering, a sub-branch of SE, which concentrates on the construction of Knowledge Based Systems, matured to a separate discipline in the last years. The above stated premises have soon be realised to be of the same importance for the field of Knowledge Engineering and gave rise to the study of more principled methodologies for Knowledge Based System construction. The term *functional specification* used in the field of SE has a number of equivalents in KE: a *conceptual model*, a *model of expertise* or a *knowledge level description*. This abstract and implementational independent description has become the basis of a number of KBS development methodologies. A particular type of conceptual model, derived from the KADS model of expertise, was introduced in [13]. KADS has become a very influential methodology in the field of KE and thus has led to a number of languages formalising the KADS models of expertise. Figure 8.1 shows the structure of the KADS model of expertise.

A model of expertise consists of four layers which are hierarchically organised. We will in short describe the contents of each layer.

- *Domain layer*: This is the layer on the bottom of the layer hierarchy, and represents knowledge about the domain of the system. A very important property of the domain layer is the way the knowledge is modelled — it should be represented in a way that is as independent as possible from the way it will be used. The domain layer serves two purposes. First it should define a conceptualisation of the domain and second it must provide all the domain knowledge required to solve the given tasks.

knowledge category	**organisation**	**knowledge types**
strategic	strategies	plans and meta rules
↓ *controls*		
task	tasks	goals, control terms, task structure
↓ *applies*		
inference	inference structure	inference actions, knowledge roles
↓ *uses*		
domain	domain theory	concept, property and relation

Figure 8.1: The four layer model of expertise

- *Inference Layer*: The direct predecessor of the domain layer in the hierarchy is the inference layer. It specifies the inference steps that constitute a problem-solving method and describes how to use the knowledge from the domain layer. This is done in two ways: the inference layer specifies the basic inference steps that can be made using the domain knowledge, and the *knowledge roles* model the premises and conclusions of the inferences. Although the inference layer specifies the basic inference steps there is no order implied concerning their application.

- *Task layer*: A task is a strategy to solve a given problem. The purpose of the task layer is to specify control over the execution of the basic inference steps defined at the inference layer. This can be done in terms of iterations, sequential application, conditional statements etc.

- *Strategy layer*: This layer, which is on top of the hierarchy, is responsible for task selection. It has to choose between various problem solving mechanisms (=tasks) which achieve the same goal.

Obviously AD^{2}L can be used to formalise the first two layers. It uses predicates, functions and single- and set-valued variables at the domain layer, and first order rules at the inference layer. The task layer and the strategy layer are included within the diagnosis system.

8.2 Future Work

The most important next step to find out the suitability of the language is to implement a corresponding compiler for a given diagnosis system. We will then use feedback from practical use to improve and enhance the language in order to make it even more attractive for knowledge engineers to use AD^{2}L.

Another very important issue is the representation of temporal information. Reasoning about change is an important AI research field, which aims to study common-sense reasoning about phenomena changing over time. Since such reasoning is crucial for a number of other AI research fields such as planning and diagnosis, we will have to find a way how to incorporate this information in AD^{2}L models.

Bibliography

[1] Raymond Reiter. *A Theory of diagnosis from first principles.* Artificial Intelligence, 32(1):57-96, April 1987.

[2] Russell Greiner, Barbara A. Smith and Ralph W. Wilkerson. *A correction to the algorithm in Reiter's theory of diagnosis.* Artificial Intelligence, 41(1):79-88, November 1989.

[3] Bruce G. Buchanan and Edward H. Shortliffe, editors. *Rule based Expert System - The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley Publishing Company, 1984

[4] Gerhard Friedrich, Georg Gottlob and Wolfgang Nejdl. *Physical impossibility instead of fault models.* In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 331-336, Boston, August 1990.

[5] Peter Struss and Oskar Dressler. *Physical negation — Integrating fault models into the general diagnostic engine.* In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1318-1323, Detroit, August 1989. Morgan Kaufmann Publishers, Inc.

[6] Luca Console and Pietro Torasso. *A spectrum of logical definitions of model-based diagnosis. Computational Intelligence*, 7(3):133-144, 1991.

[7] Russell Greiner, Barbara A. Smith and Ralph W. Wilkerson. *A correction to the algorithm in Reiter's theory of diagnosis.* Artificial Intelligence, 41(1):79-88, November 1989.

[8] Gerhard Friedrich and Wolfgang Nejdl. *MOMO — Model-based diagnosis for everybody.* In *Proceedings of the 6th IEEE Conference on AI Applications*, pages 206-213, Santa Barbara, CA, March 1990.

[9] Johan de Kleer and Brian C. Williams. *Diagnosing with behavioural modes.* In *Proceedings of the International Joint Conference on Atrificial Intelligence*, pages 1324-1330, Detroit, August 1989. Morgan Kaufmann Publishers, Inc.

[10] Peter Struss. *Diagnosis as a process* In *Readings in Model-Based Diagnosis*, pages 408-418. Morgan Kaufmann Publishers, Inc., 1992.

[11] W.Hamscherm, Luca Console, J. de Kleer, editors. *Readings in Model-Based Diagnosis*. Morgan Kaufmann Publishers, Inc., 1992.

[12] Stuart Russel and Peter Norvig. *Artificial Intelligence, A Modern Approach*, pages 413-522. Prentice Hall, Inc., 1995.

[13] B.J.Wielinga, A.Th.Schreiber and J.A.Breuker. *K*ADS: A Modelling Approach to Knowledge Engineering. In *K*nowledge Acquisition, vol 4, no 1, 1992.

[14] Markus Stumptner and Franz Wotawa. *Diagnosing Tree-Structured Systems* in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1997.

[15] Yousri El Fattah and Rina Dechter. *Diagnosing tree-decomposable circuits* in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1995.

[16] Adnan Darwiche. *Model-Based Diagnosis using Causal Networks* in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1995.

[17] Peter Fröhlich and Wolfgang Nejdl. *A Static Model-Based Engine for Model-Based Reasoning* in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1997.