PM-1 3 1/2"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file - Votre référence

Our file - Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

LOGIC PROGRAMS FOR CONSISTENCY-BASED DIAGNOSIS

by

Gregory W. Bond, B.Sc., M.Eng.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada

July 1, 1994

© Gregory W. Bond 1994

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

THE AUTHOR HAS GRANTED AN
IRREVOCABLE NON-EXCLUSIVE
LICENCE ALLOWING THE NATIONAL
LIBRARY OF CANADA TO
REPRODUCE, LOAN, DISTRIBUTE OR
SELL COPIES OF HIS/HER THESIS BY
ANY MEANS AND IN ANY FORM OR
FORMAT, MAKING THIS THESIS
AVAILABLE TO INTERESTED
PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE
IRREVOCABLE ET NON EXCLUSIVE
PERMETTANT A LA BIBLIOTHEQUE
NATIONALE DU CANADA DE
REPRODUIRE, PRETER, DISTRIBUER
OU VENDRE DES COPIES DE SA
THESE DE QUELQUE MANIERE ET
SOUS QUELQUE FORME QUE CE SOIT
POUR METTRE DES EXEMPLAIRES DE
CETTE THESE A LA DISPOSITION DES
PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP
OF THE COPYRIGHT IN HIS/HER
THESIS. NEITHER THE THESIS NOR
SUBSTANTIAL EXTRACTS FROM IT
MAY BE PRINTED OR OTHERWISE
REPRODUCED WITHOUT HIS/HER
PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE
DU DROIT D'AUTEUR QUI PROTEGE
SA THESE. NI LA THESE NI DES
EXTRAITS SUBSTANTIELS DE CELLE-
CI NE DOIVENT ETRE IMPRIMES OU
AUTREMENT REPRODUITS SANS SON
AUTORISATION.

ISBN 0-315-98441-4

Name GREGORY W. BOND

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science

SUBJECT TERM

0984

U·M·I

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0725
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language
General 0679
Ancient 0289
Linguistics 0290
Modern 0291
Literature
General 0401
Classical 0294
Comparative 0295
Medieval 0297
Modern 0298
African 0316
American 0591
Asian 0305
Canadian (English) 0352
Canadian (French) 0355
English 0593
Germanic 0311
Latin American 0312
Middle Eastern 0315
Romance 0313
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
General 0318
Biblical Studies 0321
Clergy 0319
History of 0320
Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
Archaeology 0324
Cultural 0326
Physical 0327
Business Administration
General 0310
Accounting 0272
Banking 0770
Management 0454
Marketing 0338
Canadian Studies 0385
Economics
General 0501
Agricultural 0503
Commerce-Business 0505
Finance 0508
History 0509
Labor 0510
Theory 0511
Folklore 0358
Geography 0366
Gerontology 0351
History
General 0578

Ancient 05/9
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
General 0473
Agronomy 0285
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0777
Wood Technology 0746
Biology
General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0379
Ecology 0329
Entomology 0353
Genetics 0369
Limnology 0793
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oceanography 0416
Physiology 0433
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0992
Dentistry 0567
Education 0350
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Therapy 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry
General 0485
Agricultural 0749
Analytical 0486
Biochemistry 0487
Inorganic 0488
Nuclear 0738
Organic 0490
Pharmaceutical 0491
Physical 0494
Polymer 0495
Radiation 0754
Mathematics 0405
Physics
General 0605
Acoustics 0986
Astronomy and Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463

Applied Sciences

Applied Mechanics 0346
Computer Science 0984

Engineering
General 0537
Aerospace 0538
Agricultural 0539
Automotive 0540
Biomedical 0541
Chemical 0542
Civil 0543
Electronics and Electrical 0544
Heat and Thermodynamics 0348
Hydraulic 0545
Industrial 0546
Marine 0547
Materials Science 0794
Mechanical 0548
Metallurgy 0743
Mining 0551
Nuclear 0552
Packaging 0549
Petroleum 0765
Sanitary and Municipal 0554
System Science 0790
Geotechnology 0478
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

Carleton University
Ottawa, Canada K1S 5B6

Text complete; leaf // omitted in numbering.

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

"LOGIC PROGRAMS FOR CONSISTENCY-BASED DIAGNOSIS"

submitted by
Gregory W. Bond, B.Sc., M.Eng.

in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

External Examiner

Carleton University
August 26, 1994

Carleton University
Ottawa, Canada K1S 5B6

Text complete; leaf 1/ omitted in numbering.

Abstract

In this dissertation we develop a solution to the problem of automating systems diagnosis, where we define diagnosis to be the task of identifying potentially faulty system components given assertions regarding intended or erroneous system behavior. The solution is applicable to systems that are characterized by the fact that (i) there exists no *a priori* knowledge concerning the modes or probability of component failure, and (ii) either there does not exist complete knowledge of the system's specification or observability of the system itself is restricted. Domains for which there exist significant numbers of systems satisfying these properties are software development domains, hardware development domains and domains concerned with the development of system specifications. The solution to the problem takes the form of a theoretical framework and a number of polynomial-time procedures for computing the diagnoses defined by the framework. The framework we develop is an instance of an existing diagnostic approach known as consistency-based diagnosis. Diagnoses are defined with respect to a set of first-order assertions and a system description based on a logic program. Using logic programs as the basis for system descriptions presents a number of advantages including ease of model construction, the ability to diagnose hierarchically structured systems and software systems, and the automatic definition of negative system behavior. Two necessary and sufficient conditions for component abnormality are defined and shown to reflect intuitive notions of faults occurring in hierarchies. We also formally show that the approach to logic program diagnosis known as declarative error diagnosis is a specialized instance of the consistency-based diagnosis framework developed here. One of the two diagnosis computation procedures developed for the framework efficiently computes diagnoses for a restricted class of assertions. We prove this procedure sound, identify conditions for completeness and prove its implementation correct. The other, more complex, procedure computes diagnoses for a more general class of assertions. We prove this procedure to be sound, discuss completeness issues and prove its implementation correct. Both procedures are implemented in Prolog and are used to compute diagnoses for a combinational logic circuit, a sequential logic circuit and a buggy logic program.

Carleton University
Ottawa, Canada K1S 5B6

Text complete; leaf vi omitted in numbering.

Acknowledgements

First and foremost, I thank Mom for applying to engineering schools on my behalf after I had given up. I thank Deirdre for understanding that sometimes it takes time to return from "space." I thank Bernie Pagurek for giving me the time I needed (and time I didn't always need) to do my work properly. I thank Gerald Karam for serving as my first academic role-model. I thank Vivienne, Darlene, Janet, Christy ... for their good natured administrative support. Finally, I thank NSERC and TRIO for supporting my research.

Carleton University
Ottawa, Canada K1S 5B6

Text complete; leaf ,||| omitted in numbering.

Contents

List of Figures	xvii
List of Symbols	xix
Chapter I. Introduction	1
1. Diagnosis	1
2. Problem	1
2.1. Software Development	3
2.2. Hardware Development	3
2.3. Specification Development	3
2.4. Other Considerations	4
3. Possible Approaches to Diagnosis	4
3.1. Consistency-Based Diagnosis	5
3.2. Declarative Error Diagnosis	5
3.3. IDA	6
3.4. "Model-Based Diagnosis of Logic Programs"	6
4. Results	6
4.1. Theoretical Framework	7
4.2. Diagnosis Computation	8
5. Overview	9
Chapter II. Logic and Logic Programming	11
1. First-order Languages	11

2. Logic Programs	12
3. Semantics	13
4. Herbrand Interpretations	15
5. Substitutions	17
6. SLD-resolution	18
6.1. Selection Rules	19
6.2. SLD-trees	19
6.3. Existentially Quantified Goals	20
7. Prolog	22
8. Negative Information	24
9. Equality Theories	28
Chapter III. Declarative Error Diagnosis	31
1. Procedural Ontology	31
2. Errors	32
3. Faults	33
3.1. Procedural Faults	33
3.2. Faults in Logic Programs	36
4. Diagnosis	37
4.1. FP Algorithm	38
4.2. IP Algorithm	39
Chapter IV. Consistency-Based Diagnosis	43
1. A Diagnosis	43
2. Minimal Diagnoses	44
3. Characterization of Diagnoses	46
4. Fault Models	47

5. Computing Diagnoses	47
5.1. ATMS	47
5.2. Generate-and-Test	48
5.2.1.	
5.2.2.	
5.2.3.	
Chapter V. A Diagnostic Framework Using Logic Programs	53
1. Introduction	53
2. Constructing the System Description	54
2.1. Step 1	55
2.2. Step 2	56
2.3. Step 3	57
3. Assertions	57
4. Consistency-Based Diagnosis	59
5. Characterizing Diagnoses	61
6. Properties of Abnormalities	61
7. Properties of Diagnoses	62
8. Structural Hierarchies	70
9. Negative Information	71
10. Discussion	71
11. Relationship to Declarative Error Diagnosis	73
11.1. Properties of Abnormalities	73
11.2. Granularity of Faults	74
11.3. Expressiveness of Π	75
11.4. Negative Information	75
11.5. Intended and Actual Interpretations	76
11.6. Knowledge of an Interpretation	76
11.7. Computing Diagnoses	77
11.8. Proofs	78

12. Relationship to Model-Based Diagnosis of Logic Programs	82
13. Relationship to IDA	83
14. Summary	84
Chapter VI. Computing Diagnoses With Labelled System Descriptions	85
1. Introduction	85
2. IDA	86
3. Labelled System Descriptions	89
3.1. Examples	90
3.2. Discussion	92
3.3. Meta-Functions	94
3.3.1	
3.3.2.	
3.3.3.	
3.3.4.	
3.3.5.	
3.4. Transforming Π to a Labelled System Description	95
4. Diagnosis Verification	96
4.1. Examples	96
4.2. Soundness and Completeness	97
4.2.1.	
4.2.2.	
4.3. Decidability	101
4.3.1.	
4.3.2.	
4.4. Proofs	103
4.4.1.	
4.4.2.	
4.4.3.	
4.4.4.	
4.4.5.	

5. Diagnosis Computation	119
5.1. Using a Generate-and-Test Approach	120
5.2. Properties of Abnormalities	121
5.3. Complexity and Efficiency	122
5.4. Expressiveness of Assertions	123
6. Comparison To Related Work	124
6.1. IDA	124
6.2. Declarative Error Diagnosis	124
7. Summary	125
Chapter VII. Computing Diagnoses for General Assertions	127
1. Introduction	127
2. Positive Δ-Supports	129
3. The Verification Procedure	134
4. Implementation	136
4.1. Implementation and Correctness of Step 1	136
4.2. Implementation and Correctness of Steps 2 and 3	138
4.3. Implementation and Correctness of Step 4	148
4.4. Summary	151
5. Computing Diagnoses	151
5.1. Complexity and Efficiency	152
5.2. Properties of Abnormalities	152
5.3. Expressiveness of Assertions	152
5.4. Examples	152
6. Comparison to Related Work	153
6.1. Completeness	153
6.2. Assertions	154
6.3. Implementation Issues	154
7. Summary	155
Chapter VIII. Conclusions	157

1. Summary of Results	157
1.1. Theoretical Framework	157
1.2. Diagnosis Computation	159
2. Future Research	160
2.1. Negative Information	160
2.2. Fault Models	161
2.3. Search Strategies	161
2.4. Measurement, Tests, Repair and Correction	162
2.5. Incremental Diagnosis Computation	162
Bibliography	163
Appendix A. Implementation of the TP Diagnosis Verification Procedure	171
Appendix B. Interface to the IDA Generator Algorithm	177
1. diagnose.pl	177
2. customize.pl	178
3. pi-io.pl	178
4. ARIAI Software Disclaimer	180
5. ARIAI User Agreement	180
6. my-init.pl	181
7. my-kdiags.pl	182
Appendix C. Computing Diagnoses for the Buggy Insertion Sort Program	185
1. isort.pl	185
2. isort-clause.pl	186
3. isort-ex.pl	187
4. Output for Example 1	188
5. Output for Example 2	189

Appendix D. Computing Diagnoses for the Full-Adder Circuit	193
1. full-adder.pl	193
2. full-adder-ex.pl	194
3. Output for Example 1	195
4. Output for Example 2	198
Appendix E. Computing Diagnoses for a Sequential Multiplier Circuit	201
1. multiplier.pl	202
2. mult-ex.pl	210
3. Output of the Example	211

Carleton University
Ottawa, Canada K1S 5B6

Text complete; leaf ~~XV~~ omitted in numbering.

List of Figures

I.1	Diagnosis as part of the debug/validation cycle	2
I.2	Model-based diagnosis	5
II.3	Schematic of two inverters in series	16
II.4	Example of an SLD-tree	21
III.5	Universe of behavior for a faulty logic program Π with respect to an intended interpretation M	32
IV.6	Schematic of a normal combinational full-adder circuit	46
IV.7	Lattice of positive <i>ab</i> -clauses for the full-adder circuit	48
V.8	Relationships between artifacts, specifications and their system descriptions	54
V.9	Specified structural interconnections of a full-adder circuit	55
V.10	An <i>and</i> gate	55
V.11	Actual structural interconnections of a faulty full-adder circuit	58
V.12	Relationship between theorems and their supporting lemmas	65
VI.13	Overview of relationship between theorems and supporting lemmas for negative assertions	106
VI.14	Overview of relationship between theorems and supporting lemmas for positive assertions	117

VII.15 Overview of relationship between Theorem VII.19 and its supporting theorems and lemmas	139
E.16 Structure of a 4X4 bit sequential multiplier circuit	201

List of Symbols

true, false	propositional constants	11
$\exists X^n F, \forall X^n F$	quantification short-form	12
$\exists F, \forall F$	quantification short-form	12
$p(t^n), p(t)$	atom	12
$e_1 \equiv e_2$	term identity	12
$\leftarrow G$	goal G	13
$\square$	empty clause	13
Π	logic program	13
$L(\Pi)$	language of Π	13
$\mu(p)$	number of clauses in Π with p head	13
$\text{PRED}(\Pi)$	set of all predicate symbols in Π	13
$I \models F$	formula F true in interpretation I	14
$\models S$	set of formulas S is valid	15
$S \models S'$	S' a semantic consequence of S	15
$H(\Pi)$	Herbrand universe of Π	15
$\hat{H}(\Pi)$	Herbrand base of Π	15
$M(\Pi)$	least Herbrand model of Π	16
θ, η, δ	substitutions	17
X/t	variable X bound to term t	17
$\text{dom}(\theta)$	set of variables appearing in domain of θ	17
$\text{vars}(E)$	set of variables appearing in an expression E	17
$\text{SS}(\Pi)$	success set of Π	18
$R(H)$	selection rule R applied to derivation history H	19

CWA(Π)	closed world assumption set of Π	24
NAF(Π)	negation as failure set of Π	25
FF(Π)	finite failure set of Π	25
<i>comp</i> (Π)	predicate completion of Π	25
IF(Π)	IF-definitions of Π	27
IFF(Π)	IF-AND-ONLY-IF definitions of Π	27
OIF(Π)	ONLY-IF definitions of Π	27
EQ(Π)	free equality axioms for Π	27
Eq	equality axioms	29
M	intended interpretation of Π	32
SD	system description	43
COMPS	set of system components	43
OBS	set of assertions	43
Δ	set of potentially abnormal components	44
<i>ab</i>	<i>abnormal</i> predicate symbol	44
$D(C_p, C_n)$	conjunction of <i>ab</i> literals	44
TP	diagnosis verification function	51
SD(Π)	system description based on Π	54
COMPS(Π)	set of predicate symbols in Π	55
INCORRECT(<i>p</i>)	component <i>p</i> incorrect	62
INCOMPLETE(<i>p</i>)	component <i>p</i> incomplete	62
<i>comp</i> $_{\Delta}$ (Π)	Δ -completion of Π	64
SD-IF(Π)	IF-definitions of SD(Π)	66
SD-OIF(Π)	ONLY-IF-definitions of SD(Π)	66
<i>comp-if</i> $_{\Delta}$ (Π)	IF-definitions of <i>comp</i> $_{\Delta}$ (Π)	66
<i>comp-oif</i> $_{\Delta}$ (Π)	ONLY-IF-definitions of <i>comp</i> $_{\Delta}$ (Π)	66
gl	label generation function	94
s	component state generation function	94
d	ground label generation function	94
al	label application function	94
rl	label removal function	95
LSD ⁻ (Π)	negative labelled system description for Π	95
LSD ⁺ (Π)	positive labelled system description for Π	95

Π_{Δ}^{-}	negative Δ -transform of Π	104
Π_{Δ}^{+}	positive Δ -transform of Π	104
$N^{\sim}$	the conjunctive form of a goal	136
$\square^{\sim}$	true	136
R_{Δ}	constrained selection rule	138

CHAPTER I

Introduction

1. Diagnosis

Systems diagnosis is the act of identifying potentially faulty system components given assertions regarding intended or erroneous system behavior. The act of diagnosis is normally carried out in an iterative fashion as part of a system debugging/validation cycle as shown in Figure I.1.

The diagnostic task requires detailed knowledge of the system being diagnosed and, for complex systems, usually only those associated with the development of the system have the requisite knowledge. However, diagnostic reasoning is a skill in itself, not always possessed by those intimate with the system undergoing diagnosis. For these reasons diagnosis can be a time consuming task involving highly skilled personnel.

In an attempt to circumvent diagnosis, some systems are designed and implemented using formal techniques which allow the design to be verified against a more abstract specification of system structure and behavior. In this way, one's confidence regarding the correctness of the implementation is raised. However, for complex systems, verification techniques will never be able to provide 100% assurance of correct behavior for the simple fact that exhaustive verification would take an unacceptably long time. Even assuming that 100% assurance could be had, faults are still possible because the specification, upon which the implementation is based, could be flawed.

Given that diagnosis is a task that will always be with us, and given that the task is inherently costly in terms of resources, it is an obvious candidate for automation.

2. Problem

The problem solved in this dissertation is the automation of diagnosis in the software development domain, the hardware development domain and in the domain of specification development. Aspects of diagnosis we do not investigate in this dissertation are suggesting corrections associated with a diagnosis, or suggesting tests or measurements to discriminate amongst candidate diagnoses. For systems in the domains we have identified, we argue that a significant number share the

I. INTRODUCTION

FIGURE I.1. Diagnosis as part of the debug/validation cycle

following properties which we take to be the assumptions for our problem:

- (1) no *a priori* knowledge exists of possible modes of component failure or probabilities of component failure,
- (2) either knowledge of specified system behavior is incomplete or observability of the system behavior itself is restricted.

Since we are interested in automating diagnosis, when we refer to "knowledge" above, we mean facts with an unambiguous meaning that are accessible by a computer. This can take the form of a knowledge-base resident in the computer or it can take the form of user responses to computer generated queries. It is our intention to develop a general diagnostic approach that does not make any assumptions about the availability of information in the domain. In fact, the two properties stated above are simply the negations of properties assumed to hold for systems in order to use most existing diagnostic approaches. As a consequence, systems satisfying the two properties cannot be diagnosed using existing approaches. We feel that a substantial number of systems exist that satisfy these two properties and therefore, automating diagnosis for this class of systems is a significant contribution.

We now describe some diagnostic domains whose systems satisfy these properties.

2.1. Software Development. The notion of typical failure modes, a notion derived from the domain of hardware diagnosis, is for the most part inapplicable to software components. Faulty software is not due to worn out software components, rather, it is due to a perceived inconsistency between the normal software behavior and its intended behavior. Furthermore, in the absence of comprehensive testing, failure probabilities of software components cannot be estimated. For these reasons the first property is satisfied by the majority of software systems. With regards to the second property, in a software development environment many software systems effectively permit unrestricted observability using debugging tools.

However, when the specification itself is either informal or non-existent and the diagnostician is unable (or unwilling) to answer every query regarding the specifications put to her by the computer then knowledge of specified system behavior is effectively incomplete. We feel this is the case for the majority of software systems undergoing development today. Software systems with complete formal specifications—systems which are, needless to say, in the minority—do not satisfy the second property and are therefore not considered here.

2.2. Hardware Development. During hardware development, typical failure characteristics or mean time between failure data are not always readily available—particularly for custom designed components. Also, since observability is restricted for most hardware systems, the hardware development domain generally possesses the second property.

2.3. Specification Development. The development of a hardware or software specifications is another domain whose systems satisfy the first property since, similar to the software development domain, a fault in a specification is not due to a worn out component but is instead due to a perceived inconsistency between the actual specified behavior and the intended specified behavior. As such, modes of component failure and probability of component failure are not applicable notions.

In many cases systems in this domain may also satisfy the second property. One can assume that the behavior of the specification itself is completely observable. However, when a specification does not have a specification of its own and the diagnostician is not in a position to answer all queries regarding the intended

specified behavior then one can conclude that knowledge of the specified system behavior is incomplete.

2.4. Other Considerations. Any approach to software systems diagnosis must also consider a number of characteristics commonly shared by software systems. One such characteristic is that the components of software systems are usually hierarchically structured. For example, a procedure is usually defined in terms of calls to other procedures. When recursion is used to define a procedure *i.e.* when a procedure calls itself, then one could say that the procedure is defined in terms of a recursive hierarchy. When faults occur in a structural hierarchy, be it hardware or software, one must take care to distinguish between components that are causing erroneous behavior in the system and components that are merely propagating erroneous behavior.

3. Possible Approaches to Diagnosis

The constraints above dictate that no existing diagnostic approach is wholly suitable. However, four approaches come close to suiting our needs: *consistency-based diagnosis* (*e.g.* [24, 82, 22]) for general systems diagnosis, *declarative error diagnosis* (*e.g.* [85, 57, 56, 33, 92, 34]) primarily for logic program diagnosis, *IDA* [65] for general systems diagnosis, and most recently, an approach proposed by Console, Friedrich and Dupré [14] primarily for logic program diagnosis. Note that no abductive approaches to diagnosis (*e.g.* [78]) are suitable since they require the existence of knowledge concerning typical fault modes. As such, the first property set out in our problem description is not satisfied.

The four approaches above are considered to be instances of the general approach to diagnosis known as *model-based diagnosis*, illustrated in an operational sense in Figure 1.2. The diagnostic engine is responsible for computing the possible diagnoses of the system given the two inputs. One input consists of a set of assertions concerning the system being diagnosed—either observations of erroneous system behavior or statements regarding intended system behavior. The other input is a formal model reflecting system structure and behavior. For hardware diagnosis from specifications, the model reflects the specified system structure and behavior. For software diagnosis or diagnosis of a specification in the absence of

FIGURE 1.2. Model-based diagnosis

formal specifications this model reflects the actual structure and behavior of the faulty software of specification.

3.1. Consistency-Based Diagnosis. This approach consists of a theoretical framework and a number of different procedures for computing diagnoses. The advantages of the approach are that it accommodates diagnoses of systems that satisfy the two properties we presented above and that a large body of results exist for the approach in such areas as testing (*e.g.* [62]), measurement (*e.g.* [24]), repair (*e.g.* [71]), and exploiting knowledge of failure modes and failure probabilities (*e.g.* [25]).

In principle, consistency-based diagnosis is a general enough framework for software diagnosis. However, the approach has traditionally been used only for hardware diagnosis and not, to our knowledge, for software diagnosis. A major reason for this is that all implementations of the approach have utilized propositional logic-based diagnostic models which are not expressive enough for modelling the vast majority of software systems. In order for this approach to be used for software diagnosis an instance of consistency-based diagnosis must be defined that can compute diagnoses from first-order diagnostic models.

3.2. Declarative Error Diagnosis. This approach also consists of a theoretical framework and a number of different procedures for computing diagnoses. The major advantage of the approach is that it can be used for software or hardware diagnosis, although it has been developed primarily for logic program diagnosis. The diagnostic models used by the approach are logic programs which, in principle, can be used to model any software system [1]. As such, the approach defines faults in the context of structural hierarchies, discriminating between the cause of errors and components propagating errors.

The major disadvantage of the approach is that it does not accommodate systems that satisfy the second property given above. For software diagnosis the diagnostic model is based on the software itself (complete system observability) and complete knowledge of a system's specified behavior is required. For hardware diagnosis the diagnostic model is based on system specifications (complete knowledge of specified behavior) and unrestricted system observability is required.

3.3. IDA. This approach accommodates systems satisfying both properties above, and it uses logic programs as diagnostic models. However the IDA framework only defines diagnoses for non-hierarchically structured systems. As such, it is not possible to use the IDA framework for general software diagnosis.

3.4. "Model-Based Diagnosis of Logic Programs". This approach for logic program diagnosis, developed independent of the approach we have developed in this dissertation, accommodates systems satisfying the two properties above and is able, in principle, to be used for general hardware or software diagnosis. The diagnostic framework defines intuitive notions of diagnoses in hierarchies that subsume those defined by the declarative error diagnosis framework.

However, the diagnosis computation procedure proposed for the framework suffers from a number of problems. Foremost of these is that there exist faults that the procedure is incapable of finding that procedures for declarative error diagnosis are able find [6]. As such, the procedure has incomplete fault coverage with respect to the space of diagnoses defined by its own framework. In addition, an implementation of the procedure does not exist. Issues associated with the implementation of the procedure are examined in [6] and the conclusion drawn is that a substantial amount of research is required before an implementation can be developed and proved correct.

4. Results

As a solution to our problem, we develop a theoretical consistency-based diagnosis framework for systems satisfying the two properties defined above, including software systems. We show that the framework serves to unify a number of heretofore disparate approaches to diagnosis. We also develop polynomial-time procedures for computing diagnoses defined by the framework, implement them and prove the implementations correct. We illustrate the capability of the framework

and procedures by diagnosing a hierarchically structured combinational full-adder circuit, a Prolog insertion sort program and a complex, hierarchically structured, sequential multiplier circuit.

4.1. Theoretical Framework. We define a theoretical framework that is an instance of consistency-based diagnosis for a particular class of diagnostic models. To construct a model for the framework one simply constructs a logic program that adequately reflects the behavior of a system's actual or specified behavior. The framework defines how the program is transformed to a diagnostic model for the framework. This approach to model construction has a number of advantages. First of all, constructing a logic program to reflect system behavior will usually be straightforward since the language of logic programs has operational semantics [31]. Therefore one constructs a simulation model of the system as a logic program. Methods for doing just this have been investigated for many domains. For example [12] describes modelling combinational and sequential digital circuits, [72] describes modelling concurrent discrete event processes, [87] describes modelling the denotational semantics of an imperative programming language, [90] describes using logic programs for structured design, [28] describes transforming Z specifications (e.g. [86]) into logic programs, [52] describes an interpreter for CSP specifications [46], and [59] describes specifying communication protocols as logic programs.

The framework also defines two types of faults that can occur in a hierarchically structured system: *incorrect components* and *incomplete components*. These notions of faults apply only to abnormal components that cause errors and not to normal components that merely propagate errors. They also provide some indication as to how a faulty component should be repaired.

We also show how declarative error diagnosis is a special instance of our framework, and therefore show that declarative error diagnosis is a special instance of consistency-based diagnosis. The immediate consequence of this result is that both approaches inherit substantial bodies of results from one another. For example, our consistency-based diagnosis framework inherits numerous efficient diagnosis procedures used for declarative error diagnosis.

Although this dissertation does not deal with the topics of testing, measurement, and repair, the framework inherits the body of results that exist for consistency-

based diagnosis in these areas since it is a consistency-based diagnosis framework.

4.2. Diagnosis Computation. While we prove that procedures for declarative error diagnosis can be used to compute faults for our framework, these procedures all assume the existence of complete knowledge of specified system behavior and complete observability of the system being diagnosed. Since systems satisfying these properties do not satisfy the second property in our problem statement, we cannot use the declarative error diagnosis procedures. Instead, we develop two alternative approaches to diagnosis computation. One approach efficiently computes diagnoses for a restricted class of assertions. The other, more complex, approach computes diagnoses for a more general class of assertions. We show how using a more general class of assertion provides the diagnostician with more expressive power and helps to eliminate diagnoses that would otherwise be computed using the more restricted class of assertions.

The first approach is a generalization of the approach used by the IDA framework discussed earlier. We prove that this approach is sound, we identify conditions for completeness, and prove that the implementation of the approach is correct. Since our consistency-based diagnosis approach is a generalization of the IDA approach we are also able to identify conditions for which the IDA framework computes consistency-based diagnoses.

We prove that the second approach is sound, discuss completeness issues and prove that its implementation is correct. We also show that the procedure has better fault coverage than the procedure proposed by Console *et al.* discussed earlier.

Both procedures are based on the generate-and-test approach to diagnosis used in the IDA framework. We borrow the generator algorithm from the IDA framework and interface it to our own test procedures. The test procedures we develop are nothing more than specialized theorem provers for first-order, Horn clause logic. Using these procedures, it is possible to use a constraint-based logic programming language (*e.g.* [48]) as the basis for a diagnostic model to provide greater expressive power.

Both procedures have been implemented in Prolog and have been used to compute diagnoses for all the examples found in this dissertation.

5. Overview

This dissertation is meant to be self-contained. All background material referenced in the development of the dissertation is contained in the first three chapters. Chapter II contains an overview of logic and logic programming, Chapter III contains an overview of declarative error diagnosis, and Chapter IV contains an overview of consistency-based diagnosis. The reader familiar with any of the material should feel free to skip the appropriate sections. We have maintained standard notation and terminology wherever possible. Deviations from these are duly noted. A List of Symbols, which cross-references symbols with the page numbers they are defined on, is provided following the List of Figures at the beginning of the dissertation.

In Chapter V we define a theoretical diagnostic framework for systems that satisfy the properties set out in the problem statement. We describe the steps taken to construct a diagnostic model based on a logic program. We present a number of properties of the framework including the characterization of two types of component abnormality: incompleteness and incorrectness. We also show how the framework serves to unify consistency-based diagnosis and declarative error diagnosis, two heretofore distinct approaches to diagnosis. Finally, we compare the framework to IDA and the framework proposed by Console *et al.*

In Chapter VI we develop two similar procedures for computing diagnoses defined by our framework. Each procedure computes diagnoses for a particular class of assertions. Since they are generalizations of the procedure used by IDA, we present the IDA approach first. We then define the basis for our procedures, including their soundness, and their conditions for completeness and decidability. We present a generate-and-test implementation of the procedure and show how the generator algorithm from the IDA framework is interfaced to our own test procedures. We also discuss the efficiency and complexity of the procedures and compare them to procedures used by IDA and declarative error diagnosis.

In Chapter VII we develop another procedure for computing diagnoses defined by our framework. However, in this case the procedure can compute diagnoses for a more general class of assertions than the procedures developed in the previous chapter. We prove the procedure sound and discuss issues related to its completeness. We also describe how the procedure can be implemented and prove

the implementation correct. We compare the procedure to the only comparable procedure, that proposed by Console *et al.*, and show that it is superior from a number of perspectives.

In Chapter VIII we summarize the results of the dissertation and discuss future directions this research can take.

In the appendices we have included the source code to the Prolog implementation of the procedure developed in Chapter VII, and examples of using the procedure to compute diagnoses for the examples used in the dissertation. We also include an example of using the procedure to compute the diagnoses for a hierarchically structured, 4X4 bit sequential multiplier circuit.

CHAPTER II

Logic and Logic Programming

This chapter contains a very brief review of logic and logic programming theory primarily paraphrased from [3], a brief review in itself. The reader that is interested in more detail should refer to [57]. The reader familiar with either of these topics should feel free to skip the appropriate sections and refer to them as a reference when reading the remainder of the dissertation.

1. First-order Languages

Logic programs are sets of certain formulas of a first-order language. A *first-order language* consists of an alphabet and all formulas defined over it. An *alphabet* consists of the following classes of symbols¹:

- *variables* denoted by symbols beginning with uppercase letters *e.g.* $X, Y, Z, \dots$, or $'_'$ (underscore) to represent an anonymous variable,
- *constants* denoted by lowercase symbols *e.g.* $a, b, c, \dots$,
- *function symbols* denoted by lowercase symbols *e.g.* $f, g, h, \dots$,
- *predicate symbols* denoted by lowercase symbols *e.g.* $p, q, r, \dots$,
- *propositional constants*, which are denoted by **true** and **false**,
- *connectives*, which are $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), $\rightarrow$ (implication) and $\leftrightarrow$ (equivalence),
- *quantifiers*, which are $\exists$ (there exists) and $\forall$ (for all),
- *parentheses*, which are (and) and the *comma*, that is: $,$.

¹Standard Edinburgh Prolog notation. [13]

We assume the set of variables is infinite and fixed. Each function and predicate symbol has a fixed *arity*, that is, the number of arguments. We assume that function symbols have a positive arity—the role of 0-ary functions is played by the constants. In contrast, 0-ary predicates are admitted. Note that each alphabet is uniquely determined by its constants, functions and predicates.

We now define two classes of strings of symbols over a given alphabet. First we define the class of *terms* as follows:

- a variable is a term,
- a constant is a term,
- if f is an n -ary function and $t_1, \dots, t_n$ are terms then $f(t_1, \dots, t_n)$ is a term.

Terms are denoted by $s, t, \dots$

We define the class of *formulas* as follows:

- if p is an n -ary predicate and $t_1, \dots, t_n$ are terms then $p(t_1, \dots, t_n)$ is a formula (called an *atomic formula* or just an *atom*),
- **true** and **false** are formulas,
- if F and G are formulas then so are $\neg F$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$ and $(F \leftrightarrow G)$,
- if F is a formula and X is a variable then $\exists XF$ and $\forall XF$ are formulas.

Often we shall write $(G \leftarrow F)$ instead of $(F \rightarrow G)$. Well known binary functions (like $+$) or predicates (like $=$) are written in infix notation. Atomic formulas are denoted by A, B and formulas in general by F, G . We use the short forms $\exists X^n$ and $\forall X^n$ to denote $\exists X_1 \dots \exists X_n$ and $\forall X_1 \dots \forall X_n$, respectively. When variable names are not important we use the short forms $\exists F$ or $\forall F$ to denote that all variables in a formula F are existentially or universally quantified, respectively. We write $p(\mathbf{t}^n)$ as a short form for $p(t_1, \dots, t_n)$, or we write just $p(\mathbf{t})$ when the arity of the tuple is not important. A term or formula with no variables is called *ground*.

Given two strings of symbols e_1 and e_2 from the alphabet we write $e_1 \equiv e_2$ when e_1 and e_2 are identical.

2. Logic Programs

An atom or its negation is called a *literal*. A *positive literal* is just an atom while a *negative literal* is the negation of an atom. Note that **true** and **false** are

not atoms.

A formula of the form $\forall(L_1 \vee \dots \vee L_m)$ where $L_1, \dots, L_m$ are literals is called a *clause*. Throughout this dissertation clauses will be written in *clausal form*. The previous formula in clausal form is

$$A_1, \dots, A_k \leftarrow B_1, \dots, B_n$$

where $A_1, \dots, A_k$ are the positive literals among $L_1, \dots, L_m$ called *conclusions* and $B_1, \dots, B_n$ are the remaining negative literals stripped of the negation symbol called *premises*.

If a clause has only one conclusion ($k = 1$) then it is called a *program clause* or a *definite clause*. Its conclusion is then called a *head* and the list of its premises, a *body*. When the set of conclusions is empty ($k = 0$) then it is called a *goal* or a *negative clause*. They have the form $\leftarrow B_1, \dots, B_n$. When both the set of premises and conclusions is empty then we talk of the *empty clause* and denote it $\square$. It is interpreted as a contradiction. A *logic program* (or just a *program*) is a finite, nonempty set of program clauses. We will use the symbol Π to refer to a logic program. With each program Π we can uniquely associate a first-order language $L(\Pi)$ whose constants, functions and predicates are those occurring in Π .

Let $\mu(p)$ denote the number of clauses in a logic program Π with a p literal head. Formally,

$$\mu(p) = |\{p(t) \leftarrow F \in \Pi\}|.$$

Let $\text{PRED}(\Pi)$ denote the set of all predicate symbols in a logic program Π . Formally,

$$\text{PRED}(\Pi) = \{p \mid p(t) \leftarrow F \in \Pi\}.$$

There are two ways of interpreting a clause $A \leftarrow B_1, \dots, B_n$. One is: to solve A solve $B_1, \dots, B_n$. The other is: A is true if $B_1, \dots, B_n$ are true. The first interpretation is called the *procedural interpretation*. The second is called the *declarative interpretation*.

3. Semantics

We now define the semantics of logic programs which are a special case of the semantics of first order logic.

An *interpretation* I for a first order language L consists of

- a nonempty set D , called the *domain* of I ,
- an assignment for each constant c in L of an element c_I of D ,
- an assignment for each n -ary function f in L of a mapping f_I from D^n to D ,
- an assignment for each n -ary predicate r in L of an n -ary relation r_I on D , i.e. a subset of D^n .

A *state* (over I) is a function assigning to each variable an element from D . Given a state σ , we extend its domain to all terms. We assign to a term t an element $\sigma(t)$ from D proceeding by induction as follows:

- for a constant c we define $\sigma(c)$ as c_I ,
- if $f(t_1, \dots, t_n)$ is a term then we define $\sigma(f(t_1, \dots, t_n))$ as $f_I(\sigma(t_1), \dots, \sigma(t_n))$,

Observe that for a ground term t , $\sigma(t)$ has the same value for all σ .

We now define the semantics of a formula. Given a formula F we define inductively its truth in a state σ over I , written as $I \models_\sigma F$, as follows,

- if $p(t_1, \dots, t_n)$ is an atomic formula then

$$I \models_\sigma p(t_1, \dots, t_n) \text{ iff } (\sigma(t_1), \dots, \sigma(t_n)) \in p_I,$$

- $I \models_\sigma \text{true}$, not $I \models_\sigma \text{false}$,
- if F and G are formulas then

$$I \models_\sigma \neg F \text{ iff not } I \models_\sigma F$$

$$I \models_\sigma F \vee G \text{ iff } I \models_\sigma F \text{ or } I \models_\sigma G$$

$$I \models_\sigma \forall X F \text{ iff } I \models_{\sigma[X/d]} F \text{ for all } d \in D.$$

Here $\sigma[X/d]$, for a state σ , an element d of D and a variable X stands for the state which differs from σ only on the variable X to which it assigns the element d .

The truth of other formulas is defined by expressing the remaining connectives and the quantifier $\exists$ in terms of $\neg$, $\vee$, and $\forall$:

$$F \wedge G \text{ as } \neg(\neg F \vee \neg G),$$

$$F \rightarrow G \text{ as } \neg F \vee G,$$

$$F \leftrightarrow G \text{ as } (F \rightarrow G) \wedge (G \rightarrow F),$$

$$\exists X F \text{ as } \neg \forall X \neg F.$$

We say, a formula F is true in the interpretation I , and write $I \models F$, when for all states σ , $I \models_{\sigma} F$.

Let S be a set of formulas. We say I is a *model* for S if every formula from S is true in I . When S has a model, we say that it is *satisfiable* or *consistent*. Otherwise we say it is *unsatisfiable* or *inconsistent*. When every interpretation is a model for S we say that S is *valid*, which we denote $\models S$.

Given another set of sentences S' we say that S *semantically implies* S' or S' is a *semantic consequence* of S , if every model of S is also a model of S' . We write then $S \models S'$. S and S' are *semantically equivalent* if both $S \models S'$ and $S' \models S$ hold.

The $\models$ operator for the first-order predicate calculus (FOPC) is monotonic as stated by the following theorem:

THEOREM II.1 (MONOTONICITY OF FOPC). *Let F and G be FOPC formulas, and let Γ be a set of FOPC formulas. If $\Gamma \models G$ then $\Gamma \cup \{F\} \models G$.*

FOPC is also subject to the following compactness law.

THEOREM II.2 (COMPACTNESS OF FOPC). *A set of FOPC formulas Γ is satisfiable iff every finite subset of Γ is satisfiable.*

4. Herbrand Interpretations

We now introduce a special class of models of logic programs called Herbrand models. Let L be a first-order language whose set of constants is not empty. By the *Herbrand universe* H_L for L we mean the set of all ground terms of L . By the *Herbrand base* $\hat{H}_L$ we mean the set of all ground atoms of L . If L is the first-order language associated with a program Π (that is if L is $L(\Pi)$) then we denote H_L and $\hat{H}_L$ by $H(\Pi)$ and $\hat{H}(\Pi)$, respectively.

EXAMPLE II.3. Program II.4 is a logic program representing the specified behavior for two inverters in series (schematically represented in Figure II.3). The Herbrand universe for this program is $\{0, 1\}$. The Herbrand base for this program is

$$\left\{ \begin{array}{l} inv(1,1), inv(1,0), inv(0,1), inv(0,0), \\ i1(1,1), i1(1,0), i1(0,1), i1(0,0), \\ i2(1,1), i2(1,0), i2(0,1), i2(0,0) \end{array} \right\}.$$

FIGURE II.3. Schematic of two inverters in series.

PROGRAM II.4 (SPECIFICATION OF TWO INVERTERS IN SERIES).

$$\text{inv}(X, Z) \leftarrow i1(X, Y), i2(Y, Z).$$

$$i1(1, 0). \quad i1(0, 1).$$

$$i2(1, 0). \quad i2(0, 1).$$

Note that if a logic program contains any function symbols then both its Herbrand universe and its Herbrand base are necessarily infinite.

EXAMPLE II.5. Consider the logic program $\Pi = \{p(f(1))\}$. For this program,

$$H(\Pi) = \{1, f(1), f(f(1)), f(f(f(1))), \dots\},$$

$$\hat{H}(\Pi) = \{p(1), p(f(1)), p(f(f(1))), p(f(f(f(1))))\}, \dots\}$$

By an *Herbrand interpretation* for L we mean an interpretation for L such that

- (1) its domain is the Herbrand universe H_L ,
- (2) each constant in L is assigned to itself,
- (3) if f is an n -ary function then it is assigned to the mapping from $(H_L)^n$ to H_L defined by assigning the ground term $f(t_1, \dots, t_n)$ to the sequence $t_1, \dots, t_n$ of ground terms,
- (4) if r is an n -ary predicate in L then it is assigned to a set of n -tuples of ground terms.

As each Herbrand interpretation for L is uniquely determined by a subset I of the Herbrand base $\hat{H}_L$ we identify Herbrand interpretations for L with (possibly empty) subsets of the Herbrand base $\hat{H}_L$.

By a *Herbrand model* for a set S of sentences we mean a Herbrand interpretation which is a model for S . A Herbrand model of S is the *least model of S* if it is included in every other Herbrand model of S . If Π is a logic program then Π has

a least Herbrand model $M(\Pi)$. Informally, the least Herbrand model $M(\Pi)$ of a program Π is the set of all ground atomic consequences of Π .

EXAMPLE II.6. If we take Π to be Program II.4 then its least Herbrand model $M(\Pi)$ is

$$\{inv(1,1), inv(0,0), i1(1,0), i1(0,1), i2(1,0), i2(0,1)\}.$$

Note that this set can be viewed as the set of all behavior that can be exhibited by all components of the series inverter circuit.

5. Substitutions

A *substitution* is a finite mapping from variables to terms and is written as $\theta = \{X_1/t_1, \dots, X_n/t_n\}$. Informally, it is to be read: "the variables $X_1, \dots, X_n$ are *bound to* $t_1, \dots, t_n$, respectively." The notation implies that the variables are different. If all $t_1, \dots, t_n$ are ground then θ is called ground. Given a substitution θ denote its domain by $dom(\theta)$.

Substitutions operate on expressions. By an *expression* we mean a term, a sequence of literals or a clause and denote it by E . For an expression E and a substitution θ , $E\theta$ stands for the result of applying θ to E which is obtained by simultaneously replacing each occurrence in E of a variable from the domain of θ by the corresponding term. The resulting expression $E\theta$ is called an *instance* of E . An instance is called *ground* if it contains no variables. θ is a *renaming* if it is a permutation of the variables from its domain. If θ is a renaming then $E\theta$ is called a *variant* of E . Given an expression E denote by $vars(E)$ the set of variables which appear in it.

Substitutions can be composed. Given substitutions $\theta = \{X_1/t_1, \dots, X_n/t_n\}$ and $\eta = \{Y_1/s_1, \dots, Y_m/s_m\}$ their *composition* $\theta\eta$ is defined by removing from the set

$$\{X_1/t_1\eta, \dots, X_n/t_n\eta, Y_1/s_1, \dots, Y_m/s_m\},$$

those pairs $X_i/t_i\eta$ for which $X_i \equiv t_i\eta$, as well as those pairs Y_i/s_i for which $Y_i \in \{X_1, \dots, X_n\}$.

If for a substitution θ we have $A\theta \equiv B\theta$ then θ is called a *unifier* of A and B and we say that A and B are *unifiable*. A unifier θ of A and B is called a *most general unifier* (or *mg*u for short) if for any other unifier η of A and B , $A\eta$ is an instance of $A\theta$. If two atoms are unifiable then they have a most general unifier.

6. SLD-resolution

Logic programs compute through a combination of two mechanisms—replacement and unification. This form of computing is essentially a specific form of theorem proving called *SLD-resolution* (“Selection rule-driven Linear resolution for Definite clauses”).

Let Π be a logic program and $N = \leftarrow A_1, \dots, A_n$ be a negative clause. Suppose that $C = A \leftarrow B_1, \dots, B_k$ is a clause from Π . If for some i , $1 \leq i \leq n$, A_i and A unify with an mgu θ , then we call

$$N' = \leftarrow (A_1, \dots, A_{i-1}, B_1, \dots, B_k, A_{i+1}, \dots, A_n)\theta$$

a *resolvent* of N and C with mgu θ .

By an *SLD-derivation* of $\Pi \cup \{N\}$ we mean a maximal sequence $N_0, N_1, \dots$ of negative clauses where $N = N_0$, together with a sequence $C_0, C_1, \dots$ of variants of clauses from Π and a sequence $\theta_0, \theta_1, \dots$ of substitutions such that for all $i = 0, 1, \dots$

- (1) N_{i+1} is a resolvent of N_i and C_i with the mgu θ_i ,
- (2) C_i does not have a variable in common with $N_0, C_0, \dots, C_{i-1}$.

The clauses $C_0, C_1, \dots$ are called the *input clauses* of the derivation. When one of the resolvents N_i is empty then it is the last negative clause of a derivation. Such a derivation is then called an *SLD-refutation*. We say a goal N *succeeds* in Π if there exists an SLD-refutation of $\Pi \cup \{N\}$. Let $SS(\Pi)$ represent the *success set* of a program Π where

$$SS(\Pi) = \{A \mid \text{the ground goal } \leftarrow A \text{ succeeds in } \Pi\}.$$

An SLD-derivation is called *failed* if it is finite and it is not a refutation.

EXAMPLE II.7. Take Π to be the series inverters program (Program II.4). The success set $SS(\Pi)$ is

$$\{inv(1,1), inv(0,0), i1(1,0), i1(0,1), i2(1,0), i2(0,1)\}.$$

Note that this set is identical to $M(\Pi)$ in Example II.6. This is no coincidence as we later show SLD-resolution to be both sound and complete.

According to the definition of an SLD-derivation, the following two choices are made in each step of constructing a new resolvent:

- choice of the selected atom,
- choice of the input clause whose conclusion unifies with the selected atom.

The first choice is determined by a *selection rule*. The second choice is determined by a *search rule*.

We say a goal N *finitely fails* in Π if all SLD-derivations of $\Pi \cup \{N\}$ are failed for some selection rule.

EXAMPLE II.8. Take Π to be the series inverters program (Program II.4). The goal $\leftarrow inv(1, 0)$ finitely fails in Π . Note that this can be interpreted to mean that the behavior $X = 1, Z = 0$ cannot be exhibited by the series inverter circuit.

Note that SLD-resolution is both sound and complete in the following sense.

THEOREM II.9 (SOUNDNESS OF SLD-RESOLUTION [11, Clark]). *Let Π be a program and $N \leftarrow A_1, \dots, A_k$ a goal. Suppose there exists an SLD-refutation of $\Pi \cup \{N\}$ with the sequence of substitutions $\theta_0, \dots, \theta_n$. Then $(A_1 \wedge \dots \wedge A_k)\theta_0 \dots \theta_n$ is a semantic consequence of Π .*

THEOREM II.10 (COMPLETENESS OF SLD-RESOLUTION [45, Hill]). *Let Π be a program and N a goal. Suppose $\Pi \cup \{N\}$ is inconsistent. Then there exists an SLD-refutation of $\Pi \cup \{N\}$.*

6.1. Selection Rules. We now examine the notion of a selection rule for an SLD-derivation in more detail. The selection of an atom by a selection rule is, in general, dependent upon the whole "history" of the derivation up to the current resolvent. By a selection rule we now mean a function which when applied to such a history with a non-empty last goal $N_k \leftarrow A_1, \dots, A_l$, yields an atom A_j , $1 \leq j \leq l$. Given a selection rule R , we say that an SLD-derivation of $\Pi \cup \{N\}$ is *via R* if all choices of the selected atoms in the derivation are performed according to R . That is, for each non-empty goal M of this SLD-derivation with a history H , $R(H)$ is the selected atom of M .

6.2. SLD-trees. When searching for a refutation of a goal, SLD-derivations are constructed with the aim of generating the empty clause. The totality of these derivations form a *search space*.

Let Π be a program, N be a goal and R a selection rule. The SLD-tree for $\Pi \cup \{N\}$ via R groups all SLD-derivations of $\Pi \cup \{N\}$ via R . Formally the *SLD-tree for $\Pi \cup \{N\}$ via R* is a tree such that

- its branches are SLD-derivations of $\Pi \cup \{N\}$ via R ,
- every node N' has exactly one descendant for every clause C of Π such that the selected atom A of N' unifies with the head of a variant C' of C . This descendant is a resolvent of N' and C' with A being the selected atom of N' .

EXAMPLE II.11. ([2, Apt and Van Emden]) Let Π denote the Program II.12. Figure II.4 shows a possible SLD-tree for $\Pi \cup \{\leftarrow \text{path}(X, c)\}$. The selected atoms are put in bold, used clauses and performed substitutions are indicated. The input clauses at level i are obtained from the original clause by adding the subscript " i " to all variables which were used earlier in the derivation in order to obtain variants of the input clauses.

PROGRAM II.12.

- (1) $\text{path}(X, Z) \leftarrow \text{arc}(X, Y), \text{path}(Y, Z).$
- (2) $\text{path}(X, X).$
- (3) $\text{arc}(b, c).$

6.3. Existentially Quantified Goals. The notion of a goal used thus far has all of its variables universally quantified. If an SLD-refutation is found for such a goal in a logic program then one can conclude that the negation of the goal, an existentially quantified conjunction of atoms, is logically implied by the logic program.

EXAMPLE II.13. For example, consider the logic program $\Pi = \{p(a)\}$. Since the implicitly universally quantified goal $\leftarrow p(X)$ succeeds in Π then we can conclude that $\Pi \models \exists X p(X)$ by the soundness of SLD-resolution.

However, it is also possible to use SLD-resolution to determine if a universally quantified conjunction of atoms is implied by a logic program. In this case all variables of the corresponding goal must be existentially quantified. In order to eliminate the existential quantifiers in the goal a technique from the domain of

FIGURE II.4. Example of an SLD-tree

automatic theorem is used that replaces the variables in the goal with unique constants not belonging to the language of the logic program called Skolem constants. The resulting goal is said to be in *Skolem standard form* [8]. Thus, a goal in Skolem standard form is just a ground goal.

The reason for eliminating the existential quantifiers of a goal is to simplify the process of deciding the satisfiability of a set of sentences. In particular, the following theorem holds:

THEOREM II.14 ([8, p. 48]). *Let S be a set of clauses that represents a Skolem standard form of a formula F . Then F is inconsistent iff S is inconsistent.*

The following is a corollary of this theorem specific to logic programs.

COROLLARY II.15. *Let Π be a logic program and $\leftarrow G$ be a goal. $\Pi \models \forall \neg G$ iff $\leftarrow G'$ succeeds in Π where G' is the Skolem standard form of $\exists G$.*

PROOF.

$$\Pi \models \forall \neg G$$

iff $\Pi \cup \{\exists G\}$ is inconsistent

iff $\Pi \cup \{G'\}$ is inconsistent (Theorem II.14)

iff $\leftarrow G'$ succeeds in Π (soundness and completeness of SLD-resolution).

□

EXAMPLE II.16. Consider the logic program $\Pi = \{p(Y)\}$. To confirm that $\Pi \models \forall X p(X)$ one must confirm that $\Pi \cup \{\exists p(X)\}$ is inconsistent. To do this, one must confirm that the goal in Skolem standard form, $\leftarrow p(\sigma)$ where $\sigma \notin L(\Pi)$, succeeds in Π , which is indeed the case.

7. Prolog

The programming language Prolog is a logic programming language supplemented with a number of extra-logical extensions and built-in predicates for the sake of efficiency and ease of use. Prolog utilizes SLD-resolution as its primary computation mechanism where the selection rule always chooses the leftmost goal literal first and the search rule chooses input clauses in the text order they are listed. Prolog's *search strategy* combines this selection rule and search rule with a backtracking mechanism in order to search for refutations for a given goal. Under the procedural interpretation of logic programs the order of goal reductions in a Prolog computation is analogous to the procedure call ordering that would be followed in a traditional sequential procedure-oriented language.

Although memory efficient, Prolog's search strategy compromises the completeness of SLD-resolution. For example, both a refutation and an infinite derivation for a goal may exist but, due to the clause text order, Prolog's search may explore the infinite derivation first. Thus, a goal N might succeed in Π but the Prolog search for its refutation may not terminate. Similarly, due to the choice of Prolog's selection rule, infinite derivations may exist for a goal that finitely fails in Π . As such, Prolog's search may not terminate in this case either. When we say *a goal N terminates in Π* we mean that a search for a refutation terminates under Prolog's search strategy. Otherwise we say the goal does not terminate in Π . Also, we say

a logic program Π is *everywhere terminating* if for all goals N , $\Pi \cup \{N\}$ has no infinite derivations using Prolog's selection rule.

A *pure Prolog program* is a Prolog program that does not incorporate any of the extra-logical predicates of Prolog such as the *cut* “!”, *assert*, *retract*. However, we permit a pure Prolog program to incorporate first-order built-in predicates such as $\leq$ when they are treated as logical first-order predicates by the underlying computational mechanism. Although this is generally not the case for implementations of the Prolog language, it is the case for implementations of *constraint logic programming* languages [48].

An example of a pure Prolog program for insertion sort is shown below.

PROGRAM II.17 (FAULT-FREE ISORT PROGRAM [85]).

- (1) $isort([X_h|X_t], Y) \leftarrow isort(X_t, Z), insert(X_h, Z, Y).$
- (2) $isort([], []).$
- (3) $insert(X, [Y_h|Y_t], [Y_h|Z]) \leftarrow X > Y_h, insert(X, Y_t, Z).$
- (4) $insert(X, [Y_h|Y_t], [X, Y_h|Y_t]) \leftarrow X \leq Y_h.$
- (5) $insert(X, [], [X]).$

Here the constant $[]$ represents an empty list and $[X_h|X_t]$ represents a list with head X_h and tail X_t . Similarly $[X_1, X_2|X_t]$ represents a list with X_1 and X_2 as the first two consecutive elements and tail X_t .

EXAMPLE II.18. The following represents the only SLD-refutation of the goal

$\leftarrow \text{isort}([2, 1], l)$ from Program II.17 using Prolog's search strategy.

$\leftarrow \text{isort}([2, 1], L)$	(1)	$\{X_h/2, X_t/[1], Y/L\}$
$\leftarrow \text{isort}([1], Z), \text{insert}(2, Z, L)$	(1)	$\{X_h/X_h^1, X_t/X_t^1, Z/Z^1\},$ $\{X_h^1/1, X_t^1/[], Y/Z\}$
$\leftarrow \text{isort}([], Z^1), \text{insert}(1, Z^1, Z),$ $\text{insert}(2, Z, L)$	(2)	$\{Z^1/[]\}$
$\leftarrow \text{insert}(1, [], Z), \text{insert}(2, Z, L)$	(5)	$\{X/1, Z/[X]\}$
$\leftarrow \text{insert}(2, [1], L)$	(3)	$\{X/X^1, Z/Z^2\},$ $\{X^1/2, Y_h/1, Y_t/[], L/[1 Z^2]\}$
$\leftarrow 2 > 1, \text{insert}(2, [], Z^2)$	(BIP)	$\{\}$
$\leftarrow \text{insert}(2, [], Z^2)$	(5)	$\{X/X^2\}, \{X^2/2, Z^2/[2]\}$
□		

The middle column indicates the input clauses from Program II.17. The right hand column indicates the mgu's used. Note that on several rows two substitutions are given. In these cases the first substitution creates a variant of the program clause. The second substitution is the mgu. (BIP) denotes the fact that $>$ is a built-in predicate. The sequence of mgu's performed binds the variable L to $[1, 2]$ through the consecutive substitutions $\{L/[1|Z^2]\}, \{Z^2/[2]\}$.

8. Negative Information

SLD-resolution is a very restricted form of reasoning because only positive facts can be deduced using it. However, we would sometimes like to invoke the *closed world assumption* [80] which is stated by the following meta-inference rule:

$$\frac{A \text{ cannot be proved from } \Pi}{\neg A}$$

where A is a ground atom. Given a program Π , consider the set

$$\text{CWA}(\Pi) = \{\neg A \mid A \text{ is a ground atom for which} \\ \text{there does not exist an SLD-refutation of } \Pi \cup \{\leftarrow A\}\}.$$

Note that $\text{CWA}(\Pi) \cup \text{M}(\Pi)$ is consistent for all Π . Unfortunately, $\text{CWA}(\Pi)$ is not computable for all programs Π . A way out of this dilemma is to adopt a more restrictive form of provability which considers $\neg A$ proved when an attempt

to prove A using SLD-resolution finitely fails. We now replace CWA with the following meta-inference rule:

$$\frac{A \text{ finitely fails in } \Pi}{\neg A}$$

called the *negation as failure* rule or NAF rule for short. The corresponding set

$$\text{NAF}(\Pi) = \{\neg A \mid A \text{ is a ground atom for which } A \text{ finitely fails in } \Pi\},$$

is computable. We let $\text{FF}(\Pi)$ denote the *finite failure set* of a program Π where $\text{FF}(\Pi) = \{A \mid \neg A \in \text{NAF}(\Pi)\}$. Note that $\text{NAF}(\Pi) \subseteq \text{CWA}(\Pi)$. Furthermore, $\text{NAF}(\Pi) = \text{CWA}(\Pi)$ if $\text{SS}(\Pi) \cup \text{FF}(\Pi) = \hat{H}(\Pi)$.

EXAMPLE II.19. Take Π to be the series inverters program (Program II.4). The finite failure set $\text{FF}(\Pi)$ is

$$\{inv(1,0), inv(0,1), i1(0,0), i1(1,1), i2(0,0), i2(1,1)\}.$$

Note that this set can be viewed as the set of all behavior that cannot be exhibited by the components of the series inverter circuit. From Examples II.7 and II.5 we also know that $\text{SS}(\Pi) \cup \text{FF}(\Pi) = \hat{H}(\Pi)$. Therefore $\text{NAF}(\Pi) = \text{CWA}(\Pi)$.

EXAMPLE II.20. Consider the logic program

$$\Pi = \left\{ \begin{array}{l} a(X) \leftarrow b(X), a(X). \\ b(1). \end{array} \right\}.$$

Here, $H(\Pi) = \{1\}$, $\hat{H}(\Pi) = \{a(1), b(1)\}$, and $\text{SS}(\Pi) = \{b(1)\}$. Note that since $a(1)$ does not finitely fail in Π then $\text{NAF}(\Pi) = \emptyset$, and $\text{FF}(\Pi) = \emptyset$. However, since $a(1)$ does not succeed in Π either then $\text{CWA}(\Pi) = \{\neg a(1)\}$. So in this case $\text{NAF}(\Pi) \neq \text{CWA}(\Pi)$.

Another way of defining the information deducible using the finite failure rule is by using the concept of the predicate completion of a program due to Clark [10]. A program can be seen as a collection of statements of the form "if ... then ...". This does not allow us to conclude negative facts because only positive conclusions are admitted. But treating the clauses as statements of the form "... iff ..." we obtain a stronger interpretation which allows us to draw negative conclusions. Thus, a logic program Π can be understood as a set of formulas obtained from a special transformation of the clauses of Π . The resulting set of formulas, denoted

$comp(\Pi)$ is called the *predicate completion* of Π , or the *completion* of Π for short. The significance of the completion of a program is that conclusions drawn by the negation as failure rule are logical consequences of the program's completion.

First assume that "=" is a unique binary predicate symbol not appearing in Π . We write $s \neq t$ as an abbreviation for $\neg(s = t)$. We successively perform the following steps, where $X_1, \dots, X_n, \dots$ are new variables.

Step 1: Remove terms: Transform each clause

$$p(t_1, \dots, t_n) \leftarrow B_1, \dots, B_m$$

of Π into

$$p(X_1, \dots, X_n) \leftarrow (X_1 = t_1) \wedge \dots \wedge (X_n = t_n) \wedge B_1 \wedge \dots \wedge B_m.$$

Step 2: Introduce existential quantifiers: Transform each formula

$$p(X_1, \dots, X_n) \leftarrow F$$

obtained in the previous step into

$$p(X_1, \dots, X_n) \leftarrow \exists Y_1 \dots \exists Y_d F,$$

where $Y_1, \dots, Y_d$ are the variables of the original clause.

Step 3: Group similar formulas: Let

$$p(X_1, \dots, X_n) \leftarrow F_1$$

...

$$p(X_1, \dots, X_n) \leftarrow F_{\mu(p)}$$

be all formulas obtained in the previous step with head predicate symbol p . Replace them by one formula

$$p(X_1, \dots, X_n) \leftarrow F_1 \vee \dots \vee F_{\mu(p)}.$$

If $F_1 \vee \dots \vee F_{\mu(p)}$ is empty replace it by **true**.

Step 4: Handle "undefined" predicate symbols: For each n -ary predicate symbol q not appearing in the head of a clause in p add a formula

$$q(X_1, \dots, X_n) \leftarrow \text{false}.$$

Step 5: Introduce universal quantifiers: Replace each formula

$$p(X_1, \dots, X_n) \leftarrow F$$

by

$$\forall X_1 \dots \forall X_n (p(X_1, \dots, X_n) \leftarrow F).$$

Step 6: Introduce equivalence: In each formula replace " $\leftarrow$ " with " $\leftrightarrow$ ". We call the intermediate form of Π obtained after Step 5 the *IF-definition associated with Π* and denote it by $\text{IF}(\Pi)$. We call the final form the *IFF-definition associated with Π* and denote it by $\text{IFF}(\Pi)$. By $\text{OIF}(\Pi)$ we denote the set of formulas obtained from $\text{IF}(\Pi)$ by replacing everywhere " $\leftarrow$ " by " $\rightarrow$ ".

EXAMPLE II.21. $\text{IFF}(\Pi)$ for the ISORT program Π (Program II.17) is the following.

$$\begin{aligned} \forall X \forall Y (\text{isort}(X, Y) \leftrightarrow & (\exists X_h \exists X_t \exists Z (X = [X_h | X_t] \wedge \text{isort}(X_t, Z) \wedge \\ & \text{insert}(X_h, Z, Y))) \vee (X = [] \wedge Y = [])). \\ \forall X \forall Y \forall Z (\text{insert}(X, Y, Z) \leftrightarrow & (\exists Y_h \exists Y_t \exists Z_t (Y = [Y_h | Y_t] \wedge Z = [Y_h | Z_t] \wedge X > Y_h \wedge \\ & \text{insert}(X, Y_t, Z_t))) \vee \\ & (\exists Y_h \exists Y_t (Y = [Y_h | Y_t] \wedge Z = [X, Y_h | Y_t] \wedge X \leq Y_h)) \vee \\ & (Y = [] \wedge Z = [X])). \end{aligned}$$

It is important that the predicate symbol " $=$ " is interpreted as identity. We first extend the interpretation of a first-order language. Let I be an interpretation of the first order language $L(\Pi)$ associated with Π . We put for any two terms t_1 and t_2 and a state σ over I

$$I \models_{\sigma} t_1 = t_2 \text{ iff } \sigma(t_1) \text{ and } \sigma(t_2) \text{ are the same elements of the domain of } I.$$

In addition we must add the following *free equality axioms* for Π denoted $\text{EQ}(\Pi)$ which enforce the proper interpretation of " $=$ ".

- (1) $f(X_1, \dots, X_n) = f(Y_1, \dots, Y_n) \rightarrow X_1 = Y_1 \wedge \dots \wedge X_n = Y_n$ for each n -ary function f .
- (2) $f(X_1, \dots, X_n) \neq h(Y_1, \dots, Y_m)$ for each n -ary function f and m -ary function h such that $f \neq h$.

- (3) $X \neq t$ for each variable X and each term t such that $X \neq t$ and X occurs in t .

Here constants are identified with 0-ary functions so (1) includes $c = c$ for every constant c as a special case, and (2) includes $c \neq d$ for all pairs of distinct constants as a special case.

Given now a program Π we define its completion $comp(\Pi)$ as the set of formulas $IFF(\Pi)$ augmented by the free equality axioms $EQ(\Pi)$.

EXAMPLE II.22. Augmenting $IFF(\Pi)$ from Example II.21 with $EQ(\Pi)$ we can prove, for example, that $comp(\Pi) \models \neg isort([2, 1, 3], [2, 3, 1])$.

The following lemmas are consequences of the previous definitions.

LEMMA II.23. *Let Π be a logic program and let $\exists p(t) \in L(\Pi)$.*

$$comp(\Pi) \models \exists p(t) \text{ iff } IFF(\Pi) \cup EQ(\Pi) \models \exists p(t).$$

LEMMA II.24. *Let Π be a logic program and let $\forall \neg p(t) \in L(\Pi)$.*

$$comp(\Pi) \models \forall \neg p(t) \text{ iff } OIF(\Pi) \cup EQ(\Pi) \models \forall \neg p(t).$$

We also have the following soundness and completeness results related to the completion of all programs Π .

THEOREM II.25 (SOUNDNESS OF NAF RULE [10, Clark]). *Let Π be a logic program. If, for a goal N , $\Pi \cup \{N\}$ has a finitely failed SLD-tree, then $comp(\Pi) \models N$.*

THEOREM II.26 (COMPLETENESS OF NAF RULE [57, Jaffar, Lassez, Lloyd]). *Let Π be a logic program. If, for a goal N , $comp(\Pi) \models N$, then $\Pi \cup \{N\}$ has a finitely failed SLD-tree.*

9. Equality Theories

We now present some results pertaining to equality theories used to construct the completion of a logic program. The following lemma due to Clark [10] clarifies the relationship between unification and the free equality axioms.

LEMMA II.27 (CLARK [10]). *Let Π be a logic program, and let $s_1, \dots, s_n$ and $t_1, \dots, t_n$ belong to $L(\Pi)$.*

(a) *If the set $\{s_1 = t_1, \dots, s_n = t_n\}$ has a unifier then for some of its mgu $\{X_1/u_1, \dots, X_k/u_k\}$*

$$EQ(\Pi) \models (s_1 = t_1 \wedge \dots \wedge s_n = t_n) \rightarrow (X_1 = u_1 \wedge \dots \wedge X_k = u_k).$$

(b) *If the set $\{s_1 = t_1, \dots, s_n = t_n\}$ has no unifier then*

$$EQ(\Pi) \models (s_1 = t_1 \wedge \dots \wedge s_n = t_n) \rightarrow \text{false}.$$

The next theorem relates the notions of equality as identity and the more usual notion of equality. The usual notion of equality is stated by the following axioms:

- (1) $X = X$,
- (2) $(X_1 = Y_1 \wedge \dots \wedge X_n = Y_n) \rightarrow f(X_1, \dots, X_n) = f(Y_1, \dots, Y_n)$, for each function symbol f , and
- (3) $(X_1 = Y_1 \wedge \dots \wedge X_n = Y_n) \rightarrow (p(X_1, \dots, X_n) \rightarrow p(Y_1, \dots, Y_n))$ for each predicate symbol p .

If we denote these axioms by Eq , then we see by the following theorem that interpreting equality as identity is the same as interpreting equality according to the axioms Eq .

THEOREM II.28 ([3]). *Let S be a set of formulas in a first-order language L including $=$. Then for every formula φ*

$$S \models \varphi \text{ iff } S \cup Eq \models_+ \varphi,$$

where $\models_+$ stands for validity with respect to interpretations of L which interpret $=$ in an arbitrary fashion.

CHAPTER III

Declarative Error Diagnosis

The field of declarative error diagnosis was initially defined by Shapiro [85] to be applicable to any pure (no side effects, no global data) sequential, relational or functional programming language. However, Shapiro also presented a specialized instance of the approach for the diagnosis of pure logic programs. It is this latter approach for which the bulk of the theoretical results obtained in the field of declarative error diagnosis apply (e.g. [57, 56, 33, 92, 34]).

Since this dissertation uses results obtained for both the general approach to declarative program diagnosis and its specialization for logic programs, this chapter presents brief overviews of both.

1. Procedural Ontology

As originally defined by Shapiro, declarative error diagnosis is defined in terms of an abstract computational model called *top level traces* in order to accommodate a large class of programs. Since the notion of a top level trace is procedural we therefore must view a logic program in terms of its procedural interpretation in order to use declarative error diagnosis. Recall from the previous chapter that for a clause $A \leftarrow B_1, \dots, B_n$ the procedural interpretation is: to solve A solve $B_1, \dots, B_n$. Let I be an interpretation for a logic program Π . An *ontology* for an interpretation I is an informal association between the elements of I and 'real-world' objects. Given a logic program Π the *procedural ontology* that underlies the procedural interpretation takes an individual literal $p(t_1, \dots, t_n)$ of $L(\Pi)$ to represent a procedure p with arguments $t_1, \dots, t_n$ where each ground t_i represents a possible argument value. As such, the Herbrand universe $H(\Pi)$ represents the

FIGURE III.5. Universe of behavior for a faulty logic program Π with respect to an intended interpretation M

set of all possible ground values arguments may take in a program Π .

2. Errors

In light of this procedural ontology for logic programs, the least Herbrand model $M(\Pi)$ of a logic program Π can be viewed as the set of all argument values computable by all procedures of Π . The observation of erroneous behavior in a logic program implies the existence of an Herbrand interpretation M such that $M \neq M(\Pi)$, where M represents the intended behavior of the procedures of Π . We call such an interpretation the *intended interpretation* of a program. Note that the intended interpretation M represents an Herbrand model of the ideal, fault-free version of the program Π , it is not necessarily a model for Π itself.

In general, if M is an intended interpretation for a logic program Π then we can identify three categories of Π 's behavior with respect to M :¹ *missing behavior*, *spurious behavior*, and *correct behavior*, as shown in Figure III.5. Obviously missing behavior and spurious behavior are considered erroneous behavior. Note that if a ground atom $p(g)$ is spurious in a program Π then the goal $\leftarrow p(g)$ succeeds in Π . Similarly, if $p(g)$ is missing in Π then the goal $\leftarrow p(g)$ does not succeed in Π .

¹At this point it should be mentioned that we deviate from the terminology normally used in the field of declarative error diagnosis. In declarative error diagnosis, an *error* is what causes a program to behave in an unexpected fashion. In this dissertation we refer to the cause of misbehavior as a *fault*. The misbehavior itself has no agreed upon name in declarative error diagnosis. In this dissertation, any such misbehavior is referred to as an *error*.

Shapiro [85] also defines three properties of programs which reflect their behavior with respect to an intended interpretation. Let Π be a program and M an intended interpretation. Π is *partially correct* in M if $M(\Pi) \subseteq M$. Therefore if Π is partially correct in M then Π can still be missing behavior. Π is *complete* in M if $M \subseteq M(\Pi)$. In this case it is still possible for Π to exhibit spurious behavior. Finally, Π is *totally correct* in M if $M = M(\Pi)$ and Π is everywhere terminating. In this case, Π is neither missing behavior nor is it capable of exhibiting spurious behavior with respect to M .

EXAMPLE III.1. Program III.2 below is a buggy insertion sort program which, among other errors, exhibits the spurious behavior $isort([2, 1, 3], [2, 3, 1])$ and is missing the intended behavior $isort([2, 1, 3], [1, 2, 3])$. Thus, an intended interpretation of this program would include the element $isort([2, 1, 3], [1, 2, 3])$ but exclude $isort([2, 1, 3], [2, 3, 1])$.

PROGRAM III.2 (BUGGY ISORT PROGRAM[85]).

$$isort([X_h|X_t], Y) \leftarrow isort(X_t, Z), insert(X_h, Z, Y).$$

$$isort([], []).$$

$$insert(X, [Y_h|Y_t], [Y_h|Z]) \leftarrow Y_h > X, insert(X, Y_t, Z).$$

$$insert(X, [Y_h|Y_t], [X, Y_h|Y_t]) \leftarrow X \leq Y_h.$$

$$insert(X, [], [X]).$$

3. Faults

In this section we examine faults both as they are defined in terms of the procedural ontology *i.e.* notions of faulty procedures, and the more specialized notions of faults defined in terms of logic programs in particular *i.e.* notions of faulty clauses and predicates. We also show that the latter notions of faults imply the existence of procedural faults. While procedural faults are applicable to program diagnosis in general, the latter apply only to logic programs.

3.1. Procedural Faults. In addition to the two types of errors Shapiro defines two categories of faulty procedures. Just as errors are defined with respect to an

intended interpretation M so are faults. The following fault definitions are slight generalizations of those originally presented in [85].

DEFINITION III.3 (INCORRECT PROCEDURE). A procedure $p \in \text{PRED}(\Pi)$ is an *incorrect procedure* in Π with respect to an Herbrand interpretation M iff there exists a clause $p(t) \leftarrow F \in \Pi$, and a substitution θ such that

$$M \models \forall \neg p(t)\theta \text{ and } M \models \forall F\theta.$$

DEFINITION III.4 (INCOMPLETE PROCEDURE). A procedure $p \in \text{PRED}(\Pi)$ is an *incomplete procedure* in Π with respect to an Herbrand interpretation M iff there exists an atom $p(s) \in L(\Pi)$ for all clauses $p(t) \leftarrow F \in \Pi$, and for all substitutions θ such that

$$M \models \forall p(s) \text{ and } (M \models \forall \neg F\theta \text{ or } p(t)\theta \not\models p(s)\theta).$$

In the context of logic programs, these notions of faults identify a faulty predicate. We now discuss the intuition behind these definitions. In general, clauses of the form $p(t) \leftarrow F$ define the behavior computable by a procedure p in terms of the behavior computable by the subprocedures called in F . Let $p(g) \leftarrow q_1(g_1), \dots, q_m(g_m)$ be a ground instance of a clause in a program Π . By the procedural interpretation of logic programs, one can say that if $q_1(g_1), \dots, q_m(g_m)$ are computable in Π then $p(g)$ is also computable in Π .

Now, if $p(g)$ is not intended behavior, i.e. $p(g)$ does not belong to some intended interpretation M , and if the behaviors $q_1(g_1), \dots, q_m(g_m)$ are intended behavior, i.e. they belong to M , then p is an incorrect procedure in M . On the other hand, if $p(g)$ is intended behavior and if, for all premises $q_1(g_1), \dots, q_m(g_m)$ of all ground instances of clauses with conclusion $p(g)$, at least one behavior $q_i(g_i)$ is not intended behavior, then p is an incomplete procedure in M .

EXAMPLE III.5. Consider the following ground instance of the first clause in the buggy ISORT program (Program III.2):

$$\text{isort}([2, 1, 3], [2, 3, 1]) \leftarrow \text{isort}([1, 3], [3, 1]), \text{insert}(2, [3, 1], [2, 3, 1]).$$

The intended interpretation for the program does not include the conclusion or either of the premises of this clause instance. Since all atoms in this clause are computable in the buggy ISORT program then all three atoms are spurious. Nevertheless, we do not conclude from this that the *isort* procedure is incorrect because

the premise $isort([1, 3], [3, 1])$ does not belong to the intended interpretation. In this example, the *isort* procedure is merely propagating erroneous behavior.

EXAMPLE III.6. Consider another ground instance of a clause in the buggy ISORT program:

$$isort(1, [3], [3, 1]) \leftarrow 3 > 1, insert(1, [], [1]).$$

Since the intended interpretation for this program includes $3 > 1$ and $insert(1, [], [1])$ but excludes $isort(1, [3], [3, 1])$ we can conclude that the *insert* procedure is incorrect. Note that since all three atoms in the clause are computable in the buggy ISORT program, the atoms $3 > 1$ and $insert(1, [], [1])$ are correct behavior and $isort(1, [3], [3, 1])$ is spurious behavior. In this example then, the erroneous behavior originates with the *insert* procedure rather than being propagated from its subprocedures.

EXAMPLE III.7. Consider the following ground instance of the first clause in the buggy ISORT program:

$$isort([2, 1, 3], [1, 2, 3]) \leftarrow isort([1, 3], [1, 3]), insert(2, [1, 3], [1, 2, 3]).$$

In this case both premises and the conclusion are members of the intended interpretation. Although $insert(2, [1, 3], [1, 2, 3])$ is not computable in the buggy ISORT program and hence is missing behavior, we cannot conclude that *isort* is incomplete because both premises belong to the intended interpretation. Thus, the existing structure of the *isort* procedure would permit the computation of $isort(2, [1, 3], [1, 2, 3])$ if $isort([1, 3], [1, 3])$ and $insert(2, [1, 3], [1, 2, 3])$ were computable. For the buggy ISORT program $isort([1, 3], [1, 3])$ is indeed computable and is therefore correct. The behavior $insert(2, [1, 3], [1, 2, 3])$ however, is not computable and is therefore missing.

EXAMPLE III.8. Consider the following ground instances of clauses of the buggy ISORT program:

$$insert(2, [1, 3], [1, 2, 3]) \leftarrow 1 > 2, insert(2, [3], [2, 3]).$$

$$insert(2, [1, 3], [1, 2, 3]) \leftarrow 2 \leq 1.$$

Note that these two clause instances are the only instances that exist with conclusion $insert(2, [1, 3], [1, 2, 3])$. The intended interpretation for the program includes $insert(2, [1, 3], [1, 2, 3])$ and $insert(2, [3], [2, 3])$ but excludes $1 > 2$ and $2 \leq 1$. Therefore the *insert* procedure is incomplete. In this case the existing structure of the *insert* procedure prohibits the computation of $insert(2, [1, 3], [1, 2, 3])$ by requiring the computation of the incorrect behavior $1 > 2$ or $2 \leq 1$.

3.2. Faults in Logic Programs. We now examine the two categories of faults defined in terms of logic programs rather than in terms of the more general procedural ontology.

DEFINITION III.9 (INCORRECT CLAUSE INSTANCE [56, Lloyd]). Let Π be a logic program and M an intended interpretation for Π . An instance $A \leftarrow F$ of a clause in Π is an *incorrect clause instance* for Π with respect to M if $M \models \neg \exists A$ and $M \models \forall F$.

DEFINITION III.10 (UNCOVERED ATOM [56, Lloyd]). Let Π be a logic program and M an intended interpretation for Π . Let A be an atom with predicate p . A is an *uncovered atom* for Π with respect to M if $M \models \forall A$ and, for every clause $A' \leftarrow F$ in the definition of p such that A and A' unify with mgu θ , say, we have that $M \models \neg \exists F\theta$.

Note that every instance of an incorrect clause instance is incorrect and every instance of an uncovered atom is uncovered. In contrast to the procedural notions of faults, these notions provide more information regarding the cause of errors occurring in a logic program *i.e.* an individual clause or atom is identified. In contrast, the procedural notions of faults only identify a faulty predicate. However, it is easy to see that the following relationships hold between the different notions of faults:

THEOREM III.11. *Let Π be a logic program and M an intended interpretation. If $p(t) \leftarrow F$ is an incorrect clause instance for Π with respect to M then p is an incorrect procedure in Π with respect to M .*

THEOREM III.12. *Let Π be a logic program and M an intended interpretation. If $p(t)$ is an uncovered atom for Π with respect to M then p is an incomplete procedure in Π with respect to M .*

EXAMPLE III.13. Referring to Example III.6 we find that the clause instance

$$\text{insert}(1, [3], [3, 1]) \leftarrow 3 > 1, \text{insert}(1, [], [1]),$$

is incorrect with respect to the intended interpretation, thereby implying that *insert* is incorrect with respect to the intended interpretation.

EXAMPLE III.14. Referring to Example III.8 we find that the atom *insert*(2, [1, 3], [1, 2, 3]) is uncovered with respect to the intended interpretation, thereby implying that *insert* is incomplete with respect to the intended interpretation.

4. Diagnosis

The following two theorems relate the existence of errors to the presence of (procedural) faults in a program Π .

THEOREM III.15 ([85, Shapiro]). *Let Π be a program and M an Herbrand interpretation. If Π is not partially correct with respect to M then Π contains an incorrect procedure with respect to M .*

THEOREM III.16 ([85, Shapiro]). *Let Π be a program and M an Herbrand interpretation. If a ground atom $p(\mathbf{g})$ belongs to M and $\leftarrow p(\mathbf{g})$ finitely fails in Π then Π contains an incomplete procedure with respect to M .*

In the first theorem, the fact that Π is not partially correct implies that Π can produce spurious behavior, and therefore contains an incorrect procedure. In the second theorem, the fact that an atom belonging to an intended interpretation M fails in Π means that Π is missing behavior, and therefore contains an incomplete procedure.

Based on the proofs of previous two theorems Shapiro defines two interactive diagnostic algorithms: FP, an algorithm for locating incorrect procedures² contributing to spurious behavior that terminates in a program Π , and IP, an algorithm for locating incomplete procedures contributing to missing behavior that terminates in Π .

Numerous alternative algorithms have been proposed in the literature since Shapiro defined the FP and IP algorithms. Some have been proposed for general

²"FP" stands for "false procedure," a synonym for incorrect procedure used in [85].

imperative program diagnosis (e.g. [84, 39]), and some for hardware diagnosis (e.g. [43, 53]), however most have been oriented towards logic program diagnosis in particular. The algorithms for logic program diagnosis locate uncovered atoms and incorrect clause instances rather than incorrect or incomplete procedures since the former provide more information regarding the nature of faults. In addition, most of the algorithms endeavor to minimize or simplify interaction with the user (e.g. [74, 27, 33, 61, 54, 30, 29, 92, 60]), or accept a more general class of logic programs such as general logic programs (e.g. [56, 70, 60]), parallel logic programs (e.g. [55, 89]), logic programs that support coroutining (e.g. [69]), or "impure" Prolog programs that include extra-logical predicates (e.g. [75]). However, all of these algorithms, like FP and IP, query the user regarding the truth of atoms in an intended interpretation in order to locate faults. Therefore, in order to provide some insight into the bases for these algorithms we now briefly examine the simplest of them: FP and IP. Note that in the following description both algorithms treat built-in predicates as fault-free.

4.1. FP Algorithm. The FP algorithm guides the search for an incorrect procedure by utilizing oracle responses to distinguish between procedures that are simply propagating spurious behavior and procedures that are the initial source of spurious behavior. The search strategy is akin to that of a standard tracing technique that single steps through the procedure calls of a computation.

The FP algorithm uses a *ground oracle* for an intended interpretation M which responds to questions of the form: " $M \models p(g)$?", where $p(g)$ is a ground atom. In english: "Is $p(g)$ intended behavior?" The ground oracle may respond with either *yes* or *no*. Since built-in predicates are treated as fault-free procedures, ground oracle queries are not made concerning their behavior. A faulty program must be executed by FP, therefore when diagnosing pure Prolog programs the Prolog computation mechanism is utilized.

ALGORITHM III.17 (FP).

Input: A program Π and a ground instance of spurious behavior $p(g)$ that terminates in Π .

Output: An ground instance of spurious behavior $q(g) \in \hat{H}(\Pi)$ produced by q , a procedure incorrect in M .

Algorithm: Execute Π such that $p(\mathbf{g})$ succeeds in Π . During the execution, whenever an atom $q(\mathbf{g})$ succeeds in Π check, using a ground oracle, whether $M \models q(\mathbf{g})$. If not, then return $q(\mathbf{g})$ and terminate.

EXAMPLE III.18. We now examine the behavior of *fp*, the Prolog implementation of the FP algorithm for the diagnosis of pure Prolog programs [85]. If we apply *fp* to Program III.2, the buggy ISORT program with the spurious behavior *isort*([2, 1, 3], [2, 3, 1]). *fp* will make a number of oracle queries (answered here by the user) and then return both an instance of spurious behavior and an instance of the clause that produced it. The oracle queries and their responses follow.

query: $M \models \text{isort}([], [])? \text{ y.}$
 query: $M \models \text{insert}(3, [], [3])? \text{ y.}$
 query: $M \models \text{isort}([3], [3])? \text{ y.}$
 query: $M \models \text{insert}(1, [], [1])? \text{ y.}$
 query: $M \models \text{insert}(1, [3], [3, 1])? \text{ n.}$

From these oracle responses the algorithm concludes that the *insert* procedure must be incorrect with respect to M . This is in agreement with Example III.6. *fp* also returns the incorrect clause instance that produced this spurious behavior, *insert*(1, [3], [3, 1]) $\leftarrow 3 > 1, \text{insert}(1, [], [1])$, which agrees with Example III.13. Examining this we see that the arguments for the $>$ are exchanged.

4.2. IP Algorithm. The IP algorithm guides the search for an incomplete procedure by utilizing oracle responses to distinguish between procedures whose structure would permit correct behavior and those whose structure prohibits correct behavior *i.e.* those procedures that are the source of missing behavior.

The IP algorithm locates incomplete procedures using an existential oracle and oracle computations. An *existential oracle* for an intended interpretation M answers questions of the form: " $M \models \exists \mathbf{X}^m(p(\mathbf{t}))?$ " where $\text{vars}(\mathbf{t}) = \{X_1, \dots, X_m\}$. The oracle response is a set of ground atoms

$$\{p(\mathbf{g}) \mid M \models p(\mathbf{g}) \text{ and } p(\mathbf{g}) \equiv p(\mathbf{t})\theta \text{ for some substitution } \theta\}.$$

An empty set implies a negative response to the oracle query. Roughly translated, the existential oracle query above is "What is the set of ground instances of $p(\mathbf{t})$ that represent intended behavior?"

We say an intended interpretation M is *I/O bounded* if, for all existential oracle queries " $M \models \exists X^m(p(t))$?" all oracle responses are finite sets. Since built-in predicates are treated as fault-free procedures, existential oracle queries are not made concerning their behavior.

An *oracle computation* for a goal A in an intended interpretation M is a set of ground premises $B_1, \dots, B_n$ of a ground conclusion A such that for all B_i , $M \models B_i$ and $A \leftarrow B_1, \dots, B_n$ is a ground instance of a clause in Π . Its construction is accomplished by a series of existential oracle queries.

ALGORITHM III.19 (IP).

Input: A program Π and a ground instance of missing behavior $p(g)$ that terminates in Π (i.e. $p(g)$ finitely fails in Π).

Output: A ground instance of missing behavior $q(g) \in \hat{H}(\Pi)$ such that q is incorrect in M .

Algorithm: The algorithm calls a recursive procedure IP with input $p(g)$. IP operates as follows. On input $p(g)$ it tries to construct an oracle computation of the goal $\leftarrow p(g)$ using existential queries. If it fails then IP returns $p(g)$. If it succeeds then it searches through the set returned by the oracle computation for a ground atom $q(g)$ that finitely fails in Π . IP then recursively calls itself with $q(g)$ and returns the output of the recursive call.

EXAMPLE III.20. We now examine the behavior of *ip*, the Prolog implementation of the IP algorithm for the diagnosis of pure Prolog programs [85]. If we apply *ip* to Program III.2, the buggy ISORT program with the missing behavior *isort*([2, 1, 3], [1, 2, 3]), *ip* will make a number of oracle queries (answered here by the user) and then return an instance of missing behavior. The *ip* algorithm proceeds as follows.

On input *isort*([2, 1, 3], [1, 2, 3]), *ip* tries to construct an oracle computation for $\leftarrow \text{isort}([2, 1, 3], [1, 2, 3])$. This goal resolves with the first clause in the buggy ISORT program producing the resolvent $\leftarrow \text{isort}([1, 3], Z), \text{insert}(2, Z, [1, 2, 3])$. In order to find the ground instances of *isort*([1, 3], Z) that are satisfied by an intended interpretation M , an existential oracle query is made. The query and its response are

query: $M \models \exists Z(\text{isort}([1, 3], Z))?$ {*isort*([1, 3], [1, 3])}.

To complete the oracle computation, an oracle query is made to confirm that the remaining atom in the resolvent is also satisfied by M . The query and its response are

query: $M \models \text{insert}(2, [1, 3], [1, 2, 3])? \{ \text{insert}(2, [1, 3], [1, 2, 3]) \}$.

Therefore, an oracle computation for the goal $\leftarrow \text{isort}([2, 1, 3], [1, 2, 3])$ in M is $\{ \text{isort}([1, 3], [1, 3]), \text{insert}(2, [1, 3], [1, 2, 3]) \}$. Now, the atom $\text{isort}([1, 3], [1, 3])$ of the oracle computation succeeds in Π but the atom $\text{insert}(2, [1, 3], [1, 2, 3])$ finitely fails in Π . Therefore ip is recursively called with input $\text{insert}(2, [1, 3], [1, 2, 3])$ and attempts to construct an oracle computation for the goal $\leftarrow \text{insert}(2, [1, 3], [1, 2, 3])$. This goal resolves with the first *insert* clause of the buggy ISORT program producing the resolvent $\leftarrow 1 > 2, \text{insert}(1, [3], [2, 3])$. Since $>$ is a built-in predicate it is assumed to be fault-free. However, the goal $\leftarrow 1 > 2$ finitely fails in Π so $1 > 2 \notin M$. Therefore it is not possible to construct an oracle computation with the first *insert* clause. Furthermore, the goal $\leftarrow \text{insert}(2, [1, 3], [1, 2, 3])$ does not resolve with the remaining *insert* clauses so it is not at all possible to construct an oracle computation for this goal. Therefore ip concludes that the *insert* procedure is incomplete with respect to M . ip also returns $\text{insert}(2, [1, 3], [1, 2, 3])$ as an uncovered atom with respect to M . This agrees with Examples III.8 and III.14. It also identifies the same faulty procedure found using fp in Example III.18.

CHAPTER IV

Consistency-Based Diagnosis

The intuition underlying the diagnostic approach known as consistency-based diagnosis was introduced by de Kleer and Williams in [24] in the form of a program called GDE—the “General Diagnostic Engine.” GDE itself was based on a program of de Kleer’s for non-monotonic reasoning, called the ATMS (“Assumption-based Truth Maintenance System”) [16, 17, 18]. The appeal of this approach for diagnosis is that it does not require any information regarding expected failure modes of system components. The approach only requires descriptions of the intended structure and behavior of a system undergoing diagnosis. In general, there are no assumptions made about the behavior of faulty system components.

In [82], Reiter formalized the intuition underlying GDE. This framework was later generalized by de Kleer, Mackworth and Reiter in [22]. In this chapter we present an overview of the theory behind consistency-based diagnosis that is primarily drawn from this latter source. We also describe the alternatives available for computing consistency-based diagnoses. Finally, we introduce terminology that will be used later in the dissertation for evaluating procedures that compute consistency-based diagnoses.

1. A Diagnosis

DEFINITION IV.1. A *system* is a triple (SD, COMPS, OBS) where:

- (1) SD, the *system description*, is a set of first-order sentences,
- (2) COMPS, the *system components*, is a finite set of constants,
- (3) OBS, the *assertions*¹, is a set of first-order sentences.

¹The set OBS has traditionally been called the *observations*. However, we generalize its

We assume that there exists a unique predicate symbol ab mentioned in SD such that $ab(c)$ is true when component $c \in \text{COMPS}$ is *abnormal* or *faulty*.

DEFINITION IV.2. Given two set of components C_p and C_n define $\mathcal{D}(C_p, C_n)$ to be the conjunction

$$\bigwedge_{c \in C_p} ab(c) \wedge \bigwedge_{c \in C_n} \neg ab(c).$$

We now define a diagnosis. A diagnosis is a sentence describing a possible state of the system, where this state is an assignment of the status normal or abnormal to each system component.

DEFINITION IV.3. Let $\Delta \subseteq \text{COMPS}$. $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ is a *diagnosis* for $(\text{SD}, \text{COMPS}, \text{OBS})$ iff

$$\text{SD} \cup \text{OBS} \cup \{\mathcal{D}(\Delta, \text{COMPS} - \Delta)\} \text{ is satisfiable.}$$

2. Minimal Diagnoses

From the previous definition it follows that a diagnosis exists for $(\text{SD}, \text{COMPS}, \text{OBS})$ iff $\text{SD} \cup \text{OBS}$ is satisfiable. Unfortunately there may be $2^{|\text{COMPS}|}$ diagnoses, therefore a parsimonious characterization of all possible diagnoses of a system is desirable. This brings us to the notion of a minimal diagnosis.

DEFINITION IV.4. A diagnosis $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ is a *minimal diagnosis* iff for no proper subset Δ' of Δ is $\mathcal{D}(\Delta', \text{COMPS} - \Delta')$ a diagnosis.

Thus a minimal diagnosis is determined by a minimal set of components which can be assumed to be faulty, while assuming the remaining components are functioning normally. Note that a consequence of the above definitions is that if $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ is a diagnosis then there is a minimal diagnosis $\mathcal{D}(\Delta', \text{COMPS} - \Delta')$ such that $\Delta' \subseteq \Delta$. However, a similar statement known as the minimal diagnosis hypothesis does not always hold. The *minimal diagnosis hypothesis* states: if $\mathcal{D}(\Delta', \text{COMPS} - \Delta')$ is a minimal diagnosis and $\Delta' \subseteq \Delta \subseteq \text{COMPS}$, then $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ is a diagnosis. Therefore, when the hypothesis holds every superset of the faulty components of a minimal diagnosis is a diagnosis. Moreover, when the hypothesis holds one can conclude that the set of all minimal diagnoses of a system characterize all possible diagnoses for the system.

meaning here.

The following definitions and theorem from [22] describe a simple syntactic test to determine whether or not the minimal diagnosis hypothesis holds for a system.

DEFINITION IV.5 (*ab*-LITERAL). Let $(SD, COMPS, OBS)$ be a system. An *ab*-literal is $ab(c)$ or $\neg ab(c)$ for some $c \in COMPS$.

DEFINITION IV.6 (IGNORANCE OF ABNORMAL BEHAVIOR CONDITION). The *ignorance of abnormal behavior condition* holds for a system $(SD, COMPS, OBS)$ if, in the (disjunctive) clausal form of $SD \cup OBS$, every occurrence of an *ab*-literal is positive.

THEOREM IV.7. If a system $(SD, COMPS, OBS)$ satisfies the ignorance of abnormal behavior condition and $\mathcal{D}(\Delta, COMPS - \Delta)$ is a diagnosis for $(SD, COMPS, OBS)$, then the minimal diagnosis hypothesis holds for $(SD, COMPS, OBS)$.

EXAMPLE IV.8. As an example of consistency-based diagnosis consider the schematic for a combinational full-adder circuit shown in Figure IV.6. A system description of its specified structure and behavior is:

$$\left\{ \begin{array}{l} ab(x_1) \vee n = xor(k, l), \\ ab(a_1) \vee q = and(k, l), \\ ab(a_2) \vee p = and(m, n), \\ ab(x_2) \vee r = xor(n, m), \\ ab(o_1) \vee s = or(p, q) \end{array} \right\}$$

Note that we have excluded the axioms defining the *xor*, *and* and *or* functions, and the equality axioms over 0 and 1, none of which include an *ab*-literal. Take $OBS = \{k = 1 \wedge l = 0 \wedge m = 1 \wedge r = 1 \wedge s = 0\}$ reflecting the fact that a full-adder artifact is behaving erroneously. It is easy to see that the ignorance of abnormal behavior condition holds for this system. As such, the minimal diagnoses of the system completely characterize the diagnoses of the system. The following three minimal diagnoses are defined: $\mathcal{D}(\{x_1\}, \{a_1, a_2, x_2, o_1\})$, $\mathcal{D}(\{x_2, o_1\}, \{x_1, a_1, a_2\})$, and $\mathcal{D}(\{x_2, a_2\}, \{x_1, a_1, o_1\})$. Since the minimal diagnosis hypothesis holds, then for example $\mathcal{D}(\{x_1, a_1, a_2\}, \{x_2, o_1\})$ is also a diagnosis for the system.

FIGURE IV.6. Schematic of a normal combinational full-adder circuit

3. Characterization of Diagnoses

The minimal diagnoses of a system can be characterized in terms of the notions of *ab*-clauses, prime implicates and prime implicants (see, for example, [51]) which we now define.

DEFINITION IV.9 (*ab*-CLAUSE). An *ab*-clause is a satisfiable disjunction of *ab*-literals. A positive *ab*-clause is an *ab*-clause all of whose literals are positive.

DEFINITION IV.10 (PRIME IMPLICATE). A *prime implicate* of a set Σ of sentences is a (disjunctive) clause C such that

- (1) $\Sigma \models C$, and
- (2) for no proper subset C' of C does $\Sigma \models C'$.

DEFINITION IV.11 (PRIME IMPLICANT). A *prime implicant* of a set Σ of sentences is a satisfiable (conjunctive) clause C such that

- (1) $C \models \Sigma$, and
- (2) for no proper subset C' of C does $C' \models \Sigma$.

The minimal diagnoses of a system are then characterized as follows:

THEOREM IV.12 ([82, Reiter]). $\mathcal{D}(\Delta, \text{COMPS}-\Delta)$ is a minimal diagnosis for a system $(SD, \text{COMPS}, S, OBS)$ iff $\bigwedge_{c \in \Delta} ab(c)$ is a prime implicant of the set of positive *ab*-clauses that are prime implicates of $SD \cup OBS$.

Using this characterization, computing the minimal diagnoses of a system is achieved by computing a class of the prime implicates of the $SD \cup OBS$ and then converting these to prime implicants. The advantage that comes with characterizing minimal diagnoses this way is that in the field of switching theory many procedures exist to find the prime implicates/implicants of a theory and to convert

sets of prime implicants to sets of prime implicants and vice-versa. However, these procedures apply only to propositional theories not first-order theories. Therefore $SD \cup OBS$ must be propositional to use one of these procedures.

4. Fault Models

Fault models are formulas in a system description that define how a system component behaves for a particular type of component failure mode. While the inclusion of fault models in a system description is not strictly required for use with consistency-based diagnosis, including them constrains the set of possible diagnoses of a system to reflect only anticipated modes of failure (e.g. [25, 88, 38]). For example, one might constrain the possible modes of failure of a combinational logic gate to be either stuck-at-1 or stuck-at-0, or one might constrain a wire to behave only as an open circuit when it fails (and not as a voltage source which is logically possible without fault models! [88]). Note that it is still possible to consider unanticipated modes of failure along with any anticipated modes of failure defined for a component. This is made possible by defining a fault mode for a component that reflects an unknown mode of failure i.e. no behavior is defined for the unknown mode [25].

The disadvantage of performing diagnosis with fault models is that the complexity of what is an exponential search without fault models, becomes exponentially worse [38]. One way to combat the complexity problem is to guide the search probabilistically, generating only one diagnosis at a time [25, 20]. In such a framework, the unknown failure mode would be given a low probability of occurrence.

5. Computing Diagnoses

5.1. ATMS. Using some form of ATMS has been the dominant approach for computing consistency-based diagnoses. This approach, which effectively determines the prime implicants of $SD \cup OBS$ and then computes the prime implicants from the prime implicants [81, 49, 50, 47], restricts system descriptions to be propositional. Diagnosis computation based on these principles is sound and complete, but it is only possible to compute all diagnoses at once using these approaches. As such, the worst case computational complexity is exponential in the number of system components [79, 83]. Recognizing that it is rare when knowledge of all possible diagnoses is required at once, most practical ATMS-based approaches

FIGURE IV.7. Lattice of positive *ab*-clauses for the full-adder circuit

sacrifice completeness for efficiency, which is usually achieved using *a priori* probability of component failure information to find only the most probable diagnoses (e.g. [24, 35, 25, 19, 20, 21, 23]).

5.2. Generate-and-Test.

5.2.1. HS-Tree

In [82, 42], Reiter describes a generate-and-test approach for computing minimal consistency-based diagnoses that is amenable to first-order system descriptions. Reiter's approach generates positive *ab*-clauses and uses a theorem prover to test whether or not the clause is an implicate of $SD \cup OBS$. The generate-and-test procedure continues until all the prime implicates have been found. To determine the set of all minimal diagnoses, the prime implicants of the prime implicates are then computed.

When ordered with respect to the subset relation, the space of all positive *ab*-clauses forms a lattice.

EXAMPLE IV.13. Figure IV.7 shows the lattice of positive *ab*-clauses for the combinational full-adder circuit of Figure IV.6.

Reiter's positive *ab*-clause generator searches through such a lattice in a bottom-up, breadth-first fashion in order to locate *ab*-clauses that are prime implicates. For example, the first test using the theorem prover determines whether or not the empty (disjunctive) clause is an implicate of $SD \cup OBS$. This is equivalent

to determining whether or not $SD \cup OBS$ is satisfiable. If the empty clause is an implicate then it must also be the only prime implicate since the empty clause is a subset of all other *ab*-clauses in the lattice. The only prime implicant corresponding to this implicate is also the empty (conjunctive) clause. Therefore $\mathcal{D}(\emptyset, COMPS)$ would be the only minimal diagnosis. On the other hand, if the empty (disjunctive) clause is not an implicate, then the *ab*-clauses in the next level of the lattice must be tested.

The collection of all search paths through the lattice form a tree that Reiter calls a *hitting set tree* or *HS-tree*. Note that using a bottom-up search strategy ensures that when an *ab*-clause is identified as an implicate, it must also be prime. Therefore no supersets of the implicate in the lattice need to be tested.

An attractive property of this approach is that the procedure for generating clauses is separated from the procedure for testing clauses. Reiter proposes that the generator access the tester by calling a ternary function TP ("theorem prover") with arguments (SD, OBS, *C*) where *C* is a positive *ab*-clause. The TP function returns "true" if *C* is an implicate of $SD \cup OBS$, otherwise it returns "false." Separating the generator from the tester in this way permits different theorem provers to be used for different classes of system descriptions and observations. Furthermore, the soundness, completeness and decidability properties of diagnosis computation become solely contingent on the soundness, completeness and decidability properties of the theorem prover employed. Note that since there is no decision procedure for determining the satisfiability of a general set of first-order formulas then the task of computing diagnoses in the general case must also be undecidable.

Like the ATMS-based approach, a potential disadvantage of this approach is that all prime implicates in the lattice must be found before any minimal diagnoses can be computed. The resulting complexity of the overall task is exponential in the number of system components [38, 83].

5.2.2. IDA

Recognizing that it is rare when one needs to know all diagnoses simultaneously, Mozetič in [65] proposes an alternative generate-and-test approach called the *incremental diagnostic algorithm* or *IDA* that directly generates one minimal diagnosis at a time thereby reducing the overall complexity of the task to polynomial. A

different implementation of the same approach is discussed in [9, 66].

In essence, the approach generates subsets Δ of components and calls a theorem prover to test whether or not $SD \cup OBS \cup \{\mathcal{D}(\Delta, COMPS - \Delta)\}$ is satisfiable. If satisfiable then $\mathcal{D}(\Delta, COMPS - \Delta)$ is a diagnosis. The space of all subsets $COMPS$ forms a lattice similar to that formed by the space of all disjunctions of positive *ab*-literals. In contrast to Reiter's disjunction generation procedure, Mozetič's procedure proceeds to search through the lattice of subsets in a top-down, depth-first fashion. This way, once an diagnosis has been identified, the procedure searches below the diagnosis in the lattice until it finds a minimal diagnosis, which is guaranteed to exist. Note that the search through the lattice is not a blind one—it exploits both the failure and the success of tests to guide the search in an efficient manner.

Like Reiter's procedure, Mozetič's subset generation procedure also accesses the tester by calling a ternary TP function with the same three arguments. However, in this case the function returns "true" if $\mathcal{D}(C, COMPS - C)$ is a diagnosis for $SD \cup OBS$, otherwise it returns "false."

Mozetič proves in [65] that if calls to the TP function are assumed to occur in constant time, then for a fixed set of k minimal diagnoses that have already been found, his procedure decides whether the next minimal diagnosis exists (and finds one) in polynomial time $\mathcal{O}(n^{2k})$, where n is the number of components in the system. Thus, when the first few minimal diagnoses are computed, the complexity of the task is a low order polynomial. However, as subsequent minimal diagnoses are computed, the complexity of the task will increase exponentially. Note that this result applies only to the computation of minimal diagnoses for system descriptions that do not include fault models. Once fault models are introduced to a system description, the task of finding even the next minimal diagnosis has exponential complexity [38].

5.2.3. Properties of TP

As mentioned above, one benefit of using a generate-and-test approach is that the soundness, completeness and decidability properties of diagnosis computation become solely contingent on the soundness, completeness and decidability properties of the theorem prover employed. At this point we introduce some formal properties associated with any TP function utilized by the IDA approach to diag-

nosis computation. In the chapters that follow we will be evaluating various TP functions with respect to these properties.

General purpose theorem provers are normally evaluated with respect to the soundness, completeness and decidability of their inference systems for affirming or refuting the satisfiability of a set of sentences. Since the TP function for IDA either confirms or denies that a sentence $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ is a diagnosis then we can view the task of TP to be one of *diagnosis verification*. Furthermore, we can speak of the soundness, completeness and decidability of a diagnosis verification function TP with respect to *diagnosis confirmation* and *diagnosis denial*. As such, the notions of diagnosis confirmation and denial are nothing more than specializations of notions normally associated with general purpose theorem provers.

The following are the formal definitions of the soundness, completeness and decidability of diagnosis confirmation and denial.

DEFINITION IV.14 (SOUNDNESS—DIAGNOSIS CONFIRMATION/DENIAL).

Let $(\text{SD}, \text{COMPS}, \text{OBS})$ be a system, $\Delta \subseteq \text{COMPS}$. A diagnosis verification function TP is sound for diagnosis confirmation (resp., denial) iff whenever $\text{TP}(\text{SD}, \text{OBS}, \Delta)$ returns "true" (resp., "false") then $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ is (resp., is not) a diagnosis for $(\text{SD}, \text{COMPS}, \text{OBS})$.

DEFINITION IV.15 (COMPLETENESS—DIAGNOSIS CONFIRMATION/DENIAL).

Let $(\text{SD}, \text{COMPS}, \text{OBS})$ be a system, $\Delta \subseteq \text{COMPS}$. A diagnosis verification function TP is complete for diagnosis confirmation (resp., denial) iff whenever $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ is (resp., is not) a diagnosis for $(\text{SD}, \text{COMPS}, \text{OBS})$ then $\text{TP}(\text{SD}, \text{OBS}, \Delta)$ returns "true" (resp., "false").

If we assume that for a system $(\text{SD}, \text{COMPS}, \text{OBS})$, a diagnosis verification function $\text{TP}(\text{SD}, \text{OBS}, \Delta)$ is total *i.e.* it is defined for all possible values of its arguments, then if TP is incomplete for diagnosis confirmation, TP must also be unsound for diagnosis denial. Similarly, under the same assumption, if TP is incomplete for diagnosis denial then TP must also be unsound for diagnosis confirmation. On the other hand, if TP is partially defined, it is possible for a diagnosis verification function TP to be simultaneously incomplete for diagnosis confirmation and sound for diagnosis denial. In such a case, $\text{TP}(\text{SD}, \text{OBS}, \Delta)$ might be undefined when $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ is a diagnosis for a system $(\text{SD},$

COMPS, OBS). Similarly, it is possible for TP to be both incomplete for diagnosis denial and sound for diagnosis confirmation.

The decidability of diagnosis confirmation and denial concerns the decidability of the questions "Is $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ a diagnosis for (SD, COMPS, OBS)?" , and "Is $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ *not* a diagnosis for (SD, COMPS, OBS)?"

DEFINITION IV.16 (PARTIAL DECIDABILITY—DIAGNOSIS CONFIRMATION). Let (SD, COMPS, OBS) be a system, $\Delta \subseteq \text{COMPS}$. The question "Is $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ a diagnosis for (SD, COMPS, OBS)?" is *partially decidable* iff there exists an algorithm to implement a diagnosis verification function TP that outputs "true" iff $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ is a diagnosis for (SD, COMPS, OBS) but may infinitely loop when $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ is not a diagnosis for (SD, COMPS, OBS).

DEFINITION IV.17 (PARTIAL DECIDABILITY—DIAGNOSIS DENIAL). Let (SD, COMPS, OBS) be a system, $\Delta \subseteq \text{COMPS}$. The question "Is $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ *not* a diagnosis for (SD, COMPS, OBS)?" is *partially decidable* iff there exists an algorithm to implement a diagnosis verification function TP that outputs "false" iff $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ is not a diagnosis for (SD, COMPS, OBS) but may infinitely loop when $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ is a diagnosis for (SD, COMPS, OBS).

Note that partial decidability for diagnosis confirmation (resp., denial) requires that the diagnosis verification function TP be both sound and complete for diagnosis confirmation (resp., denial).

DEFINITION IV.18 (TOTAL DECIDABILITY). Let (SD, COMPS, OBS) be a system, $\Delta \subseteq \text{COMPS}$. The question of diagnosis confirmation or diagnosis denial is *totally decidable* iff there exists an algorithm to implement TP which is capable of distinguishing between diagnoses and non-diagnoses for a system (SD, COMPS, OBS) for all sentences $\mathcal{D}(\Delta, \text{COMPS} - \Delta)$ without ever entering an infinite loop.

Note that if either diagnosis confirmation or diagnosis denial are totally decidable then this implies that both questions must be partially decidable.

CHAPTER V

A Consistency-Based Diagnostic Framework Using Logic Program-Based System Descriptions

1. Introduction

In practice, only propositional system descriptions have been used for consistency-based diagnosis. However, in order to use consistency-based diagnosis for software diagnosis a more expressive modelling language is required. In this chapter we present a consistency-based diagnosis framework that utilizes system descriptions based on logic programs. System descriptions of this type are adequate for modelling and diagnosis of software. We present an example of diagnosing a buggy insertion sort program in the absence of formal specifications to illustrate this capability. As far as we are aware, no other formal consistency-based diagnosis framework uses first-order system descriptions or is capable of software diagnosis. We also show how system descriptions based on logic programs can be constructed to model hierarchically structured systems. This is beneficial since many systems are viewed as structural hierarchies and as such, it is only natural to diagnose such systems in hierarchical terms. We present an example of diagnosing a hierarchical full-adder circuit using a description of the circuit's specified structure and behavior to illustrate this capability.

The consistency-based diagnosis framework we present here also serves to unify a number of heretofore disparate diagnostic approaches. Most significantly we show how the approach known as declarative error diagnosis is just an instance of the consistency-based diagnosis framework. An immediate consequence of establishing this relationship is that each approach inherits the substantial body of results

FIGURE V.8. Relationships between artifacts, specifications and their system descriptions

belonging to the other. In general, we show that the consistency-based diagnosis framework is superior in many respects to other comparable diagnostic approaches.

2. Constructing the System Description

We begin by presenting a step-by-step procedure for constructing a system description based on a logic program. We distinguish between the system undergoing diagnosis, known as the *artifact* and the system description used for consistency-based diagnosis. Depending on the diagnostic domain, a system description either reflects *specified* structure and behaviour of an artifact, or *actual* structure and behaviour of an artifact. The former type of system description, which we call a *specification system description* is typically used in the hardware development domain. The latter, which we call an *artifact system description* would be used in the software development domain when no concrete specifications are available but the software artifact itself is available. Figure V.8 illustrates the relationships between artifacts, specifications and their system descriptions.

Since consistency-based diagnosis has traditionally been used to diagnose physical systems and not software, in the following discussion we describe how to construct a system description based on a logic program that reflects the specified structure and behaviour of a combinational full-adder circuit shown in Figure V.9. Later we present a logic program which reflects the actual structure and behaviour of a faulty software artifact.

There are three steps involved in constructing a specification system description of an artifact:

- (1) constructing a logic program Π that *adequately* reflects the specified structure and behaviour of the artifact,
- (2) constructing the predicate completion $comp(\Pi)$ of Π , and

FIGURE V.9. Specified structural interconnections of a full-adder circuit

FIGURE V.10. An *and* gate

(3) syntactically transforming $comp(\Pi)$ to the system description $SD(\Pi)$.

2.1. Step 1. We stress the word 'adequate' above since it is important that a particular ontology be used when constructing the logic program Π . In particular we adopt the procedural ontology used for declarative error diagnosis. Namely, a predicate symbol p belonging to the language of Π , denoted $L(\Pi)$, must represent an actual system component. As such, let $COMPS(\Pi)$ represent the set of predicate symbols in a logic program Π . Also, a term t belonging to the Herbrand universe of Π , denoted $H(\Pi)$, represents a possible component port value. For example to model the *and* gate labelled $a1$ in Figure V.10, one might use the predicate symbol $a1$ to denote the component and the terms 1 and 0 to denote its possible input and output port values. Using this ontology, component behaviour is defined as a relation over n -tuples of possible port values. Thus, we would take $a1$ to represent the subset $\{(0,0,0), (0,1,0), (1,0,0), (1,1,1)\}$ of the set $\{0,1\} \times \{0,1\} \times \{0,1\}$, where we take the first and second ordinates of each tuple to represent the *and* gate's two inputs I_1 and I_2 and we take the third ordinate to represent the gate's output O_1 .

A predicate in Π therefore defines the possible specified behaviour of the component it represents. For example, a logic program defining the specified behaviour

of the *and* gate *a1* is $\Pi = \{a1(0,0,0), a1(0,1,0), a1(1,0,0), a1(1,1,1)\}$. Since *a1*(1,0,0) belongs to the least Herbrand model of this program, denoted $M(\Pi)$, then we know that for inputs 1 and 0, *a1*'s output is 0.

The immediate descendants of a parent component will be referred to as the parent's children. In terms of the procedural interpretation of a logic program, the hierarchical relationship that exists between a parent component and its children is represented by a clause in a logic program such that the clause head represents the parent and the atoms of the clause body represent its children. For example, the first clause of Program V.1 represents the hierarchical relationship that exists between the parent *fullAdder* component and its children depicted in Figure V.9. The remaining clauses in Program V.1 define the behaviour of the child components themselves. In this case the child components possess no children of their own.

PROGRAM V.1.

$$\begin{aligned} fullAdder(K, L, M, R, S) \leftarrow & x1(K, L, N), a1(K, L, Q), a2(M, N, P), \\ & x2(N, M, R), o1(P, Q, S). \\ x1(0,0,0). \quad x1(0,1,1). \quad x1(1,0,1). \quad x1(1,1,0). \\ a1(X,0,0). \quad a1(0,X,0). \quad a1(1,1,1). \\ a2(X,0,0). \quad a2(0,X,0). \quad a2(1,1,1). \\ x2(0,0,0). \quad x2(0,1,1). \quad x2(1,0,1). \quad x2(1,1,0). \\ o1(1,X,1). \quad o1(X,1,1). \quad o1(0,0,0). \end{aligned}$$

Thus, the procedural interpretation of a logic program allows us to interpret a logic program as a structural hierarchy. Indeed, since recursive procedure calls are permitted in logic programs, then it is possible to model the notion of recursive hierarchies. Furthermore, since a logic program has operational semantics, this stage of the system description may be used to simulate or query the specified positive behaviour of the system being modelled.

2.2. Step 2. In defining the specified behaviour of the components with a logic program Π we construct the predicate completion of Π , denoted $comp(\Pi)$, to explicitly define the behaviour that cannot be exhibited by a component. Thus, taking Π to be Program V.1, we can say that since $comp(\Pi) \models \neg fullAdder(1,0,1,1,0)$

then we know that *fullAdder* cannot produce the outputs $R = 1$ and $S = 0$ for the inputs $K = 1$, $L = 0$ and $M = 1$. By supplementing the operational semantics of a logic program Π with the negation as failure meta-rule, it is possible to use Π to simulate or query the specified negative behaviour of the system being modelled.

Recall that the completion $comp(\Pi)$ consists of two sets of sentences: $IFF(\Pi)$ and $EQ(\Pi)$. Sentences of $IFF(\Pi)$ are of the form $\forall \mathbf{X}(p(\mathbf{X}) \leftrightarrow F)$ where each sentence completely defines the specified behaviour of a component p . Sentences of $EQ(\Pi)$ constitute an equality theory that ensures equality between terms is interpreted as identity.

2.3. Step 3. The final step is a simple syntactic transformation that introduces the distinguished unary predicate *ab* to the completion $comp(\Pi)$ rendering a system description based on a logic program Π , denoted $SD(\Pi)$, where

$$SD(\Pi) = \{\forall \mathbf{X}(p(\mathbf{X}) \leftrightarrow F) \vee ab(p) \mid \forall \mathbf{X}(p(\mathbf{X}) \leftrightarrow F) \in IFF(\Pi)\} \cup EQ(\Pi).$$

Thus, each sentence $\forall \mathbf{X}(p(\mathbf{X}) \leftrightarrow F)$ belonging to $IFF(\Pi)$ is disjoined with an atom $ab(p)$ in $SD(\Pi)$. Constructed this way a system description $SD(\Pi)$ logically defines the relationship between the abnormal behaviour of a component p and the normal behaviour of p .

3. Assertions

Since the framework we present here accommodates both specification system descriptions and artifact system descriptions we need to clarify the interpretation of assertions in both contexts.

When using a specification system description, assertions should reflect the actual behaviour of a (presumably faulty) artifact. On the other hand, when using an artifact system description, assertions should reflect the intended behavior of a (presumably fault-free) specification.

Formally, we define assertions with respect to an interpretation I of a logic program Π . When Π is the basis for a specification system description then we call I the *actual interpretation* of an artifact. An actual interpretation I reflects the actual behavior of an artifact. Therefore, when using a specification system description, assertions must be true in an actual interpretation I . Similarly, when Π is the basis for an artifact system description we call I the *intended interpretation*

FIGURE V.11. Actual structural interconnections of a faulty full-adder circuit

of an artifact. An intended interpretation I reflects the intended behavior of an artifact. Therefore when using an artifact system description, assertions must be true in an intended interpretation I . This latter relationship between assertions and interpretations is the same as that used for declarative error diagnosis [85, 56, 33].

Note that an actual interpretation I of a faulty artifact cannot be a model for a logic program Π that is the basis for a specification system description $SD(\Pi)$. This is because models of Π only reflect the specified behavior of the artifact, whereas an actual interpretation reflects the erroneous behavior of a faulty artifact. Similarly, an intended interpretation I of a faulty artifact cannot be a model for a logic program Π that is the basis for an artifact system description $SD(\Pi)$. This is because models of Π reflect the erroneous behavior of the artifact, whereas an intended interpretation only reflects the specified behavior of the artifact.

EXAMPLE V.2. For example, consider the faulty full-adder artifact depicted in Figure V.11. Since this artifact has its output lines crossed it exhibits the erroneous behaviour $fullAdder(1,0,1,1,0)$ instead of the specified behaviour $fullAdder(1,0,1,0,1)$. Therefore $fullAdder(1,0,1,1,0)$ and $\neg fullAdder(1,0,1,0,1)$ must be true in any actual interpretation I . Note that since Π represents the specified structure and behaviour of an artifact and since I is a model for the behaviour of the faulty artifact then I cannot be a model for the completion of Program V.1.

4. Consistency-Based Diagnosis

With the notions of a system description and assertions formally defined we can now use the framework for defining consistency based diagnoses. What follows are examples of hardware diagnosis and software diagnosis in the context of the framework.

EXAMPLE V.3. As an example of hardware diagnosis, take Π to be Program V.1, the logic program defining the specified structure and behaviour of the full-adder. Let $OBS = \{fullAdder(1,0,1,1,0)\}$, where the assertion $fullAdder(1,0,1,1,0)$ corresponds to erroneous behaviour exhibited by the *fullAdder* component depicted in Figure V.11. The minimal diagnoses defined by consistency-based diagnosis for the system $(SD(\Pi), COMPS(\Pi), OBS)$ are

$$\begin{aligned} &\mathcal{D}(\{fullAdder\}, \{x1, a1, x2, a2, o1\}), \\ &\mathcal{D}(\{x1\}, \{fullAdder, a1, a2, x2, o1\}), \\ &\mathcal{D}(\{x2, o1\}, \{fullAdder, x1, a1, a2\}), \\ &\mathcal{D}(\{x2, a2\}, \{fullAdder, x1, a1, o1\}). \end{aligned}$$

Note that in this example the parent is the sole faulty component in the first diagnosis and child components constitute the faulty components in the other diagnoses thereby illustrating how this framework distinguishes between components that could be causing the observed error and those components that are merely propagating errors in a hierarchically structured system. We examine the meaning of faults occurring in a hierarchical system in a following section.

EXAMPLE V.4. As a further example of diagnosing the faulty full-adder artifact, assume that enough of the behavior of components *x1* and *o1* is observable to be able to make the negative assertions $\neg x1(1,0,0)$ and $\forall X \neg o1(1, X, 0)$. Therefore we let the set of assertions be $OBS = \{fullAdder(1,0,1,1,0), \neg x1(1,0,0), \forall X \neg o1(1, X, 0)\}$. For the system $(SD(\Pi), COMPS(\Pi), OBS)$, consistency-based diagnosis defines the following minimal diagnoses:

$$\begin{aligned} &\mathcal{D}(\{fullAdder\}, \{x1, a1, x2, a2, o1\}), \\ &\mathcal{D}(\{x2, a2\}, \{fullAdder, x1, a1, o1\}). \end{aligned}$$

Note that by including negative assertions the number of candidate diagnoses has been reduced from the previous example.

In order to use the diagnostic framework for software diagnosis in the absence of concrete, formal specifications we only have to change our interpretation of some elements of the framework. In particular, the logic program Π upon which the system description $SD(\Pi)$ is based must represent the *actual* structure and behaviour of the software artifact so that $SD(\Pi)$ is an artifact system description. In the domain of logic program diagnosis, Π is nothing more than the artifact itself. In addition, the assertions OBS must reflect the *specified* behaviour of the artifact. In this case an interpretation I that is a model for all assertions of OBS is an *intended interpretation*. Note that in this context the framework is also suitable for the diagnosis of faulty specifications.

We now present an example of logic program diagnosis that originates with Shapiro [85] (see Example III.18).

EXAMPLE V.5. Program III.2 in Chapter III is a buggy insertion sort logic program—the bug being that the arguments of the $>$ test in the first *insert* clause are exchanged. As a result, Program III.2 exhibits the spurious behaviour *isort*([2, 1, 3], [2, 3, 1]). Thus, an intended interpretation I of this program would include the element *isort*([2, 1, 3], [1, 2, 3]).

In Shapiro's framework, an algorithm queries the diagnostician (or "oracle") regarding the truth of different literals in the intended Herbrand interpretation M . We can use the diagnostician's responses to these queries as the basis for a set of assertions OBS. In this case, Shapiro's algorithm would ascertain the following assertions:

$$\left\{ \begin{array}{l} \text{isort}([], []), \text{insert}(3, [], [3]), \text{isort}([3], [3]), \text{insert}(1, [], [1]), \\ \neg \text{isort}([2, 1, 3], [2, 3, 1]), \neg \text{insert}(1, [3], [3, 1]) \end{array} \right\}.$$

If we let Π be Program III.2 so that the built-in predicates $>$ and $\leq$ are assumed to be normal, and we let OBS be the set of assertions above then consistency-based diagnosis defines only one minimal diagnosis for $(SD(\Pi), \text{COMPS}(\Pi), \text{OBS})$ such that $\Delta = \{\text{insert}\}$. This agrees with the output obtained from Shapiro's algorithm in Example III.18. Later in this chapter we will show how it is possible to use

this framework to obtain diagnoses of an equivalent granularity to those obtained using declarative error diagnosis.

5. Characterizing Diagnoses

The diagnoses for a system $(SD(\Pi), COMPS(\Pi), OBS)$ can be parsimoniously characterized in terms of the *minimal* diagnoses of the system. That is to say, the following theorem, which is an instance of the minimal diagnosis hypothesis, holds.

THEOREM V.6. *Let Π be a logic program, $\Delta \subseteq COMPS(\Pi)$, and OBS be a set of sentences of $L(\Pi)$ containing no ab literals. If $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a minimal diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$ then for all $\Delta \subseteq \Delta' \subseteq COMPS(\Pi)$, $\mathcal{D}(\Delta', COMPS(\Pi) - \Delta')$ is a diagnosis.*

PROOF. Follows from the ignorance of abnormal behavior condition (Definition IV.6) and the form of sentences in $SD(\Pi) - EQ(\Pi)$ given in Step 3. $\square$

6. Properties of Abnormalities

In this section we show how a system description $SD(\Pi)$ implicitly defines two sufficient, and in some cases necessary and sufficient, conditions for component abnormality in a system. The two fault conditions, called *incompleteness* and *incorrectness*, are logical reconstructions of notions first defined by Shapiro to characterize erroneous procedures in programs as part of his declarative error diagnosis framework [85]. We have found that these notions also intuitively characterize faults in general hierarchical systems.

In a hierarchically structured system we wish to designate only those components which are the *cause* of erroneous behavior as faulty, not those components merely propagating erroneous behavior. The notions of incorrectness and incompleteness capture precisely this distinction. The essence of these fault definitions is that a parent component that behaves counter to its specified behavior is considered faulty when one can determine that its child components are not interconnected as specified. These general definitions subsume the special case when a component has no children, in which case the component is considered faulty simply when it behaves counter to its specified behavior. The essential difference between incorrectness and incompleteness is the following:

- the structure of an incorrect component is such that it will exhibit behaviour it is not specified to exhibit, whereas
- the structure of an incomplete component is such that it will not exhibit behaviour it is specified to exhibit.

The former notion of a fault is familiar. However, the latter notion is also required in order to define faults in nondeterministically behaving systems (see [7] and also Chapter III).

We now show that the definitions for component incorrectness and incompleteness are implied by a system description $SD(\Pi)$ based on a logic program Π . Note that in the following presentation we take Π to represent the specified structure and behaviour of an artifact so the definitions of incorrectness and incompleteness will be opposite to that defined when Π represents the actual structure and behaviour of an artifact. Recall that for each component $p \in COMPS(\Pi)$ there is a sentence belonging to $SD(\Pi) - EQ(\Pi)$ of the form:

$$(\forall X(p(X) \leftrightarrow G)) \vee ab(p),$$

which can also be written

$$(\neg \forall X(p(X) \rightarrow G) \vee \neg \forall Y(G \rightarrow p(Y))) \rightarrow ab(p).$$

Based on this latter form we introduce the following notation:

(V.1) define $INCORRECT(p)$ to be $\neg \forall X(p(X) \rightarrow G)$, and

(V.2) define $INCOMPLETE(p)$ to be $\neg \forall Y(G \rightarrow p(Y))$.

So each sentence belonging to $SD(\Pi) - EQ(\Pi)$ can be written

$$(V.3) \quad (INCORRECT(p) \vee INCOMPLETE(p)) \rightarrow ab(p).$$

The informal meaning of (V.1) or (V.2) being that if a component $p \in COMPS(\Pi)$ is deduced to be either incorrect as defined by (V.1) or incomplete as defined by (V.2) then this implies the abnormality of the component.

7. Properties of Diagnoses

We now present a number of theorems that relate minimal diagnoses obtained using consistency-based diagnosis for a system $(SD(\Pi), COMPS(\Pi), OBS)$ to the notions of faults defined by (V.1) and (V.2). We leave the proofs of these theorems

to the end of this section. Informally the first theorem states that all components of a minimal diagnosis must be incorrect or incomplete. Therefore incorrectness or incompleteness constitute necessary and sufficient conditions for component abnormality in a system.

THEOREM V.7. *Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$ and $\text{OBS} \subseteq L(\Pi)$. If $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a minimal diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ then for all $p \in \Delta$,*

$$\text{SD}(\Pi) \cup \text{OBS} \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \text{INCORRECT}(p) \vee \text{INCOMPLETE}(p).$$

EXAMPLE V.8. By the previous theorem, all faulty components of the minimal diagnoses in Examples V.3, V.4 and V.5 must be incorrect or incomplete.

By restricting the class of assertions used with consistency-based diagnosis we can determine if a faulty component is only incorrect or only incomplete *i.e.* we can determine if incorrectness or incompleteness alone are necessary and sufficient conditions for component abnormality. We first define the restricted classes of assertions.

DEFINITION V.9 (POSITIVE ASSERTION). Let Π be a logic program. A sentence $S \in L(\Pi)$ is a positive assertion iff S is of the form $\exists p(t)$.

DEFINITION V.10 (NEGATIVE ASSERTION). Let Π be a logic program. A sentence $S \in L(\Pi)$ is a negative assertion iff S is of the form $\forall \neg p(t)$.

THEOREM V.11. *Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$, $\text{OBS} = \text{OBS}^- \cup \text{OBS}^+$ where OBS^+ is a set of positive assertions and OBS^- is a set of negative assertions such that for all $F \in \text{OBS}^-$, $\text{comp}(\Pi) \models F$. If $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a minimal diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ then*

$$\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \text{OBS} \models \bigwedge_{q \in \Delta} \text{INCORRECT}(q).$$

THEOREM V.12. *Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$, $\text{OBS} = \text{OBS}^- \cup \text{OBS}^+$ where OBS^- is a set of negative assertions and OBS^+ is a set of positive assertions such that for all $F \in \text{OBS}^+$, $\text{comp}(\Pi) \models F$. If $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a minimal diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ then*

$$\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \text{OBS} \models \bigwedge_{q \in \Delta} \text{INCOMPLETE}(q).$$

Informally, Theorem V.11 states that since $SD(\Pi)$ is a specification system description then if a set of assertions consists of only literals such that the negative literals are assertions reflecting specified behaviour then the abnormal components of minimal diagnoses generated using consistency-based diagnosis are incorrect. Similarly, Theorem V.12 states that if a set of assertions consists only of literals such that the positive literals are assertions reflecting specified behaviour then the abnormal components of minimal diagnoses are incomplete.

EXAMPLE V.13. Consider Example V.3. Since the only assertion in OBS is positive then Theorem V.11 applies to the minimal diagnoses obtained. Thus, the abnormal components of the minimal diagnoses are all incorrect.

EXAMPLE V.14. Consider Example V.4. Since the negative assertions in OBS are logically implied by $comp(\Pi)$ then Theorem V.11 applies to the minimal diagnoses obtained. Thus, the abnormal components of the minimal diagnoses are all incorrect.

EXAMPLE V.15. In Example V.5, Π represents the actual structure and behavior of the software artifact. In this context, the definitions of incorrectness and incompleteness given by (V.1) and (V.2) must be reversed. Referring to Theorem V.12 we can deduce that the abnormal component *insert* must be incorrect, which agrees with the output of Shapiro's algorithm in Example III.18.

We now present the proofs of Theorems V.7, V.11 and V.12. The relationship between the proofs of these theorems and their supporting lemmas is shown in Figure V.12. We proceed with the proofs bottom-up.

The first lemma that we prove is used in proofs throughout the remainder of this dissertation. The lemma defines the relationship that exists between a set of sentences $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\}$ and a subset of $comp(\Pi)$. The subset of $comp(\Pi)$ is defined as follows:

DEFINITION V.16 (Δ -COMPLETION OF Π). Let Π be a logic program and $\Delta \subseteq COMPS(\Pi)$. The Δ -completion of Π , denoted $comp_{\Delta}(\Pi)$ is the set of sentences

$$\{\forall \mathbf{X}(p(\mathbf{X}) \leftrightarrow F) \in IFF(\Pi) | p \in COMPS(\Pi) - \Delta\} \cup EQ(\Pi).$$

FIGURE V.12. Relationship between theorems and their supporting lemmas

Thus, the Δ -completion of Π defines only those predicates designated as normal with respect to a sentence $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$. The relationship between $\text{comp}_{\Delta}(\Pi)$ and $\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\}$ is stated by the following lemma.

LEMMA V.17. *Let Π be a logic program and $\Delta \subseteq \text{COMPS}(\Pi)$. The set of sentences $\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\}$ is semantically equal to $\text{comp}_{\Delta}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\}$.*

PROOF. Let I be an interpretation. The proof follows directly from the fact that for all $(F \vee ab(p)) \in \text{SD}(\Pi)$,

$$\begin{aligned}
 I \models (F \vee ab(p)) \wedge ab(p) &\text{ iff } I \models ab(p), \text{ and} \\
 I \models (F \vee ab(p)) \wedge \neg ab(p) &\text{ iff } I \models \neg ab(p) \wedge F.
 \end{aligned}$$

□

With the previous lemma, one is able to replace a system description $\text{SD}(\Pi)$ with a subset of the completion of Π , and vice versa. We are now prepared to prove Theorem V.7.

PROOF. (Theorem V.7)

By way of contradiction assume there exists a $p \in \Delta$ and an interpretation I such

that

$$I \models \text{SD}(\Pi) \cup \text{OBS} \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \{\neg \text{INCORRECT}(p) \wedge \neg \text{INCOMPLETE}(p)\}.$$

By Lemma V.17 we have

$$I \models \text{comp}_{\Delta}(\Pi) \cup \text{OBS} \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \{\neg \text{INCORRECT}(p) \wedge \neg \text{INCOMPLETE}(p)\}.$$

Since $\text{comp}_{\Delta}(\Pi)$, OBS and $\{\neg \text{INCORRECT}(p) \wedge \neg \text{INCOMPLETE}(p)\}$ do not contain *ab* literals then

$$\text{comp}_{\Delta}(\Pi) \cup \text{OBS} \cup \{\mathcal{D}(\Delta', \text{COMPS}(\Pi) - \Delta')\} \cup \{\neg \text{INCORRECT}(p) \wedge \neg \text{INCOMPLETE}(p)\} \text{ is satisfiable,}$$

where $\Delta' = \Delta - \{p\}$. Furthermore, by (V.1) and (V.2)

$$\begin{aligned} & \text{comp}_{\Delta}(\Pi) \cup \{\neg \text{INCORRECT}(p) \wedge \neg \text{INCOMPLETE}(p)\} \\ &= \text{comp}_{\Delta'}(\Pi). \end{aligned}$$

so by (V.4) and Lemma V.17

$$\text{SD}(\Pi) \cup \text{OBS} \cup \{\mathcal{D}(\Delta', \text{COMPS}(\Pi) - \Delta')\} \text{ is satisfiable.}$$

Therefore $\mathcal{D}(\Delta', \text{COMPS}(\Pi) - \Delta')$ is a diagnosis. But $\mathcal{D}(\Delta', \text{COMPS}(\Pi) - \Delta')$ can't be a diagnosis since $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a minimal diagnosis and $\Delta' \subset \Delta$, which is a contradiction. $\square$

To prove remaining lemmas and theorems we require the following notation.

Let $\text{SD-IF}(\Pi) = \{\forall(p(t) \leftarrow F) \vee ab(p) \mid \forall(p(t) \leftarrow F) \in \text{IF}(\Pi)\} \cup \text{EQ}(\Pi)$, and

let $\text{SD-OIF}(\Pi) = \{\forall(p(t) \rightarrow F) \vee ab(p) \mid \forall(p(t) \rightarrow F) \in \text{OIF}(\Pi)\} \cup \text{EQ}(\Pi)$.

Here $\text{SD-IF}(\Pi)$ and $\text{SD-OIF}(\Pi)$ respectively represent the IF-definitions and the ONLY-IF-definitions of $\text{SD}(\Pi)$. Note that $\text{SD-IF}(\Pi) \cup \text{SD-OIF}(\Pi)$ is semantically equivalent to $\text{SD}(\Pi)$. Similarly,

Let $\text{comp-if}_{\Delta}(\Pi) = \{\forall(p(t) \leftarrow F) \in \text{IF}(\Pi) \mid p \in \text{COMPS}(\Pi) - \Delta\} \cup \text{EQ}(\Pi)$, and

let $\text{comp-oif}_{\Delta}(\Pi) = \{\forall(p(t) \rightarrow F) \in \text{OIF}(\Pi) \mid p \in \text{COMPS}(\Pi) - \Delta\} \cup \text{EQ}(\Pi)$.

Here, $comp\text{-}if_{\Delta}(\Pi)$ and $comp\text{-}oif_{\Delta}(\Pi)$ respectively represent the IF-definitions and the ONLY-IF-definitions of $comp_{\Delta}(\Pi)$. Also note that $comp\text{-}if_{\Delta}(\Pi) \cup comp\text{-}oif_{\Delta}(\Pi)$ is semantically equivalent to $comp_{\Delta}(\Pi)$.

We now present the final two lemmas required to prove the remaining two theorems.

LEMMA V.18. *Let Π be a logic program, $\Delta \subseteq COMPS(\Pi)$, $OBS = OBS^- \cup OBS^+$ where OBS^- is a set of negative assertions and OBS^+ is a set of positive assertions such that for all $L \in OBS^+$, $comp(\Pi) \models L$. $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$ iff $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD\text{-}IF(\Pi), COMPS(\Pi), OBS)$.*

PROOF.

($\Rightarrow$)

Since $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$ then

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS \text{ is satisfiable.}$$

Therefore, by the compactness of FOPC,

$$\text{SD-IF}(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS \text{ is satisfiable.}$$

Thus, $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD\text{-}IF(\Pi), COMPS(\Pi), OBS)$.

($\Leftarrow$)

Since $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD\text{-}IF(\Pi), COMPS(\Pi), OBS)$ then

$$SD\text{-}IF(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS \text{ is satisfiable,}$$

and by Lemma V.17,

$$comp\text{-}if_{\Delta}(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS \text{ is satisfiable.}$$

So by the compactness of FOPC

$$(V.5) \quad comp\text{-}if_{\Delta}(\Pi) \cup OBS \text{ is satisfiable.}$$

Since $comp_{\Delta}(\Pi) \subseteq comp(\Pi)$, then $comp_{\Delta}(\Pi)$ is satisfiable by the compactness of FOPC. We now prove that

$$(V.6) \quad comp_{\Delta}(\Pi) \cup OBS \text{ is satisfiable.}$$

By way of contradiction assume $comp_{\Delta}(\Pi) \cup OBS$ is not satisfiable. Since $comp_{\Delta}(\Pi)$ is satisfiable then for some assertion $L \in OBS$, $comp_{\Delta}(\Pi) \models \neg L$. There are two cases to consider: (i) $L \in OBS^+$, (ii) $L \in OBS^-$. For case (i) Since $comp_{\Delta}(\Pi) \models \neg L$ it follows from monotonicity of FOPC that $comp(\Pi) \models \neg L$ for some $L \in OBS^+$. But $comp(\Pi) \models L$ for all $L \in OBS^+$ which is a contradiction. For case (ii), since $comp-if_{\Delta}(\Pi) \cup comp-oif_{\Delta}(\Pi)$ is semantically equivalent to $comp_{\Delta}(\Pi)$ then $comp-if_{\Delta}(\Pi) \cup comp-oif_{\Delta}(\Pi) \models \neg L$ for some $L \in OBS^-$. So by Lemma II.23, $comp-if_{\Delta}(\Pi) \models \neg L$. But by (V.5), $comp-if_{\Delta}(\Pi) \cup OBS$ is satisfiable which is a contradiction. Therefore (V.6) is true.

Since (V.6) is true and $comp_{\Delta}(\Pi) \cup OBS$ contains no *ab* literals then

$$comp_{\Delta}(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS \text{ is satisfiable,}$$

and by Lemma V.17 it follows that,

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS \text{ is satisfiable.}$$

Thus, $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$. $\square$

LEMMA V.19. Let Π be a logic program, $\Delta \subseteq COMPS(\Pi)$, $OBS = OBS^- \cup OBS^+$ where OBS^+ is a set of positive assertions and OBS^- is a set of negative assertions such that for all $L \in OBS^-$, $comp(\Pi) \models L$. $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$ iff $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD-OIF(\Pi), COMPS(\Pi), OBS)$.

PROOF. Follows *mutatis mutandis* from proof of Lemma V.18. $\square$

We now present the proofs of Theorems V.11 and V.12.

PROOF. (Theorem V.11)

We first prove by way of contradiction that

(V.7)

$$comp-if_{\Delta}(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS \models \bigwedge_{q \in \Delta} INCORRECT(q).$$

Assume there exists an interpretation I such that

$$I \models comp-if_{\Delta}(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS, \text{ and}$$

$$I \models \bigvee_{q \in \Delta} \neg INCORRECT(q).$$

So for at least one $q' \in \Delta$.

$$\text{comp-if}_{\Delta}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \text{OBS} \cup \{\neg \text{INCORRECT}(q')\}$$

is satisfiable. Or, by the compactness of FOPC,

$$(V.8) \quad \text{comp-if}_{\Delta}(\Pi) \cup \text{OBS} \cup \{\neg \text{INCORRECT}(q')\} \text{ is satisfiable.}$$

Now by (V.1),

$$\text{comp-if}_{\Delta}(\Pi) = \{\neg \text{INCORRECT}(p) \mid p \in \text{COMPS}(\Pi) - \Delta\}.$$

So if we let $\Delta' = \Delta - \{q'\}$ then by (V.8) we know

$$\text{comp-if}_{\Delta'}(\Pi) \cup \text{OBS} \text{ is satisfiable.}$$

Since $\text{comp-if}_{\Delta'}(\Pi) \cup \text{OBS}$ contains no *ab* literals then

$$\text{comp-if}_{\Delta'}(\Pi) \cup \{\mathcal{D}(\Delta', \text{COMPS}(\Pi) - \Delta')\} \cup \text{OBS} \text{ is satisfiable,}$$

and by Lemma V.17 it follows that,

$$\text{SD-IF}(\Pi) \cup \{\mathcal{D}(\Delta', \text{COMPS}(\Pi) - \Delta')\} \cup \text{OBS} \text{ is satisfiable.}$$

So by Lemma V.18, $\mathcal{D}(\Delta', \text{COMPS}(\Pi) - \Delta')$ is a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ where $\Delta' \subset \Delta$. But $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a minimal diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ which is a contradiction. Therefore (V.7) is true.

So by Lemma V.17,

$$\text{SD-IF}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \text{OBS} \models \bigwedge_{q \in \Delta} \text{INCORRECT}(q),$$

and by monotonicity of FOPC,

$$\text{SD-IF}(\Pi) \cup \text{SD-OIF}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \text{OBS} \models \bigwedge_{q \in \Delta} \text{INCORRECT}(q),$$

and since $\text{SD-IF}(\Pi) \cup \text{SD-OIF}(\Pi)$ is semantically equivalent to $\text{SD}(\Pi)$ then

$$\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \text{OBS} \models \bigwedge_{q \in \Delta} \text{INCORRECT}(q).$$

□

PROOF. (Theorem V.12)

Proof follows *mutatis mutandis* from proof of Theorem V.11. □

8. Structural Hierarchies

In our diagnostic framework the system description $SD(\Pi)$ defines the relationship between faults occurring among system components in a structural hierarchy. We are aware of only one other diagnostic framework [37] that proposes definitions for the same purpose. In this framework the relationship between faults occurring in a hierarchy are defined by a set of axioms called a hierarchical theory.

DEFINITION V.20 ([37, Definition 4]). A *hierarchical theory* is a set of *ab*-clauses of the form

$$ab(c_i) \leftrightarrow ab(c_j) \vee \dots \vee ab(c_k),$$

where all literals contained in such an equivalence are unique. The left-hand side contains exactly one literal. The right-hand side contains at least one literal. There are no two equivalences having the same literal on the left-hand side, and there are no two equivalences having the same literal on the right-hand side.

From [37]: "Such a hierarchical theory defines a set of directed trees where $ab(c_i)$ is the direct predecessor of each $ab(c_l) \in \{ab(c_j), \dots, ab(c_k)\}$. Each equivalence uniquely defines all successors of an $ab(c_i)$. The nodes are labelled by these literals."

EXAMPLE V.21. As an example, consider the full-adder circuit shown in Figure V.9. Using the definition of a hierarchical theory we have

$$ab(fullAdder) \leftrightarrow ab(x1) \vee ab(x2) \vee ab(a1) \vee ab(a2) \vee ab(o1).$$

For any diagnoses satisfying the above equivalence, the parent full-adder component is defined to be abnormal iff at least one of its child components is defined to be abnormal. A consequence of defining the hierarchical full-adder this way is that it is not possible for only the full-adder component itself to be faulty, as is the case depicted in Figure V.11. In contrast, the diagnoses defined for the full-adder in Example V.3 admit the possibility of only the parent component failing or only the child components failing.

Another difficulty encountered with the definition of a hierarchical theory is that it does not admit recursive hierarchies, such as those encountered in software systems. This is because the definition of a hierarchical theory insists that all literals contained in an equivalence are unique. On the other hand, our own framework

does not distinguish between recursive and non-recursive hierarchies. For example, the buggy ISORT program (Program III.2) defines a recursive hierarchy which is diagnosed using our framework in Example V.5.

In light of these findings we feel that the notions of faults occurring in hierarchies defined by our framework are more general and intuitive than those defined by a hierarchical theory in [37].

9. Negative Information

In Step 2 of the construction of a system description $SD(\Pi)$, the negative behavior is defined by $comp(\Pi)$, the completion of Π . As noted in Chapter II, $comp(\Pi)$ defines a computable subset of the negative behavior that would be defined with respect to the, in general, non-computable closed world assumption of Π . However, if Π is everywhere terminating then the negative information defined by both approaches are identical. As mentioned in Chapter II, a necessary and sufficient condition for equality of negative information is that $SS(\Pi) \cup FF(\Pi) = \hat{H}(\Pi)$. If, however, $SS(\Pi) \cup FF(\Pi) \neq \hat{H}(\Pi)$ then it is possible to obtain counter-intuitive diagnoses in the framework as illustrated by the following example.

EXAMPLE V.22. For example consider the following program. Let $\Pi = \{p(a) \leftarrow p(a)\}$. The goal $\leftarrow p(a)$ does not succeed nor does it finitely fail in Π . If one were to attempt to use consistency-based diagnosis to diagnose this as a missing behavior error with respect to the artifact system description Π using the assertion $OBS = \{\neg p(a)\}$, then only the minimal diagnosis $\mathcal{D}(\emptyset, \{p\})$ would be defined for $(SD(\Pi), COMPS(\Pi), OBS)$. However, consistency-based diagnosis would define $\mathcal{D}(\emptyset, \{p\})$ as a diagnosis for the assertion $\{p(a)\}$ as well.

10. Discussion

The consistency-based diagnosis framework we have defined is nothing more than an instance of general consistency-based diagnosis for a particular class of system descriptions $SD(\Pi)$ based on a logic program Π . Because the framework is a consistency-based diagnosis framework it inherits many of the results obtained for consistency-based diagnosis such as theories of testing and measurement (e.g. [82, 24, 62]), theories of repair (e.g. [36]), and probabilistic search techniques (e.g. [19, 20, 21, 23]). In addition, system descriptions for this framework are first-order

which offers a number of advantages over strictly propositional models which have been used for consistency-based diagnosis to date. In particular, the advantages of using $SD(\Pi)$ as a system description are:

- the conciseness afforded by a first-order modelling language as compared to a propositional language.
- the expressiveness afforded by a first-order modelling language as compared to a propositional language—in particular, the language of Horn clauses can be used to define any computable function [3] so that, in principle, it is adequate for modelling any software system, including nondeterministically behaving systems [7].
- the ability to model hierarchical systems, including systems with recursive hierarchies, such as the recursively defined *isort* procedure in Program III.2,
- the ability to use Π not only as the basis for $SD(\Pi)$ but also to simulate or query both positive and negative system behavior.
- the automatic definition of the negative behavior of the system by syntactically transforming Π to its completion $comp(\Pi)$.
- the parsimonious characterization of system diagnoses in terms of its minimal diagnoses, and
- the definition of two types of component abnormalities in the absence of fault models in $SD(\Pi)$.

The disadvantages of using a system description $SD(\Pi)$ based on a logic program Π are:

- a theorem prover for computing consistency-based diagnoses will be more complicated and potentially incomplete or undecidable as compared to complete, decidable theorem provers available for propositional system descriptions.
- the negative behavior of the system is defined in terms of negation as failure *i.e.* it is defined in terms of the operational semantics of Π rather than in terms of a more general declarative semantics of Π .
- although two notions of faults are implicitly defined, $SD(\Pi)$ does not admit explicit fault models.
- Π cannot include the negation as failure operator *i.e.* Π cannot be a general logic program.

11. Relationship to Declarative Error Diagnosis

We now clarify the relationship between the consistency-based diagnosis framework we have just developed and the approach to logic program diagnosis known as declarative error diagnosis (see Chapter III).

11.1. Properties of Abnormalities. Although "incorrect" and "incomplete" are names for faults in this framework and in declarative error diagnosis, we have yet to prove any relationship exists between them. We now present two theorems which state that Definitions III.3 and III.4 in Chapter III are, in fact, identical to the notions of faults defined in this framework. We leave the proofs of the theorems to the end of this section. Note that since formulas (V.1) and (V.2) are defined with respect to an actual interpretation, they have the opposite meaning to the terms of the same name defined in Chapter III.

THEOREM V.23. *Let Π be a logic program and let M be an Herbrand interpretation such that $M \models EQ(\Pi)$. A component $p \in COMPS(\Pi)$ is incorrect in Π with respect to M iff $M \models INCORRECT(p)$.*

THEOREM V.24. *Let Π be a logic program and let M be an Herbrand interpretation such that $M \models EQ(\Pi)$. A component $p \in COMPS(\Pi)$ is incomplete in Π with respect to M iff $M \models INCOMPLETE(p)$.*

Thus, Shapiro's notions of errors are implicitly defined by a system description $SD(\Pi)$ based on a logic program Π . As such, the framework presented here effectively forms a bridge between the two previously unrelated approaches to diagnosis. As a consequence, not only can the theory and procedures for consistency-based diagnosis be used for logic program diagnosis, but the theory and procedures developed for declarative error diagnosis can also be applied to the domain of consistency-based diagnosis.

In Chapter III, Theorems III.15 and III.16 relate the existence of errors to the presence of faults in the context of the declarative error diagnosis framework. In the context of our own framework, these theorems are simply instances of Theorems V.12 and V.11, respectively, which relate classes of assertions to the properties of abnormalities in minimal diagnoses. To see why this is the case, note that if a program Π is not partially correct in an intended interpretation M then a negative assertion exists regarding M such that $M(\Pi)$ does not satisfy the

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

assertion. Similarly, if a ground atom $p(\mathbf{g})$ in an intended interpretation $\mathbf{M}$ finitely fails in Π then there exists a positive assertion regarding $\mathbf{M}$, namely $p(\mathbf{g})$, such that $\mathbf{M}(\Pi)$ does not satisfy $p(\mathbf{g})$.

11.2. Granularity of Faults. In the consistency-based diagnosis framework, the modelling ontology used in Step 1 of the construction of $\text{SD}(\Pi)$ dictates that predicates defined by the logic program Π correspond to potentially incorrect or incomplete system components. This contrasts with the notions of errors usually defined for declarative error diagnosis, namely *uncovered atoms* and *incorrect clause instances* (see Chapter III). The finer grained notions of errors used for declarative error diagnosis are intuitively meaningful in the context of logic program diagnosis but they lose their meaning using our framework's ontology. However, by exploiting the fact that these finer grained notions of errors imply incorrectness and incompleteness (Theorems III.11 and III.12) we can use the consistency-based diagnosis framework to generate diagnoses in terms of uncovered atoms and incorrect clause instances.

All that is required is to add a step between Steps 1 and 2 of the construction of a system description $\text{SD}(\Pi)$. This added step transforms the logic program Π to a logic program Π' in which each clause of Π defines a unique predicate in Π' so that each clause in Π is identified with a unique component in the consistency-based diagnosis framework's ontology. Formally:

$$\begin{aligned} \text{Let } \Pi'' = \{ (p_n(\mathbf{t}) \leftarrow F) \mid (p(\mathbf{t}) \leftarrow F) \in \Pi, \\ p_n \text{ is a unique predicate name in } \Pi'', \\ \text{and for all } k \neq n, (p_k(\mathbf{t}) \leftarrow F) \notin \Pi'' \}. \end{aligned}$$

$$\text{Let } \Pi' = \Pi'' \cup \{ (p(\mathbf{X}) \leftarrow p_n(\mathbf{X})) \mid p_n \text{ is a predicate name in } \Pi'' \}.$$

Using this simple syntactic transformation, Π' would be used as the logic program for Step 3. However, the set of potentially faulty components COMPS should be only those predicates defined in Π'' . Therefore using $\text{SD}(\Pi')$ as a system description and COMPS as the set of system components, we can, in effect, consider each *clause* in Π as a potentially faulty component using our consistency-based diagnosis framework.

PROGRAM V.26.

$$\begin{aligned}
&isort(X, Y) \leftarrow isort_1(X, Y). \\
&isort(X, Y) \leftarrow isort_2(X, Y). \\
&isort_1([X_h|X_t], Y) \leftarrow isort(X_t, Z), insert(X_h, Z, Y). \\
&isort_2([], []). \\
\\
&insert(X, Y, Z) \leftarrow insert_1(X, Y, Z). \\
&insert(X, Y, Z) \leftarrow insert_2(X, Y, Z). \\
&insert(X, Y, Z) \leftarrow insert_3(X, Y, Z). \\
&insert_1(X, [Y_h|Y_t], [Y_h|Z]) \leftarrow Y_h > X, insert(X, Y_t, Z). \\
&insert_2(X, [Y_h|Y_t], [X, Y_h|Y_t]) \leftarrow X \leq Y_h. \\
&insert_3(X, [], [X]).
\end{aligned}$$

EXAMPLE V.25. As an example, let Π' be Program V.26, where Program V.26 is the buggy ISORT Program III.2 transformed in the aforementioned fashion. Taking COMPS to be the set $\{isort_1, isort_2, insert_1, insert_2, insert_3\}$, and using the same set of assertions OBS used in Example V.5 we find there exists only one minimal diagnosis for $(SD(\Pi'), COMPS, OBS)$ such that $\Delta = \{insert_1\}$ where $insert_1$ must be incorrect by Theorem V.12. This is identical to the diagnosis obtained using declarative error diagnosis in Example III.18 which finds the first *insert* clause of Program III.2 to have an incorrect clause instance.

11.3. Expressiveness of Π . In the consistency-based diagnosis framework a system description $SD(\Pi)$ is based on a definite logic program Π . On the other hand, declarative error diagnosis defines its faults and errors in terms of general logic programs [56] *i.e.* logic programs whose clause bodies may include negative literals. Although the languages of definite logic programs and general logic programs are equally expressive, the latter language has the potential for more concise modelling.

11.4. Negative Information. Declarative error diagnosis suffers from the same problems with respect to negative information as the consistency-based diag-

nosis framework does. To see how this is the case consider the following example.

EXAMPLE V.27. Let $H = p(a) \leftarrow p(a)$. Clearly, H is non-terminating for the goal $\leftarrow p(a)$, so we can take $p(a)$ to be missing behavior. Now, one would expect there to be a fault in H associated with this error but if one examines the definitions for incorrect clause instance and uncovered atom (Definitions III.9 and III.10), we see that neither applies. Therefore, these two notions of faults, and the associated notions of incorrect and incomplete procedures, inadequately capture our intuitive notions of faults that can occur in a logic program.

One solution to this problem is proposed by Ferrand in [34] in which faults and errors for declarative error diagnosis are defined independent of the operational semantics of a logic program.

11.5. Intended and Actual Interpretations. In order to apply consistency-based diagnosis to the tasks of both hardware and software diagnosis we have clarified and exploited a duality that exists between the notions of intended and actual interpretations of systems. Establishing the relationship between the consistency-based diagnosis framework and declarative error diagnosis means that this same duality can be exploited for the declarative error diagnosis of both hardware and software. In [53], this duality has been correctly exploited for the declarative error diagnosis of digital hardware. However, in [43] the existence of a duality goes unrecognized and consequently, we have found that the proposed approach for declarative error diagnosis of digital hardware is incorrect in general.

11.6. Knowledge of an Interpretation. Faults in declarative error diagnosis are defined with respect to a single actual or intended interpretation. In contrast, diagnoses in consistency-based diagnosis are defined with respect to a set of assertions which may be true in numerous possible interpretations. As such, it is usually the case that more knowledge of an interpretation is required to identify a fault for declarative error diagnosis than is required for consistency-based diagnosis. For example, in Example V.3 four minimal diagnoses are defined using the consistency-based diagnosis framework for the single assertion *fullAdder*(1,0,1,1,0). Here, the assertion *fullAdder*(1,0,1,1,0) is the only information available regarding the actual interpretation of the artifact. However, using declarative error diagnosis it is

not possible to identify any faults with just this one assertion. In order to identify a faulty component, information is also required regarding the behavior of the full-adder's child components with respect to the actual interpretation. Thus, the consistency-based diagnosis framework is preferable to the declarative error diagnosis framework when there only exists partial knowledge of an intended or actual interpretation.

When using declarative error diagnosis for hardware diagnosis, an intended interpretation must reflect the behavior of the faulty artifact. As such, in a worst-case scenario, complete knowledge of the actual interpretation for the artifact must be available in order to identify faults *i.e.* the hardware artifact must be completely observable. One approach to declarative error diagnosis for hardware diagnosis [53] takes into account limited observability but diagnoses obtained are less refined than those obtained using the consistency-based diagnosis approach. For example, in contrast to the four minimal diagnoses obtained in Example V.3, the approach taken in [53] merely identifies the full-adder and all of its sub-components as a faulty module for the single assertion *fullAdder*(1,0,1,1,0)

11.7. Computing Diagnoses. As discussed in Chapter III, computing diagnoses for declarative error diagnosis is an interactive process in which an oracle (normally the user) is queried regarding the truth of literals in a given interpretation. Querying an oracle is necessary for reasons we have just discussed—faults in declarative error diagnosis are defined with respect to just one actual or intended interpretation. In contrast, consistency-based diagnoses are defined with respect to possible interpretations. As a consequence, diagnosis computation for consistency-based diagnosis is autonomous.

Since an oracle is necessary for most declarative error diagnosis computation procedures, these are best used in diagnostic domains for which exists complete knowledge of the interpretation, *i.e.* complete observability of an artifact or complete knowledge of specified behavior. In this context declarative error diagnosis procedures are very efficient at locating one fault at a time. However, when knowledge of an interpretation is incomplete then a consistency-based diagnosis procedure is better suited. Procedures for computing consistency-based diagnoses for propositional system descriptions are discussed in Chapter IV. The next two chapters will discuss procedures for the class of system descriptions $SD(\Pi)$.

11.8. Proofs. We now present the proofs of Theorems V.23 and V.24. Again, take note that formulas (V.1) and (V.2) are defined with respect to an actual interpretation and for this reason we reverse the meanings of Definitions III.3 and III.4 in the following proofs.

PROOF. (Theorem V.23)

($\Rightarrow$)

We are given that there exists an atom $p(\mathbf{s}^n) \in L(\Pi)$ such that for all clauses $p(\mathbf{t}^n) \leftarrow F \in \Pi$ and all substitutions η

$$(V.9) \quad (p(\mathbf{s}^n)\eta \not\equiv p(\mathbf{t}^n)\eta \text{ or } M \models \forall \neg F\eta) \text{ and } M \models \forall p(\mathbf{s}^n).$$

If $p(\mathbf{s}^n)\eta \not\equiv p(\mathbf{t}^n)\eta$ then by Lemma II.27(b),

$$EQ(\Pi) \models \neg(s_1 = t_1 \wedge \cdots \wedge s_n = t_n)\eta,$$

and since $M \models EQ(\Pi)$ then $p(\mathbf{s}^n)\eta \not\equiv p(\mathbf{t}^n)\eta$ implies

$$M \models \neg(s_1 = t_1 \wedge \cdots \wedge s_n = t_n)\eta.$$

Therefore from (V.9), for all clauses $p(\mathbf{t}^n) \leftarrow F \in \Pi$ and all substitutions η

$$\begin{aligned} M &\models (s_1 \neq t_1 \vee \cdots \vee s_n \neq t_n \vee \neg F)\eta, \text{ or} \\ M &\models (\neg(s_1 = t_1 \wedge \cdots \wedge s_n = t_n \wedge F))\eta, \text{ or} \\ M &\models \forall \mathbf{Z}^l \forall \mathbf{Y}^m \neg(s_1 = t_1 \wedge \cdots \wedge s_n = t_n \wedge F), \\ &\text{where } \{Y_1, \dots, Y_m\} = \text{vars}(p(\mathbf{t}^n) \leftarrow F), \text{ and} \\ &\{Z_1, \dots, Z_l\} = \text{vars}(\mathbf{s}^n). \end{aligned}$$

It follows that for all clauses $p(\mathbf{t}^n) \leftarrow F \in \Pi$,

$$M \models \forall \mathbf{Y}^m \neg(s_1 = t_1 \wedge \cdots \wedge s_n = t_n \wedge F).$$

so from (V.9) we have

$$\begin{aligned} M &\models \forall \mathbf{Y}^m \neg(X_1 = t_1 \wedge \cdots \wedge X_n = t_n \wedge F)\theta \text{ and } M \models p(\mathbf{X}^n)\theta, \\ &\text{where } \theta = \{X_1/s_1, \dots, X_n/s_n\}. \end{aligned}$$

Therefore,

$$\begin{aligned} M \models & (p(X^n) \wedge (\neg \exists Y^{m_1}(X^n = t^{n_1} \wedge F_1)) \wedge \dots \\ & \wedge (\neg \exists Y^{m_{\mu(p)}}(X^n = t^{n_{\mu(p)}} \wedge F_{\mu(p)})))\theta, \text{ or} \\ M \models & \exists X^n(p(X^n) \wedge (\neg \exists Y^{m_1}(X^n = t^{n_1} \wedge F_1)) \wedge \dots \\ & \wedge (\neg \exists Y^{m_{\mu(p)}}(X^n = t^{n_{\mu(p)}} \wedge F_{\mu(p)}))), \text{ or} \\ M \models & \text{INCORRECT}(p). \end{aligned}$$

($\Leftarrow$)

Since $M \models \text{INCORRECT}(p)$ then

$$\begin{aligned} M \models & \exists X^n(p(X^n) \wedge (\neg \exists Y^{m_1}(X^n = t^{n_1} \wedge F_1)) \wedge \dots \\ & \wedge (\neg \exists Y^{m_{\mu(p)}}(X^n = t^{n_{\mu(p)}} \wedge F_{\mu(p)}))), \text{ or} \\ M \models & (p(X^n) \wedge (\neg \exists Y^{m_1}(X^n = t^{n_1} \wedge F_1)) \wedge \dots \\ & \wedge (\neg \exists Y^{m_{\mu(p)}}(X^n = t^{n_{\mu(p)}} \wedge F_{\mu(p)})))\theta, \text{ where} \\ \theta = & \{X_1/\sigma_1, \dots, X_n/\sigma_n\}, \sigma_i \notin L(\Pi). \end{aligned}$$

Therefore for all clauses $p(t^n) \leftarrow F \in \Pi$,

$$(V.10) \quad M \models p(X^n)\theta \text{ and } M \models \forall Y^m(t^n \neq X^n \vee \neg F)\theta.$$

Now by (V.1), for all clauses $p(t^n) \leftarrow F \in \Pi$, $\text{vars}(p(t^n) \leftarrow F) \cap \text{dom}(\theta) = \emptyset$.

Therefore from (V.10) we have for all clauses $p(t^n) \leftarrow F \in \Pi$,

$$\begin{aligned} M \models & \forall Y^m(t_1 \neq \sigma_1 \vee \dots \vee t_n \neq \sigma_n \vee \neg F), \text{ or} \\ M \models & \forall Y^m(\neg(t_1 = \sigma_1 \wedge \dots \wedge t_n = \sigma_n) \vee \neg F). \end{aligned}$$

So for all clauses $p(t^n) \leftarrow F \in \Pi$ and for all substitutions η ,

$$(V.11) \quad M \models \neg(t_1 = \sigma_1 \wedge \dots \wedge t_n = \sigma_n)\eta \vee \neg F\eta.$$

Now, for all models M of $\text{EQ}(\Pi)$,

$$M \models \neg(t_1 = \sigma_1 \wedge \dots \wedge t_n = \sigma_n)\eta \text{ implies } p(\sigma_1, \dots, \sigma_n)\eta \neq p(t_1, \dots, t_n)\eta$$

It therefore follows from (V.10) and (V.11) that since $M \models \text{EQ}(\Pi)$ then for all clauses $p(t^n) \leftarrow F \in \Pi$ and for all substitutions η ,

$$(p(\sigma_1, \dots, \sigma_n)\eta \neq p(t_1, \dots, t_n)\eta \text{ or } M \models \neg F\eta) \text{ and } M \models p(\sigma_1, \dots, \sigma_n).$$

Thus, there exists an atom $p(s^n) \in L(\Pi)$ such that for all clauses $p(t^n) \leftarrow F \in \Pi$ and for all substitutions η ,

$$(p(s^n)\eta \neq p(t^n)\eta \text{ or } M \models \forall \neg F\eta) \text{ and } M \models \forall p(s^n).$$

□

PROOF. (Theorem V.24)

($\Rightarrow$)

Let $X_1, \dots, X_n$ be variables such that $\{X_1, \dots, X_n\} \cap \text{vars}(p(t^n) \leftarrow F) = \emptyset$. Now, we are given that there exists a substitution η such that

$$\begin{aligned} & M \models \forall \neg p(t^n)\eta, \text{ or} \\ & M \models \neg p(t^n)\eta, \text{ or} \\ (V.12) \quad & M \models \neg p(X^n)\theta\eta, \\ & \text{where } \theta = \{X_1/t_1, \dots, X_n/t_n\}. \end{aligned}$$

Since

$$EQ(\Pi) \models (t_1 = t_1 \wedge \dots \wedge t_n = t_n)\eta,$$

then

$$EQ(\Pi) \models (X_1 = t_1 \wedge \dots \wedge X_n = t_n)\theta\eta,$$

and since $M \models EQ(\Pi)$ then

$$(V.13) \quad M \models (X_1 = t_1 \wedge \dots \wedge X_n = t_n)\theta\eta.$$

Now, since we are given $M \models \forall F\eta$ and since $\text{vars}(F) \cap \text{dom}(\theta) = \emptyset$ then

$$(V.14) \quad M \models F\theta\eta.$$

So by (V.12), (V.13), and (V.14) we have

$$M \models (\neg p(X^n) \wedge X^n = t^n \wedge F)\theta\eta.$$

If we let $\text{vars}(p(t^n) \leftarrow F) = \{Y_1, \dots, Y_m\}$, then

$$M \models \exists X^n \exists Y^m (\neg p(X^n) \wedge X^n = t^n \wedge F),$$

and since $\{X_1, \dots, X_n\} \cap \text{vars}(p(t^n) \leftarrow F) = \emptyset$ then

$$\begin{aligned} M \models \exists X^n (\neg p(X^n) \wedge \exists Y^m (X^n = t^n \wedge F)), \text{ so} \\ M \models \exists X^n (\neg p(X^n) \wedge (\exists Y^{m_1} (X^n = t^{n_1} \wedge F_1) \vee \dots \\ \vee \exists Y^{m_{\mu(p)}} (X^n = t^{n_{\mu(p)}} \wedge F_{\mu(p)}))), \text{ so} \\ M \models \text{INCOMPLETE}(p). \end{aligned}$$

($\Leftarrow$)

We are given

$$\begin{aligned} M \models \text{INCOMPLETE}(p), \text{ so} \\ M \models \exists X^n (\neg p(X^n) \wedge (\exists Y^{m_1} (X^n = t^{n_1} \wedge F_1) \vee \dots \\ \vee \exists Y^{m_{\mu(p)}} (X^n = t^{n_{\mu(p)}} \wedge F_{\mu(p)}))), \end{aligned}$$

and by (V.2) we know that X and Y variables are distinct so

$$\begin{aligned} M \models \exists X^n \exists Y^{m_1} \dots \exists Y^{m_{\mu(p)}} (\neg p(X^n) \wedge ((X^n = t^{n_1} \wedge F_1) \vee \dots \\ \vee (X^n = t^{n_{\mu(p)}} \wedge F_{\mu(p)}))). \end{aligned}$$

Therefore for some clause $p(t^n) \leftarrow F \in \Pi$,

$$\begin{aligned} M \models \exists X^n \exists Y^m (\neg p(X^n) \wedge X^n = t^n \wedge F), \text{ or} \\ M \models (\neg p(X^n) \wedge X^n = t^n \wedge F)\theta\eta, \\ \text{where } \theta = \{X_1/\sigma_1, \dots, X_n/\sigma_n\}, \sigma_i \notin L(\Pi), \text{ and} \\ \eta = \{Y_1/\delta_1, \dots, Y_m/\delta_m\}, \delta_i \notin L(\Pi). \end{aligned}$$

Since $\{X_1, \dots, X_n\} \cap \text{dom}(\eta) = \emptyset$ and $\text{vars}(F) \cap \text{dom}(\theta) = \emptyset$ then

$$\begin{aligned} M \models \neg p(X^n)\theta \wedge F\eta \wedge (X^n = t^n)\theta\eta, \text{ or} \\ \text{(V.15)} \quad M \models \neg p(\sigma_1, \dots, \sigma_n) \wedge F\eta \wedge t_1\eta = \sigma_1 \wedge \dots \wedge t_n\eta = \sigma_n. \end{aligned}$$

Now since $M \models \text{EQ}(\Pi)$ then by Theorem II.22

$$M \models (X_1 = Y_1 \wedge \dots \wedge X_n = Y_n) \rightarrow (\neg p(Y^n) \rightarrow \neg p(X^n)).$$

So from (V.15) it follows that

$$\begin{aligned}
 & M \models \neg p(t_1\eta, \dots, t_n\eta), \text{ or} \\
 & M \models \neg p(t^n)\eta, \text{ or} \\
 (V.16) \quad & M \models \forall \neg p(t^n)\eta.
 \end{aligned}$$

Thus, from (V.15) and (V.16) it follows that there exists a clause $p(t^n) \leftarrow F \in \Pi$ and a substitution η such that

$$M \models \forall \neg p(t^n)\eta \text{ and } M \models \forall F\eta.$$

□

12. Relationship to Model-Based Diagnosis of Logic Programs

In [14], Console, Friedrich and Dupré present a framework for the diagnosis of definite logic programs and a procedure for computing the diagnoses defined by the framework. The framework is only loosely based on consistency-based diagnosis: rather than identifying abnormal predicates, diagnoses in their framework suggest which, if any, clauses should be removed from the program, and which, if any, atoms to add to the program, in order to correct program behavior with respect to a set of assertions. In addition, they define a preference order on the candidate diagnoses which reflects the relative ease of program correction associated with each diagnosis. Recall that faults in our consistency-based diagnosis framework indicate that a procedure is incomplete or incorrect, or that there exists an incorrect clause instance or an uncovered atom. While these notions of faults provide some information regarding the correction of components, i.e. incomplete components are missing behavior and incorrect components exhibit spurious behavior, they don't provide as explicit information as provided by the framework in [14].

Console *et al.* informally show that the space of diagnoses defined by their framework includes those faults defined for declarative error diagnosis. As such, their framework shares much in common with the consistency-based diagnosis framework presented here. However, it is important to note that because the framework proposed by Console *et al.* is not an instance of consistency-based diagnosis then it is potentially difficult to transfer results from the domain of consistency-based diagnosis to their framework. This is not an issue with the consistency-based diagnosis framework presented here. Furthermore, we will show

in a future chapter that the diagnosis computation procedure proposed for their framework has worse diagnostic coverage than a comparable procedure we develop for our own consistency-based diagnosis framework.

13. Relationship to IDA

In [65] a generate-and-test approach to diagnosis computation is proposed using a procedure called IDA ("Incremental Diagnosis Algorithm") (see Chapter IV). The framework uses a logic program as a description of a system's structure and behavior but, unlike the consistency-based diagnosis framework presented here, the IDA approach to diagnosis is presented in terms of a model-based diagnostic framework only informally related to consistency-based diagnosis.

The significance of the IDA approach is two-fold. First the approach clearly illustrates the effectiveness of logic program based-system descriptions for diagnosis. Examples of system descriptions used are a qualitative model of the human heart, a combinational 500-bit ripple carry adder with 2500 components [65] and an active analog filter designed-for-testability [67].¹ Second, IDA shows the utility of a generate-and-test approach to diagnosis when using logic program-based system descriptions. Both of these characteristics are shared with the consistency-based diagnosis framework presented here.

One characteristic not shared with our consistency-based diagnosis framework is that the IDA framework does not directly support the modelling and diagnosis of hierarchical systems. For this reason the IDA framework is not suitable for software diagnosis. On the other hand, the IDA framework supports the inclusion of fault models with system descriptions.

In the next chapter we show how the IDA framework, excluding fault models, is actually an instance of the consistency-based diagnosis framework presented here. As a result, the IDA framework inherits results associated with our consistency-based diagnosis framework and with consistency-based diagnosis in general.

¹The system descriptions mentioned here are all modelled using a constraint logic programming language rather than pure Horn clause logic. However, since the constraint solving mechanism for such a constraint language is transparent then the language is essentially equivalent to the language of Horn clause logic i.e. the system descriptions are essentially logic programs.

14. Summary

We have presented a formal consistency-based diagnosis framework that uses system descriptions based on logic programs. The language associated with this class of system descriptions is shown to be expressive enough for modelling and diagnosing software systems and structural hierarchies. Since the framework is suited to diagnosis from specifications or diagnosis in the absence of specifications we have clarified the meanings of assertions in the two diagnostic contexts. Diagnoses of the framework are characterized in terms of their minimal diagnoses. Necessary and sufficient conditions for component abnormality have been identified. These conditions, named incorrectness and incompleteness, indicate the nature of a fault.

We have shown how declarative error diagnosis is an instance of the consistency-based diagnosis framework with the consequence that the bodies of results for each approach are applicable to one another. We have also compared the framework to two other comparable diagnostic frameworks and found them wanting in a number of respects.

CHAPTER VI

Computing Diagnoses With Labelled System Descriptions

1. Introduction

In the previous chapter we defined a consistency-based diagnosis framework for a class of systems of the form $(SD(\Pi), COMPS(\Pi), OBS)$ where $SD(\Pi)$ is a system description based on a logic program Π . In this chapter we develop procedures to compute diagnoses for this class of systems where OBS is either a set of positive assertions or a set of negative assertions (see Chapter V). Why—one might ask—do we need more diagnosis computation procedures when we established in the last chapter that the procedures developed for declarative error diagnosis can be used for systems of the form $(SD(\Pi), COMPS(\Pi), OBS)$? The reason is that all such procedures require an oracle to provide knowledge regarding the state of interpretations. As such, these procedures are efficient for generating diagnoses one fault at a time only when complete knowledge of interpretation is available. However, when incomplete knowledge is available *i.e.* when an artifact is not completely observable or when the specified behavior of an artifact is not completely known then alternative procedures are required. In this chapter we present just this type of procedure.

We use a generate-and-test approach to diagnosis computation similar to that used in the IDA framework [65] (see Chapter IV) *i.e.* a procedure generates sentences of the form $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ and a diagnosis verification function TP ("theorem prover" [82]) verifies whether the sentence is a diagnosis or not. In this chapter we define two diagnosis verification functions, TP^+ and TP^- , for verification with respect to sets of positive and negative assertions, respectively. We

show how TP^+ is a generalization of the TP function used in the IDA framework. Establishing this relationship shows how the IDA framework is a specialized instance of consistency-based diagnosis—a conclusion whose only basis up until now was intuition.

The theory underlying the TP^+ and TP^- functions is defined in terms of labelled system descriptions—logic programs that are nothing more than syntactic transformations of the logic program Π upon which a system description $SD(\Pi)$ is based. With regards to this theory, we identify conditions for which diagnosis verification is sound, complete and decidable. We then describe how TP^+ and TP^- are implemented and prove the implementation correct. The implementation proves to be efficient as all that is required is to check whether or not an assertion succeeds or finitely fails in a labelled system description. In the course of this presentation we use the familiar examples of the hierarchically structured full-adder circuit (see Chapters IV and V) and the buggy insertion sort procedure (see Chapters III and V).

2. IDA

Initially we present a brief overview of the TP function implementation used by the IDA framework [65] (see Chapter III). The approach is straightforward and understanding it will help in understanding the more general approach we present later.

EXAMPLE VI.1. Program VI.2 is an example of a system description of the full-adder circuit specification (see Figure V.9) for the IDA framework. In contrast to Program V.1, the program used as the basis for a specification system description in the previous chapter, the head of each clause here has an additional argument. For example, the clause

$$x1(ok, X, Y, Z) \leftarrow xor(X, Y, Z)$$

defines the *normal* behavior of component $x1$, as indicated by the *ok* term in the clause head. Therefore, the goal $\leftarrow x1(ok, 1, 0, 1)$ would succeed since this represents normal behavior for $x1$. However, the goal $\leftarrow x1(ok, 1, 0, 0)$ would finitely fail since this represents abnormal behavior which a normally behaving

component cannot exhibit. On the other hand the clause

$$x1(ab, -, -, -)$$

defines the *abnormal* behavior of the component $x1$. In this case, both $\leftarrow x1(ab, 1, 0, 1)$ and $\leftarrow x1(ab, 1, 0, 0)$ would succeed because abnormal behavior for $x1$ is completely unconstrained. Thus, these two clauses for $x1$ can be used to verify whether or not an assertion regarding the behavior of $x1$ is consistent with the behavior defined for the component for a particular fault state: normal or abnormal. Recall from Chapter IV that this is precisely what one wants to verify using a diagnosis verification function TP.

The *fullAdder* clause simply specifies the interconnections between the components $x1$, $x2$, $a1$, $a2$, and $o1$. Note that there are no clauses here that explicitly state the *ok* or *ab* states for the *fullAdder* itself. As such the *fullAdder* cannot be considered as a potentially faulty system component using this system description i.e. in this example $COMPS = \{x1, x2, a1, a2, o1\}$.

A system description in the IDA framework must include a distinguished predicate like the *fullAdder* predicate in Program VI.2 which represents the relation between a possible fault state of the system components (normal or abnormal) and its behavior. Any assertions made regarding a system must be in terms of such a predicate. In general, assertions in the IDA framework are of the form $p(L, t)$ where L is a label that indicates a possible fault state for each system component and t is a tuple that indicates the port values of system components.

EXAMPLE VI.3. For example, assume that a full-adder artifact exhibits the erroneous behavior $K = 1$, $L = 0$, $M = 1$, $R = 1$, and $S = 0$. The assertion that is used to determine whether or not $\mathcal{D}(\{x1\}, \{x2, a1, a2, o1\})$ is a diagnosis would be $\leftarrow fullAdder([ab, ok, ok, ok, ok], 1, 0, 1, 1, 0)$.

In the IDA framework, a diagnosis for an observation is defined to be a (not necessarily ground) label L such that $SD \models \forall p(L, t)$, where SD is a system description such as Program VI.2. Therefore to verify whether or not a sentence $\mathcal{D}(\Delta, COMPS - \Delta)$ is a diagnosis with respect to a positive assertion one simply checks whether or not the associated assertion succeeds or finitely fails, respectively, in the system description.

PROGRAM VI.2.

$$\begin{aligned} \text{fullAdder}([X1, X2, A1, A2, O1], K, L, M, R, S) \leftarrow & x1(X1, K, L, N), \\ & a1(A1, K, L, Q), \\ & a2(A2, M, N, P), \\ & x2(X2, N, M, R), \\ & o1(O1, P, Q, S). \end{aligned}$$

$$\begin{aligned} x1(ok, X, Y, Z) &\leftarrow xor(X, Y, Z). \\ x1(ab, -, -, -). \\ x2(ok, X, Y, Z) &\leftarrow xor(X, Y, Z). \\ x2(ab, -, -, -). \\ a1(ok, X, Y, Z) &\leftarrow and(X, Y, Z). \\ a1(ab, -, -, -). \\ a2(ok, X, Y, Z) &\leftarrow and(X, Y, Z). \\ a2(ab, -, -, -). \\ o1(ok, X, Y, Z) &\leftarrow or(X, Y, Z). \\ o1(ab, -, -, -). \\ and(X, 0, 0). \quad and(0, X, 0). \quad and(1, 1, 1). \\ or(1, X, 1). \quad or(X, 1, 1). \quad or(0, 0, 0). \\ xor(0, 0, 0). \quad xor(0, 1, 1). \quad xor(1, 0, 1). \quad xor(1, 1, 0). \end{aligned}$$

EXAMPLE VI.4. For example, since the goal

$$\leftarrow \text{fullAdder}([ab, ok, ok, ok, ok], 1, 0, 1, 1, 0)$$

succeeds in Program VI.2 then $\mathcal{D}(\{x1\}, \{x2, a1, a2, o1\})$ is a diagnosis. Note that this agrees with one of the minimal diagnoses defined by consistency-based diagnosis in Example V.3.

On an intuitive level, this approach to diagnosis verification seems sound with respect to the definition of a consistency-based diagnosis. However, it is important to note that no claim is made in [65] that the IDA framework computes consistency-based diagnoses. The IDA framework is presented only as a model-based diagnostic

framework.

In addition to the possible fault states *ok* representing normal component behavior and *ab* representing unconstrained abnormal behavior, Mozetič's framework permits fault models (constrained abnormal behavior—see Chapter IV) to be defined for a component in a manner similar to that accomplished by means of fault models in traditional consistency-based diagnosis

EXAMPLE VI.5. The component *x1* might have the fault models: output stuck-at-1 (*s1*) or output stuck-at-0 (*s0*). This would be represented by adding the following clauses to Program VI.2.

$$x1(s1, -, -, 1). \quad x1(s0, -, -, 0).$$

In summary, although the approach to diagnosis verification used by IDA accepts system descriptions based on logic programs, the approach does not accept system descriptions of hierarchically structured systems. Since most software systems consist of hierarchies of functions, procedures, modules, *etc.*, then diagnosis of all but the simplest monolithic software programs is impossible using this approach. In addition, the framework defines diagnoses only with respect to positive assertions. Lastly, diagnoses defined by the framework are not formally defined to be consistency-based diagnoses.

In the remainder of this chapter we present an approach to diagnosis verification for the consistency-based diagnosis framework presented in the previous chapter. The approach, which is a generalization of that used by the IDA framework, permits verifying diagnoses for hierarchically structured systems and for either positive or negative assertions. One not entirely surprising result related to the development of our approach is that we show diagnoses defined by the IDA framework for system descriptions that do not use fault models are also consistency-based diagnoses.

3. Labelled System Descriptions

The diagnosis verification approach we develop for systems of the form $(SD(\Pi), COMPS(\Pi), OBS)$ is based on nothing more than SLD-resolution combined with the negation-as-failure (NAF) rule for logic programs (see Chapter II). The key to the approach is the underlying representation of the system description $SD(\Pi)$

based on a logic program Π . We utilize two forms of representation, called labelled system descriptions (LSD's), which are simply syntactic transformations of Π . A positive labelled system description, denoted $\text{LSD}^+(\Pi)$ is used to verify whether or not a sentence $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a diagnosis for a system $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ where OBS is a set of positive assertions. A negative labelled system description, denoted $\text{LSD}^-(\Pi)$ is used to verify whether or not a sentence $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a diagnosis for a system $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ where OBS is a set of negative assertions.

3.1. Examples. $\text{LSD}^-(\Pi)$ is formed from Π by adding a label argument to each atom occurring in each clause of Π . If $|\text{COMPS}(\Pi)| = m$ then the label is an m -element list structure $[x_1, \dots, x_m]$, each element x_i , $1 \leq i \leq m$, being a variable or one of the constants *ok* or *ab*. If we impose an arbitrary total order on the elements of $\text{COMPS}(\Pi)$ then the meaning of a list element is interpreted as follows:

- $x_i = \text{ok}$ signifies the i^{th} element of $\text{COMPS}(\Pi)$ is normal;
- $x_i = \text{ab}$ signifies the i^{th} element of $\text{COMPS}(\Pi)$ is abnormal;
- x_i is a variable signifies the i^{th} element of $\text{COMPS}(\Pi)$ is either normal or abnormal;

EXAMPLE VI.6. As an example, Program VI.7 is $\text{LSD}^-(\Pi)$ for the the full-adder Program V.1. Here $\text{COMPS}(\Pi) = \{\text{fullAdder}, x1, x2, a1, a2, o1\}$. We treat the predicates *xor*, *and*, and *or* as fault-free. For the purposes of labelling we have taken the ordering on $\text{COMPS}(\Pi)$ to be the order it is written above.

The elements of the labels of $\text{LSD}^-(\Pi)$ are either variables or the constant *ok*. As such the labels of $\text{LSD}^-(\Pi)$ serve to explicitly indicate the normal structure and behavior of a program Π . No clauses exist with labels defining the abnormal behavior of components so that the abnormal behavior of Π is implicitly defined by the NAF rule. In effect, $\text{LSD}^-(\Pi)$ defines the abnormal behavior of components to be entirely negative.

EXAMPLE VI.8. Since the goal

$$\leftarrow x1([-, ab, -, -, -, -], X, Y, Z)$$

PROGRAM VI.7 (LSD⁻(Π) FOR THE FULL-ADDER SPECIFICATION).

$$\begin{aligned}
 fullAdder([ok, \neg, \neg, \neg, \neg, \neg], K, L, M, R, S) &\leftarrow x1([ok, \neg, \neg, \neg, \neg, \neg], K, L, N), \\
 &\quad a1([ok, \neg, \neg, \neg, \neg, \neg], K, L, Q), \\
 &\quad a2([ok, \neg, \neg, \neg, \neg, \neg], M, N, P), \\
 &\quad x2([ok, \neg, \neg, \neg, \neg, \neg], N, M, R), \\
 &\quad o1([ok, \neg, \neg, \neg, \neg, \neg], P, Q, S). \\
 x1([\neg, ok, \neg, \neg, \neg, \neg], X, Y, Z) &\leftarrow xor(X, Y, Z). \\
 x2([\neg, \neg, ok, \neg, \neg, \neg], X, Y, Z) &\leftarrow xor(X, Y, Z). \\
 a1([\neg, \neg, \neg, ok, \neg, \neg], X, Y, Z) &\leftarrow and(X, Y, Z). \\
 a2([\neg, \neg, \neg, \neg, ok, \neg], X, Y, Z) &\leftarrow and(X, Y, Z). \\
 o1([\neg, \neg, \neg, \neg, \neg, ok], X, Y, Z) &\leftarrow or(X, Y, Z). \\
 and(X, 0, 0). \quad and(0, X, 0). \quad and(1, 1, 1). \\
 or(1, X, 1). \quad or(X, 1, 1). \quad or(0, 0, 0). \\
 xor(0, 0, 0). \quad xor(0, 1, 1). \quad xor(1, 0, 1). \quad xor(1, 1, 0).
 \end{aligned}$$

finitely fails in Program VI.7 then we can conclude from the soundness of the NAF rule that

$$comp(LSD^-(\Pi)) \models \forall \neg x1([\neg, ab, \neg, \neg, \neg, \neg], X, Y, Z).$$

This means that all combinations of port values, X , Y and Z reflect negative behavior of $x1$ when $x1$ is assumed to be behaving abnormally.

Since LSD⁻(Π) is only used to verify negative assertions about components then it is necessary only to confirm that a negative assertion finitely fails in LSD⁻(Π) to confirm that it is possible negative behavior.

EXAMPLE VI.9. For example, since the goal

$$\leftarrow x1([\neg, ok, \neg, \neg, \neg, \neg], 1, 0, 0)$$

finitely fails in Program VI.7, then it is normal negative behavior of $x1$. Similarly, since the goal

$$\leftarrow x1([\neg, ab, \neg, \neg, \neg, \neg], 1, 0, 0)$$

finitely fails in Program VI.7, then it is also a possible abnormal negative behavior of $x1$. On the other hand, since the goal

$$\leftarrow x1([-, ok, -, -, -, -], 1, 0, 1)$$

succeeds in Program VI.7, then it is not normal negative behavior of $x1$.

In contrast to a negative labelled system description, the labels of a positive labelled system description $LSD^+(\Pi)$ explicitly indicate both the normal behavior of a program Π and its abnormal behavior. In effect, $LSD^+(\Pi)$ defines the abnormal behavior of components to be entirely positive. Since $LSD^+(\Pi)$ is only used to verify positive assertions about components, then it is necessary only to confirm that a positive assertion succeeds in $LSD^+(\Pi)$ to confirm that it is possible positive behavior.

EXAMPLE VI.10. As an example, Program VI.12 is $LSD^+(\Pi)$ for the the full-adder Program V.1. We make the same assumptions concerning $COMPS(\Pi)$ as we did for Program VI.7 in Example VI.6.

EXAMPLE VI.11. Since the goal

$$\leftarrow x1([-, ok, -, -, -, -], 1, 0, 1)$$

succeeds in Program VI.7, then it is normal positive behavior of $x1$. Similarly, since the goal

$$\leftarrow x1([-, ab, -, -, -, -], 1, 0, 1)$$

finitely fails in Program VI.7, then it is also a possible abnormal positive behavior of $x1$. On the other hand, since the goal

$$\leftarrow x1([-, ok, -, -, -, -], 1, 0, 0)$$

finitely fails in Program VI.7, then it is not normal positive behavior of $x1$.

3.2. Discussion. Note that in both Programs VI.7 and VI.12, the *fullAdder* component and all of its subcomponents are represented in the labels. This permits the normal and abnormal behavior of all components including *fullAdder* to be verified. This contrasts with the system description employed by the IDA framework which only permits the behavior of components in flat-hierarchies to be verified. Also, whereas the positive labelled system description Program VI.12 is

PROGRAM VI.12 (LSD⁺(Π) FOR THE FULL-ADDER SPECIFICATION).

```

fullAdder([ok, -, -, -, -], K, L, M, R, S)  $\leftarrow$  x1([ok, -, -, -, -], K, L, N),
                                         a1([ok, -, -, -, -], K, L, Q),
                                         a2([ok, -, -, -, -], M, N, P),
                                         x2([ok, -, -, -, -], N, M, R),
                                         o1([ok, -, -, -, -], P, Q, S).

x1([-, ok, -, -, -], X, Y, Z)  $\leftarrow$  xor(X, Y, Z).
x2([-, -, ck, -, -, -], X, Y, Z)  $\leftarrow$  xor(X, Y, Z).
a1([-, -, -, ok, -, -], X, Y, Z)  $\leftarrow$  and(X, Y, Z).
a2([-, -, -, -, ok, -], X, Y, Z)  $\leftarrow$  and(X, Y, Z).
o1([-, -, -, -, -, ok], X, Y, Z)  $\leftarrow$  or(X, Y, Z).

fullAdder([ab, -, -, -, -], -, -, -, -).
x1([-, ab, -, -, -], -, -, -).
x2([-, -, ab, -, -, -], -, -, -).
a1([-, -, -, ab, -, -], -, -, -).
a2([-, -, -, -, ab, -], -, -, -).
o1([-, -, -, -, -, ab], -, -, -).

and(X, 0, 0). and(0, X, 0). and(1, 1, 1).
or(1, X, 1). or(X, 1, 1). or(0, 0, 0).
xor(0, 0, 0). xor(0, 1, 1). xor(1, 0, 1). xor(1, 1, 0).

```

similar to the IDA system description Program VI.2, the negative labelled system description Program VI.7 has no counterpart in the IDA framework. This is because the IDA framework only defines diagnoses with respect to positive assertions, and not negative assertions.

3.3. Meta-Functions. Before we present the formal definitions of labelled system descriptions we first define five meta-functions for label generation, label application and label removal. These functions are used throughout the remainder of the chapter to simplify the presentation of definitions and proofs. We consider these functions to be meta-functions so as to distinguish them from functions of a first order language. In defining the meta-functions we take Π to be a logic program, $|\text{COMPS}(\Pi)| = m$, and $\text{COMPS}(\Pi)$ to be totally ordered.

3.3.1. Label Generation

Let p_k be the k^{th} component of $\text{COMPS}(\Pi)$, $1 \leq k \leq m$. The function gl is the *label generation function* which takes a clause F belonging to $L(\Pi)$ as its first argument, a component identifier $p_k \in \text{COMPS}(\Pi)$ as its second argument and a constant $s_k = ab$ or $s_k = ok$ as its third argument such that

$$gl(F, p_k, s_k) = [X_1, \dots, X_{k-1}, s_k, X_{k+1}, \dots, X_m],$$

where $\{X_1, \dots, X_{k-1}, s_k, X_{k+1}, \dots, X_m\} \cap \text{vars}(F) = \emptyset$.

3.3.2. Component State Generation

The function s is the *component state generation function* which takes a component identifier $p \in \text{COMPS}(\Pi)$ as its first argument and a set $\Delta \subseteq \text{COMPS}(\Pi)$ as its second argument such that

$$s(p, \Delta) = \begin{cases} ab & \text{if } p \in \Delta, \\ ok & \text{if } p \in \text{COMPS}(\Pi) - \Delta. \end{cases}$$

3.3.3. Ground Label Generation

d is the *ground label generation function* which takes a set $\Delta \subseteq \text{COMPS}(\Pi)$ as its only argument such that

$$d(\Delta) = [s(p_1, \Delta), \dots, s(p_m, \Delta)],$$

where for all $1 \leq i \leq m$, p_i represents the i^{th} component of $\text{COMPS}(\Pi)$.

3.3.4. Label Application

al is the *label application function* which either takes a clause $F = p_0(t_0) \leftarrow p_1(t_1), \dots, p_j(t_j)$, $j \geq 0$ belonging to $L(\Pi)$ or a goal $G = \leftarrow q_1(s_1), \dots, q_k(s_k)$, $k \geq 1$ belonging to $L(\Pi)$ as its first argument, and a label L as its second argument such that

$$al(F, L) = p_0(L, t_0) \leftarrow p_1(L, t_1), \dots, p_j(L, t_j), \text{ and} \\ al(G, L) = \leftarrow q_1(L, s_1), \dots, q_k(L, s_k).$$

3.3.5. Label Removal

Let $p_0(t_0) \leftarrow p_1(t_1), \dots, p_j(t_j)$, $j \geq 0$ be a clause belonging to $L(\Pi)$, let $\leftarrow q_1(s_1), \dots, q_k(s_k)$, $k \geq 1$ be a goal belonging to $L(\Pi)$ and let L be a label. rl is the *label removal function* which either takes a labelled clause $F = p_0(L, t_0) \leftarrow p_1(L, t_1), \dots, p_j(L, t_j)$, $j \geq 0$ or a labelled goal $G = \leftarrow q_1(L, s_1), \dots, q_k(L, s_k)$, $k \geq 1$ as its only argument such that

$$rl(F) = p_0(t_0) \leftarrow p_1(t_1), \dots, p_j(t_j), \text{ and} \\ rl(G) = \leftarrow q_1(s_1), \dots, q_k(s_k).$$

3.4. Transforming Π to a Labelled System Description. The formal definitions of the labelled system descriptions $LSD^-(\Pi)$ and $LSD^+(\Pi)$ of a logic program Π follow.

DEFINITION VI.13 (NEGATIVE LABELLED SYSTEM DESCRIPTION). Let Π be a logic program and let $LSD^-(\Pi)$ be the *negative labelled system description* for Π .

$$LSD^-(\Pi) = \{al((p(t) \leftarrow F), L) \mid (p(t) \leftarrow F) \in \Pi \text{ (} F \text{ possibly empty),} \\ L = gl((p(t) \leftarrow F), p, ok)\}.$$

DEFINITION VI.14 (POSITIVE LABELLED SYSTEM DESCRIPTION). Let Π be a logic program and let $LSD^+(\Pi)$ be the *positive labelled system description* for Π .

$$LSD^+(\Pi) = LSD^-(\Pi) \cup$$

$$\{al(p(X), L) \mid p \text{ is an } n\text{-ary predicate of } \Pi \text{ and } L = gl(p(X), p, ab)\}.$$

4. Diagnosis Verification

In this section we first present examples of using labelled system descriptions for diagnosis verification. Then we present the soundness, completeness and decidability properties of the approach. We leave the proofs of the various properties and their supporting lemmas to the end of this section.

4.1. Examples. With programs $LSD^-(\Pi)$ and $LSD^+(\Pi)$ it becomes a simple matter of using SLD-resolution and the NAF rule to verify whether or not a sentence $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$ where OBS is a set of either negative or positive assertions. To verify that $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis with respect to a negative assertion, one must confirm that a goal associated with the negative assertion finitely fails in the negative labelled system description. To verify that $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis with respect to a positive assertion, one must confirm that a goal associated with the positive assertion succeeds in the positive labelled system description.

EXAMPLE VI.15. Consider the assertion $\neg fullAdder(1, 0, 1, 0, 1)$ which reflects the fact that the specified behavior $K = 1, L = 0, M = 1, R = 0, S = 1$ is not exhibited by a faulty full-adder artifact (see Figure V.11). Since the assertion is negative then to determine whether or not $\mathcal{D}(\emptyset, COMPS(\Pi))$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), \{\neg fullAdder(1, 0, 1, 0, 1)\})$ where Π is the full-adder specification Program V.1, we check whether or not

$$comp(LSD^-(\Pi)) \models \neg fullAdder([ok, ok, ok, ok, ok], 1, 0, 1, 0, 1).$$

Here we take $LSD^-(\Pi)$ to be Program VI.7. This amounts to checking whether or not the goal

$$\leftarrow fullAdder([ok, ok, ok, ok, ok], 1, 0, 1, 0, 1)$$

finitely fails in $LSD^-(\Pi)$ or not. Since

$$LSD^-(\Pi) \cup \{\leftarrow fullAdder([ok, ok, ok, ok, ok], 1, 0, 1, 0, 1)\}$$

has an SLD-refutation then by the soundness of SLD-resolution

$$comp(LSD^-(\Pi)) \models fullAdder([ok, ok, ok, ok, ok], 1, 0, 1, 0, 1).$$

Therefore we can conclude that $\mathcal{D}(\emptyset, \text{COMPS}(\Pi))$ is not a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \{\neg \text{fullAdder}(1, 0, 1, 0, 1)\})$.

EXAMPLE VI.16. Consider the assertion $\text{fullAdder}(1, 0, 1, 1, 0)$ which reflects the fact that the erroneous behavior $K = 1, L = 0, M = 1, R = 1, S = 0$ is exhibited by a faulty full-adder artifact. Since the assertion is positive then to determine whether or not $\mathcal{D}(\{x1\}, \{\text{fullAdder}, x2, a1, a2, o1\})$ is a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \{\text{fullAdder}(1, 0, 1, 1, 0)\})$ where Π is the full-adder specification Program V.1, we check whether or not

$$\text{comp}(\text{LSD}^+(\Pi)) \models \text{fullAdder}([ok, ab, ok, ok, ok, ok], 1, 0, 1, 1, 0).$$

Here we take $\text{LSD}^+(\Pi)$ to be Program VI.12. This amounts to checking whether or not the goal

$$\leftarrow \text{fullAdder}([ok, ab, ok, ok, ok, ok], 1, 0, 1, 1, 0)$$

succeeds in $\text{LSD}^+(\Pi)$ or not. Since

$$\text{LSD}^+(\Pi) \cup \{\leftarrow \text{fullAdder}([ok, ab, ok, ok, ok, ok], 1, 0, 1, 1, 0)\}$$

has an SLD-refutation then by the soundness of SLD-resolution

$$\text{comp}(\text{LSD}^+(\Pi)) \models \text{fullAdder}([ok, ab, ok, ok, ok, ok], 1, 0, 1, 1, 0).$$

Therefore we can conclude that $\mathcal{D}(\{x1\}, \{\text{fullAdder}, x2, a1, a2, o1\})$ is a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \{\text{fullAdder}(1, 0, 1, 1, 0)\})$.

4.2. Soundness and Completeness. Intuitively, the approach to diagnosis verification presented in the previous section seems sound. In this section we show that for systems $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ where OBS is either a set of negative or positive assertions, this approach is sound and complete for diagnosis denial and sound for diagnosis confirmation (see Chapter IV). In addition, we show that for an appropriate subclass of negative and positive assertions OBS and logic programs Π , our approach is complete for diagnosis confirmation. We first present the results of using the approach for sets of negative assertions. This is followed by the results for using the approach for sets of positive assertions. The proofs are left until after the discussion of decidability.

4.2.1. Results for $LSD^-(\Pi)$

THEOREM VI.17 (SOUNDNESS—DENIAL FOR $LSD^-(\Pi)$). *Let Π be a logic program, OBS be a set of negative assertions and $\Delta \subseteq COMPS(\Pi)$. If for some $\forall \neg p(t) \in OBS$, $comp(LSD^-(\Pi)) \models \exists p(d(\Delta), t)$ then $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is not a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$.*

THEOREM VI.18 (COMPLETENESS—DENIAL FOR $LSD^-(\Pi)$). *Let Π be a logic program, OBS be a set of positive or negative assertions and $\Delta \subseteq COMPS(\Pi)$. If $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is not a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$ then for some $\forall \neg p(t) \in OBS$, $comp(LSD^-(\Pi)) \models \exists p(d(\Delta), t)$ or for some $\exists p(t) \in OBS$, $comp(LSD^-(\Pi)) \models \forall \neg p(d(\Delta), t)$.*

An immediate consequence of Theorems VI.17 and VI.18 is the following corollary.

COROLLARY VI.19. *Let Π be a logic program, OBS be a set of negative assertions and $\Delta \subseteq COMPS(\Pi)$. $comp(LSD^-(\Pi)) \models \exists p(d(\Delta), t)$ for some $\forall \neg p(t) \in OBS$ iff $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is not a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$.*

THEOREM VI.20 (SOUNDNESS—CONFIRMATION FOR $LSD^-(\Pi)$). *Let Π be a logic program, OBS be a set of positive or negative assertions and $\Delta \subseteq COMPS(\Pi)$. If for all $\forall \neg p(t) \in OBS$, $comp(LSD^-(\Pi)) \models \forall \neg p(d(\Delta), t)$ and for all $\exists p(t) \in OBS$, $comp(LSD^-(\Pi)) \models \exists p(d(\Delta), t)$ then $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$.*

For completeness—confirmation to hold we restrict the class of logic programs and negative assertions.

THEOREM VI.21 (COMPLETENESS—CONFIRMATION FOR $LSD^-(\Pi)$). *Let Π be a logic program and $\Delta \subseteq COMPS(\Pi)$. Let OBS be a set of negative assertions such that for all $\forall \neg p(t) \in OBS$, $\Pi \cup \{\leftarrow p(t)\}$ has no infinite SLD-derivations for some selection rule. If $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$ then $comp(LSD^-(\Pi)) \models \forall \neg p(d(\Delta), t)$.*

From Theorems VI.20 and VI.17 one can see that using our approach diagnosis confirmation is sound for systems of the form $(SD(\Pi), COMPS(\Pi), OBS)$ where OBS is a set of negative assertions. However, note that the completeness theorem constrains literals $\forall \neg p(t) \in OBS$ and logic programs Π so that $\Pi \cup \{\leftarrow p(t)\}$

has no infinite SLD-derivations for some selection rule. Without this constraint completeness is not possible as illustrated by the following example.

EXAMPLE VI.22.

$$\text{Let } \Pi = \left\{ \begin{array}{l} b(X) \leftarrow c(X). \\ b(X) \leftarrow a(X). \\ a(X) \leftarrow a(X). \\ c(1). \end{array} \right\}, \text{ and let } \text{OBS} = \{\neg b(1)\}.$$

Note that while $\Pi \models b(1)$, $\Pi \cup \{\leftarrow b(1)\}$ also has an infinite SLD-derivation with resolvents

$$\leftarrow b(1), \leftarrow a(1), \leftarrow a(1), \leftarrow a(1), \dots$$

for all selection rules. Now $\mathcal{D}(\{c\}, \{a, b\})$ is a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \{\neg b(1)\})$ but $\text{comp}(\text{LSD}^-(\Pi)) \not\models \neg b(d(\{c\}), 1)$, and $\text{comp}(\text{LSD}^-(\Pi)) \not\models b(d(\{c\}), 1)$. Thus, for a given logic program Π and a negative assertion $\text{OBS} = \{\forall \neg p(t)\}$ one cannot always verify whether or not $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ using a labelled system description $\text{LSD}^-(\Pi)$.

4.2.2. Results for $\text{LSD}^+(\Pi)$

THEOREM VI.23 (SOUNDNESS—DENIAL FOR $\text{LSD}^+(\Pi)$). *Let Π be a logic program, OBS be a set of positive assertions and $\Delta \subseteq \text{COMPS}(\Pi)$. If for some $\exists p(t) \in \text{OBS}$, $\text{comp}(\text{LSD}^+(\Pi)) \models \forall \neg p(d(\Delta), t)$ then $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is not a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$.*

THEOREM VI.24 (COMPLETENESS—DENIAL FOR $\text{LSD}^+(\Pi)$). *Let Π be a logic program, OBS be a set of positive or negative assertions and $\Delta \subseteq \text{COMPS}(\Pi)$. If $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is not a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ then for some $\exists p(t) \in \text{OBS}$, $\text{comp}(\text{LSD}^+(\Pi)) \models \forall \neg p(d(\Delta), t)$ or for some $\forall \neg p(t) \in \text{OBS}$, $\text{comp}(\text{LSD}^+(\Pi)) \models \exists p(d(\Delta), t)$.*

An immediate consequence of Theorems VI.23 and VI.24 is the following corollary.

COROLLARY VI.25. *Let Π be a logic program, OBS be a set of positive assertions and $\Delta \subseteq \text{COMPS}(\Pi)$. $\text{comp}(\text{LSD}^+(\Pi)) \models \forall \neg p(d(\Delta), t)$ for some $\exists p(t) \in \text{OBS}$ iff $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is not a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$.*

THEOREM VI.26 (SOUNDNESS—CONFIRMATION FOR $LSD^+(\Pi)$). *Let Π be a logic program, OBS be a set of positive or negative assertions and $\Delta \subseteq COMPS(\Pi)$. If for all $\exists p(t) \in OBS$, $comp(LSD^+(\Pi)) \models \exists p(d(\Delta), t)$ and for all $\forall \neg p(t) \in OBS$, $comp(LSD^+(\Pi)) \models \forall \neg p(d(\Delta), t)$ then $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$.*

For completeness—confirmation to hold we must consider a more restricted class of logic programs and positive assertions than was required to ensure completeness—confirmation for $LSD^-(\Pi)$. Before we present the actual completeness theorem we define the notion of a positive Δ -transform of a logic program Π , denoted Π_{Δ}^+ to be the set

$$\{(p(t) \leftarrow F) \in \Pi, (F \text{ possibly empty}) \mid p \in COMPS(\Pi) - \Delta\} \cup \{p(X) \mid p \in \Delta\}.$$

Thus, Π_{Δ}^+ contains all the clauses that Π does for components assumed to be normal. However, Π_{Δ}^+ contains a clause of the form $p(X)$ for components assumed to be faulty. We examine the notion of a Δ -transform in detail in a later section when we present the proofs of the completeness and soundness theorems. For now it is only necessary to know how a positive Δ -transform is defined in order to understand the following completeness theorem.

THEOREM VI.27 (COMPLETENESS—CONFIRMATION FOR $LSD^+(\Pi)$). *Let Π be a logic program and $\Delta \subseteq COMPS(\Pi)$. Let OBS be a set of positive assertions such that for all $\exists p(t) \in OBS$, $\Pi_{\Delta}^+ \cup \{\leftarrow p(t)\}$ has no infinite SLD-derivations for some selection rule. If $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$ then $comp(LSD^+(\Pi)) \models \exists p(d(\Delta), t)$.*

From Theorems VI.26 and VI.23 one can see that using our approach diagnosis confirmation is sound for systems $(SD(\Pi), COMPS(\Pi), OBS)$ where OBS is a set of positive assertions. However, for completeness of diagnosis confirmation the class of observations OBS and logic programs Π is constrained so that for all $\exists p(t) \in OBS$, $\Pi_{\Delta}^+ \cup \{\leftarrow p(t)\}$ has no infinite SLD-derivations for some selection rule. Note that this constraint deals with the positive Δ -transform Π_{Δ}^+ of Π rather than Π itself. The importance of this constraint with respect to completeness is illustrated by the following example.

EXAMPLE VI.28.

Let $\Pi = \{(b(X) \leftarrow b(X), a(X))\}$, and let $OBS = \{b(1)\}$.

Note that $comp(\Pi) \models \neg a(1)$ and $comp(\Pi) \models \neg b(1)$. Therefore there must exist a selection rule such that $\Pi \cup \{\leftarrow b(1)\}$ has no infinite SLD-derivations. However, the satisfaction of this condition is not sufficient to ensure completeness. This is because although $\mathcal{D}(\{a\}, \{b\})$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$, we have $comp(\Pi_{\Delta}^+) \not\models \neg b(1)$, and $comp(\Pi_{\Delta}^+) \not\models b(1)$. Furthermore, $comp(LSD^+(\Pi)) \not\models \neg b(d(\{a\}), 1)$ and $comp(LSD^+(\Pi)) \not\models b(d(\{a\}), 1)$. Thus, for a given logic program Π , a positive assertion $OBS = \{\exists p(t)\}$ and $\Delta \subseteq COMPS(\Pi)$, one cannot always verify when $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$ using a labelled system description $LSD^+(\Pi)$.

4.3. Decidability. The decidability results follow directly from the soundness and completeness results of the previous sections. For the meaning of decidability in the context of diagnosis verification see Chapter IV.

4.3.1. Results For $LSD^-(\Pi)$

In this section we show that diagnosis denial is partially decidable for $LSD^-(\Pi)$ and that for a restricted class of logic programs Π and negative assertions OBS , diagnosis confirmation is totally decidable.

THEOREM VI.29 (PARTIAL DECIDABILITY—DENIAL FOR $LSD^-(\Pi)$). *Let Π be a logic program, OBS be a finite set of negative assertions and $\Delta \subseteq COMPS(\Pi)$. The question “Is $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ not a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$?” is partially decidable.*

PROOF. From Corollary VI.19, the question posed is equivalent to “Is there some $\forall \neg p(t) \in OBS$ such that $comp(LSD^-(\Pi)) \models \exists p(d(\Delta), t)$?” By the soundness and completeness of SLD-resolution, this question is equivalent to “Does $\leftarrow p(d(\Delta), t)$ succeed in $LSD^-(\Pi)$ for some $\forall \neg p(t) \in OBS$?” Since OBS is finite we can envisage concurrently searching for an SLD-refutation for all members $\forall \neg p(t)$ of OBS and terminating the procedure when for some $\forall \neg p(t) \in OBS$, we find that $comp(LSD^-(\Pi)) \models \exists p(d(\Delta), t)$. Thus, by appealing to Church’s thesis [15], the original question is partially decidable. $\square$

THEOREM VI.30 (TOTAL DECIDABILITY FOR $LSD^-(\Pi)$). *Let Π be a logic program and $\Delta \subseteq COMPS(\Pi)$. Let OBS be a finite set of negative assertions such that for all $\forall \neg p(t) \in OBS$, $\Pi \cup \{\leftarrow p(t)\}$ has no infinite SLD-derivations for some selection rule. The question "Is $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$?" is totally decidable.*

PROOF. From the soundness and completeness theorems for diagnosis confirmation (Theorem VI.20 and Theorem VI.21) the question is equivalent to "Does $comp(LSD^-(\Pi)) \models \forall \neg p(d(\Delta), t)$ for all $\forall \neg p(t) \in OBS$?" By the soundness and completeness of the NAF rule this question is, in turn, equivalent to "For all $\forall \neg p(t) \in OBS$, does $\leftarrow p(d(\Delta), t)$ finitely fail in $LSD^-(\Pi)$?" Since OBS is finite we can envisage successively confirming for each member of OBS that for a given fair selection rule, all SLD-derivations are failed. This procedure will terminate after the finite failure of all members of OBS has been confirmed. Therefore, by appealing to Church's thesis, the question is partially decidable. In addition, we know from Theorem VI.29 that the question "Is $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ not a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$?" is partially decidable. Therefore the question "Is $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$?" is totally decidable. $\square$

With regards to the condition used in the total decidability theorem, the question of whether or not $\Pi \cup \{\leftarrow p(t)\}$ has any infinite SLD-derivations for some selection rule is the same as determining whether Π will halt for some goal $\leftarrow p(t)$. This is the well known halting problem which is undecidable in general. However, the question has been shown to be decidable for certain classes of logic programs and goals (see, for example [76]).

4.3.2. Results For $LSD^+(\Pi)$

In this section we show that diagnosis denial for $LSD^+(\Pi)$ is partially decidable and that for a restricted class of logic programs Π and positive assertions OBS , diagnosis confirmation is totally decidable.

THEOREM VI.31 (PARTIAL DECIDABILITY—DENIAL FOR $LSD^+(\Pi)$). *Let Π be a logic program, OBS be a finite set of positive assertions and $\Delta \subseteq COMPS(\Pi)$. The question "Is $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ not a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$?" is partially decidable.*

PROOF. From Corollary VI.25, the question posed is equivalent to "Is there some $\exists p(t) \in \text{OBS}$ such that $\text{comp}(\text{LSD}^+(\Pi)) \models \forall \neg p(d(\Delta), t)$?" By the soundness and completeness of the NAF rule this question is, in turn, equivalent to "Does $\leftarrow p(d(\Delta), t)$ finitely fail in $\text{LSD}^+(\Pi)$ for some $\exists p(t) \in \text{OBS}$?" For each $\exists p(t) \in \text{OBS}$ the question is partially decidable since finite failure is partially decidable. Furthermore, since OBS is finite we can envisage concurrently testing each member of OBS for finite failure and terminating the procedure when for some $\exists p(t) \in \text{OBS}$, we find that $\text{comp}(\text{LSD}^+(\Pi)) \models \forall \neg p(d(\Delta), t)$. Thus, by appealing to Church's thesis, the original question is partially decidable. $\square$

THEOREM VI.32 (TOTAL DECIDABILITY FOR $\text{LSD}^+(\Pi)$). *Let Π be a logic program, OBS be a finite set of positive assertions such that for all $\exists p(t) \in \text{OBS}$, $\Pi_{\Delta}^+ \cup \{\leftarrow p(t)\}$ has no infinite SLD-derivations for some selection rule and $\Delta \subseteq \text{COMPS}(\Pi)$. The question "Is $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$?" is totally decidable.*

PROOF. From the soundness and completeness theorems for diagnosis confirmation (Theorems VI.26 and Theorem VI.27) the question is equivalent to "Does $\text{comp}(\text{LSD}^+(\Pi)) \models \exists p(d(\Delta), t)$ for all $\exists p(t) \in \text{OBS}$?" By the soundness and completeness of SLD-resolution, this question is equivalent to "Does $\leftarrow p(d(\Delta), t)$ succeed in $\text{LSD}^+(\Pi)$ for all $\exists p(t) \in \text{OBS}$?" Since OBS is finite we can envisage successively searching for an SLD-refutation for all members $\exists p(t)$ of OBS and terminating once a refutation has been found for all members of OBS. Therefore, by appealing to Church's thesis, the question is partially decidable. In addition, we know from Theorem VI.31 that the question "Is $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ not a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$?" is partially decidable. Therefore the question "Is $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$?" is totally decidable. $\square$

4.4. Proofs. We now develop the proofs for the soundness and completeness results for $\text{LSD}^-(\Pi)$ and $\text{LSD}^+(\Pi)$. First however, it is necessary to introduce the notion of a Δ -transform.

4.4.1. Δ -Transforms

In order to simplify the development of our results we introduce the notion of a negative Δ -transform Π_{Δ}^- , and a positive Δ -transform Π_{Δ}^+ for a logic program Π .

The Δ -transforms Π_{Δ}^{-} and Π_{Δ}^{+} are logic programs that serve as a bridge between a system description $SD(\Pi)$ and the labelled system descriptions $LSD^{-}(\Pi)$ and $LSD^{+}(\Pi)$, respectively. We also use the notions of Δ -transforms extensively in the next chapter. We first define the notions of Δ -transforms. Then we present some properties related to the completion and Δ -completion (see Chapters II and V, respectively) of Δ -transforms.

DEFINITION VI.33 (NEGATIVE Δ -TRANSFORM). Let Π be a logic program and $\Delta \subseteq COMPS(\Pi)$. Let Π_{Δ}^{-} denote the *negative Δ -transform of Π* .

$$\Pi_{\Delta}^{-} = \{(p(t) \leftarrow F) \in \Pi, (F \text{ possibly empty}) \mid p \in COMPS(\Pi) - \Delta\}.$$

DEFINITION VI.34 (POSITIVE Δ -TRANSFORM). Let Π be a logic program and $\Delta \subseteq COMPS(\Pi)$. Let Π_{Δ}^{+} denote the *positive Δ -transform of Π* .

$$\Pi_{\Delta}^{+} = \{(p(t) \leftarrow F) \in \Pi, (F \text{ possibly empty}) \mid p \in COMPS(\Pi) - \Delta\} \cup \{p(X) \mid p \in \Delta\}.$$

To simplify our exposition in this chapter and the next, we take the equality theory that is included in the completion of a Δ -transform to be $EQ(\Pi)$. Formally,

$$\begin{aligned} comp(\Pi_{\Delta}^{-}) &= IFF(\Pi_{\Delta}^{-}) \cup EQ(\Pi), \text{ and} \\ comp(\Pi_{\Delta}^{+}) &= IFF(\Pi_{\Delta}^{+}) \cup EQ(\Pi). \end{aligned}$$

The following are some properties of a negative Δ -transform.

LEMMA VI.35. *Let Π be a logic program and $\Delta \subseteq COMPS(\Pi)$.*

- (a) $comp_{\Delta}(\Pi) = comp_{\Delta}(\Pi_{\Delta}^{-})$,
- (b) $comp(\Pi_{\Delta}^{-}) \models comp_{\Delta}(\Pi)$, and
- (c) $comp_{\Delta}(\Pi_{\Delta}^{-}) \models \Pi_{\Delta}^{-}$.

PROOF.

(a) Follows directly from the definitions of Π_{Δ}^{-} , the completion of a program and the Δ -completion of a program.

(b) From the definitions of Π_{Δ}^{-} , the completion of a program and the Δ -completion of a program it follows that $comp_{\Delta}(\Pi) \subseteq comp(\Pi_{\Delta}^{-})$. Thus, $comp(\Pi_{\Delta}^{-}) \models comp_{\Delta}(\Pi)$.

(c) From the definitions of Π_{Δ}^{-} , the completion of a program and the Δ -completion of a program it follows that $IF(\Pi_{\Delta}^{-}) \cup EQ(\Pi) \subseteq comp_{\Delta}(\Pi_{\Delta}^{-})$. Since $IF(\Pi_{\Delta}^{-}) \cup EQ(\Pi)$ is semantically equal to Π_{Δ}^{-} then $comp_{\Delta}(\Pi_{\Delta}^{-}) \models \Pi_{\Delta}^{-}$.

□

The following are some properties of a positive Δ -transform.

LEMMA VI.36. *Let Π be a logic program and $\Delta \subseteq COMPS(\Pi)$.*

(a) $comp_{\Delta}(\Pi) = comp_{\Delta}(\Pi_{\Delta}^{+})$,

(b) $comp(\Pi_{\Delta}^{+}) \models comp_{\Delta}(\Pi)$.

PROOF.

(a) Follows directly from the definitions of Π_{Δ}^{+} , the completion of a program and the Δ -completion of a program.

(b) From the definitions of Π_{Δ}^{+} , the completion of a program and the Δ -completion of a program it follows that $comp_{\Delta}(\Pi) \subseteq comp(\Pi_{\Delta}^{+})$. Thus, $comp(\Pi_{\Delta}^{+}) \models comp_{\Delta}(\Pi)$.

□

4.4.2. Supporting Lemmas For $LSD^{-}(\Pi)$

In this section we present the supporting theorems and lemmas required for the proofs of diagnostic soundness and completeness using $LSD^{-}(\Pi)$. An overview of the theorems and lemmas required is shown in Figure VI.13. The proofs of the soundness and completeness properties themselves are included in the next section.

Lemmas VI.37 and VI.38 formalize the relationship between a system description $SD(\Pi)$ and a negative Δ -transform Π_{Δ}^{-} . Lemma VI.41 formalizes the relationship between a negative Δ -transform Π_{Δ}^{-} and a negative labelled system description $LSD^{-}(\Pi)$.

LEMMA VI.37. *Let Π be a logic program, OBS be a set of sentences of $L(\Pi)$ and $\Delta \subseteq COMPS(\Pi)$. If $comp(\Pi_{\Delta}^{-}) \cup OBS$ is satisfiable then $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$.*

PROOF. Since $comp(\Pi_{\Delta}^{-}) \cup OBS$ is satisfiable, then by Lemma VI.35(b) and the compactness of FOPC, $comp_{\Delta}(\Pi) \cup OBS$ is satisfiable. Now, since ab is not a predicate in $L(\Pi)$ then

$$comp_{\Delta}(\Pi) \cup OBS \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \text{ is satisfiable.}$$

FIGURE VI.13. Overview of relationship between theorems and supporting lemmas for negative assertions

So by Lemma V.17

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS \text{ is satisfiable.}$$

Therefore $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$. $\square$

LEMMA VI.38. *Let Π be a logic program, $\forall \neg p(t) \in L(\Pi)$, $OBS = \{\forall \neg p(t)\}$, and $\Delta \subseteq COMPS(\Pi)$. If $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$ and $\Pi \cup \{\leftarrow p(t)\}$ has no infinite SLD-derivations for some selection rule then $comp(\Pi_{\Delta}^-) \models \forall \neg p(t)$.*

In order to prove Lemma VI.38 we require the following two lemmas.

LEMMA VI.39. *Let Π be a logic program, OBS be a set of negative assertions and $\Delta \subseteq COMPS(\Pi)$. If $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$ then $comp(\Pi_{\Delta}^-) \cup OBS$ is satisfiable.*

PROOF. Since $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$ then

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS \text{ is satisfiable.}$$

So by Lemma V.17,

$$comp_{\Delta}(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS \text{ is satisfiable,}$$

and by Lemma VI.35(a)

$$comp_{\Delta}(\Pi_{\Delta}^-) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS \text{ is satisfiable.}$$

Now since ab is not a predicate in $L(\Pi)$ then by Lemma V.17 we know that

$$(VI.1) \quad SD(\Pi_{\Delta}^-) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS \text{ is satisfiable.}$$

Therefore $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi_{\Delta}^-), COMPS(\Pi), OBS)$.

Now, there must exist a minimal diagnosis $\mathcal{D}(\Delta', COMPS(\Pi) - \Delta')$, $\Delta' \subseteq \Delta$ for $(SD(\Pi_{\Delta}^-), COMPS(\Pi), OBS)$. Furthermore, by Theorem V.12,

$$SD(\Pi_{\Delta}^-) \cup \{\mathcal{D}(\Delta', COMPS(\Pi) - \Delta')\} \cup OBS \models \bigwedge_{q \in \Delta'} INCOMPLETE(q).$$

By the definition of Π_{Δ}^{-} and formula (V.2), for all $q \in \Delta'$,

$$\begin{aligned} \text{INCORRECT}(q) &\leftrightarrow \neg(\forall X(q(X) \leftarrow \text{false})), \\ \text{or } \text{INCORRECT}(q) &\leftrightarrow \text{false}. \end{aligned}$$

Therefore, $\Delta' = \emptyset$ can be the only minimal diagnosis because otherwise for all $\emptyset \subset \Delta' \subseteq \Delta$,

$$\text{SD}(\Pi_{\Delta}^{-}) \cup \{\mathcal{D}(\Delta', \text{COMPS}(\Pi) - \Delta')\} \cup \text{OBS} \models \text{false}.$$

But this would contradict (VI.1) for $\Delta' = \Delta$. Thus,

$$\text{SD}(\Pi_{\Delta}^{-}) \cup \{\mathcal{D}(\emptyset, \text{COMPS}(\Pi))\} \cup \text{OBS} \text{ is satisfiable.}$$

Or, by Lemma V.17,

$$\text{comp}(\Pi_{\Delta}^{-}) \cup \{\mathcal{D}(\emptyset, \text{COMPS}(\Pi))\} \cup \text{OBS} \text{ is satisfiable.}$$

Therefore, by the compactness of FOPC, $\text{comp}(\Pi_{\Delta}^{-}) \cup \text{OBS}$ is satisfiable. $\square$

LEMMA VI.40. *Let Π be a logic program, $\forall \neg p(t) \in L(\Pi)$, $\text{OBS} = \{\forall \neg p(t)\}$, and $\Delta \subseteq \text{COMPS}(\Pi)$. If $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ and $R_0, \dots, R_n, \square$ are the resolvents of an SLD-refutation of $\Pi \cup \{\leftarrow p(t)\}$ where $R_0 = \leftarrow p(t)$ then there exists a resolvent R_i , $0 \leq i \leq n$, and an atom $q(t)$, $q \in \Delta$ such that $q(t)$ is an atom in the resolvent R_i .*

PROOF. By way of contradiction assume that for all $q \in \Delta$, no atom $q(t)$ occurs in any resolvent R_i , $0 \leq i \leq n$, of an SLD-refutation with resolvents $R_0, \dots, R_n, \square$ of $\Pi \cup \{\leftarrow p(t)\}$ where $R_0 = \leftarrow p(t)$.

Therefore no clauses defining predicates of $q \in \Delta$ are required to derive an SLD-refutation. So by the definition of Π_{Δ}^{-} , $R_0, \dots, R_n, \square$ are resolvents of an SLD-refutation of $\Pi_{\Delta}^{-} \cup \{\leftarrow p(t)\}$. Thus, by the soundness of SLD-resolution $\text{comp}(\Pi_{\Delta}^{-}) \models \exists p(t)$, or

$$\text{comp}(\Pi_{\Delta}^{-}) \cup \text{OBS} \text{ is not satisfiable.}$$

But since $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ then by Lemma VI.39, $\text{comp}(\Pi_{\Delta}^{-}) \cup \text{OBS}$ is satisfiable, which is a contradiction. $\square$

PROOF OF LEMMA VI.38. Since $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ then by Lemma VI.40, all SLD-refutations of $\Pi \cup \{\leftarrow p(t)\}$ with resolvents $R_0, \dots, R_n, \square$ where $R_0 = \leftarrow p(t)$, possess a resolvent R_i , $0 \leq i \leq n$, containing an atom $q(t)$, $q \in \Delta$. Therefore $\Pi_{\Delta}^- \cup \{\leftarrow p(t)\}$ can have no SLD-refutations. Furthermore, since $\Pi \cup \{\leftarrow p(t)\}$ has no infinite SLD-refutations for some selection rule then $\Pi_{\Delta}^- \cup \{\leftarrow p(t)\}$ can have no infinite SLD-derivations for the same selection rule since $\Pi_{\Delta}^- \subseteq \Pi$. Therefore, for some selection rule, all SLD-derivations of $\Pi_{\Delta}^- \cup \{\leftarrow p(t)\}$ are finitely failed. Thus, by the soundness of the NAF rule, $\text{comp}(\Pi_{\Delta}^-) \models \forall \neg p(t)$. $\square$

Now we proceed to formalize the relationship between Π_{Δ}^- and $\text{LSD}^-(\Pi)$. This relationship is stated by the following lemma.

LEMMA VI.41. *Let Π be a logic program, $\forall \neg p(t) \in L(\Pi)$ and $\Delta \subseteq \text{COMPS}(\Pi)$. $\text{comp}(\Pi_{\Delta}^-) \models \forall \neg p(t)$ iff $\text{comp}(\text{LSD}^-(\Pi)) \models \forall \neg p(d(\Delta), t)$.*

Although this lemma is simple to state, it is rather complicated to prove. We proceed with the proof in a bottom-up fashion as illustrated by Figure VI.13.

LEMMA VI.42. *Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$, $\text{COMPS}(\Pi) = \{p_1, \dots, p_m\}$ be totally ordered as shown, p_k be the k^{th} member of $\text{COMPS}(\Pi)$, $1 \leq k \leq m$, $(p_k(t) \leftarrow G) \in L(\Pi)$, and $L = \text{gl}((p_k(t) \leftarrow G), p_k, f)$.*

There exists a substitution η such that $L\eta \equiv d(\Delta)$ iff $f = s(p_k, \Delta)$.

PROOF.

($\Rightarrow$)

By definition of gl ,

$$L = [X_1, \dots, X_{k-1}, f, X_{k+1}, \dots, X_m],$$

and by the definition of d ,

$$d(\Delta) = [s(p_1, \Delta), \dots, s(p_m, \Delta)].$$

Since $L\eta \equiv d(\Delta)$ then

$\eta = \{X_1/s(p_1, \Delta), \dots, X_{k-1}/s(p_{k-1}, \Delta), X_{k+1}/s(p_{k+1}, \Delta), \dots, X_m/s(p_m, \Delta)\}$, and $f = s(p_k, \Delta)$.

($\Leftarrow$)

By the definition of gl

$$L = [X_1, \dots, X_{k-1}, s(p_k, \Delta), x_{k+1}, \dots, x_m],$$

and by the definition of d ,

$$d(\Delta) = [s(p_1, \Delta), \dots, s(p_m, \Delta)].$$

Now if we let

$$\eta = \{X_1/s(p_1, \Delta), \dots, X_{k-1}/s(p_{k-1}, \Delta), X_{k+1}/s(p_{k+1}, \Delta), \dots, X_m/s(p_m, \Delta)\},$$

then we have $L\eta \equiv d(\Delta)$. $\square$

LEMMA VI.43. *Let Π be a logic program, $\Delta \subseteq COMPS(\Pi)$, $(p(s) \leftarrow G) \in L(\Pi)$, $\leftarrow p(t) \in L(\Pi)$, $L = gl((p(s) \leftarrow G), p, f)$, and $al((p(s) \leftarrow G), L) \in LSD^-(\Pi)/LSD^+(\Pi)$.*

$f = s(p, \Delta)$ and there exists a substitution θ such that $t\theta \equiv s\theta$ iff $\leftarrow p(d(\Delta), t)$ and $al((p(s) \leftarrow G), L)$ resolve producing a resolvent equivalent to $(al(\leftarrow G, d(\Delta)))\theta$.

PROOF.

($\Rightarrow$)

Since $f = s(p, \Delta)$ then, by Lemma VI.42 there exists a substitution η such that $L\eta \equiv d(\Delta)$. Since $s\theta \equiv t\theta$ then $p(d(\Delta), t)\eta\theta \equiv p(L, s)\eta\theta$.

Since $p(L, s)$ is the head atom of a clause $al((p(s) \leftarrow G), L)$ then $\leftarrow p(d(\Delta), t)$ and $al((p(s) \leftarrow G), L)$ resolve with resolvent $al(\leftarrow G, L)\eta\theta$, which is equivalent to $(al(\leftarrow G, d(\Delta)))\theta$.

($\Leftarrow$)

Since $\leftarrow p(d(\Delta), t)$ and $al((p(s) \leftarrow G), L)$ resolve then there exists a substitution ζ such that $p(d(\Delta), t)\zeta \equiv p(L, s)\zeta$. Now, by definition of gl , $vars(L) \cap vars(p(s) \leftarrow G) = \emptyset$ so there must exist substitutions η and θ such that $\zeta = \eta\theta$ and $L\eta \equiv d(\Delta)$ and $s\theta \equiv t\theta$. Therefore by Lemma VI.42, $f = s(p, \Delta)$. $\square$

LEMMA VI.44. *Let Π be a logic program, $\Delta \subseteq COMPS(\Pi)$, $(p(s) \leftarrow G) \in L(\Pi)$, and $L = gl((p(s) \leftarrow G), p, f)$. $al((p(s) \leftarrow G), L) \in LSD^-(\Pi)/LSD^+(\Pi)$ and $f = s(p, \Delta)$ iff $(p(s) \leftarrow G) \in \Pi_{\Delta}^-/\Pi_{\Delta}^+$.*

PROOF.

($\Rightarrow$)

Since $al((p(s) \leftarrow G), L) \in LSD^-(\Pi)/LSD^+(\Pi)$ there are two cases to consider.

- (1) If $f = ok$ then by the definition of $LSD^-(\Pi)/LSD^+(\Pi)$, $p \in COMPS(\Pi) - \Delta$ and $(p(s) \leftarrow G) \in \Pi$. By the definition of $\Pi_{\Delta}^-/\Pi_{\Delta}^+$, if $p \in COMPS(\Pi) - \Delta$ and $(p(s) \leftarrow G) \in \Pi$ then $(p(s) \leftarrow G) \in \Pi_{\Delta}^-/\Pi_{\Delta}^+$.
- (2) If $f = ab$ then by the definition of $LSD^+(\Pi)$ ($f = ok$ for all clauses of $LSD^-(\Pi)$), $p \in \Delta$ and $p(s) \leftarrow G$ is of the form $p(X)$. By the definition of Π_{Δ}^+ , if $p \in \Delta$ then $p(X) \in \Pi_{\Delta}^+$.

($\Leftarrow$)

Since $(p(s) \leftarrow G) \in \Pi_{\Delta}^-/\Pi_{\Delta}^+$ there are two cases to consider.

- (1) If $p \in COMPS(\Pi) - \Delta$ then by the definition of $\Pi_{\Delta}^-/\Pi_{\Delta}^+$, $(p(s) \leftarrow G) \in \Pi$. By the definition of $LSD^-(\Pi)/LSD^+(\Pi)$, if $(p(s) \leftarrow G) \in \Pi$ then $al((p(s) \leftarrow G), L) \in LSD^-(\Pi)/LSD^+(\Pi)$ where $f = ok$. Since $s(p, \Delta) = ok$ then $f = s(p, \Delta)$.
- (2) If $p \in \Delta$ then by the definition of Π_{Δ}^+ ($p \in COMPS(\Pi) - \Delta$ for all clauses of Π_{Δ}^-), $p(s) \leftarrow G$ is of the form $p(X)$. By the definition of $LSD^+(\Pi)$, $al(p(X), L) \in LSD^+(\Pi)$ where $f = ab$. Since $s(p, \Delta) = ab$ then $f = s(p, \Delta)$.

□

LEMMA VI.45. *Let Π be a logic program, and $\Delta \subseteq COMPS(\Pi)$. If $N_0, \dots, N_n$ are resolvents of an SLD-derivation of $LSD^-(\Pi) \cup \{N_0\}/LSD^+(\Pi) \cup \{N_0\}$ where N_0 is of the form $al(\leftarrow F, d(\Delta))$, $\leftarrow F \in L(\Pi)$, then for all $1 \leq i \leq n$, N_i are of the form $al(\leftarrow F_i, d(\Delta))$ where $\leftarrow F_i \in L(\Pi)$.*

PROOF. By induction on the length n of an SLD-derivation of $LSD^-(\Pi) \cup \{N_0\}/LSD^+(\Pi) \cup \{N_0\}$.

(Basis Step)

An SLD-derivation of length $n = 1$ of $LSD^-(\Pi) \cup \{N_0\}/LSD^+(\Pi) \cup \{N_0\}$ is $N_0 = al(\leftarrow F, d(\Delta))$ for some $\leftarrow F \in L(\Pi)$.

(Induction Hypothesis)

If $N_0, \dots, N_n$ are resolvents of an SLD-derivation of length $n > 1$ of $LSD^-(\Pi) \cup \{N_0\}/LSD^+(\Pi) \cup \{N_0\}$ where N_0 is of the form $al(\leftarrow F, d(\Delta))$, $\leftarrow F \in L(\Pi)$, then for all $1 \leq i \leq n$, N_i are of the form $al(\leftarrow F_i, d(\Delta))$ where $\leftarrow F_i \in L(\Pi)$.

(Induction Step)

Assume $N_0, \dots, N_n$ are resolvents of an SLD-derivation of length $n + 1$, $n > 1$, of $LSD^-(\Pi) \cup \{N_0\}/LSD^+(\Pi) \cup \{N_0\}$ where N_0 is of the form $al(\leftarrow F, d(\Delta))$,

$\leftarrow F \in L(\Pi)$. Now N_1 is a resolvent of N_0 and some clause of $\text{LSD}^-(\Pi)/\text{LSD}^+(\Pi)$. Let

$$N_0 = \text{al}((\leftarrow p_1(t_1), \dots, p_m(t_m)), d(\Delta)).$$

So by Lemma VI.43, a selected atom $p_k(d(\Delta), t_k)$, $1 \leq k \leq m$ of N_0 that resolves with a clause $\text{al}((p(s) \leftarrow G), L)$ of $\text{LSD}^-(\Pi)/\text{LSD}^+(\Pi)$ will produce a resolvent equivalent to $(\text{al}(\leftarrow G, d(\Delta)))\theta$, where $t_k\theta \equiv s\theta$.

So

$$N_1 = (\text{al}((\leftarrow p_1(t_1), \dots, p_{k-1}(t_{k-1}), G, p_{k+1}(t_{k+1}), \dots, p_m(t_m)), d(\Delta)))\theta.$$

Now, $N_1, \dots, N_n$ are the resolvents of an SLD-derivation of length $n > 1$ of $\text{LSD}^-(\Pi) \cup \{N_1\}/\text{LSD}^+(\Pi) \cup \{N_1\}$ where N_1 is of the form $\text{al}(\leftarrow F', d(\Delta))$, $\leftarrow F' \in L(\Pi)$. Therefore, by the induction hypothesis, for all $2 \leq i \leq n$, N_i are of the form $\text{al}(\leftarrow F_i, d(\Delta))$ where $\leftarrow F_i \in L(\Pi)$. Furthermore, since N_1 is a resolvent of N_0 and a clause of $\text{LSD}^-(\Pi)/\text{LSD}^+(\Pi)$, then $N_0, N_1, \dots, N_n$ are resolvents of an SLD-derivation of $\text{LSD}^-(\Pi) \cup \{N_0\}/\text{LSD}^+(\Pi) \cup \{N_0\}$ such that for all $0 \leq i \leq n$, N_i is of the form $\text{al}(\leftarrow F_i, d(\Delta))$ where $\leftarrow F_i \in L(\Pi)$. $\square$

LEMMA VI.46. *Let Π be a logic program, and $\Delta \subseteq \text{COMPS}(\Pi)$. If N_1 is a resolvent of N_0 and a clause of $\text{LSD}^-(\Pi)/\text{LSD}^+(\Pi)$ and $N_0 = \text{al}(\leftarrow F, d(\Delta))$, $\leftarrow F \in L(\Pi)$ then $\text{rl}(N_1)$ is a resolvent of $\text{rl}(N_0)$ and a clause of $\Pi_{\Delta}^-/\Pi_{\Delta}^+$.*

PROOF. Let $\leftarrow F = \leftarrow p_1(t_1), \dots, p_n(t_n)$, $n > 0$.

Thus,

$$N_0 = \text{al}((\leftarrow p_1(t_1), \dots, p_n(t_n)), d(\Delta)).$$

Let N_1 be a resolvent of N_0 and a clause $C = \text{al}((p_k(s) \leftarrow G), L)$ of $\text{LSD}^-(\Pi)/\text{LSD}^+(\Pi)$, where $L = \text{gl}((p_k(s) \leftarrow G), p_k, f)$. For any $1 \leq k \leq n$, a selected atom of N_0 is of the form $p_k(d(\Delta), t_k)$. Therefore by Lemma VI.43, the resolvent of $\leftarrow p_k(d(\Delta), t_k)$ and C is equivalent to $(\text{al}(\leftarrow G, d(\Delta)))\theta$ where $f = s(p_k, \Delta)$ and $s\theta \equiv t_k\theta$.

Thus,

$$N_1 = \text{al}((\leftarrow p_1(t_1), \dots, p_{k-1}(t_{k-1}), G, p_{k+1}(t_{k+1}), \dots, p_n(t_n))\theta, d(\Delta)).$$

By Lemma VI.44, since $C \in \text{LSD}^-(\Pi)/\text{LSD}^+(\Pi)$ and $f = s(p_k, \Delta)$ then $(p_k(s) \leftarrow G) \in \Pi_{\Delta}^-/\Pi_{\Delta}^+$. Since $s\theta \equiv t_k\theta$ then an atom $p_k(t_k)$ of $\text{rl}(N_0)$ will unify with the head

of $p_k(s) \leftarrow G$. Therefore, $rl(N_0)$ and $(p_k(s) \leftarrow G) \in \Pi_{\Delta}^-/\Pi_{\Delta}^+$ resolve producing the resolvent

$$\leftarrow (p_1(t_1), \dots, p_{k-1}(t_{k-1}), G, p_{k+1}(t_{k+1}), \dots, p_n(t_n))\theta,$$

which is equal to $rl(N_1)$. $\square$

LEMMA VI.47. *Let Π be a logic program, and $\Delta \subseteq COMPS(\Pi)$. If N_1 is a resolvent of N_0 and a clause of $\Pi_{\Delta}^-/\Pi_{\Delta}^+$ and $N_0 = \leftarrow F$, $\leftarrow F \in L(\Pi)$ then $al(N_1, d(\Delta))$ is a resolvent of $al(N_0, d(\Delta))$ and a clause of $LSD^-(\Pi)/LSD^+(\Pi)$.*

PROOF. Let $N_0 = \leftarrow p_1(t_1), \dots, p_n(t_n)$, $n > 0$. Let N_1 be a resolvent of N_0 and a clause $C = (p_k(s) \leftarrow G)$ of $\Pi_{\Delta}^-/\Pi_{\Delta}^+$. Now, let $\leftarrow G\theta$ be the resolvent of a selected atom $p_k(t_k)$, $1 \leq k \leq n$, of N_0 and C with unifier θ such that $s\theta \equiv t_k\theta$.

Thus,

$$N_1 = \leftarrow (p_1(t_1), \dots, p_{k-1}(t_{k-1}), G, p_{k+1}(t_{k+1}), \dots, p_n(t_n))\theta.$$

Since $C \in \Pi_{\Delta}^-/\Pi_{\Delta}^+$, then by Lemma VI.44, $al(C, L) \in LSD^-(\Pi)/LSD^+(\Pi)$, where $L = gl(C, p_k, f)$, and $f = s(p_k, \Delta)$. Since

$$al(N_0, d(\Delta)) = al(\leftarrow p_1(t_1), \dots, p_n(t_n), d(\Delta)),$$

then a selected atom of $al(N_0, d(\Delta))$ would be of the form $p_k(d(\Delta), t_k)$ for any $1 \leq k \leq n$.

Since $f = s(p_k, \Delta)$ and $t_k\theta \equiv s\theta$ then by Lemma VI.43, $\leftarrow p_k(d(\Delta), t_k)$ and $al((p_k(s) \leftarrow G), L)$ resolve producing a resolvent equivalent to

$$al(\leftarrow (p_1(t_1), \dots, p_{k-1}(t_{k-1}), G, p_{k+1}(t_{k+1}), \dots, p_n(t_n))\theta, d(\Delta)),$$

which is equal to $al(N_1, d(\Delta))$. $\square$

LEMMA VI.48. *Let Π be a logic program, and $\Delta \subseteq COMPS(\Pi)$. If $N'_0, \dots, N'_n$ are the resolvents of an SLD-derivation of $LSD^-(\Pi) \cup \{N'_0\}/LSD^+(\Pi) \cup \{N'_0\}$ where $N'_0 = al(\leftarrow F, d(\Delta))$ for some $\leftarrow F \in L(\Pi)$ then $N_0, \dots, N_n$ are the resolvents of an SLD-derivation of $\Pi_{\Delta}^- \cup \{N_0\}/\Pi_{\Delta}^+ \cup \{N_0\}$ such that for all $0 \leq i \leq n$, $N_i = rl(N'_i)$.*

PROOF. By way of contradiction assume that $N_0, \dots, N_n$ are not the resolvents of an SLD-derivation of $\Pi_{\Delta}^- \cup \{N_0\}/\Pi_{\Delta}^+ \cup \{N_0\}$ where $N_i = rl(N'_i)$ for all $0 \leq i \leq n$. By the definition of an SLD-derivation there are two cases to consider.

- (1) Assume there exists some N_k , $0 \leq k \leq n$ such that N_k is not a resolvent of N_{k-1} and a clause C of $\Pi_{\Delta}^-/\Pi_{\Delta}^+$. Since $N'_0 = \text{al}(\leftarrow F, d(\Delta))$, $\leftarrow F \in L(\Pi)$, then by Lemma VI.45, the resolvent N'_{k-1} is of the form $\text{al}(\leftarrow G, d(\Delta))$, $\leftarrow G \in L(\Pi)$, for all $1 \leq k \leq n$. We also know that since N'_k is a resolvent of N'_{k-1} and a clause C' of $\text{LSD}^-(\Pi)/\text{LSD}^+(\Pi)$ and N'_{k-1} is of the form $\text{al}(\leftarrow G, d(\Delta))$, $\leftarrow G \in L(\Pi)$, then by Lemma VI.46 $\text{rl}(N'_k)$ is a resolvent of $\text{rl}(N'_{k-1})$ and a clause $\text{rl}(C')$ of $\Pi_{\Delta}^-/\Pi_{\Delta}^+$, for all $1 \leq k \leq n$. Since $N_k = \text{rl}(N'_k)$ and $N_{k-1} = \text{rl}(N'_{k-1})$ then for all $1 \leq k \leq n$, N_k is a resolvent of N_{k-1} and a clause of $\Pi_{\Delta}^-/\Pi_{\Delta}^+$. But this contradicts our initial assumption.
- (2) Assume $N_0, \dots, N_n$ is not a maximal sequence of resolvents. Therefore there must exist a resolvent N_{n+1} of N_n and a clause C of $\Pi_{\Delta}^-/\Pi_{\Delta}^+$. By Lemma VI.47, there exists a resolvent N'_{n+1} of N'_n and a clause C' of $\text{LSD}^-(\Pi)/\text{LSD}^+(\Pi)$. Therefore $N'_0, \dots, N'_n, N'_{n+1}$ are the resolvents of an SLD-derivation of $\text{LSD}^-(\Pi) \cup \{N'_0\}/\text{LSD}^+(\Pi) \cup \{N'_0\}$. But $N'_0, \dots, N'_n$ is a maximal sequence of resolvents which is a contradiction.

□

LEMMA VI.49. Let Π be a logic program, and $\Delta \subseteq \text{COMPS}(\Pi)$. If $N_0, \dots, N_n$ are the resolvents of an SLD-derivation of $\Pi_{\Delta}^- \cup \{N_0\}/\Pi_{\Delta}^+ \cup \{N_0\}$ where $N_0 = \leftarrow F$ for some $\leftarrow F \in L(\Pi)$ then $N'_0, \dots, N'_n$ are the resolvents of an SLD-derivation of $\text{LSD}^-(\Pi) \cup \{N'_0\}/\text{LSD}^+(\Pi) \cup \{N'_0\}$ such that for all $0 \leq i \leq n$, $N_i = \text{al}(N'_i, d(\Delta))$.

PROOF. By way of contradiction assume that $N'_0, \dots, N'_n$ are not the resolvents of an SLD-derivation of $\text{LSD}^-(\Pi) \cup \{N'_0\}/\text{LSD}^+(\Pi) \cup \{N'_0\}$ where $N'_i = \text{al}(N_i, d(\Delta))$ for all $0 \leq i \leq n$. By the definition of an SLD-derivation there are two cases to consider.

- (1) Assume there exists some N'_k , $0 \leq k \leq n$ such that N'_k is not a resolvent of N'_{k-1} and a clause C' of $\text{LSD}^-(\Pi)/\text{LSD}^+(\Pi)$. However, for all $1 \leq k \leq n$, since N_k is a resolvent of N_{k-1} and a clause C of $\Pi_{\Delta}^-/\Pi_{\Delta}^+$ then by Lemma VI.47, $\text{al}(N_k, d(\Delta))$ is a resolvent of $\text{al}(N_{k-1}, d(\Delta))$ and a clause C' of $\text{LSD}^-(\Pi)/\text{LSD}^+(\Pi)$. Since $N'_k = \text{al}(N_k, d(\Delta))$ and $N'_{k-1} = \text{al}(N_{k-1}, d(\Delta))$ then N'_k is a resolvent of N'_{k-1} and a clause of $\text{LSD}^-(\Pi)/\text{LSD}^+(\Pi)$, for all $1 \leq k \leq n$. But this contradicts our initial assumption.

- (2) Assume $N'_0, \dots, N'_n$ is not a maximal sequence of resolvents. Therefore there must exist a resolvent N'_{n+1} of N'_n and a clause C' of $\text{LSD}^-(\Pi)/\text{LSD}^+(\Pi)$. By Lemma VI.46, there exists a resolvent N_{n+1} of N_n and a clause C of $\Pi^-_{\Delta}/\Pi^+_{\Delta}$. Therefore $N_0, \dots, N_n, N_{n+1}$ are the resolvents of an SLD-derivation of $\Pi^-_{\Delta} \cup \{N_0\}/\Pi^+_{\Delta} \cup \{N_0\}$. But $N_0, \dots, N_n$ is a maximal sequence of resolvents which is a contradiction.

□

PROOF OF LEMMA VI.41.

($\Rightarrow$)

Since $\text{comp}(\Pi^-_{\Delta}) \models \forall \neg p(t)$ then by the completeness of the NAF rule for some selection rule, all SLD-derivations of $\Pi^-_{\Delta} \cup \{\leftarrow p(t)\}$ are finite and failed. Therefore, for the same selection rule, all SLD-derivations of $\text{LSD}^-(\Pi) \cup \{\leftarrow p(d(\Delta), t)\}$ are finite and failed by Lemma VI.48. Thus, by the soundness of the NAF rule, $\text{comp}(\text{LSD}^-(\Pi)) \models \forall \neg p(d(\Delta), t)$.

($\Leftarrow$)

Since $\text{comp}(\text{LSD}^-(\Pi)) \models \forall \neg p(d(\Delta), t)$, then by the completeness of the NAF rule for some selection rule, all SLD-derivations of $\text{LSD}^-(\Pi) \cup \{\leftarrow p(d(\Delta), t)\}$ are finite and failed. Therefore, for the same selection rule, all SLD-derivations of $\Pi^-_{\Delta} \cup \{\leftarrow p(t)\}$ are finite and failed by Lemma VI.49. Thus, by the soundness of the NAF rule, $\text{comp}(\Pi^-_{\Delta}) \models \forall \neg p(t)$. □

COROLLARY VI.50. Let Π be a logic program, $\exists p(t) \in L(\Pi)$ and $\Delta \subseteq \text{COMPS}(\Pi)$. $\text{comp}(\Pi^-_{\Delta}) \models \exists p(t)$ iff $\text{comp}(\text{LSD}^-(\Pi)) \models \exists p(d(\Delta), t)$.

PROOF. Follows *mutatis mutandis* from proof of Lemma VI.41. □

4.4.3. Soundness and Completeness Proofs for $\text{LSD}^-(\Pi)$

PROOF. (Theorem VI.17: Soundness—Denial For $\text{LSD}^-(\Pi)$)

By Corollary VI.50, we know that for some $\forall \neg p(t) \in \text{OBS}$, $\text{comp}(\Pi^-_{\Delta}) \models \exists p(t)$. Therefore $\text{comp}(\Pi^-_{\Delta}) \cup \text{OBS}$ is not satisfiable. By Lemma VI.39, $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is not a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$. □

PROOF. (Theorem VI.18: Completeness—Denial For $\text{LSD}^-(\Pi)$)

By Lemma VI.37, $\text{comp}(\Pi^-_{\Delta}) \cup \text{OBS}$ is not satisfiable. So for some $\forall \neg p(t) \in \text{OBS}$, $\text{comp}(\Pi^-_{\Delta}) \models \exists p(t)$ or for some $\exists p(t) \in \text{OBS}$, $\text{comp}(\Pi^-_{\Delta}) \models \forall \neg p(t)$. By

Corollary VI.50 and Lemma VI.41 it follows that for some $\forall \neg p(t) \in \text{OBS}$, $\text{comp}(\text{LSD}^-(\Pi)) \models \exists p(d(\Delta), t)$ or for some $\exists p(t) \in \text{OBS}$, $\text{comp}(\text{LSD}^-(\Pi)) \models \forall \neg p(d(\Delta), t)$. $\square$

PROOF. (Theorem VI.20: Soundness—Confirmation for $\text{LSD}^-(\Pi)$)

From Lemma VI.41 and Corollary VI.50 we know that for all $\forall \neg p(t) \in \text{OBS}$, $\text{comp}(\Pi_\Delta^-) \models \forall \neg p(t)$, and for all $\exists p(t) \in \text{OBS}$, $\text{comp}(\Pi_\Delta^-) \models \exists p(t)$. Therefore $\text{comp}(\Pi_\Delta^-) \cup \text{OBS}$ is satisfiable and by Lemma VI.37, $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$. $\square$

PROOF. (Theorem VI.21: Completeness—Confirmation For $\text{LSD}^-(\Pi)$)

Since $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ then for all $\forall \neg p(t) \in \text{OBS}$,

$$\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \{\forall \neg p(t)\} \text{ is satisfiable.}$$

Therefore, for all $\forall \neg p(t) \in \text{OBS}$, $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \{\forall \neg p(t)\})$. So by Lemma VI.38, for all $\forall \neg p(t) \in \text{OBS}$, $\text{comp}(\Pi_\Delta^-) \models \forall \neg p(t)$. Therefore, for all $\forall \neg p(t) \in \text{OBS}$, $\text{comp}(\text{LSD}^-(\Pi)) \models \forall \neg p(d(\Delta), t)$ by Lemma VI.41. $\square$

4.4.4. Supporting Lemmas For $\text{LSD}^+(\Pi)$

In this section we present the supporting theorems and lemmas required for the proofs of diagnostic soundness and completeness using $\text{LSD}^+(\Pi)$. An overview of the theorems and lemmas required is shown in Figure VI.14. The proofs of the soundness and completeness properties themselves are included in the next section.

Lemmas VI.51 and VI.52 formalize the relationship between a system description $\text{SD}(\Pi)$ and a positive Δ -transform Π_Δ^+ . Lemma VI.54 formalizes the relationship between a positive Δ -transform Π_Δ^+ and a positive labelled system description $\text{LSD}^+(\Pi)$. Note that the proof of Lemma VI.54 relies on the same lemmas that were used to prove Lemma VI.41 for the case of negative assertions.

LEMMA VI.51. *Let Π be a logic program, OBS be a set of sentences of $L(\Pi)$ and $\Delta \subseteq \text{COMPS}(\Pi)$. If $\text{comp}(\Pi_\Delta^+) \cup \text{OBS}$ is satisfiable then $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$.*

FIGURE VI.14. Overview of relationship between theorems and supporting lemmas for positive assertions

PROOF. Since $comp(\Pi_{\Delta}^+) \cup OBS$ is satisfiable, then by Lemma VI.36(b) and the compactness of FOPC, $comp_{\Delta}(\Pi) \cup OBS$ is satisfiable. Since ab is not a predicate in Π then

$$comp_{\Delta}(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS \text{ is satisfiable.}$$

So by Lemma V.17 we know

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS$$

is satisfiable. Therefore $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$. $\square$

LEMMA VI.52. Let Π be a logic program, $\exists p(t) \in L(\Pi)$, $OBS = \{\exists p(t)\}$, and $\Delta \subseteq COMPS(\Pi)$. If $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi),$

OBS) and $\Pi_{\Delta}^+ \cup \{\leftarrow p(t)\}$ has no infinite SLD-derivations for some selection rule then $\text{comp}(\Pi_{\Delta}^+) \models \exists p(t)$.

In order to prove Lemma VI.52 we require the following lemma.

LEMMA VI.53. *Let Π be a logic program, *OBS* be a set of positive assertions and $\Delta \subseteq \text{COMPS}(\Pi)$. If $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), \text{COMPS}(\Pi), \text{OBS})$ then $\text{comp}(\Pi_{\Delta}^+) \cup \text{OBS}$ is satisfiable.*

PROOF. The proof follows *mutatis mutandis* from the proof of Lemma VI.39. $\square$

PROOF OF LEMMA VI.52. Since $\Pi_{\Delta}^+ \cup \{\leftarrow p(t)\}$ has no infinite SLD-derivations for some selection rule then by the soundness of the NAF rule and SLD-resolution either $\text{comp}(\Pi_{\Delta}^+) \models \forall \neg p(t)$ or $\text{comp}(\Pi_{\Delta}^+) \models \exists p(t)$. The former case cannot be true by Lemma VI.53. Therefore $\text{comp}(\Pi_{\Delta}^+) \models \exists p(t)$. $\square$

We now proceed to formalize the relationship between the positive Δ -transform Π_{Δ}^+ and the positive labelled system description $\text{LSD}^+(\Pi)$. This relationship is stated by the following lemma.

LEMMA VI.54. *Let Π be a logic program, $\exists p(t) \in L(\Pi)$ and $\Delta \subseteq \text{COMPS}(\Pi)$. $\text{comp}(\Pi_{\Delta}^+) \models \exists p(t)$ iff $\text{comp}(\text{LSD}^+(\Pi)) \models \exists p(d(\Delta), t)$.*

PROOF.

($\Rightarrow$)

Since $\text{comp}(\Pi_{\Delta}^+) \models \exists p(t)$ then by the completeness of SLD-resolution there exists an SLD-refutation for $\Pi_{\Delta}^+ \cup \{\leftarrow p(t)\}$. Therefore, by Lemma VI.48, there exists an SLD-refutation of $\text{LSD}^+(\Pi) \cup \{\leftarrow p(d(\Delta), t)\}$. Thus, by the soundness of SLD-resolution, $\text{comp}(\text{LSD}^+(\Pi)) \models \exists p(d(\Delta), t)$.

($\Leftarrow$)

Since $\text{comp}(\text{LSD}^+(\Pi)) \models \exists p(d(\Delta), t)$ then by the completeness of SLD-resolution there exists an SLD-refutation for $\text{LSD}^+(\Pi) \cup \{\leftarrow p(d(\Delta), t)\}$. Therefore, by Lemma VI.49, there exists an SLD-refutation for $\Pi_{\Delta}^+ \cup \{\leftarrow p(t)\}$. Thus, by the soundness of SLD-resolution, $\text{comp}(\Pi_{\Delta}^+) \models \exists p(t)$. $\square$

COROLLARY VI.55. *Let Π be a logic program, $\forall \neg p(t) \in L(\Pi)$ and $\Delta \subseteq \text{COMPS}(\Pi)$. $\text{comp}(\Pi_{\Delta}^+) \models \forall \neg p(t)$ iff $\text{comp}(\text{LSD}^+(\Pi)) \models \forall \neg p(d(\Delta), t)$.*

PROOF. Follows *mutatis mutandis* from proof of Lemma VI.54. $\square$

4.4.5. Soundness and Completeness Proofs For $LSD^+(\Pi)$

PROOF. (Theorem VI.23: Soundness—Denial for $LSD^+(\Pi)$)

By Corollary VI.55, we know that for some $\exists p(t) \in OBS$, $comp(\Pi_\Delta^+) \models \forall \neg p(t)$. Therefore $comp(\Pi_\Delta^+) \cup OBS$ is not satisfiable. By Lemma VI.53, $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is not a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$. $\square$

PROOF. (Theorem VI.24: Completeness—Denial For $LSD^+(\Pi)$)

By Lemma VI.51, $comp(\Pi_\Delta^+) \cup OBS$ is not satisfiable. So for some $\exists p(t) \in OBS$, $comp(\Pi_\Delta^+) \models \forall \neg p(t)$ or for some $\forall \neg p(t) \in OBS$, $comp(\Pi_\Delta^+) \models \exists p(t)$. By Corollary VI.55 and Lemma VI.54 it follows that for some $\exists p(t) \in OBS$, $comp(LSD^+(\Pi)) \models \forall \neg p(d(\Delta), t)$ or for some $\forall \neg p(t) \in OBS$, $comp(LSD^+(\Pi)) \models \exists p(d(\Delta), t)$. $\square$

PROOF. (Theorem VI.26: Soundness—Confirmation for $LSD^+(\Pi)$)

From Lemma VI.54 and Corollary VI.55 we know that for all $\exists p(t) \in OBS$, $comp(\Pi_\Delta^+) \models \exists p(t)$ and for all $\forall \neg p(t) \in OBS$, $comp(\Pi_\Delta^+) \models \forall \neg p(t)$. Therefore $comp(\Pi_\Delta^+) \cup OBS$ is satisfiable and by Lemma VI.51, $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$. $\square$

PROOF. (Theorem VI.27: Completeness—Confirmation For $LSD^+(\Pi)$)

Since $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS)$ then for all $\exists p(t) \in OBS$,

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup \{\exists p(t)\} \text{ is satisfiable.}$$

Therefore, for all $\exists p(t) \in OBS$, $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), \{\exists p(t)\})$. So by Lemma VI.52, for all $\exists p(t) \in OBS$, $comp(\Pi_\Delta^+) \models \exists p(t)$. Therefore, for all $\exists p(t) \in OBS$, $comp(LSD^+(\Pi)) \models \exists p(d(\Delta), t)$ by Lemma VI.54. $\square$

5. Diagnosis Computation

In this section we describe how the approach to diagnosis verification we have defined in the previous sections can be used for computing minimal diagnoses. We present examples of diagnosis computation and discuss how one can determine the type of fault associated with abnormalities computed by the procedure. In addition we discuss the complexity and efficiency of the approach.

5.1. Using a Generate-and-Test Approach. Having determined the soundness, completeness and decidability of this approach to diagnosis verification, we can now incorporate the approach into a diagnosis computation procedure. We use a generate-and-test computation procedure, where a diagnosis verification procedure plays the role of the tester (see Chapter IV). We use the generator algorithm employed by the IDA procedure in [65]. Since the generator algorithm calls the tester via a ternary TP diagnosis verification function, we now define the appropriate functions for our own tester.

Let Π be a logic program, OBS be a finite set of positive assertions, and $\Delta \subseteq \text{COMPS}(\Pi)$.

$$\text{TP}^+(\Pi, \text{OBS}, \Delta) = \begin{cases} \text{'true' if } \text{comp}(\text{LSD}^+(\Pi)) \models \exists p(d(\Delta), t) \\ \text{for all } \exists p(t) \in \text{OBS}, \\ \text{'false' if } \text{comp}(\text{LSD}^+(\Pi)) \models \forall \neg p(d(\Delta), t) \\ \text{for some } \exists p(t) \in \text{OBS}. \end{cases}$$

Let Π be a logic program, OBS be a finite set of negative assertions, and $\Delta \subseteq \text{COMPS}(\Pi)$.

$$\text{TP}^-(\Pi, \text{OBS}, \Delta) = \begin{cases} \text{'true' if } \text{comp}(\text{LSD}^-(\Pi)) \models \forall \neg p(d(\Delta), t) \\ \text{for all } \forall \neg p(t) \in \text{OBS}, \\ \text{'false' if } \text{comp}(\text{LSD}^-(\Pi)) \models \exists p(d(\Delta), t) \\ \text{for some } \forall \neg p(t) \in \text{OBS}. \end{cases}$$

Note that the functions TP^+ and TP^- simply provide an interface for the generator algorithm. By the soundness theorems for diagnosis confirmation (Theorems VI.26 and VI.20), if $\text{TP}^+(\Pi, \text{OBS}, \Delta) = \text{'true'}$ or $\text{TP}^-(\Pi, \text{OBS}, \Delta) = \text{'true'}$ then $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$. Similarly, by the soundness theorems for diagnosis denial (Theorems VI.23 and VI.17), if $\text{TP}^+(\Pi, \text{OBS}, \Delta) = \text{'false'}$ or $\text{TP}^-(\Pi, \text{OBS}, \Delta) = \text{'false'}$ then $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is not a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$. Furthermore, as described by the decidability theorems (Theorems VI.31 and VI.32, and Theorems VI.29 and VI.30) the functions TP^+ and TP^- are partially decidable for the purposes of diagnosis denial and are totally decidable for the appropriate class of logic programs Π and assertions OBS.

Now, to compute the minimal diagnoses of a system $(SD(\Pi), COMPS(\Pi), OBS)$ where OBS is a set of positive assertions, one interfaces the IDA diagnosis generation algorithm to TP^+ . When OBS is a set of negative assertions one interfaces the IDA diagnosis generation algorithm to TP^- .

Note that only in the case when TP^+ or TP^- are totally decidable is the diagnosis computation procedure guaranteed to terminate. On the other hand, the soundness properties of TP^+ and TP^- combined with the correctness of the generator algorithm ensure that whenever the diagnosis computation procedure does terminate, the resulting set of minimal diagnoses generated are minimal diagnoses for a system $(SD(\Pi), COMPS(\Pi), OBS)$.

EXAMPLE VI.56. Take Π to be the full-adder specification given in Program V.1 and $OBS = \{fullAdder(1, 0, 1, 1, 0)\}$. Since this assertion is positive then we can use the IDA generation procedure interfaced to TP^+ to generate the minimal diagnoses for the system $(SD(\Pi), COMPS(\Pi), OBS)$. The minimal diagnoses computed using this procedure are the same as those defined in Example V.3.

EXAMPLE VI.57. Assume Π is the buggy ISORT Program III.2. In Example III.20 we illustrate how a declarative error diagnosis procedure is used to locate a fault causing missing behavior in the buggy ISORT program. If we take the set of assertions to reflect the responses provided by the oracle in that example we have

$$OBS = \{isort([2, 1, 3], [1, 2, 3]), isort([1, 3], [1, 3]), insert(2, [1, 3], [1, 2, 3])\}.$$

Since all assertions are positive then we can use the IDA generation procedure interfaced to TP^+ to generate the minimal diagnoses for the system $(SD(\Pi), COMPS(\Pi), OBS)$. The only minimal diagnosis computed is $\mathcal{D}(\{insert\}, \{isort\})$ which agrees with the result returned by the declarative error diagnosis procedure.

5.2. Properties of Abnormalities. Recall from the previous chapter that abnormal components in minimal diagnoses defined for a system $(SD(\Pi), COMPS(\Pi), OBS)$ are either incorrect or incomplete by Theorems V.11 and V.12. These theorems can be used to categorize abnormal components in minimal diagnoses returned by our diagnosis computation procedure. In particular, Theorem V.11 applies to all minimal diagnoses computed for systems $(SD(\Pi), COMPS(\Pi), OBS)$

where OBS is a set of positive assertions and Theorem V.12 applies to all minimal diagnoses computed where OBS is a set of negative assertions.

EXAMPLE VI.58. In Example VI.56, the abnormal components in all computed minimal diagnoses must be incorrect by Theorem V.11. In Example VI.57, the *insert* predicate must be incomplete by Theorem V.11 (recall that because $SD(\Pi)$ is an artifact system description in this case, then the meanings of incorrect and incomplete are reversed in Theorems V.7, V.11 and V.12). This is also how declarative error diagnosis characterizes the faulty predicate in Example III.20.

5.3. Complexity and Efficiency. In [65], it is proven that the computation of the next minimal diagnosis takes polynomial time using the IDA approach (see Chapter IV). However, this applies only if calls to the TP function take constant time. Although the complexity of SLD-resolution and the negation-as-failure rule are undecidable in general, the complexity for a number of sub-classes of logic programs is decidable—ranging from linear time to exponential (*e.g.* [26]). As a result, assuming calls to TP^+ and TP^- take constant time is not an unreasonable assumption to make in many cases.

SLD-resolution and the negation-as-failure rule are used to implement TP^+ and TP^- in order to determine the success or failure of a goal in the logic programs $LSD^+(\Pi)$ and $LSD^-(\Pi)$. Using a Prolog compiler it is possible to compile the programs so that calls to TP^+ and TP^- are more efficient. However by compiling the programs, Prolog's search strategy and (unfair) selection rule must be used thereby potentially compromising the completeness of diagnosis verification.

Since the implementations of both procedures interpret logic programs, the efficiency of both approaches can potentially be improved by partially evaluating Π with respect to the sets of assertions (*e.g.* [58]) before constructing $LSD^+(\Pi)$ or $LSD^-(\Pi)$. Also, more expressive and concise logic programs can be written using a constraint-based logic programming language (*e.g.* [48]). Although their interpretation is no longer strictly in terms of SLD-resolution, the semantics of these languages can be viewed as the same as for logic programs. As such, they are suitable for the diagnostic computation procedures.

With regards to applying the procedures to "real-world" examples, we are optimistic. The generate-and-test approach to diagnosis used by the IDA procedure [65], upon which our procedures are based, has been shown to be superior to

probabilistically-guided search techniques used by ATMS-based approaches (e.g. [20]) and in [66], Mozetič reports that the IDA approach is also superior to sophisticated critical set procedures [9]. For example, the IDA procedure has been used to compute the first 5 diagnoses for a 5000 component combinational ripple adder circuit in only 4 CPU seconds.

Mozetič notes that the majority of diagnosis computation time is spent in the test procedure, not the generator algorithm and therefore emphasis should be placed on reducing the cost of calls to the former. He claims that model abstraction, efficient constraint propagation and model compilation, each yielded at least a 10-fold improvement in diagnostic efficiency. Since research in logic programming encompasses all of these areas, we are able to use results obtained in these areas to further increase the efficiency of our own diagnosis procedures.

5.4. Expressiveness of Assertions. A couple of tricks can be employed to extend the effective expressiveness of the class of positive assertions accepted by the diagnosis verification function TP^+ . Apart from existentially quantified atoms, universally quantified atoms can also be used indirectly. This is done by including an atom in the set of assertions that corresponds to the universally quantified atom with its variables replaced by Skolem constants. For example, to use the assertion $\forall p(\mathbf{X}^m)$, one should include the atom $p(\sigma_1, \dots, \sigma_m)$ in the set of assertions where σ_i is a Skolem constant. This ensures that the diagnosis verification function TP^+ retains its soundness and completeness properties (see Corollary II.15).

It is also possible to indirectly use assertions of the form $\exists(A_1 \wedge \dots \wedge A_n)$. To do this, one includes $\exists \rho(\mathbf{X}^m)$ in the set of assertions and introduces a supplementary clause to the logic program Π of the form

$$\rho(\mathbf{X}^m) \leftarrow A_1, \dots, A_n$$

where ρ is a predicate not mentioned in $L(\Pi)$, and $\{X_1, \dots, X_m\} = \text{vars}(A_1 \wedge \dots \wedge A_n)$. In order to use assertions of the form $\forall(A_1 \wedge \dots \wedge A_n)$ we combine the two techniques for extending the expressiveness of the set of assertions. The same supplementary clause is introduced to Π as just described but instead of including $\exists \rho(\mathbf{X}^m)$ in the set of assertions one includes its Skolem standard form, $\rho(\sigma_1, \dots, \sigma_m)$.

6. Comparison To Related Work

6.1. IDA. Earlier in this chapter we pointed out that system descriptions in the IDA framework [65] cannot represent hierarchically structured systems in general, or software systems in particular. Furthermore we noted that the IDA procedure only defines diagnoses with respect to positive assertions whereas our own procedure can compute diagnoses for sets of either positive or negative assertions. However, despite the differences between the two frameworks, there are conditions under which diagnoses in the IDA framework correspond to consistency-based diagnoses for a class of systems $(SD(\Pi), COMPS(\Pi), OBS)$.

It is possible to view system descriptions SD in the IDA framework as transformations from a logic program Π in the same way a labelled system description $LSD^+(\Pi)$ is a transformation from a logic program Π . Viewed this way the IDA labelling scheme restricts system descriptions to transformations from logic programs Π with only single level structural hierarchies. For example, it is not possible to consider the *fullAdder* component and its subcomponents together as fault candidates since they reside across two levels of the system's structural hierarchy. Furthermore, constructing a system description for the buggy ISORT program (Program III.2) would not be possible due to the existence of a recursive structural hierarchy.

Now, consider a system description SD in the IDA framework whose components are restricted to being labelled as either *ok* or *ab* (no fault models) and assume that SD is a transformation from some logic program Π . It is clear that SD is equivalent to the positive labelled system description $LSD^+(\Pi)$ for the purposes of verifying diagnoses with respect to positive assertions. As such, diagnoses obtained for this class of system descriptions are, by the soundness theorems (Theorems VI.26 and VI.23), diagnoses for the system $(SD(\Pi), COMPS(\Pi), OBS)$ where OBS is a finite set of positive assertions. Therefore for this class of system descriptions a diagnosis defined by the IDA framework corresponds to a consistency-based diagnosis.

6.2. Declarative Error Diagnosis. To reiterate from the introduction to this chapter, diagnostic procedures for declarative error diagnosis require an oracle, which is equivalent to possessing complete knowledge of the specified behavior of an artifact or unrestricted observability of an artifact, depending on the type of system description being used. When an oracle is available then using a declarative

error diagnosis procedure is an efficient way to identify abnormal components in a system, one at a time.

On the other hand, when an oracle is not available the computation procedure we define in this chapter for systems (SD(Π), COMPS(Π), OBS) is preferable since it computes diagnoses based only on a given set of assertions—no other information is required.

7. Summary

In this chapter we have developed a consistency-based diagnosis computation procedure for systems (SD(Π), COMPS(Π), OBS) where OBS is either a set of positive or negative assertions. The procedure is proven to be sound and conditions are identified for which it is also complete. The procedure is able to calculate the next diagnosis in polynomial time and can be implemented efficiently in Prolog. Abnormal components computed by the procedure can be classified as either incorrect or incomplete to give some indication as to the nature of the component failure. By using Skolem constants in place of variables in positive assertions or by using supplementary clauses it is possible to effectively extend the class positive assertions accepted by the computation procedure to universally or existentially quantified conjunctions of atoms. The computation procedure is also shown to be a generalized instance of that used by the IDA framework and, as such, we identify conditions for which IDA-defined diagnoses are also consistency-based diagnoses. Although the procedure presented in this chapter is an efficient approach to diagnosis computation, one potential disadvantage is that it does not accept sets of assertions consisting of both positive and negative assertions. However, in the next chapter a procedure, albeit a more complicated procedure, is developed for just this class of assertions.

CHAPTER VII

Computing Diagnoses For A More General Class of Assertions

1. Introduction

In the previous chapter we developed algorithms that compute the diagnoses for systems $(SD(\Pi), COMPS(\Pi), OBS)$ where OBS is restricted to either positive or negative assertions. While these algorithms provide an efficient approach to computing diagnoses, there are limitations associated with restricting the classes of assertions in this way. The following is an example of one such limitation.

EXAMPLE VII.1. Recall from Example V.3 that the following minimal diagnoses were obtained for the system $(SD(\Pi), COMPS(\Pi), \{fullAdder(1, 0, 1, 1, 0)\})$, where Π is Program V.1 representing the specified structure and behavior of a combinational full-adder circuit:

$$\begin{aligned} &\mathcal{D}(\{fullAdder\}, \{x1, a1, x2, a2, o1\}), \\ &\mathcal{D}(\{x1\}, \{fullAdder, a1, a2, x2, o1\}), \\ &\mathcal{D}(\{x2, o1\}, \{fullAdder, x1, a1, a2\}), \\ &\mathcal{D}(\{x2, a2\}, \{fullAdder, x1, a1, o1\}). \end{aligned}$$

Now assume that it is possible to observe the behavior of all the subcomponents of the full-adder artifact and further assume that we find they exhibit their normal

behavior. The following set OBS of positive assertions reflects this fact:

$$\left\{ \begin{array}{l} \text{fullAdder}(1, 0, 1, 1, 0), \\ x1(1, 0, 1), x1(0, 1, 1), x1(0, 0, 0), x1(1, 1, 0), \\ a1(0, 0, 0), a1(1, 0, 0), a1(0, 1, 0), a1(1, 1, 1), \\ a2(0, 0, 0), a2(1, 0, 0), a2(0, 1, 0), a2(1, 1, 1), \\ x2(0, 0, 0), x2(1, 0, 1), x2(0, 1, 1), x2(1, 1, 0), \\ o1(0, 0, 0), o1(1, 0, 1), o1(0, 1, 1), o1(1, 1, 1) \end{array} \right\}.$$

Intuitively, one might think that such an observation would implicate the full-adder component itself as the only possible fault. However, for the system $(SD(\Pi), COMPS(\Pi), OBS)$ the same minimal diagnoses are defined as for the system $(SD(\Pi), COMPS(\Pi), \{\text{fullAdder}(1, 0, 1, 1, 0)\})$.¹ In order to eliminate possible minimal diagnoses, the set of assertions should include negative assertions as well. For example, for the system

$$(SD(\Pi), COMPS(\Pi), \{\text{fullAdder}(1, 0, 1, 1, 0), \neg x1(1, 0, 0), \forall X \neg o1(1, X, 0)\}),$$

the only minimal diagnoses defined are

$$\begin{aligned} & \mathcal{D}(\{x2, a2\}, \{\text{fullAdder}, x1, a1, o1\}), \text{ and} \\ & \mathcal{D}(\{\text{fullAdder}\}, \{x1, a1, x2, a2, o1\}). \end{aligned}$$

The advantage gained by permitting OBS to consist of both positive and negative assertions is that it provides the diagnostician with more expressive power and helps to eliminate diagnoses that would otherwise be computed using either negative or positive assertions alone. In this chapter we present a procedure for verifying diagnoses for systems of the form $(SD(\Pi), COMPS(\Pi), OBS)$ where OBS is a set of both positive or negative assertions i.e. OBS consists of both existentially quantified positive literals or universally quantified negative literals. In order to compute all the minimal diagnoses of a system $(SD(\Pi), COMPS(\Pi), OBS)$, the

¹This seemingly counter-intuitive result is not an artifact of the diagnostic theory but due to the lack of a closure constraint on the domain of the system description. Such a constraint could be stated as follows: $\forall X (X = 1 \vee X = 0)$. However this constraint is non-Horn and therefore cannot be included in a logic program Π upon which a system description $SD(\Pi)$ is based. A possible solution to this problem would be to use a constraint-based logic programming language that implicitly supports such closure constraints on variables with finite domains.

verification procedure is interfaced to the IDA diagnosis generation algorithm [65] in the same way the verification algorithms TP^+ and TP^- of the previous chapter were. The resulting procedure computes the next diagnosis in polynomial time.

In [14], Console, Friedrich and Dupré present a generalized approach to declarative error diagnosis of logic programs consisting of a model-based diagnostic framework and a procedure for computing the diagnoses defined by the framework. Their procedure accepts assertions in the form of universally quantified positive literals or universally quantified negative literals. However, as we later show, the procedure presented in this chapter has better fault coverage than their procedure i.e. there exist diagnoses which their procedure will not compute but the procedure presented here will. The procedure presented here also computes diagnoses for a larger class of assertions.

The following is an overview of the remainder of this chapter. We first present the theory underlying the verification procedure in order to prove it sound for both diagnosis confirmation and diagnosis denial. We also discuss issues related to the completeness of a procedure based on the given theory. Next we present the verification procedure and its implementation and prove the implementation correct with respect to the underlying theory. As our presentation proceeds we carry along the examples of the full-adder circuit and the insertion sort algorithm to illustrate various concepts. We conclude by comparing the resulting diagnosis computation procedure to the procedure proposed in [14].

2. Positive Δ -Supports

Initially we define the notion of a Δ -literal.

DEFINITION VII.2 (Δ -LITERAL). Let Π be a logic program and $\Delta \subseteq \text{COMPS}(\Pi)$. A literal B belonging to $L(\Pi)$ is a Δ -literal iff the predicate symbol of B belongs to Δ .

The basis for our diagnosis verification procedure is the notion of a positive Δ -support which we now define.

DEFINITION VII.3 (POSITIVE Δ -SUPPORT). Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$. Let $\text{OBS}^+ = \{A_1, \dots, A_n\}$ be a set of positive assertions. Let D be a sentence of the form $\exists(B_1 \wedge \dots \wedge B_k)$, $k \geq 1$, where each B_i , $1 \leq i \leq k$, is a

positive Δ -literal or the propositional constant true. D is a *positive Δ -support* for OBS^+ with respect to Π iff

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models D \rightarrow (A_1 \wedge \dots \wedge A_n).$$

Informally speaking, a positive Δ -support can be viewed as a kind of abductive explanation (e.g. [78]) for the set of positive assertions OBS^+ where the facts are $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\}$, and the possible hypotheses are existentially quantified conjuncts of positive Δ -literals.² In this context, a positive Δ -support serves as a constraint on the behavior of faulty components that explains the system behavior specified by the positive assertions.

EXAMPLE VII.4. As an example, let Π denote Program V.1, the full-adder circuit specification. For $OBS^+ = \{fullAdder(1,0,1,1,0)\}$ and $\Delta = \{x1, a2, o1\}$ we have

$$\begin{aligned} SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models \\ \exists X(x1(1,0,0) \wedge a2(1,0,X) \wedge o1(X,0,0)) \rightarrow fullAdder(1,0,1,1,0). \end{aligned}$$

As such, $\exists X(x1(1,0,0) \wedge a2(1,0,X) \wedge o1(X,0,0))$ is a positive Δ -support for OBS^+ with respect to Π .

The notion of a positive Δ -support serves as the basis for a sufficient condition for diagnosis confirmation.

THEOREM VII.5 (SOUNDNESS DIAGNOSIS CONFIRMATION). *Let Π be a logic program, $\Delta \subseteq COMPS(\Pi)$. Let $OBS^+ = \{A_1, \dots, A_k\}$, $OBS^- = \{B_1, \dots, B_l\}$ be sets of positive and negative assertions, respectively. Let D be a positive Δ -support for OBS^+ with respect to Π . If $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup \{D\} \cup OBS^-$ is satisfiable then $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS^+ \cup OBS^-)$.*

²In Poole's Theorist framework [77] the user provides a set of closed formulas F , called the *facts*, and a set of open formulas H , called the *possible hypotheses*. A *scenario* is a set $D \cup F$ where D is a set of instances of elements of H such that $D \cup F$ is consistent. An *explanation* of a formula G is a scenario that logically implies G . Note that in the Theorist framework, the sentences of D are ground whereas in our framework positive Δ -supports are, in general, not ground.

PROOF. We are given that $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup \{D\} \cup OBS^-$ is satisfiable. By the definition of a positive Δ -support we have

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models D \rightarrow (A_1 \wedge \dots \wedge A_n).$$

Therefore,

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup \{D\} \cup OBS^+ \cup OBS^- \text{ is satisfiable.}$$

By the compactness of FOPC,

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS^+ \cup OBS^- \text{ is satisfiable.}$$

Thus, by the definition of a diagnosis, $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS^+ \cup OBS^-)$. $\square$

So, to confirm that $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis, one must find a positive Δ -support that satisfies the aforementioned condition.

EXAMPLE VII.6. Let Π denote Program III.2, the buggy insertion sort program. From Example III.18, the assertions evoked from the oracle by Shapiro's *fp* algorithm when diagnosing this program are

$$OBS^+ = \{isort([], []), insert(3, [], [3]), isort([3], [3]), insert(1, [], [1])\}, \text{ and}$$

$$OBS^- = \{\neg isort([2, 1, 3], [2, 3, 1]), \neg insert(1, [3], [3, 1])\}$$

For $\Delta = \{insert\}$, $insert(1, [], [1]) \wedge insert(3, [], [3])$ is a positive Δ -support for OBS^+ with respect to Π . Since

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup \{insert(1, [], [1]) \wedge insert(3, [], [3])\} \cup OBS^-$$

is satisfiable, then by the soundness confirmation theorem above, $\mathcal{D}(\{insert\}, \{isort\})$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS^+ \cup OBS^-)$ which agrees with the result obtained by Shapiro's algorithm.

To characterize a sufficient condition for diagnosis denial the notion of a complete set of positive Δ -supports is required.

DEFINITION VII.7 (COMPLETE SET OF POSITIVE Δ -SUPPORTS). Let Π be a logic program, $\Delta \subseteq COMPS(\Pi)$. Let $OBS^+ = \{A_1, \dots, A_k\}$ be a set of positive assertions. Let $D = \{D_1, \dots, D_m\}$, $m \geq 1$, be a set of positive Δ -supports for

OBS^+ with respect to Π . D is a *complete set of positive Δ -supports for OBS^+ with respect to Π* iff

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models (A_1 \wedge \dots \wedge A_n) \rightarrow (D_1 \vee \dots \vee D_m).$$

Therefore, if the positive assertions are logically implied by $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\}$ then at least one positive Δ -support from a complete set of positive Δ -supports must be logically implied by $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\}$. In this sense, a complete set of positive Δ -supports constitutes a closure condition on the possible behaviors of possibly faulty system components.

EXAMPLE VII.8. Let Π denote Program V.1, the full-adder specification. For $OBS^+ = \{fullAdder(1, 0, 1, 1, 0)\}$ and $\Delta = \{x1, x2, a1, o1\}$, the following set of two positive Δ -supports constitutes a complete set of positive Δ -supports for OBS^+ with respect to Π :

$$\left\{ \begin{array}{l} (\exists X(a1(1, 0, X) \wedge o1(0, X, 0) \wedge x1(1, 0, 0) \wedge x2(0, 1, 1))), \\ (\exists X(a1(1, 0, X) \wedge o1(0, X, 0) \wedge x1(1, 0, 1) \wedge x2(1, 1, 1))) \end{array} \right\}.$$

So to verify that a sentence $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is *not* a diagnosis for a system $(SD(\Pi), COMPS(\Pi), OBS^+ \cup OBS^-)$, we have the following sufficient condition for diagnosis denial.

THEOREM VII.9 (SOUNDNESS DIAGNOSIS DENIAL (I)). Let Π be a logic program, $\Delta \subseteq COMPS(\Pi)$. Let $OBS^+ = \{A_1, \dots, A_k\}$, $OBS^- = \{B_1, \dots, B_l\}$ be sets of positive and negative assertions respectively. Let $\{D_1, \dots, D_m\}$, $m \geq 1$, be a complete set of positive Δ -supports for OBS^+ with respect to Π . If for all D_i , $1 \leq i \leq m$,

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup \{D_i\} \cup OBS^-$$

is not satisfiable, then $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is not a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS^+ \cup OBS^-)$.

PROOF. We are given that for all D_i , $1 \leq i \leq m$,

$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup \{D_i\} \cup OBS^-$ is not satisfiable, so

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup \{D_i\} \models \neg B_1 \vee \dots \vee \neg B_l, \text{ or}$$

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models D_i \rightarrow (\neg B_1 \vee \dots \vee \neg B_l).$$

Therefore,

$$\begin{aligned} & \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models (D_1 \vee \dots \vee D_m) \rightarrow (\neg B_1 \vee \dots \vee \neg B_l), \text{ or} \\ & \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models (B_1 \wedge \dots \wedge B_l) \rightarrow \neg(D_1 \vee \dots \vee D_m), \text{ or} \\ \text{(VII.1)} \quad & \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \text{OBS}^- \models \neg(D_1 \vee \dots \vee D_m). \end{aligned}$$

However, by the definition of a complete set of positive Δ -supports,

$$\begin{aligned} & \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models (A_1 \wedge \dots \wedge A_k) \rightarrow (D_1 \vee \dots \vee D_m), \text{ or} \\ \text{(VII.2)} \quad & \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \text{OBS}^+ \models (D_1 \vee \dots \vee D_m). \end{aligned}$$

Therefore, by (VII.1) and (VII.2) we have

$$\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \text{OBS}^+ \cup \text{OBS}^- \text{ is not satisfiable.}$$

Thus, by the definition of a diagnosis, $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is not a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS}^+ \cup \text{OBS}^-)$. $\square$

Therefore, to deny that $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is a diagnosis, one only needs to identify a complete set of positive Δ -supports that satisfies the aforementioned condition.

EXAMPLE VII.10. To continue with the previous example, we find that for $\text{OBS}^- = \{\neg x1(1, 0, 0), \forall X \neg o1(1, X, 0)\}$,

$$\begin{aligned} & \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \\ & \quad \{\exists X (o1(1, 0, X) \wedge o1(0, X, 0) \wedge x1(1, 0, 0) \wedge x2(0, 1, 1))\} \cup \text{OBS}^- \end{aligned}$$

is not satisfiable, and

$$\begin{aligned} & \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \\ & \quad \{\exists X (o1(1, 0, X) \wedge o1(0, X, 0) \wedge x1(1, 0, 1) \wedge x2(1, 1, 1))\} \cup \text{OBS}^- \end{aligned}$$

is not satisfiable. Therefore, by the soundness denial theorem above, $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is not a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS}^+ \cup \text{OBS}^-)$.

Theorems VII.5 and VII.9 constitute sufficient conditions for the soundness of diagnosis confirmation and denial respectively. Unfortunately, the converse of each theorem does not hold. As such, identifying completeness conditions for diagnosis confirmation and denial in this context remains an open research problem. The

following are two counter-examples that disprove the converse of Theorems VII.5 and VII.9.

EXAMPLE VII.11 (INCOMPLETENESS DIAGNOSIS CONFIRMATION). Consider the logic program $\Pi = \{f(X) \leftarrow f(X)\}$. In this case, $\mathcal{D}(\emptyset, \{f\})$ is defined as a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \{\exists X f(X)\})$. However, no positive Δ -supports exist for $\{\exists X f(X)\}$ with respect to Π .

EXAMPLE VII.12 (INCOMPLETENESS DIAGNOSIS DENIAL). Consider the logic program

$$\Pi = \left\{ \begin{array}{l} b \leftarrow f(X), g(X). \\ c \leftarrow f(X), g(X). \\ f(0). \\ f(\text{succ}(X)) \leftarrow f(X). \\ g(X). \end{array} \right\}.$$

If $\text{OBS}^+ = \{b\}$ and $\text{OBS}^- = \{\neg c\}$, then $\mathcal{D}(\{g\}, \{b, c, f\})$ is not defined to be a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS}^+ \cup \text{OBS}^-)$. However, OBS^+ has an infinite number of positive Δ -supports for OBS^+ with respect to Π : $g(0), g(\text{succ}(0)), g(\text{succ}(\text{succ}(0))), \dots$. As such, a complete set of positive Δ -supports for OBS^+ with respect to Π does not exist.

3. The Verification Procedure

In this section we present a sound procedure to verify diagnoses for systems $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS})$ where OBS is a set of positive or negative assertions. The interface to the procedure takes the form of a quaternary function TP ("theorem prover," [82]) that takes as its arguments a logic program Π , sets of positive and negative assertions OBS^+ and OBS^- respectively, and a set $\Delta \subseteq \text{COMPS}(\Pi)$. The function is defined as follows:

$$\text{TP}(\Pi, \text{OBS}^+, \text{OBS}^-, \Delta) =$$

$$\left\{ \begin{array}{ll} \text{'true' if} & \mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta) \text{ is a diagnosis for} \\ & (\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS}^+ \cup \text{OBS}^-), \\ \text{'false' if} & \mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta) \text{ is not a diagnosis for} \\ & (\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS}^+ \cup \text{OBS}^-). \end{array} \right.$$

PROCEDURE VII.13 (TP PROCEDURE FOR DIAGNOSIS VERIFICATION).

```

    CS :=  $\emptyset$ 
     $\Delta$ -VALID := false
    CS-COMplete := false
  Step 1: if  $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models \neg A$  for all  $A \in OBS^+$  then
    repeat
  Step 2:   generate a positive  $\Delta$ -support  $S$  for  $OBS^+$  w.r.t.  $\Pi$ 
           CS :=  $CS \cup \{S\}$ 
  Step 3:   if CS a complete set of positive  $\Delta$ -supports for  $OBS^+$  w.r.t.  $\Pi$ 
           then CS-COMplete := true
  Step 4:   if  $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup \{S\} \cup OBS^-$  satisfiable
           then  $\Delta$ -VALID := true
    until  $\Delta$ -VALID or CS-COMplete
  endif
  return  $\Delta$ -VALID

```

Procedure VII.13, the procedure that implements TP, is based on the soundness confirmation and soundness denial theorems of the previous section. The procedure is sound for diagnosis confirmation since it only returns 'true' when the condition for soundness confirmation (Theorem VII.5) is met. There are two conditions for which the procedure can return 'false.' One condition corresponds to the condition set out in the soundness denial theorem (I) (Theorem VII.9). However, the procedure will also return 'false' when $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models \neg A$ for some $A \in OBS^+$. In such a situation, no positive Δ -supports exist for OBS^+ with respect to Π . Denying a diagnosis in this case is also sound, as shown by the following theorem.

THEOREM VII.14 (SOUNDNESS DIAGNOSIS DENIAL (II)). *Let Π be a logic program, $\Delta \subseteq COMPS(\Pi)$. Let OBS^+ and OBS^- be sets of positive and negative assertions respectively. If for some $A \in OBS^+$, $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models \neg A$ then $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is not a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS^+ \cup OBS^-)$.*

PROOF. Since $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models \neg A$ then $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS^+$ is not satisfiable. Therefore, $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup OBS^-$

$\Delta\} \cup \text{OBS}^+ \cup \text{OBS}^-$ is not satisfiable. Thus, $\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)$ is not a diagnosis for $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS}^+ \cup \text{OBS}^-)$. $\square$

Note that because completeness conditions have not yet been identified for diagnosis verification in this context, the procedure is not guaranteed to terminate in general.

4. Implementation

We now describe the implementation of Steps 1 through 4 of the diagnosis verification procedure and prove that the implementation is correct. In Appendix A, we present a simple Prolog meta-interpreter that implements the verification procedure as it is described here. We first describe a notation that is used in the following sections. Given a goal $N \leftarrow A_1, \dots, A_k$ we denote by $N^\sim$ the formula $A_1 \wedge \dots \wedge A_k$. Then $\square^\sim$ is the empty conjunction which we identify with true.

4.1. Implementation and Correctness of Step 1. The implementation of Step 1 is described by the following theorem. The theorem refers to the notion of the positive Δ -transform, Π_Δ^+ , of a logic program Π defined in the previous chapter.

THEOREM VII.15. *Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$, $\text{OBS}^+ = \{\exists A_1, \dots, \exists A_n\}$ be a set of positive assertions such that no pair of unique assertions shares variables. If the goal $\leftarrow A_1, \dots, A_n$ fails in Π_Δ^+ then $\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \neg \exists A_i$, for some $1 \leq i \leq n$.*

Thus, this step of the verification procedure consists of nothing more than a test for finite failure of a goal in a logic program. The implementation of this step is integrated with Steps 2 and 3 in the Prolog implementation of the procedure given in Appendix A. Note that this step will not terminate if the SLD-tree for this step has an infinite branch. To prove the implementation of this step correct we require the following lemma.

LEMMA VII.16. *Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$. If a goal D fails in Π_Δ^+ then $\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \neg \exists D^\sim$.*

PROOF. By the soundness of the NAF rule, $\text{comp}(\Pi_\Delta^+) \models \neg \exists D^\sim$. So by Lemma II.24 (VII.3)

$$\text{OIF}(\Pi_\Delta^+) \cup \text{EQ}(\Pi) \models \neg \exists D^\sim.$$

Now, $OIF(\Pi_{\Delta}^+)$ is equal to

$$\{\forall X(p(X) \rightarrow F) \in OIF(\Pi) | p \in COMPS(\Pi) - \Delta\} \\ \cup \{\forall X(p(X) \rightarrow \text{true}) | p \in \Delta\} \cup EQ(\Pi),$$

which is semantically equivalent to

$$(VII.4) \quad \{\forall X(p(X) \rightarrow F) \in OIF(\Pi) | p \in COMPS(\Pi) - \Delta\} \cup EQ(\Pi).$$

So by substitution of (VII.4) in (VII.3) we have

$$(VII.5) \quad \{\forall X(p(X) \rightarrow F) \in OIF(\Pi) | p \in COMPS(\Pi) - \Delta\} \cup EQ(\Pi) \models \neg \exists D \sim.$$

Since

$$(VII.6) \quad comp_{\Delta}(\Pi) = \{\forall X(p(X) \leftrightarrow F) \in IFF(\Pi) | p \in COMPS(\Pi) - \Delta\} \cup EQ(\Pi),$$

then by monotonicity of FOPC, (VII.5) and (VII.6) we have

$$comp_{\Delta}(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models \neg \exists D \sim.$$

Thus, by Lemma V.17 we have

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models \neg \exists D \sim.$$

□

The proof of Theorem VII.15 proceeds as follows.

PROOF. (Theorem VII.15)

Since the goal $\leftarrow A_1, \dots, A_n$ fails in Π_{Δ}^+ then by Lemma VII.16

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models \neg \exists (A_1 \wedge \dots \wedge A_n).$$

Since the assertions share no variables then

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models \neg (\exists A_1 \wedge \dots \wedge \exists A_n), \text{ or} \\ SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models \neg \exists A_1 \vee \dots \vee \neg \exists A_n.$$

Thus, $SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \models \neg \exists A_i$, for some $1 \leq i \leq n$. □

4.2. Implementation and Correctness of Steps 2 and 3. Steps 2 and 3 require the ability to compute a complete set of positive Δ -supports for a set of positive assertions with respect to a logic program. The implementation of Steps 2 and 3 is formally described by a theorem that will be presented momentarily. However, we must first define some terms used in the theorem.

DEFINITION VII.17 (Δ -GOAL). Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$. G is a Δ -goal iff $G = \leftarrow q_1(s_1), \dots, q_k(s_k)$, where $k \geq 1$ and $q_i \in \Delta$ for $1 \leq i \leq k$.

DEFINITION VII.18 (AN R_Δ SELECTION RULE). Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$, $\leftarrow Q$ be a goal and H be a derivation history of $\Pi \cup \{\leftarrow Q\}$ with last goal G . Define a selection rule R_Δ to be any selection rule that obeys the following constraint:

$$R_\Delta(H) = p(t) \text{ if } p(t) \text{ is a } \Delta\text{-literal in } G \text{ and } G \text{ consists only of } \Delta\text{-literals, or if } p(t) \text{ is a non-}\Delta\text{-literal of } G$$

Defined this way, a selection rule R_Δ simply delays the selection of Δ -literals in a derivation until the last goal consists only of Δ -literals.

We now present a theorem that describes how to compute a complete set of positive Δ -supports for a set of positive assertions with respect to a logic program. The proof of this theorem is left until the end of this section.

THEOREM VII.19. Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$. Let $\text{OBS}^+ = \{\exists A_1, \dots, \exists A_n\}$ be a set of positive assertions such that no unique pair of assertions shares variables. Let T denote an SLD-tree for $\Pi_\Delta^+ \cup \{\leftarrow A_1, \dots, A_n\}$ via R_Δ such that T contains no infinite branches. For each successful branch of T let D denote the first Δ -goal of the branch, or $\square$ if the branch contains no Δ -goals. If $D_1, \dots, D_m$, $m \geq 1$, represent all such goals in T then $\{\exists D_1^*, \dots, \exists D_m^*\}$ is a complete set of positive Δ -supports for OBS^+ with respect to Π .

Thus, a complete set of positive Δ -supports for a set of positive assertions is computed by searching an SLD-tree for particular goals. In particular, Step 2 of the verification procedure is accomplished by following a successful branch of the SLD-tree until a positive Δ -support is encountered. Step 3 of the procedure is accomplished when there are no more successful branches to follow. As shown in Appendix A, Steps 2 and 3 are easily implemented as a Prolog meta-interpreter.

$X \longrightarrow Y$
 indicates that X used in proof of Y

FIGURE VII.15. Overview of relationship between Theorem VII.19 and its supporting theorems and lemmas

Note that Step 2 will not terminate if the SLD-tree has an infinite branch and Step 3 will not terminate if the SLD-tree has infinite breadth. This is in accordance with the incompleteness of the approach discussed earlier.

At this point we develop the proof of Theorem VII.19. The proof for the theorem depends on a number of other theorems and lemmas as shown in Figure VII.15. The reader not interested in the proof may skip ahead to the next section where the implementation and correctness of Step 4 of the verification procedure is discussed. Referring to Figure VII.15 we proceed with the proof of the theorem in a bottom-up fashion.

The following lemma refers to the negative Δ -transform, Π_{Δ}^{-} , of a logic program Π defined in the previous chapter.

LEMMA VII.20. *Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$, $S \in L(\Pi)$. If $\Pi_{\Delta}^{-} \models S$ then $\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models S$.*

PROOF. Since $\Pi_{\Delta}^{-} \models S$ then by Lemmas VI.35(c) and (b), $\text{comp}_{\Delta}(\Pi) \models S$. So by the monotonicity of FOPC we have

$$\text{comp}_{\Delta}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models S,$$

and by Lemma V.17, $\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models S$. $\square$

LEMMA VII.21. *Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$.*

(a) *If $D_0, \dots, D_n$ is a failed SLD-derivation for $\Pi_{\Delta}^+ \cup \{D_0\}$ via R_{Δ} then $D_0, \dots, D_n$ is a failed SLD-derivation for $\Pi_{\Delta}^- \cup \{D_0\}$.*

(b) *If $D_0, \dots, D_n, \dots, \square$ is an SLD-refutation for $\Pi_{\Delta}^+ \cup \{D_0\}$ via R_{Δ} where D_n is the first Δ -goal then $D_0, \dots, D_n$ is a failed SLD-derivation for $\Pi_{\Delta}^- \cup \{D_0\}$.*

(c) *If $D_0, \dots, D_n, \square$ is an SLD-refutation for $\Pi_{\Delta}^+ \cup \{D_0\}$ via R_{Δ} where the refutation contains no Δ -goals then $D_0, \dots, D_n, \square$ is an SLD-refutation for $\Pi_{\Delta}^- \cup \{D_0\}$.*

PROOF.

(a) Since the failed derivation is constructed via R_{Δ} and since derivations with Δ -goals are guaranteed to succeed in Π_{Δ}^+ then the derivation $D_0, \dots, D_n$ is constructed by resolving goals only with clauses defining predicates belonging to $\text{COMPS}(\Pi) - \Delta$. Since these clauses also belong to Π_{Δ}^- then $D_0, \dots, D_n$ is also a partial derivation for $\Pi_{\Delta}^- \cup \{D_0\}$. Furthermore, since the goal D_n immediately fails in Π_{Δ}^+ then it must also immediately fail in Π_{Δ}^- because $\Pi_{\Delta}^- \subseteq \Pi_{\Delta}^+$. Thus, $D_0, \dots, D_n$ is a failed SLD-derivation for $\Pi_{\Delta}^- \cup \{D_0\}$.

(b) Since the refutation $D_0, \dots, D_n, \dots, \square$ is via R_{Δ} and D_n represents the first Δ -goal in the refutation then the partial derivation $D_0, \dots, D_n$ is constructed by resolving goals with clauses defining predicates belonging to $\text{COMPS}(\Pi) - \Delta$. Since these clauses also belong to Π_{Δ}^- then $D_0, \dots, D_n$ is also a partial derivation for $\Pi_{\Delta}^- \cup \{D_0\}$. However, the Δ -goal D_n must immediately fail in Π_{Δ}^- since Π_{Δ}^- contains no clauses of predicates belonging to Δ . Thus, $D_0, \dots, D_n$ is a failed SLD-derivation for $\Pi_{\Delta}^- \cup \{D_0\}$.

(c) Since the refutation $D_0, \dots, D_n, \square$ contains no Δ -goals then the refutation is constructed by resolving goals in the refutation with clauses defining predicates belonging to $\text{COMPS}(\Pi) - \Delta$. Since these clauses also belong to Π_{Δ}^- then $D_0, \dots, D_n, \square$ is also an SLD-refutation for $\Pi_{\Delta}^- \cup \{D_0\}$. $\square$

THEOREM VII.22. *Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$. Let $\text{OBS}^+ = \{\exists A_1, \dots, \exists A_n\}$ be a set of positive assertions. Let $D_0, \dots, D_k, \dots, \square$, $n \geq 0$, be an SLD-refutation for $\Pi_{\Delta}^+ \cup \{D_0\}$ via R_{Δ} where $D_0 = \{\leftarrow A_1, \dots, A_n\}$.*

(a) *If D_k is the first Δ -goal in the refutation then $\exists D_k^{\sim}$ is a positive Δ -support for OBS^+ with respect to Π .*

(b) If the refutation contains no Δ -goals then $\square^\sim$ is a positive Δ -support for OBS^+ with respect to Π .

PROOF.

(a) By Lemma VII.21(b), $D_0, \dots, D_k$ is a failed SLD-derivation for $\Pi_\Delta^- \cup \{D_0\}$. Assuming that the substitutions $\theta_0, \dots, \theta_k$ are used in the derivation so that their composition $\theta_0 \dots \theta_k = \theta$ then the soundness of SLD-resolution dictates that

$$\begin{aligned}\Pi_\Delta^- &\models D_k^\sim \rightarrow D_0^\sim \theta, \text{ or} \\ \Pi_\Delta^- &\models \exists D_k^\sim \rightarrow \exists D_0^\sim \theta, \text{ or} \\ \Pi_\Delta^- &\models \exists D_k^\sim \rightarrow \exists D_0^\sim.\end{aligned}$$

So, by Lemma VII.20

$$\begin{aligned}\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} &\models \exists D_k^\sim \rightarrow \exists D_0^\sim, \text{ or} \\ \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} &\models \exists D_k^\sim \rightarrow \exists (A_1 \wedge \dots \wedge A_n), \text{ or} \\ \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} &\models \exists D_k^\sim \rightarrow (\exists A_1 \wedge \dots \wedge \exists A_n).\end{aligned}$$

Therefore, by the definition of a positive Δ -support, $\exists D_k^\sim$ is a positive Δ -support for OBS^+ with respect to Π .

(b) By Lemma VII.21(c), $D_0, \dots, D_k, \square$ is an SLD-refutation for $\Pi_\Delta^- \cup \{D_0\}$. By the soundness of SLD-resolution we have

$$\begin{aligned}\Pi_\Delta^- &\models \exists D_0^\sim, \text{ or} \\ \Pi_\Delta^- &\models \text{true} \rightarrow \exists D_0^\sim, \text{ or} \\ \Pi_\Delta^- &\models \square^\sim \rightarrow \exists D_0^\sim.\end{aligned}$$

So, by Lemma VII.20

$$\begin{aligned}\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} &\models \square^\sim \rightarrow \exists D_0^\sim, \text{ or} \\ \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} &\models \square^\sim \rightarrow \exists (A_1 \wedge \dots \wedge A_n), \text{ or} \\ \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} &\models \square^\sim \rightarrow (\exists A_1 \wedge \dots \wedge \exists A_n).\end{aligned}$$

Therefore, by the definition of a positive Δ -support, $\square^\sim$ is a positive Δ -support for OBS^+ with respect to Π .

□

LEMMA VII.23. Let Π be a logic program. Let the terms $p(t^n)$ and $p(s^n)$, the formula G , and the variables $Z_1, \dots, Z_l$ belong to $L(\Pi)$. Let Γ be a set of sentences such that $\Gamma \subseteq L(\Pi)$ and $\Gamma \supseteq EQ(\Pi)$. If there does not exist a substitution η such that $p(t^n)\eta \equiv p(s^n)\eta$ then

$$\Gamma \models \neg \exists Z^l (t_1 = s_1 \wedge \dots \wedge t_n = s_n \wedge G), \quad l \geq 0.$$

PROOF. Since there does not exist a substitution η such that $p(t^n)\eta \equiv p(s^n)\eta$ then by Lemma II.27(b)

$$EQ(\Pi) \models \neg(t_1 = s_1 \wedge \dots \wedge t_n = s_n).$$

Since $\Gamma \supseteq EQ(\Pi)$ then by the monotonicity of FOPC,

$$\begin{aligned} \Gamma &\models \neg(t_1 = s_1 \wedge \dots \wedge t_n = s_n), \text{ so} \\ \Gamma &\models \neg(t_1 = s_1 \wedge \dots \wedge t_n = s_n \wedge G), \text{ so} \\ \Gamma &\models \forall Z^l \neg(t_1 = s_1 \wedge \dots \wedge t_n = s_n \wedge G), \text{ so} \\ \Gamma &\models \neg \exists Z^l (t_1 = s_1 \wedge \dots \wedge t_n = s_n \wedge G), \quad l \geq 0. \end{aligned}$$

□

LEMMA VII.24. Let Π be a logic program. Let the terms $p(t^n)$ and $p(s^n)$, the formula G , and the variables $Z_1, \dots, Z_l$ belong to $L(\Pi)$. Let Γ be a set of sentences such that $\Gamma \subseteq L(\Pi)$ and $\Gamma \supseteq EQ(\Pi)$. If there exists a substitution $\eta = \{X_1/u_1, \dots, X_k/u_k\}$ such that $p(t^n)\eta \equiv p(s^n)\eta$ then

$$\Gamma \models \exists Z^l (t_1 = s_1 \wedge \dots \wedge t_n = s_n \wedge G) \rightarrow \exists Z^l G\eta, \quad l \geq 0.$$

PROOF. Since there exists a substitution $\eta = \{X_1/u_1, \dots, X_k/u_k\}$ such that $p(t^n)\eta \equiv p(s^n)\eta$, then by Lemma II.27(a)

$$\begin{aligned} EQ(\Pi) &\models (t_1 = s_1 \wedge \dots \wedge t_n = s_n) \rightarrow (X_1 = u_1 \wedge \dots \wedge X_k = u_k), \text{ so} \\ EQ(\Pi) &\models \forall Z^l ((t_1 = s_1 \wedge \dots \wedge t_n = s_n) \rightarrow (X_1 = u_1 \wedge \dots \wedge X_k = u_k)). \end{aligned}$$

Since $\Gamma \supseteq EQ(\Pi)$ then by the monotonicity of FOPC

$$\Gamma \models \forall Z^l ((t_1 = s_1 \wedge \dots \wedge t_n = s_n) \rightarrow (X_1 = u_1 \wedge \dots \wedge X_k = u_k)).$$

It follows that

$$\begin{aligned} \Gamma \models (t_1 = s_1 \wedge \dots \wedge t_n = s_n)\delta \rightarrow (X_1 = u_1 \wedge \dots \wedge X_k = u_k)\delta, \\ \text{where } \delta = \{Z_1/\sigma_1, \dots, Z_l/\sigma_l\}, \text{ and } \sigma_i \notin L(\Pi), 0 \leq i \leq l, \text{ or} \\ \text{(VII.7) } \Gamma \cup \{(t_1 = s_1 \wedge \dots \wedge t_n = s_n)\delta\} \models (X_1 = u_1 \wedge \dots \wedge X_k = u_k)\delta. \end{aligned}$$

Now, since $\Gamma \supseteq \text{EQ}(\Pi)$ then by Theorem II.28 we have

$$\begin{aligned} \Gamma \models (X_1 = u_1 \wedge \dots \wedge X_k = u_k) \rightarrow (G \rightarrow G\eta), \text{ or} \\ \Gamma \models \forall Z'((X_1 = u_1 \wedge \dots \wedge X_k = u_k) \rightarrow (G \rightarrow G\eta)), \text{ or} \\ \Gamma \models (X_1 = u_1 \wedge \dots \wedge X_k = u_k)\delta \rightarrow (G\delta \rightarrow G\eta\delta). \end{aligned}$$

So by the monotonicity of FOPC

$$\begin{aligned} \text{(VII.8) } \Gamma \cup \{(t_1 = s_1 \wedge \dots \wedge t_n = s_n)\delta\} \models \\ (X_1 = u_1 \wedge \dots \wedge X_k = u_k)\delta \rightarrow (G\delta \rightarrow G\eta\delta). \end{aligned}$$

So from (VII.7) and (VII.8) we have

$$\begin{aligned} \Gamma \cup \{(t_1 = s_1 \wedge \dots \wedge t_n = s_n)\delta\} \models G\delta \rightarrow G\eta\delta, \text{ or} \\ \Gamma \models ((t_1 = s_1 \wedge \dots \wedge t_n = s_n)\delta \wedge G\delta) \rightarrow G\eta\delta, \text{ or} \\ \Gamma \models \exists Z'(t_1 = s_1 \wedge \dots \wedge t_n = s_n \wedge G) \rightarrow \exists Z'G\eta, l \geq 0. \end{aligned}$$

□

The final lemma required for the proof of correctness of Steps 2 and 3 of the verification procedure is the following.

LEMMA VII.25. *Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$. Let H be a goal and T be an SLD-tree of $\Pi_{\Delta} \cup \{H\}$. If $D_1, \dots, D_m$, $m \geq 1$ are goals of T such that each branch of T contains exactly one of them then*

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \forall(H \sim \rightarrow (\exists Z^{l_1} D_1 \sim \vee \dots \vee \exists Z^{l_m} D_m \sim)),$$

where for all $1 \leq i \leq m$, $l_i \geq 0$, and $\{Z_1, \dots, Z_{l_i}\} = \text{vars}(D_i) - \text{vars}(H)$.

PROOF. Without loss of generality, we take the goal $H = \leftarrow p(t^n)$. The proof proceeds by induction on k , the total number of goals that precede the goals $D_1, \dots, D_m$ on the branches of T . Let D denote the multiset $\{D_1, \dots, D_m\}$.

$k = 0$: In this case the root goal $\leftarrow p(t^n)$ immediately fails so $m = 1$ and $D_1^\sim = p(t^n)$. Therefore $\text{vars}(D_1^\sim) - \text{vars}(p(t^n)) = \emptyset$ and so $l_1 = 0$. Now, since

$$\models \forall(p(t^n) \rightarrow p(t^n)),$$

then

$$\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \forall(p(t^n) \rightarrow p(t^n)).$$

Therefore,

$$\begin{aligned} \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} &\models \forall(p(t^n) \rightarrow \exists Z^l D_1^\sim), \text{ or} \\ \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} &\models \forall(H^\sim \rightarrow \exists Z^l D_1^\sim). \end{aligned}$$

$k = 1$: In this case there exists only one predecessor goal for all $D_1, \dots, D_m$, namely the root goal $\leftarrow p(t^n)$. Note that in this case p must belong to $\text{COMPS}(\Pi) - \Delta$ since $\leftarrow p(t^n)$ resolves with at least one clause of Π_Δ^- and no clauses with Δ -literal heads belong to Π_Δ^- . Since $p \in \text{COMPS}(\Pi) - \Delta$ then there exists a sentence of the form

$$(VII.9) \quad \forall Y^n (p(Y^n) \leftrightarrow (F_1 \vee \dots \vee F_{\mu(p)})) \in \text{comp}_\Delta(\Pi),$$

where for all $1 \leq i \leq \mu(p)$, each F_i in (VII.9) is of the form

$$(VII.10) \quad \exists Z^l (EQ \wedge G),$$

where EQ denotes $Y_1 = s_1 \wedge \dots \wedge Y_n = s_n$,

G denotes $A_1 \wedge \dots \wedge A_k$,

$p(s^n) \leftarrow A_1, \dots, A_k \in \Pi_\Delta^-$, and

$\text{vars}(p(s^n) \leftarrow A_1, \dots, A_k) = \{Z_1, \dots, Z_l\}$, $l \geq 0$.

From (VII.9) we have

$$(VII.11) \quad \text{comp}_\Delta(\Pi) \models \forall(p(t^n) \leftrightarrow (F_1 \vee \dots \vee F_{\mu(p)})\theta),$$

where $\theta = \{Y_1/t_1, \dots, Y_n/t_n\}$.

Let R^- represent the set of F_i from (VII.11) such that the clause $p(s^n) \leftarrow A_1, \dots, A_k \in \Pi_\Delta^-$ associated with F_i fails to resolve with the goal $\leftarrow p(t^n)$. Let

R^+ represent the set remaining from (VII.11) i.e. $R^+ = \{F_1, \dots, F_{\mu(p)}\} - R^-$. So we can rewrite (VII.11) as

$$(VII.12) \quad comp_{\Delta}(\Pi) \models \forall(p(t^n) \leftrightarrow (\bigvee_{F \in R^-} F\theta \vee \bigvee_{F \in R^+} F\theta)).$$

Now, for all $F \in R^-$ consider the clause $p(s^n) \leftarrow A_1, \dots, A_k \in \Pi_{\Delta}^-$ associated with F . Since $F \in R^-$ then there does not exist a substitution η such that $p(t^n)\eta \equiv p(s^n)\eta$. In addition, since each F denotes a formula of the form given by (VII.10) then by Lemma VII.23 we have for all $F \in R^-$

$$comp_{\Delta}(\Pi) \models F\theta \leftrightarrow \text{false}.$$

By replacement in (VII.12) we have

$$(VII.13) \quad \begin{aligned} comp_{\Delta}(\Pi) \models \forall(p(t^n) \leftrightarrow \bigvee_{F \in R^+} F\theta), \text{ or} \\ comp_{\Delta}(\Pi) \cup \{p(t^n)\} \models \bigvee_{F \in R^+} F\theta. \end{aligned}$$

Now, for all $F \in R^+$ consider the clause $p(s^n) \leftarrow A_1, \dots, A_k \in \Pi_{\Delta}^-$ associated with F . Since $F \in R^+$ then there exists a substitution η such that $p(t^n)\eta \equiv p(s^n)\eta$. In addition, since each F denotes a formula of the form given by (VII.10) then by Lemma VII.24 we have for all $F \in R^+$

$$(VII.14) \quad comp_{\Delta}(\Pi) \cup \{p(t^n)\} \models F\theta \rightarrow \exists Z^i G\theta\eta.$$

Note that $dom(\theta) \cap vars(G) = \emptyset$ so that $G\theta = G$. Therefore from (VII.14) we have for all $F \in R^+$

$$(VII.15) \quad comp_{\Delta}(\Pi) \cup \{p(t^n)\} \models F\theta \rightarrow \exists Z^i G\eta.$$

By the completeness of SLD-resolution, for all $F \in R^+$ where F is of the form $\exists Z^i (EQ \wedge G)$, there exists a $D_i \in D$ such that $D_i^{\sim} \equiv G\eta$. So from (VII.15) we have that for all $F \in R^+$, there exists a $D_i \in D$ such that

$$(VII.16) \quad comp_{\Delta}(\Pi) \cup \{p(t^n)\} \models F\theta \rightarrow \exists Z^i D_i^{\sim}.$$

So from (VII.13) and (VII.16) we derive

$$\begin{aligned} comp_{\Delta}(\Pi) \cup \{p(t^n)\} \models \exists Z^1 D_1^{\sim} \vee \dots \vee \exists Z^l D_l^{\sim}, \text{ or} \\ comp_{\Delta}(\Pi) \models \forall(p(t^n) \rightarrow (\exists Z^1 D_1^{\sim} \vee \dots \vee \exists Z^l D_l^{\sim})). \end{aligned}$$

By the monotonicity of FOPC

$$\text{comp}_\Delta(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \forall(p(t^n) \rightarrow (\exists Z^{l_1} D_1^\sim \vee \dots \vee \exists Z^{l_m} D_m^\sim)).$$

Therefore, by Lemma V.17

$$\begin{aligned} \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} &\models \forall(p(t^n) \rightarrow (\exists Z^{l_1} D_1^\sim \vee \dots \vee \exists Z^{l_m} D_m^\sim)), \text{ or} \\ \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} &\models \forall(H^\sim \rightarrow (\exists Z^{l_1} D_1^\sim \vee \dots \vee \exists Z^{l_m} D_m^\sim)). \end{aligned}$$

Induction Hypothesis: Assume a total of $k > 1$ goals precede the goals $D_1, \dots, D_m$ in T . Then

$$\begin{aligned} \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} &\models \forall(H^\sim \rightarrow (\exists Z^{l_1} D_1^\sim \vee \dots \vee \exists Z^{l_m} D_m^\sim)), \\ \text{where for all } 1 \leq i \leq m, \{Z_1, \dots, Z_{l_i}\} &= \text{vars}(D_i) - \text{vars}(H), l_i \geq 0. \end{aligned}$$

Induction Step: Assume that a total of $k+1$ goals precede the goals $D_1, \dots, D_m$ in T . Let E be a goal on a branch of T whose immediate successors are some sub-multiset D' of D that occur at a maximal depth in T . So by the induction hypothesis

$$\begin{aligned} \text{(VII.17)} \quad \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} &\models \forall(p(t^n) \rightarrow (\bigvee_{D_i \in D-D'} \exists Z^{l_i} D_i^\sim \vee \exists X^k E^\sim)), \\ \text{where } \{X_1, \dots, X_k\} &= \text{vars}(E) - \text{vars}(p(t^n)), k \geq 0, \text{ and} \\ \text{for all } i, \{Z_1, \dots, Z_{l_i}\} &= \text{vars}(D_i) - \text{vars}(p(t^n)), l_i \geq 0. \end{aligned}$$

Now consider the subtree T' of T rooted at E and let $q(t)$ be the selected atom of E in T' . Since the tree T' has only the one predecessor goal E with respect to the goals belonging to D' then by following the proof for the case $k = 1$ we have

$$\begin{aligned} \text{(VII.18)} \quad \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} &\models \forall(q(t) \rightarrow \bigvee_{D_i \in D'} \exists Z^{l_i} D_i^\sim), \\ \text{where for all } i, \{Z_1, \dots, Z_{l_i}\} &= \text{vars}(D_i) - \text{vars}(q(t)), l_i \geq 0. \end{aligned}$$

Since $\text{vars}(E) \cap \text{vars}(D_i) \subseteq \text{vars}(q(t))$ then

$$\text{(VII.19)} \quad \text{vars}(D_i) - \text{vars}(q(t)) = \text{vars}(D_i) - \text{vars}(E).$$

So from (VII.18) and (VII.19) we have

$$\begin{aligned}
 & \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \forall(E^\sim \rightarrow \bigvee_{D_i \in D'} \exists Z^{l_i} D_i^\sim), \text{ or} \\
 & \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \exists X^k E^\sim \rightarrow \exists X^k \bigvee_{D_i \in D'} \exists Z^{l_i} D_i^\sim, \text{ or} \\
 & \text{(VII.20)} \quad \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \exists X^k E^\sim \rightarrow \bigvee_{D_i \in D'} \exists X^k \exists Z^{l_i} D_i^\sim,
 \end{aligned}$$

where for all i , $\{Z_1, \dots, Z_{l_i}\} = \text{vars}(D_i) - \text{vars}(E)$, $l_i \geq 0$.

From (VII.17) and (VII.20) we have

$$\begin{aligned}
 & \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \\
 & \text{(VII.21)} \quad \forall(p(t^n) \rightarrow (\bigvee_{D_i \in D-D'} \exists Z^{l_i} D_i \vee \bigvee_{D_i \in D'} \exists X^k \exists Z^{l_i} D_i^\sim)), \\
 & \text{where for all } i, \{Z_1, \dots, Z_{l_i}\} = \text{vars}(D_i) - \text{vars}(p(t^n)), \quad l_i \geq 0, \\
 & \quad \{X_1, \dots, X_k\} = \text{vars}(E) - \text{vars}(p(t^n)), \quad k \geq 0, \text{ and} \\
 & \text{for all } j, \{Z_1, \dots, Z_{l_j}\} = \text{vars}(D_j) - \text{vars}(E), \quad l_j \geq 0.
 \end{aligned}$$

Now,

$$\begin{aligned}
 & \text{(VII.22)} \quad (\text{vars}(D_j) - \text{vars}(E)) \cup (\text{vars}(E) - \text{vars}(p(t^n))) = \\
 & \quad (\text{vars}(D_j) - \text{vars}(p(t^n))).
 \end{aligned}$$

So from (VII.21) and (VII.22) we have

$$\begin{aligned}
 & \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \forall(p(t^n) \rightarrow (\exists Z^{l_1} D_1^\sim \vee \dots \vee \exists Z^{l_m} D_m^\sim)), \text{ or} \\
 & \text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \forall(H^\sim \rightarrow (\exists Z^{l_1} D_1^\sim \vee \dots \vee \exists Z^{l_m} D_m^\sim)), \\
 & \text{where for all } 1 \leq i \leq m, \{Z_1, \dots, Z_{l_i}\} = \text{vars}(D_i) - \text{vars}(H), \quad l_i \geq 0.
 \end{aligned}$$

□

We are now prepared to prove Theorem VII.19, thereby proving the correctness of Steps 2 and 3 of the diagnosis verification procedure.

PROOF. (Theorem VII.19) By Theorem VII.22, for all $1 \leq i \leq m$, $\exists D_i^\sim$ is a positive Δ -support for OBS^+ with respect to Π . Now let D^+ denote the multiset $\{D_1, \dots, D_m\}$ and let $D^- = \{D_{m+1}, \dots, D_k\}$ represent the multiset of leaf goals on

failing branches of T such that each branch contains exactly one goal in D^- . Note that since T contains no infinite branches then $D^+ \cup D^-$ denotes a multiset of goals such that each branch of T contains exactly one of them. From Lemma VII.21 we also know that $D^+ \cup D^-$ denotes a multiset of goals of an SLD-tree T' for $\Pi_{\Delta}^- \cup \{G\}$ where $G = \leftarrow A_1, \dots, A_n$. Since $\Pi_{\Delta}^- \subseteq \Pi_{\Delta}^+$ then T' contains no more branches than T so that each branch of T' contains exactly one goal from $D^+ \cup D^-$. Therefore by Lemma VII.25

(VII.23)

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \forall(G^{\sim} \rightarrow (\bigvee_{D_i \in D^-} \exists Z^{l_i} D_i^{\sim} \vee \bigvee_{D_j \in D^-} \exists Z^{l_j} D_j^{\sim})),$$

where for all $1 \leq i \leq m$, $\{Z_1, \dots, Z_{l_i}\} = \text{vars}(D_i) - \text{vars}(G)$, $l_i \geq 0$, and
for all $m+1 \leq j \leq k$, $\{Z_1, \dots, Z_{l_j}\} = \text{vars}(D_j) - \text{vars}(G)$, $l_j \geq 0$.

Now, all goals $D \in D^-$ fail in Π_{Δ}^+ . Therefore by Lemma VII.16, for all $D \in D^-$

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \exists D^{\sim} \leftrightarrow \text{false}.$$

So by replacement in (VII.23) we have

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \forall(G^{\sim} \rightarrow \bigvee_{D_i \in D^+} \exists Z^{l_i} D_i^{\sim}), \text{ or}$$

(VII.24)

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models \exists G^{\sim} \rightarrow (\exists D_1^{\sim} \vee \dots \vee \exists D_m^{\sim}).$$

Since the positive assertions share no variables then $\exists(A_1 \wedge \dots \wedge A_n)$ is logically equivalent to $\exists A_1 \wedge \dots \wedge \exists A_n$. So by replacement in (VII.24) we have

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \models (\exists A_1 \wedge \dots \wedge \exists A_n) \rightarrow (\exists D_1^{\sim} \vee \dots \vee \exists D_m^{\sim}).$$

Therefore $\{\exists D_1^{\sim}, \dots, \exists D_m^{\sim}\}$ is a complete set of positive Δ -supports for OBS^+ with respect to Π . $\square$

4.3. Implementation and Correctness of Step 4. Step 4 determines the satisfiability of the set of sentences

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \{S\} \cup \text{OBS}^-,$$

where S is a positive Δ -support for OBS^+ with respect to Π . By the soundness confirmation theorem (Theorem VII.5), we know that if this sentence is satisfiable then $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS^+ \cup OBS^-)$. On the other hand, if this sentence is not satisfiable for all positive Δ -supports in a complete set of positive Δ -supports then by the soundness denial theorem (I) (Theorem VII.9), $\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)$ is not a diagnosis for $(SD(\Pi), COMPS(\Pi), OBS^+ \cup OBS^-)$.

In order to compute whether the set of sentences is satisfiable or not we utilize a technique from the domain of automatic theorem proving which permits the existential quantifiers of a positive Δ -support to be eliminated by Skolemizing the existentially quantified variables (see Chapter II). Since a positive Δ -support is of the form $\exists(A_1 \wedge \dots \wedge A_n)$ where each A_i is a positive literal then in Skolem standard form the positive Δ -support would be of the form $A'_1 \wedge \dots \wedge A'_k$ where A'_i denotes A_i with its variables replaced with Skolem constants. Thus, a positive Δ -support in Skolem standard form is just a conjunction of ground atoms.

EXAMPLE VII.26. In Example VII.4 we determined that

$$\exists X(x1(1, 0, 0) \wedge a2(1, 0, X) \wedge o1(X, 0, 0)),$$

was a positive Δ -support. After converting this formula to Skolem standard form we obtain

$$x1(1, 0, 0) \wedge a2(1, 0, \sigma) \wedge o1(\sigma, 0, 0),$$

where σ is the Skolem constant replacing the existentially quantified variable X .

We exploit Theorem II.14 in the proof of one of the following two lemmas which describe how to implement the satisfiability check of Step 4 of the diagnosis verification procedure.

LEMMA VII.27. *Let Π be a logic program, $\Delta \subseteq COMPS(\Pi)$. Let $OBS^- = \{\forall \neg A_1, \dots, \forall \neg A_k\}$ be a set of negative assertions. Let D be a positive Δ -support for some set of positive assertions OBS^+ with respect to Π . Let D_σ represent the set of ground atoms which constitute the Skolem standard form of D . If for all $\forall \neg A \in OBS^-$, $\leftarrow A$ finitely fails in $\Pi_\Delta^- \cup D_\sigma$ then*

$$SD(\Pi) \cup \{\mathcal{D}(\Delta, COMPS(\Pi) - \Delta)\} \cup \{D\} \cup OBS^- \text{ is satisfiable.}$$

LEMMA VII.28. Let Π be a logic program, $\Delta \subseteq \text{COMPS}(\Pi)$. Let OBS^- be a set of negative assertions and $\forall \neg A \in \text{OBS}^-$. Let D be a positive Δ -support for some set of positive assertions OBS^+ with respect to Π . Let D_σ represent the set of ground atoms which constitute the Skolem standard form of D . If $\leftarrow A$ succeeds in $\Pi_\Delta^- \cup D_\sigma$ then

$\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \{D\} \cup \text{OBS}^-$ is not satisfiable.

Thus, Step 4 consists of simply determining whether a set of goals succeeds or finitely fails in a logic program. Note that if the SLD-tree for this step has an infinite branch then the step may not terminate. We now present the proofs of these two lemmas.

PROOF. (Lemma VII.27)

Since $\leftarrow A$ finitely fails in $\Pi_\Delta^- \cup D_\sigma$ for all $\forall \neg A \in \text{OBS}^-$ then by the soundness of the NAF rule,

$$\text{comp}(\Pi_\Delta^- \cup D_\sigma) \models \forall \neg A_1 \wedge \dots \wedge \forall \neg A_k.$$

Furthermore,

$$\text{comp}(\Pi_\Delta^- \cup D_\sigma) \models D.$$

Now, since $\text{comp}(\Pi_\Delta^- \cup D_\sigma)$ is satisfiable then

$$(VII.25) \quad \text{comp}(\Pi_\Delta^- \cup D_\sigma) \cup \{D\} \cup \text{OBS}^- \text{ is satisfiable,}$$

and since

$$(VII.26) \quad \{\forall X(p(X) \leftrightarrow F) \in \text{comp}_\Delta(\Pi)\} \cup \text{EQ}(\Pi) \subseteq \text{comp}(\Pi_\Delta^- \cup D_\sigma)$$

then by compactness of FOPC, (VII.25) and (VII.26) we have

$$\text{comp}_\Delta(\Pi) \cup \{D\} \cup \text{OBS}^- \text{ is satisfiable.}$$

Since ab is not a predicate in $L(\Pi)$ then

$$\text{comp}_\Delta(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \{D\} \cup \text{OBS}^- \text{ is satisfiable.}$$

Thus, by Lemma V.17

$$\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \{D\} \cup \text{OBS}^- \text{ is satisfiable.}$$

□

PROOF. (Lemma VII.28)

Since $\leftarrow A$ succeeds in $\Pi_{\Delta}^- \cup D_{\sigma}$ then by the soundness of SLD-resolution

$$\Pi_{\Delta}^- \cup D_{\sigma} \cup \{\forall \neg A\} \text{ is not satisfiable.}$$

So by Theorem II.14

$$\Pi_{\Delta}^- \cup \{D\} \cup \{\forall \neg A\} \text{ is not satisfiable.}$$

By Lemmas VI.35(c) and (a) $\text{comp}_{\Delta}(\Pi) \models \Pi_{\Delta}^-$ so by compactness of FOPC

$$\text{comp}_{\Delta}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \{D\} \cup \text{OBS}^- \text{ is not satisfiable.}$$

Therefore by Lemma V.17

$$\text{SD}(\Pi) \cup \{\mathcal{D}(\Delta, \text{COMPS}(\Pi) - \Delta)\} \cup \{D\} \cup \text{OBS}^- \text{ is not satisfiable.}$$

□

4.4. Summary. The previous sections have shown how a procedure to verify diagnoses based on the notion of positive Δ -supports can be implemented. We have proven such an implementation to be correct with respect to the underlying theory of positive Δ -supports. Steps 1 and 4 require nothing more than SLD-resolution and the NAF rule to verify success or failure of goals in a logic program. Steps 2 and 3 require the use of a simple SLD-resolution meta-interpreter. An implementation of the procedure using Prolog is included in Appendix A. Since Steps 1 and 4 require only SLD-resolution and the NAF rule then these steps may be compiled to execute efficiently. However, this is not possible with Steps 2 and 3 since they require a meta-interpreter.

5. Computing Diagnoses

In order to use the verification procedure TP to compute minimal diagnoses for a system $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS}^+ \cup \text{OBS}^-)$, we use a generate-and-test approach identical to that used in the previous chapter i.e. we use the generator algorithm from the IDA framework [65] (see Chapter IV) and have it call the TP procedure for testing purposes. We have interfaced the Prolog implementation of the TP procedure given in Appendix A to the Prolog implementation of the IDA generator algorithm. The Prolog code necessary to interface the two procedures is included in Appendix B.

5.1. Complexity and Efficiency. The resulting diagnosis computation procedure computes the next diagnosis in polynomial time assuming successive calls to the TP function take constant time. As discussed in the previous chapter, whereas there are cases for which this assumption does not hold, we feel this is a reasonable assumption to make in general.

With one exception, all the comments regarding the efficiency of the approach developed in the previous chapter apply to the procedure developed in this chapter *e.g.* the possibility of partially evaluating Π with respect to the set of assertions and the ability to use a constraint-based logic programming dialect. In the previous chapter the possibility also existed for compiling the logic programs $LSD^+(\Pi)$ and $LSD^-(\Pi)$ called by the TP^+ and TP^- procedures, respectively. For the TP procedure developed here, program compilation is not an option since the program effectively changes as the value of Δ and the positive Δ -support change.

5.2. Properties of Abnormalities. Recall from Chapter V that for this consistency-based diagnosis framework abnormal components in minimal diagnoses can be classified as incorrect or incomplete. As such Theorems V.7, V.11 and V.12 apply to the diagnoses computed using the approach described here so that one is able to gain some insight into the nature of the fault that has been identified in the same way this was possible for the procedures in the last chapter.

5.3. Expressiveness of Assertions. Using the same methods described in the previous chapter, it is possible to extend the effective expressiveness of the class of positive assertions accepted by the diagnosis verification procedure to sets of universally or existentially quantified conjunctions of positive literals.

5.4. Examples. Examples of computing minimal diagnoses using the TP procedure coupled with the IDA generator algorithm are included in the appendices.

- Appendix C gives examples of using the procedure to diagnose the buggy insertion sort program (Program III.2). We show how the procedure can diagnose faults at both the coarse procedural level and at the finer clausal level.
- Appendix D gives examples of using the procedure to diagnose the faulty full-adder circuit (see Figure V.11).

- Appendix E gives an example of diagnosing a hierarchical sequential logic circuit. The example circuit is a 4X4 bit sequential multiplier that was initially used as an example by Hamscher and Davis in [44] to illustrate the difficulty of diagnosing sequential circuits. In our example we illustrate the expressive power of logic programs by using a sequential circuit modelling scheme originally developed by Clocksin in [12].

6. Comparison to Related Work

In the paper entitled "Model-Based Diagnosis Meets Error Diagnosis in Logic Programs," by L. Console, G. Friedrich and D.T. Dupré [14], a model-based diagnostic framework is described that defines diagnoses in faulty logic programs (see Chapter V). Their paper also presents a procedure for computing the diagnoses defined by the framework. The procedure proposed in [14] is the only approach we are aware of that is comparable to the one presented here. Whereas declarative error diagnosis procedures require a user to answer questions about intended program behavior, the diagnostic procedure proposed in [14] computes diagnoses given whatever initial assertions regarding intended program behavior the user can provide. Assertions can be universally quantified negative literals or universally quantified positive literals. In addition, whereas declarative error diagnosis computes faults one at a time, the diagnostic procedure proposed in [14] computes sets of diagnoses that are consistent with the assertions provided by the user. Since the framework proposed in [14] can define many diagnoses for a given set of assertions, a preference ordering on diagnoses is defined. The diagnosis computation procedure is then able to compute diagnoses in decreasing order of preference.

In [6], the results of a critical analysis of [14] are presented. The analysis shows that a property claimed to be true in [14] is false. The consequence of this finding is that diagnosis computation procedure proposed in [14] fails to compute some diagnoses defined by the framework. Also explored in [6] are issues related to the implementation of the diagnosis computation procedure—issues not discussed in [14]. In this section we review the findings presented in [6] and contrast these with properties of the diagnosis computation procedure presented in this chapter.

6.1. Completeness. The diagnostic framework presented by Console *et al.* in [14] defines diagnoses in such a way that all errors defined by declarative error di-

agnosis are also defined to be diagnoses for the framework. However, the diagnosis computation procedure proposed in [14] is not always able to compute diagnoses defined by their framework. In contrast, the procedures defined for declarative error diagnosis and the procedure presented in this chapter are able to compute these same diagnoses.

EXAMPLE VII.29. As an example let Π denote the following logic program:

$$\{a(X) \leftarrow b(X, Y), c(Y).\}$$

Now take the set of positive assertions to be $OBS^+ = \{b(1, 1), c(2)\}$, and take the set of negative assertions to be $OBS^- = \{\neg a(1)\}$. Given these assertions, the diagnostic framework proposed in [14] defines a diagnosis to exist for which only b and c are faulty. However for these same assertions the diagnosis computation procedure proposed in [14] computes no diagnoses for which only b and c are faulty. On the other hand, using Shapiro's *ip* algorithm [85] for missing solutions, one would find using the assertions in OBS^+ as oracle responses that both b and c were incomplete and atoms $b(1, 1)$ and $c(2)$ were uncovered with respect to the intended interpretation. Also, using the diagnostic framework defined in this dissertation, $\mathcal{D}(\{b, c\}, \{a\})$ is a minimal diagnosis for $(SD(\Pi), COMPS(\Pi), OBS^+ \cup OBS^-)$, and by Theorem V.11, both b and c must be incomplete i.e. there exist uncovered atoms for b and c —in this case these atoms are the positive assertions themselves. The TP procedure defined in this chapter also computes $\mathcal{D}(\{b, c\}, \{a\})$ as a minimal diagnosis defined by our framework.

6.2. Assertions. The procedure proposed in [14] only computes diagnoses for universally quantified positive literals. This contrasts with the procedure presented in this chapter which directly computes diagnoses for existentially quantified positive literals, and indirectly computes diagnoses for universally or existentially quantified conjunctions of positive literals as discussed earlier.

6.3. Implementation Issues. Although implementation issues for the diagnosis computation procedure are not discussed in [14], one can infer from the description of the procedure that the following procedures are required: first-order term subsumption, propositional prime implicant/implicate generation/transformation, and confirming/denying logical implication of positive/negative assertions by a (general) logic program. With one exception, these procedures

and their associated soundness, completeness and decidability properties, are well understood.

The exception occurs during a step of the procedure that determines whether or not

$$\Pi \cup D \models \forall X p(X), \text{ for all } \forall X p(X) \in \text{OBS}^+.$$

Here Π denotes a logic program and D denotes a set of general logic program clauses of the form (for unary predicates): $p(X) \leftarrow X \neq t_1, \dots, X \neq t_n$, for $n \geq 0$. To perform this step of the procedure one must use a theorem prover that provides semantics for both negative terms and for the equality predicate. One might get the impression that what is required is an SLDNF-resolution theorem prover [57] that supports a meta-level definition of equality as identity. However, this is not the case for a number of reasons described in [6]. In particular, it is shown that an unsafe selection rule is required in some instances due to the presence of disequality terms. Furthermore, since the variables of the positive assertions are universally quantified in their framework, the variables must be Skolemized (see Chapter II) in order to use SLDNF-resolution. As a result, supporting a meta-level notion of equality as identity can be shown to be inadequate.

Both of these implementation issues necessitate the use of a more complicated, potentially less complete theorem prover than that used by the procedure described earlier in this chapter. Moreover, the correctness of an implementation of their procedure will remain unknown until a particular theorem proving approach is chosen. This contrasts with the implementation of the procedure presented in this chapter which has been proven correct.

7. Summary

In this chapter we have presented a procedure for verifying diagnoses for systems of the form $(\text{SD}(\Pi), \text{COMPS}(\Pi), \text{OBS}^+ \cup \text{OBS}^-)$ where OBS^+ and OBS^- are sets of positive and negative assertions, respectively. The procedure, based on the notion of a positive Δ -support has been proven sound although completeness conditions remain to be identified. An implementation of the procedure is presented and proven correct. When combined with the IDA diagnosis generation algorithm, the next minimal diagnosis is computable in polynomial time.

While potentially not as efficient an approach as that presented in the previous chapter, the approach presented in this chapter has the advantage of computing diagnoses for a more general class of assertions. This provides the diagnostician with the ability to better express test results or expectations. It also provides greater potential for refining diagnoses.

Finally, we have shown that the procedure presented in this chapter has better fault coverage, is easier to implement and can compute diagnoses for a larger class of assertions than the only other comparable procedure.

CHAPTER VIII

Conclusions

1. Summary of Results

The problem we set out to solve in this dissertation was to automate diagnosis for systems satisfying two properties. These properties essentially specify the absence of diagnostic information for a particular system. For most diagnostic approaches, the negation of one or both of these properties is assumed. However, we have argued that a significant number of systems exist which satisfy these properties and therefore are not diagnosable using the other approaches. Examples of such systems are most software systems, some hardware systems, and most system specifications. In this dissertation the problem is solved by way of developing a theoretical diagnostic framework and polynomial-time procedures for computing the diagnoses defined by this framework. The framework and procedures developed are shown to be superior in a number of respects to existing frameworks and procedures.

1.1. Theoretical Framework. Our approach is defined in terms of a consistency-based diagnosis framework for the class of systems $(SD(\Pi), COMPS(\Pi), OBS)$ where Π is a logic program. The logic program Π upon which the system description $SD(\Pi)$ is based is simply a simulation model of the system to diagnose. As such, the modeller can take advantage of a wide variety of existing methods for system modelling using logic programs, including methods for modelling software, hardware and specifications.

The language of logic programs is doubly useful for diagnosis when the artifact in question is a software system or a specification. Not only is the language capable,

in principle, of modelling any such artifact but it also possesses a formal semantics that we can exploit to evaluate the soundness and completeness of the diagnostic framework with respect to the software or specification language semantics. Note that soundness and completeness with respect to the artifact language semantics are notions distinct from soundness and completeness with respect to the modelling language semantics. The latter properties relate the diagnostic procedures to the diagnostic framework. These properties have been explored for the two diagnostic procedures we develop in this dissertation. On the other hand, if the artifact language semantics are formal then a formal mapping can be established between the modelling language semantics and the artifact language semantics. The properties of this mapping will make it possible to determine if faults that exist in an artifact will be defined as faults by the framework (completeness), or if faults defined to exist by the framework truly exist in the artifact (soundness). For example, for logic program diagnosis in this dissertation we utilize an identity mapping from logic program constructs (the artifact language) to logic program constructs (the modelling language). Obviously such a mapping ensures soundness and completeness of diagnosis with respect to the artifact language semantics. What will be interesting to further explore are the mappings required for the diagnosis of other programming or specification languages.

In our framework, the completion of a logic program Π has been incorporated into a system description $SD(\Pi)$ in order to automatically define the negative behavior associated with a system whose positive behavior is defined by Π . This relieves the modeller of having to explicitly provide the clauses defining negative behavior.

Since systems are frequently modelled in terms of structural hierarchies it is important that any diagnostic approach be able to diagnose systems in hierarchical terms. Two notions of faults in structural hierarchies—incompleteness and incorrectness—have been defined in our framework and shown to constitute sufficient, and sometimes necessary and sufficient characteristics of abnormal components. This is a rather surprising result given that system descriptions $SD(\Pi)$ contain no fault models. However, like fault models these notions of faults provide a general indication of how a component has failed and how it should be repaired. These notions of faults are also defined to distinguish between components

causing erroneous behavior in a hierarchy and components propagating erroneous behavior—something we would hope any hierarchical approach to diagnosis would be capable of doing.

We have also shown that our consistency-based diagnosis framework is a generalization of declarative error diagnosis—a result that has ramifications for both consistency-based diagnosis and declarative error diagnosis. For instance, it is possible to use our general consistency-based diagnosis framework for software diagnosis in general and logic program diagnosis in particular when the constraints on such software systems required by declarative error diagnosis cannot be met. On the other hand, if a software system satisfies the constraints for declarative error diagnosis then one of the numerous efficient declarative error diagnosis procedures can be used to locate faults defined by our own consistency-based diagnosis framework.

By virtue of being an instance of general consistency-based diagnosis theory, our framework has inherited results obtained for consistency-based diagnosis in such areas as testing, measurement and correction. These results are inherited by declarative error diagnosis too as a result of the relationship we have shown to exist between our consistency-based diagnosis framework and declarative error diagnosis.

1.2. Diagnosis Computation. We have developed two generate-and-test approaches to diagnosis computation that utilize IDA's polynomial-time minimal diagnosis generation algorithm. Both approaches have been shown to be more general than comparable diagnosis computation procedures. As such, both approaches are capable of computing potentially more diagnoses for more general classes of systems than other comparable approaches.

One approach computes diagnoses for systems whose assertions are sets of either positive or negative assertions. The test part of the generate-and-test procedure simply checks whether assertions succeed or finitely fail in a logic program. Since it is possible to compile the logic programs used, the implementation of this approach can be quite efficient. The second approach computes diagnoses for sets of assertions consisting of both positive and negative assertions using a meta-interpreter. While more complex than the previous approach, this approach computes diagnoses for a more general class of systems. Using both approaches, the abnormal

components of computed minimal diagnoses can be categorized as incomplete or incorrect. By replacing variables in assertions with Skolem constants or introducing supplementary clauses it is also possible to use both approaches to compute diagnoses for the class of positive assertions that includes universally or existentially quantified conjunctions of positive literals.

Since both approaches interpret logic programs, their efficiency can potentially be improved by partially evaluating the logic programs with respect to the sets of assertions or using a constraint-based logic programming language. Considering the success of the IDA procedure, we also have reason to be optimistic regarding the ability of the procedures to scale-up to "real-world" examples.

Both approaches have been proven to be sound and their implementations proven correct. For the first approach, conditions for completeness have been identified. For the second approach we have shown instances where incompleteness can occur but have stopped short of identifying general completeness conditions. Indeed, it is quite likely that no complete diagnosis computation procedure exists for the general class of systems ($SD(\Pi)$, $COMPS(\Pi)$, OBS) due to the expressiveness afforded by system descriptions $SD(\Pi)$. However, we feel this is an acceptable trade-off given the ability to model and diagnose general software systems. Furthermore, the termination analysis of logic programs is an active field of research (e.g. see [76, 4]) whose results will help to identify classes of system descriptions $SD(\Pi)$ for which diagnostic completeness is guaranteed.

2. Future Research

2.1. Negative Information. We have chosen to use the negation as failure formalism to define the negative behavior associated with system description of our framework. This particular semantics for negation was chosen because it is computable whereas more general forms such as the closed world assumption are not. However, as shown in Chapter V, using negation as failure semantics can sometimes lead to counter-intuitive diagnoses. As such, the framework might be altered to utilize a different but still computable negation semantics (e.g. see [32, 40, 5]).

Also, system descriptions for our framework can only be based on normal logic programs, not general logic programs which allow negative literals to be included in

the bodies of program clauses. While the expressiveness of general logic programs is the same as for normal logic programs, the language of general logic programs provides the potential for constructing more concise system descriptions.

2.2. Fault Models. Although not a requirement for the framework as defined in our problem statement, one disadvantage of the framework we have developed is that it does not accept system descriptions that include fault models. While their presence exponentially increases the size of the diagnostic search space, they also help to eliminate counter-intuitive diagnoses. In addition, by associating an *a priori* probability of failure with each failure mode, it is possible to utilize probabilistic search techniques to find most probable diagnoses.

As shown in Chapter VI, system descriptions of the IDA framework accommodate fault models. Since the diagnostic approach developed for our framework in Chapter VI is closely related to the IDA approach, we feel that it should not be too difficult to alter our framework accordingly. Note however, that by incorporating arbitrary fault models in a system description, it is possible that many of the properties defined for the framework, for example the characterization of diagnoses in terms of minimal diagnoses, and the necessary and sufficient conditions for component abnormality, will no longer hold.

2.3. Search Strategies. While the lattice search strategy employed by our diagnosis computation procedures is very efficient, calls to the diagnosis verification procedures TP could stand to be made more efficient. We have already discussed the possibilities of compiling or partially evaluating the logic programs called, or using efficient constraint-based logic programming languages. However, other options also exist.

Since we have shown declarative error diagnosis to be an instance of our own consistency-based diagnosis framework we should be able to utilize search strategies employed by declarative error diagnosis procedures. One particularly promising approach exploits the term dependencies in SLD-trees to effectively reduce the number of branch traversals required [74]. To utilize this approach, the individual terms which constitute the arguments of each assertion must be categorized as correct or incorrect.

Since our diagnostic framework defines diagnoses for structural hierarchies then using a top-down search strategy through the hierarchy can reduce the potential

diagnostic search space. This topic has been investigated in the context of using model abstractions for model-based diagnosis (e.g. [41, 64, 73]) and for declarative error diagnosis (e.g. [56, 92]). The important topics to be investigated when developing such a strategy for the framework are its soundness and completeness.

2.4. Measurement, Tests, Repair and Correction. Our framework and procedures only solve one problem that exists as part of the overall problem of automating the debugging process (see Figure I.1). An obvious future direction for this research is to suggest possible repairs or corrections associated with a diagnosis, or suggest measurements or tests whose results will eliminate candidate diagnoses defined for a prior set of assertions.

In Chapter VII it was pointed out that a positive Δ -support can be interpreted as a hypothesis concerning the positive behaviors of components assumed to be faulty. As such a positive Δ -support can be seen as the basis for a discriminating test or a correction. Some results that appear applicable to our framework have been obtained by Console *et al.* [14] as part of their model-based diagnosis framework, and by McIlraith, Reiter and Tsybenko for first-order theories in general [82, 63, 62, 91].

Since our framework is a generalization of declarative error diagnosis another promising approach might be to exploit the theory of inductive logic program synthesis introduced by Shapiro as the second part of his dissertation in which he also defines declarative error diagnosis [85]. This theory suggests corrections for faults located in logic programs using declarative error diagnosis. A recent summary of work in this area is found in [68].

2.5. Incremental Diagnosis Computation. Recognizing the fact that diagnosis is traditionally part of a debugging cycle (see Figure I.1), some incremental diagnosis computation algorithms have been developed for consistency-based diagnosis (e.g. [50, 47]). These algorithms use information from computing the diagnoses for a set of assertions OBS in order to reduce the complexity of computing the diagnoses for a set of assertions $OBS \cup \{S\}$ where S is an additional assertion that may reflect the outcome of a test or measurement. Whereas the existing incremental algorithms are for propositional systems (SD, COMPS, OBS), a first-order algorithm will be required for systems (SD(Π), COMPS(Π), OBS).

Bibliography

- [1] H. Andréka and I. Németi. The generalized completeness of Horn predicate logic as a programming language. *Acta Cybernetica*, 4:3-10, 1978.
- [2] K.R. Apt and M.H. van Emden. Contributions to the theory of logic programming. *Journal of the A.C.M.*, 29(3):841-862, 1982.
- [3] Krzysztof R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Formal Models and Semantics*, volume B, chapter 10, pages 493-574. MIT Press/Elsevier, 1990.
- [4] Krzysztof R. Apt and Dino Pedreschi. Proving termination of general logic programs. In T. Ito and A.R. Meyer, editors, *Proceedings of the International Conference on Theoretical Aspects of Computer Science (TACS '91)*, number 526 in Lecture Notes in Computer Science, pages 265-289. Springer-Verlag, September 1991.
- [5] N. Bidoit. Negation in rule-based database languages: a survey. *Theoretical Computer Science*, 78:3-83, 1991.
- [6] G. W. Bond and B. Pagurek. A critical analysis of "Model-Based Diagnosis Meets Error Diagnosis in Logic Programs". Technical Report SCE-94-15, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, K1S 5B6, June 1994.
- [7] Gregory W. Bond and Bernie Pagurek. Consistency based diagnosis of nondeterministic systems. In *Working Papers of the 3rd International Workshop on Principles of Diagnosis (DX-92)*, pages 212-218, 1992.
- [8] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Computer Science Classics. Academic Press, 1973.
- [9] Rita L. Childress and Marco Valtorta. Polynomial-time model-based diagnosis with the critical set algorithm. In *Working Papers of the Fourth International Workshop on Principles of Diagnosis (DX-93)*, pages 166-177, 1993.
- [10] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293-322. Plenum Press, New York, 1978.

- [11] K.L. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Dept. of Computing, Imperial College, London, 1979.
- [12] W.F. Clocksin. Logic programming and digital circuit analysis. *Journal of Logic Programming*, 4:59-82, 1987.
- [13] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.
- [14] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-93)*, volume 2, pages 1494-1499, 1993. Also in *Working Papers of the 4th International Workshop on Principles of Diagnosis (DX-93)*, pp. 212-222.
- [15] N. J. Cutland. *Computability: an introduction to recursive function theory*. Cambridge University Press, 1988.
- [16] Johan de Kleer. An assumption-based ATMS. *Artificial Intelligence*, 28:127-162, 1986.
- [17] Johan de Kleer. Extending the ATMS. *Artificial Intelligence*, 28:163-196, 1986.
- [18] Johan de Kleer. Problem solving with the ATMS. *Artificial Intelligence*, 28:197-224, 1986.
- [19] Johan de Kleer. Using crude probability estimates to guide diagnosis. *Artificial Intelligence*, 45:381-391, 1990.
- [20] Johan de Kleer. Focusing on probable diagnoses. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI-91)*, volume 2, pages 842-848, July 1991.
- [21] Johan de Kleer. Optimizing focusing model-based diagnosis. In *Working Papers of the Third International Workshop on Principles of Diagnosis (DX-92)*, pages 26-29, 1992.
- [22] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56:197-222, 1992.
- [23] Johan de Kleer and Olivier Raiman. How to diagnose very well with very little information. In *Working Papers of the Fourth International Workshop on Principles of Diagnosis (DX-93)*, pages 160-165, 1993.
- [24] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97-130, 1987.
- [25] Johan de Kleer and Brian C. Williams. Diagnosis with behavioral modes. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, volume 2, pages 1324-1330, 1989.
- [26] Saumya K. Debray. On the complexity of dataflow analysis of logic programs. Technical report, Dept. of Computer Science, University of Arizona, Tucson, AZ 85721, USA, 1994.
- [27] Nachum Dershowitz and Yuh-Jeng Lee. Deductive debugging. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 298-306. IEEE Computer Society Press, 1987.
- [28] A.J.J. Dick, P.J. Krause, and J. Cozens. Computer-aided transformation of Z into Prolog. In J.E. Nicholls, editor, *Z User Workshop, Oxford 1989*, Workshops in Computing, pages 71-85. Springer-Verlag, 1990.
- [29] Wlodek Drabent, Simin Nadjm-Tehrani, and Jan Maluszyński. Algorithmic debugging with

- assertions. In H. Abramson and M.H. Rogers, editors, *Meta-programming in logic programming*, pages 501-521. MIT Press, 1989.
- [30] Wlodek Drabnet, Simin Hadjm-Tehrani, and Jan Maluszynski. The use of assertions in algorithmic debugging. In Institute for New Generation Computer Technology (ICOT), editor, *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 2, pages 573-581. Springer-Verlag, 1988.
- [31] M.H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733-742, October 1976.
- [32] K. Eshghi and R.A. Kowalski. Abduction compared with negation by failure. In G. Levi and M. Martelli, editors, *Logic Programming—Proceedings of the Sixth International Conference*, pages 234-254. MIT Press, 1989.
- [33] Gérard Ferrand. Error diagnosis in logic programming, an adaptation of E.Y. Shapiro's method. *Journal of Logic Programming*, 4:177-198, 1987.
- [34] Gérard Ferrand. The notions of symptom and error in declarative diagnosis of logic programs. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging (AADEBUG '93)*, pages 33-49, May 1993.
- [35] K.D. Forbus and Johan de Kleer. Focusing the ATMS. In *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI-88)*, pages 193-198, 1988.
- [36] G. Friedrich, G. Gottlob, and W. Nejdl. Formalizing the repair process. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI '92)*, pages 709-713, 1992.
- [37] Gerhard Friedrich. Theory diagnoses: A concise characterization of faulty systems. In *Working Papers of the 3rd International Workshop on Principles of Diagnosis (DX-92)*, pages 117-125, 1992.
- [38] Gerhard Friedrich, Georg Gottlob, and Wolfgang Nejdl. Physical impossibility instead of fault models. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, volume 1, pages 331-336, 1990.
- [39] Peter Fritzson, Tibor Gyimothy, Mariam Kamkar, and Nahid Shahmehri. Generalized algorithmic debugging and testing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 317-326. ACM Press, June 1991.
- [40] Hector Geffner. Beyond negation as failure. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on the Principles of Knowledge Representation and Reasoning (KR '91)*, pages 218-229, 1991.
- [41] Michael R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24:411-436, 1984.
- [42] 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, 1989. Research Note.
- [43] Ajay Gupta. Hardware diagnosis as program debugging. In *International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 524-526, 1987.

- [44] Walter Hamscher and Randall Davis. Diagnosing circuits with state: An inherently under-constrained problem. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-84)*, Austin, Texas, pages 142-147. American Association on Artificial Intelligence (AAAI), 1984.
- [45] R. Hill. LUSH-resolution and its completeness. Technical Report DCL Memo 78, Dept. of Artificial Intelligence, University of Edinburgh, 1974.
- [46] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [47] Peter Jackson. Computing prime implicates incrementally. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, number 607 in Lecture Notes in Artificial Intelligence (subseries of Lecture Notes in Computer Science), pages 253-267. Springer-Verlag, 1992.
- [48] J. Jaffar and J. L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111-119, 1987.
- [49] Alex Kean and George Tsiknis. Assumption based reasoning and clause management systems. Technical Report 90-9, Department of Computer Science, University of British Columbia, 1990.
- [50] Alex Kean and George Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9:185-206, 1990.
- [51] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, 1978.
- [52] D.G. Kourie. The design and use of a Prolog trace generator for CSP. *Software—Practice and Experience*, 17(7):423-438, 1987.
- [53] Krzysztof Kuchcinski, Wlodzimierz Drabent, and Jan Maluszynski. Automatic diagnosis of VLSI digital circuits using algorithmic debugging. In *Local Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging (AADEBUG '93)*, pages 382-392, 1993.
- [54] Yossi Lichtenstein and Ehud Shapiro. Abstract algorithmic debugging. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, volume 1 of *MIT Press Series in Logic Programming*, pages 512-531. MIT Press, 1988.
- [55] Yossi Lichtenstein and Ehud Shapiro. Concurrent algorithmic debugging. Technical Report CS87-20, The Weizmann Institute of Science, Dept. of Computer Science, Rehovot 76100, Israel, 1987. Also published in *Proceedings of ACM SIGPOS/SIGPLAN Workshop on Distributed and Parallel Debugging*, Madison, May 1988.
- [56] J.W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5:133-154, 1987.
- [57] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
- [58] J.W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217-242, 1991.
- [59] Luigi Logrippo, Didier Simon, and Hasan Ural. Executable description of the OSI transport service in Prolog. In Y. Yemini, R. Strom, and S. Yemini, editors, *Proceedings of Protocol*

- Specification, Testing and Verification IV*, pages 279–293, 1985.
- [60] Lunjin Lu. A generic declarative diagnoser for normal logic programs. In *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning (LPAR '94)*, 1994.
 - [61] M. Maeji and T. Kanamori. Top-down zooming diagnosis of logic programs. Technical Report TR-290, ICOT, August 1987.
 - [62] S. McIlraith. Generating tests using abduction. In *Working Papers of the 4th International Workshop on Principles of Diagnosis*, pages 223–235, 1993.
 - [63] S. McIlraith and R. Reiter. On tests for hypothetical reasoning. In L. Console, W. Hamscher, and J. de Kleer, editors, *Readings in Model-Based Diagnosis*, pages 89–96. Morgan-Kaufmann, 1992.
 - [64] Igor Mozetič. Hierarchical model-based diagnosis. *International Journal of Man-Machine Studies*, 35(3):329–362, 1991. Also in *Readings in Model-Based Diagnosis*, W. Hamscher, L. Console, J. de Kleer eds., Morgan Kaufmann, 1992.
 - [65] Igor Mozetič. A polynomial-time algorithm for model-based diagnosis. In B. Neumann, editor, *10th European Conference on Artificial Intelligence (ECAI 92)*, pages 729–733, 1992.
 - [66] Igor Mozetič. Computing minimal diagnoses with critical set algorithms. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 657–661. John Wiley and Sons, 1994.
 - [67] Igor Mozetič, Franc Novak, Marina Santo-Zarnik, and Anton Biasizzo. Diagnosing analog circuits designed-for-testability by using CLP($\mathcal{R}$). In *Working Papers of the Fourth International Workshop on Principles of Diagnosis (DX-93)*, pages 105–120, 1993.
 - [68] Stephen Muggleton. Inductive logic programming: derivations, successes and shortcomings. *SIGART Bulletin*, 5(1):5–11, January 1994.
 - [69] Lee Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10:255–285, 1991.
 - [70] Lee Naish, Philip W. Dart, and Justin Zobel. The NU-Prolog debugging environment. In G. Levi and M. Martelli, editors, *Proceedings of the 6th International Conference on Logic Programming (ICLP '89)*, pages 521–536. MIT Press, 1989.
 - [71] Wolfgang Nejdl and Gerhard Friedrich. Constructive utility in model-based diagnosis/repair systems. In *Working Papers of the Fourth International Workshop on Principles of Diagnosis (DX-93)*, pages 296–307, 1993.
 - [72] J. S. Ostroff. Constraint logic programming for reasoning about discrete event processes. *Journal of Logic Programming*, 11:243–270, 1991.
 - [73] D.J. Out and R.P. van Rikxoort. Constructing hierarchic models in linear time. In *Working Papers of the 3rd International Workshop on Principles of Diagnosis (DX-92)*, pages 40–49, 1992.
 - [74] Luis Moniz Pereira. Rational debugging in Prolog programs. In Ehud Shapiro, editor, *Third International Conference in Logic Programming*, number 225 in Lecture Notes in Computer

- Science, pages 203-210. Springer-Verlag, 1986.
- [75] Luis Moniz Pereira and Miguel Calejo. A framework for Prolog debugging. In R.A. Kowalski and K. A. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming*, volume 1 of *MIT Press Series in Logic Programming*, pages 481-495. MIT Press, 1988.
 - [76] Lutz Plümer. *Termination Proofs in Logic Programs*. Number 446 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1990.
 - [77] D. L. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36:22-47, 1988.
 - [78] David Poole. Explanation and prediction: an architecture for default and abductive reasoning. *Computational Intelligence*, 5(2):97-110, May 1989.
 - [79] Gregory M. Provan. A complexity analysis of assumption-based truth maintenance systems. In Barbara Smith and Gerald Kelleher, editors, *Reason Maintenance Systems and Their Applications*, chapter 6, pages 98-113. Ellis Horwood, 1988.
 - [80] R. Reiter. On closed world databases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 119-140. Plenum Publishing, 1978.
 - [81] R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems: Preliminary report. In *Proceedings of AAAI-87*, pages 183-188. AAAI, 1987.
 - [82] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57-95, 1987.
 - [83] Bart Selman and Hector J. Levesque. Abductive and default reasoning: A computational core. In *Proceedings of the 8th National Conference on Artificial Intelligence*, volume 1, pages 343-348, 1990.
 - [84] Nahid Shahmehri, Mariam Kamkar, and Peter Fritzson. Semi-automatic bug localization in software maintenance. In *Proceedings of the Conference on Software Maintenance-1990*, pages 30-36. IEEE Computer Society Press, 1990.
 - [85] Ehud Y. Shapiro. *Algorithmic Program Debugging*. ACM Distinguished Dissertation Series. MIT Press, 1983. Doctoral Dissertation May 1982, Dept. of Computer Science, Yale University.
 - [86] J.M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988.
 - [87] Susan Stepney, Dave Whitley, David Cooper, and Colin Grant. A demonstrably correct compiler. *Formal Aspects of Computing*, 3:58-101, 1991.
 - [88] Peter Struss and Oskar Dressler. Physical negation: Integrating fault models into the general diagnostic engine. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, volume 2, pages 1318-1323, 1989.
 - [89] Akikazu Takeuchi. Algorithmic debugging of GHC programs and its implementation in GHC. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2 of *MIT Press Series in Logic Programming*, pages 180-196. MIT Press, 1987.

- [90] T.H. Tse, T.Y. Chen, F.T. Chan, H.Y. Chen, and H.L. Xie. The application of Prolog to structured design. *Software—Practice and Experience*, 24, 1994.
- [91] Yury V. Tsybenko. On choice the actual diagnosis in assumption-based model. In *Working Papers of the Fourth International Workshop on Principles of Diagnosis (DX-93)*, pages 308-315, 1993.
- [92] S. Y. Yan. Foundations of declarative debugging in arbitrary logic programming. *International Journal of Man-Machine Studies*, 32:215-232, 1990.

3 of/de 3

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

APPENDIX A

Implementation of the TP Diagnosis Verification Procedure

```
% (c) Copyright 1994, Carleton University

% GWBond July 1993
% GWBond revised May 1994
% bond@sce.carleton.ca

:- module(tp,[tp/3,tpVerbose/3]).

/*

Implementation of the function "TP" for diagnosis verification using
mixed positive and negative assertions. From the Ph.D. dissertation
"Logic Programs for Consistency Based Diagnosis," by G.W. Bond,
Dept. of Systems and Computer Engineering, Carleton University,
Ottawa, Canada, 1994.

Calls to Quintus Prolog built-in predicates are commented with
'Quintus'.

*/

:- ensure_loaded(library(basics)).
:- ensure_loaded(library(ordsets)).
:- ensure_loaded([library(occurs),library(strings)]).

/*

tp(+OBS_PLUS,+OBS_MINUS,+DELTA)

"TP" (theorem prover) is a 4 argument function that returns either
'true' or 'false'. Here the function is implemented as a 3 argument
predicate "tp" that succeeds if the diagnosis is confirmed or fails if
```

172 A. IMPLEMENTATION OF THE TP DIAGNOSIS VERIFICATION PROCEDURE

the diagnosis is denied. We exclude the first argument PI, representing a logic program, from the list of arguments in the implementation. Instead, the predicates loadPi/1 and unloadPi/0 are provided in module 'pi-io' to load and unload resp. a logic program for use with the 'tp' module predicates.

OBS_PLUS is a list of atoms representing existentially quantified positive assertions, OBS_MINUS is a list of atoms representing universally quantified negative assertions, and DELTA is a list of predicate name/arity terms that are to be considered as possible fault candidates.

First generate a positive delta-support for the positive assertions OBSP with respect to the logic program loaded into the dynamic clause space of 'tp'. Then Skolemize the variables in the support. Finally check whether the support is valid w.r.t. the negative assertions OBSM.

*/

```
tp(OBSP,OBSM,DELTA) :-
    genSupport(OBSP,DELTA,PSUPPORT),
    skolemizeSupport(PSUPPORT,SPSUPPORT),
    validSupport(OBSM,DELTA,SPSUPPORT).
```

/*

```
tpVerbose(+OBS_PLUS,+OBS_MINUS,+DELTA)
```

Same as tp/3 except that execution status is printed to standard output.

*/

```
tpVerbose(OBSP,OBSM,DELTA) :-
    format('~n*** Generating supports for ~p...~n',[DELTA]), % Quintus
    genSupport(OBSP,DELTA,PSUPPORT),
    skolemizeSupport(PSUPPORT,SPSUPPORT),
    format('~n*** ~p is a support.~n',[SPSUPPORT]), % Quintus
    onBacktracking(formac('~n*** Support failed, generating new support...~n',[])),
    validSupport(OBSM,DELTA,SPSUPPORT),
    format('~n*** ~p is a diagnosis.~n',[DELTA]). % Quintus
```

```
tpVerbose(_,_,DELTA) :-
    format('~n*** No (more) supports.~n',[]), % Quintus
    format('~n*** ~p is not a diagnosis.~n',[DELTA]), % Quintus
    !,fail.
```

A. IMPLEMENTATION OF THE TP DIAGNOSIS VERIFICATION PROCEDURE 173

/*

skolemizeSupport(+INLITERALS,-OUTLITERALS)

Transforms list of literals INLITERALS into Skolemized list of
literals OUTLITERALS.

*/

skolemizeSupport([],[]) :- !.

skolemizeSupport([H|T1],[H|T2]) :-

ground(H),!, % Quintus

skolemizeSupport(T1,T2).

skolemizeSupport([H1|T1],[H2|T2]) :-

copy_term(H1,H2), % Quintus

skolemize(H2),!

skolemizeSupport(T1,T2).

skolemize(T) :-

ground(T). % Quintus

skolemize(T) :-

varInTerm(V,T),!

gensym(V), % Quintus

skolemize(T).

varInTerm(V,T) :-

sub_term(V,T), % Quintus

var(V).

/*

genSupport(+OBSP,+DELTA,-PSUPPORT)

Finds a positive delta-support for the list of atoms OBSP with respect
to the logic program PI loaded into the dynamic clause space of the
'tp' module. DELTA is a list of name/arity terms representing
potentially faulty components. PSUPPORT is a list of atoms that
constitute the support. An empty list denotes the propositional
symbol 'true'. This predicate implements Steps 1, 2 and 3 of the TP
procedure as described in the dissertation. The predicate initially
succeeds and returns empty support if OBSP is an empty list. Predicate
initially fails when at least one positive assertion fails to succeed.
On backtracking, finds another support in the SLD-tree. Fails on
backtracking if no (more) supports exist i.e. fails on backtracking
after a complete set of supports has been found.

*/

174 A. IMPLEMENTATION OF THE TP DIAGNOSIS VERIFICATION PROCEDURE

```

genSupport([],_,[]) :- !.
genSupport(OBSP,DELTA,PSUPPORT) :-
    succeedInPDP(OBSP,DELTA,PSUPPORT).

/*

succeedIn PDP(+OBSP,+DELTA,-PSUPPORT)

Finds a positive delta-support in the form of a list of atoms PSUPPORT
for a non-empty list OBSP of atoms which represent a non-empty set of
positive assertions. This is achieved by converting the list of
assertions OBSP into a multi-term goal and then using a
meta-interpreter to evaluate the goal using SLD-resolution in the
positive delta-transform PI-delta-plus of a logic program PI. Terms in
support are sorted only for output purposes when called by tpVerbose/3.

*/

succeedInPDP(OBSP,DELTA,PSUPPORT) :-
    goalClause(OBSP,GOBSP),
    evalPDP(GOBSP,DELTA,X/X,SUPPORT/[]),
    list_to_ord_set(SUPPORT,PSUPPORT).      % Quintus

/*

goalClause(+ATOMLIST,-GOAL)

Convert a list of atoms ATOMLIST to a single goal GOAL.

*/

goalClause([X],X).
goalClause([X|Y],(X,Z)) :-
    goalClause(Y,Z).

/*

evalPDP(+GOAL,+DELTA,+INSUPPORT,-OUTSUPPORT)

Meta-interpreter for a goal GOAL in the positive delta-transform
PI-delta-plus of a logic program PI. DELTA is a list of name/arity
terms that denote possible faulty components. INSUPPORT and OUTSUPPORT
are difference lists that denote the initial and final list of atoms
resp. that constitute a positive delta support for GOAL.

*/

```

A. IMPLEMENTATION OF THE TP DIAGNOSIS VERIFICATION PROCEDURE 175

```

evalPDP(A,DELTA,X/[A|Y],X/Y) :- % succeed immediately and add literal
    \+ A = (_,_),                % to support if literal is a delta-literal
    deltaLiteral(A,DELTA),!.
evalPDP(A,_,X/Y,X/Y) :-          % do not consider built-in predicates as
    \+ A = (_,_),                % fault candidates - evaluate normally
    predicate_property(A,built_in),!.    % Quintus
A.
evalPDP((A,B),DELTA,X/Y1,X/Y3) :- % evaluate a goal of the form (A,B)
    evalPDP(A,DELTA,X/Y1,X/Y2),
    evalPDP(B,DELTA,X/Y2,X/Y3).
evalPDP(A,DELTA,X/Y1,X/Y2) :- % evaluate a non-delta literal
    \+ A = (_,_),
    clause(A,B),                  % Quintus
    evalPDP(B,DELTA,X/Y1,X/Y2).

```

/*

validSupport(+OBSM,+DELTA,+PSUPPORT)

Succeeds if all negative assertions of the list of atoms OBSM fail in the logic program consisting of the negative delta-transform PI-delta-minus of the logic program PI, and the the (Skolemized) positive delta-support, represented by the list of Skolemized atoms PSUPPORT. DELTA is a list of name/arity terms representing possible faulty components. This predicate represents Step 4 of the TP procedure as described in the dissertation.

*/

```

validSupport([],_,_) :- !.
validSupport([OBS|T],DELTA,PSUPPORT) :-
    \+ evalPDM(OBS,DELTA,PSUPPORT),
    validSupport(T,DELTA,PSUPPORT).

```

/*

evalPDM(+GOAL,+DELTA,+PSUPPORT)

Meta-interpreter for a goal GOAL in the negative delta-transform PI-delta-minus of a logic program PI and the Skolemized terms of a positive delta-support represented by the list of terms PSUPPORT. DELTA is a list of name/arity terms that denote possible faulty components.

*/

176 A. IMPLEMENTATION OF THE TP DIAGNOSIS VERIFICATION PROCEDURE

```

evalPDM(A,DELTA,PSUPPORT) :- % delta-literal goal resolves with a
    \+ A = (.,.),           % term in the support
    deltaLiteral(A,DELTA),!,
    member(A,PSUPPORT).      % Quintus
evalPDM(A,.,.) :-           % built-in predicates evaluated normally
    \+ A = (.,.),
    predicate_property(A,built_in),!, % Quintus
    A.
evalPDM((A,B),DELTA,PSUPPORT) :- % goal of the form (A,B) evaluated
    evalPDM(A,DELTA,PSUPPORT),
    evalPDM(B,DELTA,PSUPPORT).
evalPDM(A,DELTA,PSUPPORT) :- % non-delta literal evaluated
    \+ A = (.,.),
    clause(A,B),             % Quintus
    evalPDM(B,DELTA,PSUPPORT).

```

/*

deltaLiteral(+LITERAL,+DELTA)

Succeed if LITERAL is a delta-literal, otherwise fail. DELTA is a list of name/arity terms that denote possible faulty components.

*/

```

deltaLiteral(A,DELTA) :-
    functor(A,NAME,ARITY),      % Quintus
    memberchk(NAME/ARITY,DELTA). % Quintus

```

/*

onBacktracking(+GOAL)

When called initially do nothing. On backtracking through initial call, evaluate GOAL.

*/

```

onBacktracking(_).
onBacktracking(X) :-
    X,!,fail.

```

APPENDIX B

Interface to the IDA Generator Algorithm

This appendix contains the initialization files and customization files for loading and interfacing the TP procedure listed in the previous appendix with Mozetič's IDA algorithm. Since two of the IDA files were altered to interface with the TP procedure we also include their listings here. The remaining files for implementing the IDA algorithm can be obtained from Igor Mozetič: contact *igor@ai.univie.ac.at*. In summary, this appendix contains the following listings:

- diagnose.pl**: the top level load file,
- customize.pl**: the interface procedures between TP and IDA,
- pi-io.pl**: IO procedures for TP,
- ARIAI Software Disclaimer**: Austrian Research Institute for Artificial Intelligence software disclaimer for altered IDA code,
- ARIAI User Agreement**: Austrian Research Institute for Artificial Intelligence user agreement for altered IDA code,
- my-init.pl**: altered version of IDA's *init.pl* file,
- my-kdiags.pl**: altered version of IDA's *kdiags.pl* file.

1. diagnose.pl

```
% (c) Copyright 1994, Carleton University

% GWBond July 1993
% bond@sce.carleton.ca

:- ensure_loaded('my-init'). % load file for IDA algorithm

% Before calling IDA's ida or k_diags predicates, ensure
```



```
% that the following fact is asserted: state_components(L),
% where L is a list of name/arity terms of potentially
% faulty system components.

:- ensure_loaded(customize). % interface procedures between
                             % IDA and TP

:- ensure_loaded('tp').      % TP diagnosis verification procedure
:- ensure_loaded('pi-io'). % Predicates for loading/unloading logic
                             % program for 'tp' module
```

2. customize.pl

```
% (c) Copyright 1994, Carleton University

% GWBond July 1993
% bond@ace.carleton.ca

% Hacked work-arounds to interface IDA
% to TP. Problematic due to
% IDA's use of global variables!

label_to_delta(LABEL_TERM,DELTA) :-
    LABEL_TERM =.. [_|LABEL],
    state_components(COMP), % Me
    label_to_delta(LABEL,COMP,DELTA).

label_to_delta([],[],[]).
label_to_delta([OK|T1],[_|T2],DELTA) :-
    state_normal(_,OK),
    label_to_delta(T1,T2,DELTA).
label_to_delta([AB|T1],[COMP|T2],[COMP|T3]) :-
    state_abnormal(_,AB),
    label_to_delta(T1,T2,T3).

state_size(N) :-
    state_components(L), % Me
    length(L,N).        % Quintus

state_normal(_,ok).
state_abnormal(_,ab).

state_functor(label).
```

3. pi-io.pl

```
% (c) Copyright 1994, Carleton University
```

[illegible]

```

        assert(tp:X),
        assertPi(S),!.
assertPi(S) :-
        close(S),!.                                % Quintus

/*

unloadPi

Removes all clauses in dynamic clause space of 'tp' module.

*/

unloadPi :-
        predicate_property(tp:X,dynamic),          % Quintus
        functor(X,NAME,ARITY),                      % Quintus
        abolish(tp:NAME/ARITY),                    % Quintus
        fail.
unloadPi :- !.

```

4. ARIAI Software Disclaimer

As unestablished, research software, this program is provided free of charge on an "as is" basis without warranty of any kind, either expressed or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose. ARIAI does not warrant that the functions contained in this program will meet the user's requirements or that the operation of this program will be uninterrupted or error-free. Acceptance and use of this program constitutes the user's understanding that he will have no recourse to ARIAI for any actual or consequential damages, including, but not limited to, lost profits or savings, arising out of the use or inability to use this program. Even if the user informs ARIAI of the possibility of such damages, ARIAI expects the user of this program to accept the risk of any harm arising out of the use of this program, or the user shall not attempt to use this program for any purpose.

5. ARIAI User Agreement

By acceptance and use of this experimental program the user agrees to the following:

- (1) This program is provided for the user's personal, non-commercial, experimental use and the user is granted permission to copy this program to the extent reasonably required for such use.
- (2) All title, ownership and rights to this program and any copies remain with ARIAI, irrespective of the ownership of the media on which the program resides.
- (3) The user is permitted to create derivative works to this program. However, all copies of the program and its derivative works must contain the ARIAI copyright notice, the RESEARCH SOFTWARE DISCLAIMER and this USER AGREEMENT.
- (4) By furnishing this program to the user, ARIAI does NOT grant either directly or by implication, estoppel, or otherwise any license under any patents, patent applications, trademarks, copyrights or other rights belonging to ARIAI or to any third party, except as expressly provided herein.
- (5) The user understands and agrees that this program and any derivative works are to be used solely for experimental uses and are not to be sold, distributed to a commercial organization, or be commercially exploited in any manner.
- (6) ARIAI requests that the user supply to ARIAI a copy of any changes, enhancements, or derivative works which the user may create. The user grants ARIAI and its subsidiaries an irrevocable, nonexclusive, worldwide and royalty-free license to use, execute, reproduce, display, perform, prepare derivative works based upon, and distribute, (INTERNALLY AND EXTERNALLY) copies of any and all such materials and derivative works thereof, and to sublicense others to do any, some, or all of the foregoing, (including supporting documentation).

6. my-init.pl

```
%% IDA (c) Copyright 1992, Austrian Research Institute for Artificial Intelligence
%%
%% File:    init
%% Author:  Igor Mozetic (igor@ai.univie.ac.at)
%% Date:    April 15, 1992

% GWBond revised July 1993 (see "% GWB" comments)
% bond@sce.carleton.ca
```

```

%
% Top level loading directives.
% An interface to the model <SD, Comps, Obs> must be defined in terms of
% the following predicates (see examples):
%
% model( Comps, Obs )      - a binary predicate which relates Comps to Obs,
% observation( Obs )      - an instance of Obs (optional),
% state_functor( Funct )  - a functor of Comps,
% state_size( N )         - the arity of Comps,
% state_normal( I, Ok )   - a constant which denotes normal state in Comps
%                           at argument position I (must be singleton),
% state_abnormal( I, Ab ) - a constant which denotes abnormal state in Comps
%                           (there can be several abnormal states at I).
%

:- op( 600, xfy, ':' ).

%load_file( File ) :- consult( File ).    % C-Prolog
load_file( File ) :- compile( File ).    % Quintus, SICStus Prolog

:- load_file( 'my-kdiags.pl' ). % GWB
:- load_file( 'hset.pl' ).
:- load_file( 'lattice.pl' ).
:- load_file( 'ords.pl' ). % sets represented by ordered lists
%:- load_file( 'bits.pl' ). % sets represented by bit vectors

```

7. my-kdiags.pl

```

%% IDA (c) Copyright 1992, Austrian Research Institute for Artificial Intelligence
%%
%% File:   kdiags.pl
%% Author: Igor Mozetic (igor@ai.univie.ac.at)
%% Date:   April 15, 1992
%
% Implements IDA - an Incremental Diagnostic Algorithm.

% GWBond revised May 1994 (see "% GWB" comments)
% bond@sce.carleton.ca

ida( Diags, Confs ) :-
    observation( Obs ),
    k_diags( 1000, Obs, Diags, Confs ).

ida( Obs, Diags, Confs ) :-
    k_diags( 1000, Obs, Diags, Confs ).

```

```

k_diags( K, Obs, Diags, Confs ) :-
    k_diags( 0, K, [], [], Diags, Confs, Obs ).

% k_diags( +N, +K, +Diags0, +Confs0, -Diags, -Confs, +Obs )
% Given an initial set of minimal Diags0 and (non-minimal) Confs0,
% computes next K-N minimal diagnoses. Returns updated Diags and Confs.

k_diags( N, K, Diags0, Confs0, Diags, Confs, Obs ) :-
    N < K,
    gen_label( Diags0, Confs0, Label ), !,
    verify_label( Label, Diags0, Confs0, Diags1, Confs1, Obs ),
    ( Diags0 == Diags1 -> N1 = N ; N1 is N+1 ),
    k_diags( N1, K, Diags1, Confs1, Diags, Confs, Obs ).
k_diags( _, _, Diags, Confs, Diags, Confs, _ ).

% verify_label( +Label, +Diags0, +Confs0, -Diags, -Confs, +Obs )
% Verifies whether Label is a diagnosis or a conflict. If it is
% a (non-minimal) Diag0 then min_diag/6 returns a minimal Diag and
% previous Diags0 are updated into new Diags. Otherwise Label is
% a Conf, and Confs0 are updated into Confs.

verify_label( Label, Diags0, Confs0, Diags, Confs, Obs ) :-
    call_model( Label, Diag0, Obs ), !,
    gen_succs( Diag0, Succs ),
    min_diag( Succs, Diag0, Confs0, Diag, Confs, Obs ),
    ord_insert( Diags0, Diag, Diags ).
% latt_insert( Diags0, Diag, Diags ).
verify_label( Conf, Diags, Confs0, Diags, Confs, _ ) :-
    latt_replace( Confs0, Conf, Confs ).

% min_diag( +Labels, +Diag0, +Confs0, -Diag, -Confs, +Obs )
% For a (non-minimal) Diag0 and a list of its direct successors Labels
% returns a minimal Diag. Also updates Confs0 into Confs.

min_diag( [], Diag, Confs, Diag, Confs, _ ) :- !. % no more Labels
min_diag( [Conf|Labels], Diag0, Confs0, Diag, Confs, Obs ) :- % Label is in Confs0,
    latt_member( Conf, Confs0 ), !, % don't call TP
    min_diag( Labels, Diag0, Confs0, Diag, Confs, Obs ).
min_diag( [Label|_], _, Confs0, Diag, Confs, Obs ) :- % Label yields Diag0,
    call_model( Label, Diag0, Obs ), % a new candidate for
    gen_succs( Diag0, Succs ), !, % a minimal Diag
    min_diag( Succs, Diag0, Confs0, Diag, Confs, Obs ).
min_diag( [Conf|Labels], Diag0, Confs0, Diag, Confs, Obs ) :- % Label is Conf
    latt_replace( Confs0, Conf, Confs1 ), !,

```

```

min_diag( Labels, Diag0, Confs1, Diag, Confs, Obs ).

% call_model( +Label, -Diag, +Obs )
% Implements the TP function.
% Label, Obs -> model( Diag, Obs )

call_model( Label, Diag, [OBSP,OBEM] ) :-          % GWB
    ords_ground_state( Label, Ground ), % to ensure P complexity
    label_to_delta(Ground,DELTA),              % GWB
    tpVerbose(OBSP,OBEM,DELTA),                % GWB
%   model( Ground, Obs ),                      % for a weak fault model % GWB
%   ords_state( Label, State ),
%   model( State, Obs ),
    !,
    state_ords( Ground, Diag ).
%   state_ords( State, Diag ).

% gen_succs( +Set, -[Sub|Subsets] )
% Subsets are immediate successors of Set.
% 3:[1,3,5] -> [2:[3,5], 2:[1,5], 2:[1,3]]

gen_succs( Set, Subsets ) :-
    ords_gen_subs( Set, Set, Subsets ).

% gen_label( +Diags, +Confs, -Label )
% Label is a complement of a hitting set of minimal Diags,
% such that it is not a SUBset of any Confs.
% [2:[1,2], 2:[1,5]] x 4:[1,2,3,4] -> 3:[1,3,4], 4:[2,3,4,5] -> 4:[2,3,4,5]
% Nondeterministic, NP complexity !

gen_label( Diags, Confs, Label ) :-
    hitting_set( Diags, Hset ),
    ords_complement( Hset, Label ),
    \+ latt_member( Label, Confs ).

```

APPENDIX C

Computing Diagnoses for the Buggy Insertion Sort Program

In this appendix we present two examples of using the implementation of the TP procedure given in Appendix A combined with the IDA algorithm to compute diagnoses for the buggy insertion sort logic program (Program III.2). The assertions used in both examples are the same as those used to illustrate Shapiro's *fp* algorithm in Example III.18 in Chapter III. The first example shows how minimal diagnoses are computed at the procedural level of granularity corresponding to Example V.5 in Chapter V. The second example illustrates how minimal diagnoses are computed at the clausal level of granularity corresponding to Example V.25 in Chapter V.

The following is a summary of the listings included in this appendix:

- isort.pl**: the buggy insertion sort logic program for computing diagnoses at the procedural level of granularity,
- isort-clause.pl**: the buggy insertion sort logic program for computing diagnoses at the clausal level of granularity,
- isort-ex.pl**: the file containing the two clauses that run the examples,
- isort-ex1.out**: the output listing for the first example,
- isort-ex2.out**: the output listing for the second example.

1. isort.pl

% (c) Copyright 1994, Carleton University

% GVBond Oct 1991

% bond@sce.carleton.ca

186C. COMPUTING DIAGNOSES FOR THE BUGGY INSERTION SORT PROGRAM

```
% buggy insertion sort program

% Program from E. Shapiro's spurious
% behavior example ref. pp. 41-42 of
% "Algorithmic Program Debugging"
```

```
isort([X|Xs],Ys) :-
    isort(Xs,Zs),
    insert(X,Zs,Ys).
isort([],[]).
```

```
insert(X,[Y|Ys],[Y|Zs]) :-
    X < Y, % bug
%     Y > X, % correct
    insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys]) :-
    Y >= X.
insert(X,[],[X]).
```

2. isort-clause.pl

```
% (c) Copyright 1994, Carleton University
```

```
% GWBond Oct 1991
% bond@sce.carleton.ca
```

```
% buggy insertion sort program
```

```
% Program transformed to refine granularity of diagnosis
% from predicate level to clause level
```

```
% program from Shapiro's spurious
% behavior example ref. pp. 41-42 of
% "Algorithmic Program Debugging"
```

```
isort(X,Y) :-
    isort1(X,Y).
isort(X,Y) :-
    isort2(X,Y).
```

```
isort1([X|Xs],Ys) :-
    isort(Xs,Zs),
    insert(X,Zs,Ys).
isort2([],[]).
```

```
insert(X,Y,Z) :-
```

```

        insert1(X,Y,Z).
insert(X,Y,Z) :-
        insert2(X,Y,Z).
insert(X,Y,Z) :-
        insert3(X,Y,Z).

insert1(X,[Y|Ys],[Y|Zs]) :-
        X < Y, % bug
%      Y > X, % correct
        insert(X,Ys,Zs).
insert2(X,[Y|Ys],[X,Y|Ys]) :-
        Y >= X.
insert3(X,[],[X]).

```

3. isort-ex.pl

% (c) Copyright 1994, Carleton University

% GWBond July 1994

% bond@sce.carleton.ca

% Examples of computing diagnoses for a buggy insertion sort logic
% program isort.pl.

% Assertions and program taken from E. Shapiro's spurious
% behavior example ref. pp. 41-42 of "Algorithmic Program Debugging"

:- ensure_loaded(diagnose).

% this example computes diagnoses at the predicate
% level of granularity

```

ex1(D) :-
    unloadPi,
    loadPiVerbose('isort.pl'),
    abolish(state_components/1),
    assert(state_components([isort/2,insert/3])),
    ida([isort([],[]),insert(3,[],[3]),isort([3],[3]),
        insert(1,[],[1]),
        [isort([2,1,3],[2,3,1]),insert(1,[3],[3,1])]],
        D,_),
    unloadPi.

```

% this example computes diagnoses at the clausal level
% of granularity

188C. COMPUTING DIAGNOSES FOR THE BUGGY INSERTION SORT PROGRAM

```
ex2(D) :-
    unloadPi,
    loadPiVerbose('isort-clause.pl'),
    abolish(state_components/1),
    assert(state_components([isort1/2,isort2/2,insert1/3,
                             insert2/3,insert3/3])),
    ida([[isort([],[]),insert(3,[],[3]),isort([3],[3]),
          insert(1,[],[1])],
         [isort([2,1,3],[2,3,1]),insert(1,[3],[3,1])]],
         D,_),
    unloadPi.
```

4. Output for Example 1

```
| ?- ex1(D).

**** Loading isort.pl... Done.

**** Generating supports for [isort/2,insert/3]...

**** [isort([],[]),isort([3],[3]),insert(1,[],[1]),insert(3,[],[3])] is a support.

**** [isort/2,insert/3] is a diagnosis.

**** Generating supports for [insert/3]...

**** [insert(1,[],[1]),insert(3,[],[3])] is a support.

**** [insert/3] is a diagnosis.

**** Generating supports for []...

**** [] is a support.

**** Support failed, generating new support...

**** No (more) supports.

**** [] is not a diagnosis.

**** Generating supports for [isort/2]...

**** [isort([],[]),isort([3],[3])] is a support.

**** Support failed, generating new support...
```

**** No (more) supports.

**** [isort/2] is not a diagnosis.

D = [1:[2]]

% This value for D indicates that just one minimal diagnosis
% exists where the diagnosis consists of just one abnormal
% component. The abnormal component is the 2nd component in
% the argument list of the asserted state_components/1
% predicate, which is insert/3.

5. Output for Example 2

| ?- ex?(D).

**** Loading isort-clause.pl... Done.

**** Generating supports for [isort1/2,isort2/2,insert1/3,insert2/3,insert3/3]...

**** [isort1([],[]),isort1([3],[3]),insert1(1,[],[1]),insert1(3,[],[3])] is a support.

**** [isort1/2,isort2/2,insert1/3,insert2/3,insert3/3] is a diagnosis.

**** Generating supports for [isort2/2,insert1/3,insert2/3,insert3/3]...

**** [isort2([],%1),isort2([],[]),insert1(1,[],[1]),insert1(3,%2,[3]),insert1(3,[],[3])] is a support.

**** [isort2/2,insert1/3,insert2/3,insert3/3] is a diagnosis.

**** Generating supports for [insert1/3,insert2/3,insert3/3]...

**** [insert1(1,[],[1]),insert1(3,[],[3])] is a support.

**** [insert1/3,insert2/3,insert3/3] is a diagnosis.

**** Generating supports for [insert2/3,insert3/3]...

**** [insert2(1,[],[1]),insert2(3,[],[3])] is a support.

**** Support failed, generating new support...

**** [insert2(3,[],[3]),insert3(1,[],[1])] is a support.

190C. COMPUTING DIAGNOSES FOR THE BUGGY INSERTION SORT PROGRAM

**** Support failed, generating new support...

**** [insert2(1,□,[1]),insert2(3,□,[3]),insert3(3,□,[3])] is a support.

**** Support failed, generating new support...

**** [insert2(3,□,[3]),insert3(1,□,[1]),insert3(3,□,[3])] is a support.

**** Support failed, generating new support...

**** [insert2(1,□,[1]),insert2(3,□,[3]),insert3(3,□,[3])] is a support.

**** Support failed, generating new support...

**** [insert2(3,□,[3]),insert3(1,□,[1]),insert3(3,□,[3])] is a support.

**** Support failed, generating new support...

**** [insert2(1,□,[1]),insert3(3,□,[3])] is a support.

**** Support failed, generating new support...

**** [insert3(1,□,[1]),insert3(3,□,[3])] is a support.

**** Support failed, generating new support...

**** No (more) supports.

**** [insert2/3,insert3/3] is not a diagnosis.

**** Generating supports for [insert1/3,insert3/3]...

**** [insert1(1,□,[1]),insert1(3,□,[3])] is a support.

**** [insert1/3,insert3/3] is a diagnosis.

**** Generating supports for [insert1/3]...

**** [insert1(1,□,[1]),insert1(3,□,[3])] is a support.

**** [insert1/3] is a diagnosis.

**** Generating supports for [isort1/2,isort2/2,insert2/3,insert3/3]...

**** [isort1(□,□),isort1([3],[3]),insert2(1,□,[1]),insert2(3,□,[3])] is a support.

**** Support failed, generating new support...

**** [isort1([],[]),isort1([3],[3]),insert2(3,[],[3]),insert3(1,[],[1])] is a support.
**** Support failed, generating new support...
**** [isort1([],[]),isort2([3],[3]),insert2(1,[],[1]),insert2(3,[],[3])] is a support.
**** Support failed, generating new support...
**** [isort1([],[]),isort2([3],[3]),insert2(3,[],[3]),insert3(1,[],[1])] is a support.
**** Support failed, generating new support...
**** [isort1([],[]),isort1([3],[3]),insert2(1,[],[1]),insert3(3,[],[3])] is a support.
**** Support failed, generating new support...
**** [isort1([],[]),isort1([3],[3]),insert3(1,[],[1]),insert3(3,[],[3])] is a support.
**** Support failed, generating new support...
**** [isort1([],[]),isort2([3],[3]),insert2(1,[],[1]),insert3(3,[],[3])] is a support.
**** Support failed, generating new support...
**** [isort1([],[]),isort2([3],[3]),insert3(1,[],[1]),insert3(3,[],[3])] is a support.
**** Support failed, generating new support...
**** [isort1([3],[3]),isort2([],[]),insert2(1,[],[1]),insert2(3,[],[3])] is a support.
**** Support failed, generating new support...
**** [isort1([3],[3]),isort2([],[]),insert2(3,[],[3]),insert3(1,[],[1])] is a support.
**** Support failed, generating new support...
**** [isort2([],[]),isort2([3],[3]),insert2(1,[],[1]),insert2(3,[],[3])] is a support.
**** Support failed, generating new support...
**** [isort2([],[]),isort2([3],[3]),insert2(3,[],[3]),insert3(1,[],[1])] is a support.
**** Support failed, generating new support...
**** [isort1([3],[3]),isort2([],[]),insert2(1,[],[1]),insert3(3,[],[3])] is a support.

192C. COMPUTING DIAGNOSES FOR THE BUGGY INSERTION SORT PROGRAM

```
**** Support failed, generating new support...

**** [isort1([3],[3]),isort2([],[]),insert3(1,[],[1]),insert3(3,[],[3])] is a support.

**** Support failed, generating new support...

**** [isort2([],[]),isort2([3],[3]),insert2(1,[],[1]),insert3(3,[],[3])] is a support.

**** Support failed, generating new support...

**** [isort2([],[]),isort2([3],[3]),insert3(1,[],[1]),insert3(3,[],[3])] is a support.

**** Support failed, generating new support...

**** No (more) supports.

**** [isort1/2,isort2/2,insert2/3,insert3/3] is not a diagnosis.

D = [1:[3]]

% This value for D indicates that just one minimal diagnosis
% exists where the diagnosis consists of just one abnormal
% component. The abnormal component is the 3rd component in
% the argument list of the asserted state_components/1
% predicate, which is insert1/3.
```

APPENDIX D

Computing Diagnoses for the Full-Adder Circuit

In this appendix we present two examples of using the implementation of the TP procedure given in Appendix A combined with the IDA algorithm to compute diagnoses for the faulty full-adder circuit (Figure V.11). The first example corresponds to Example V.3, and the second example corresponds to Example V.4.

The following is a summary of the listings included in this appendix:

- full-adder.pl:** the full-adder specification,
- full-adder-ex.pl:** the file containing the two clauses defining the two examples,
- full-adder-ex1.out:** the output listing for the first example,
- full-adder-ex2.out:** the output listing for the second example.

1. full-adder.pl

```
% (c) Copyright 1994, Carleton University
```

```
% GWBond Oct 1991
```

```
% bond@sce.carleton.ca
```

```
% full adder circuit specification
```

```
fullAdder(X,L,M,R,S) :-
```

```
    x1(X,L,M),
```

```
    a1(X,L,Q),
```

```
    a2(M,M,P),
```

```
    x2(M,M,R),
```

```
    o1(P,Q,S).
```

```
x1(X,Y,Z) :-
```



```

xor(X,Y,Z).
x2(X,Y,Z) :-
    xor(X,Y,Z).
a1(X,Y,Z) :-
    and(X,Y,Z).
a2(X,Y,Z) :-
    and(X,Y,Z).
o1(X,Y,Z) :-
    or(X,Y,Z).

```

```

and(1,0,0).
and(0,0,0).
and(0,1,0).
and(1,1,1).
or(1,1,1).
or(0,1,1).
or(1,0,1).
or(0,0,0).
xor(0,0,0).
xor(1,0,1).
xor(0,1,1).
xor(1,1,0).

```

2. full-adder-ex.pl

```
% (c) Copyright 1994, Carleton University
```

```
% GWBond July 1993
```

```
% bond@sce.carleton.ca
```

```
% full-adder diagnosis examples
```

```
:- ensure_loaded(diagnose).
```

```
% this example computes diagnoses for a single positive assertion
```

```

ex1(D) :-
    unloadPi,
    loadPiVerbose('full-adder.pl'),
    abolish(state_components/1),
    assert(state_components([fullAdder/5,x1/3,x2/3,a1/3,a2/3,o1/3])),
    ida([[fullAdder(1,0,1,1,0)],[]],D,_),
    abolish(state_components/1),
    unloadPi.

```

```
% this example computes diagnoses for a set of positive and negative
```

% assertions

```
ex2(D) :-
    unloadPi,
    loadPiVerbose('full-adder.pl'),
    abolish(state_components/1),
    assert(state_components([fullAdder/5,x1/3,x2/3,a1/3,a2/3,o1/3])),
    ida([fullAdder(1,0,1,1,0)], [x1(1,0,0),o1(1,_,0)]], D, _),
    abolish(state_components/1),
    unloadPi.
```

3. Output for Example 1

| ?- ex1(D).

**** Loading full-adder.pl... Done.

**** Generating supports for [fullAdder/5,x1/3,x2/3,a1/3,a2/3,o1/3]...

**** [fullAdder(1,0,1,1,0)] is a support.

**** [fullAdder/5,x1/3,x2/3,a1/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [x1/3,x2/3,a1/3,a2/3,o1/3]...

**** [a1(1,0,%1),a2(1,%3,%2),o1(%5,%4,0),x1(1,0,%6),x2(%7,1,1)] is a support.

**** [x1/3,x2/3,a1/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [x2/3,a1/3,a2/3,o1/3]...

**** [a1(1,0,%8),a2(1,1,%9),o1(%11,%10,0),x2(1,1,1)] is a support.

**** [x2/3,a1/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [a1/3,a2/3,o1/3]...

**** No (more) supports.

**** [a1/3,a2/3,o1/3] is not a diagnosis.

**** Generating supports for [x2/3,a2/3,o1/3]...

**** [a2(1,1,%12),o1(%13,0,0),x2(1,1,1)] is a support.

```
**** [x2/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [x2/3,o1/3]...

**** [o1(1,0,0),x2(1,1,1)] is a support.

**** [x2/3,o1/3] is a diagnosis.

**** Generating supports for [x2/3]...

**** No (more) supports.

**** [x2/3] is not a diagnosis.

**** Generating supports for [fullAdder/5,x1/3,a1/3,a2/3,o1/3]...

**** [fullAdder(1,0,1,1,0)] is a support.

**** [fullAdder/5,x1/3,a1/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [x1/3,a1/3,a2/3,o1/3]...

**** [a1(1,0,x14),a2(1,0,x15),o1(x17,x16,0),x1(1,0,0)] is a support.

**** [x1/3,a1/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [x1/3,a2/3,o1/3]...

**** [a2(1,0,x18),o1(x19,0,0),x1(1,0,0)] is a support.

**** [x1/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [x1/3,o1/3]...

**** [o1(0,0,0),x1(1,0,0)] is a support.

**** [x1/3,o1/3] is a diagnosis.

**** Generating supports for [x1/3]...

**** [x1(1,0,0)] is a support.

**** [x1/3] is a diagnosis.

**** Generating supports for [fullAdder/5,a1/3,a2/3,o1/3]...

**** [fullAdder(1,0,1,1,0)] is a support.
```

```

**** [fullAdder/5,a1/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [fullAdder/5,a2/3,o1/3]...

**** [fullAdder(1,0,1,1,0)] is a support.

**** [fullAdder/5,a2/3,o1/3] is a diagnosis.

**** Generating supports for [fullAdder/5,o1/3]...

**** [fullAdder(1,0,1,1,0)] is a support.

**** [fullAdder/5,o1/3] is a diagnosis.

**** Generating supports for [fullAdder/5]...

**** [fullAdder(1,0,1,1,0)] is a support.

**** [fullAdder/5] is a diagnosis.

**** Generating supports for [x2/3,a1/3,a2/3]...

**** [a1(1,0,0),a2(1,1,0),x2(1,1,1)] is a support.

**** [x2/3,a1/3,a2/3] is a diagnosis.

**** Generating supports for [x2/3,a2/3]...

**** [a2(1,1,0),x2(1,1,1)] is a support.

**** [x2/3,a2/3] is a diagnosis.

**** Generating supports for [x2/3,a1/3]...

**** No (more) supports.

**** [x2/3,a1/3] is not a diagnosis.

D = [1:[1],1:[2],2:[3,5],2:[3,6]]

% This value for D indicates that 4 minimal diagnoses
% exist. An element of the form n:[m1,...,mn] belonging to D
% indicates that there exists a minimal diagnosis with n
% abnormal components, and that these components are the
% elements located at positions m1,...,mn in the argument list
% of the asserted state_components/1 predicate. Therefore, in

```

% this example we have 1:[fullAdder/5], 1:[x1/3],
 % 2:[x2/3,a2/3], 2:[x2/3,o1/3].

4. Output for Example 2

```
| ?- ex2(D).

**** Loading full-adder.pl... Done.

**** Generating supports for [fullAdder/5,x1/3,x2/3,a1/3,a2/3,o1/3]...

**** [fullAdder(1,0,1,1,0)] is a support.

**** [fullAdder/5,x1/3,x2/3,a1/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [x1/3,x2/3,a1/3,a2/3,o1/3]...

**** [a1(1,0,%20),a2(1,%22,%21),o1(%24,%23,0),x1(1,0,%25),x2(%26,1,1)] is a support.

**** [x1/3,x2/3,a1/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [x2/3,a1/3,a2/3,o1/3]...

**** [a1(1,0,%27),a2(1,1,%28),o1(%30,%29,0),x2(1,1,1)] is a support.

**** [x2/3,a1/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [a1/3,a2/3,o1/3]...

**** No (more) supports.

**** [a1/3,a2/3,o1/3] is not a diagnosis.

**** Generating supports for [x2/3,a2/3,o1/3]...

**** [a2(1,1,%31),o1(%32,0,0),x2(1,1,1)] is a support.

**** [x2/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [x2/3,o1/3]...

**** [o1(1,0,0),x2(1,1,1)] is a support.

**** Support failed, generating new support...
```

**** No (more) supports.

**** [x2/3,o1/3] is not a diagnosis.

**** Generating supports for [x2/3,a2/3]...

**** [a2(1,1,0),x2(1,1,1)] is a support.

**** [x2/3,a2/3] is a diagnosis.

**** Generating supports for [fullAdder/5,x1/3,a1/3,a2/3,o1/3]...

**** [fullAdder(1,0,1,1,0)] is a support.

**** [fullAdder/5,x1/3,a1/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [x1/3,a1/3,a2/3,o1/3]...

**** [a1(1,0,%33),a2(1,0,%34),o1(%36,%35,0),x1(1,0,0)] is a support.

**** Support failed, generating new support...

**** No (more) supports.

**** [x1/3,a1/3,a2/3,o1/3] is not a diagnosis.

**** Generating supports for [fullAdder/5,a1/3,a2/3,o1/3]...

**** [fullAdder(1,0,1,1,0)] is a support.

**** [fullAdder/5,a1/3,a2/3,o1/3] is a diagnosis.

**** Generating supports for [fullAdder/5,a2/3,o1/3]...

**** [fullAdder(1,0,1,1,0)] is a support.

**** [fullAdder/5,a2/3,o1/3] is a diagnosis.

**** Generating supports for [fullAdder/5,o1/3]...

**** [fullAdder(1,0,1,1,0)] is a support.

**** [fullAdder/5,o1/3] is a diagnosis.

**** Generating supports for [fullAdder/5]...

**** [fullAdder(1,0,1,1,0)] is a support.

```

**** [fullAdder/5] is a diagnosis.

**** Generating supports for [x1/3,x2/3,a1/3,o1/3]...

**** [a1(1,0,%37),o1(0,%38,0),x1(1,0,0),x2(0,1,1)] is a support.

**** Support failed, generating new support...

**** [a1(1,0,%39),o1(1,%40,0),x1(1,0,1),x2(1,1,1)] is a support.

**** Support failed, generating new support...

**** No (more) supports.

**** [x1/3,x2/3,a1/3,o1/3] is not a diagnosis.

D = [1:[1],2:[3,5]]

% This value for D indicates that 2 minimal diagnoses
% exist. An element of the form n:[m1,...,mn] belonging to D
% indicates that there exists a minimal diagnosis with n
% abnormal components, and that these components are the
% elements located at positions m1,...,mn in the argument list
% of the asserted state_components/1 predicate. Therefore, in
% this example we have 1:[fullAdder/5], and 2:[x2/3,a2/3].

```

APPENDIX E

Computing Diagnoses for a Sequential Multiplier Circuit

In this appendix we present an example of using the implementation of the TP procedure given in Appendix A combined with the IDA algorithm to compute diagnoses for a faulty 4X4 bit sequential multiplier circuit. The specified structure of this circuit is shown in Figure E.16. This circuit was originally used as an example by Hamscher and Davis in [44] to illustrate the difficulty of diagnosing sequential circuits. In our example we illustrate the expressive power of logic programs by using a sequential circuit modelling scheme originally developed by Clocksin in [12] for the purpose of behavior simulation, signal flow determination, gate assignment problems and verification of hypothetical states. In the following example, we compute diagnoses assuming that all components below the top two levels of the structural hierarchy are normal.

The following is a summary of the listings included in this appendix:

- multiplier.pl:** the multiplier specification,
- mult-ex.pl:** the file containing the clause defining the diagnosis example,
- mult-ex.out:** the output listing for the example—here we only show the end result of the diagnosis.

FIGURE E.16. Structure of a 4X4 bit sequential multiplier circuit

202 E. COMPUTING DIAGNOSES FOR A SEQUENTIAL MULTIPLIER CIRCUIT

1. multiplier.pl

% (c) Copyright 1994, Carleton University

% GWBond April 1993

% bond@sce.carleton.ca

/*

Definitional diagnostic spec model of 4x4 sequential multiplier with
8-bit result register.

Multiplier consists of:

- a 4-bit shift-down register A,
- an 8-bit shift-up register B,
- an 8-bit load and store register Q,
- an 8-bit driver, and
- an 8-bit adder.

All registers are constructed from D flip-flops.
The driver is constructed from and gates.
The adder is constructed from full adders.

Circuit from: Hanscher and Davis, "Diagnosing Circuits with State: an
Inherently Underconstrained Problem," AAAI-84.

Definitional representation of circuits described in: W.F. Clocksin,
"Logic Programming and Digital Circuit Analysis", Journal of Logic
Programming, 1987.

*/

/*

mult_test(+A3,...,+A0,+B7,...,+B0,+CL,-Q7,...,-Q0)

Multiplier test circuit. Initializes the multiplier's A and B
registers to A3,...,A0 and B7,...,B0 respectively. Note that only
bottom 4 bits of B register are used for multiplication with A
register. The top 4 bits should be zero. Q register is initialized to
0. For a given list of clock pulses CL the test circuit records the Q
register next state outputs for successive clock pulses as lists
Q7,...,Q0.

*/

mult_test(A3,A2,A1,A0,

```

B7,B6,B5,B4,B3,B2,B1,B0,
CL,Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0) :-
    mult_init(A3,A2,A1,A0,
              B7,B6,B5,B4,B3,B2,B1,B0,
              NQ7,NQ6,NQ5,NQ4,NQ3,NQ2,NQ1,NQ0,
              NA3,NA2,NA1,NA0,
              NB7,NB6,NB5,NB4,NB3,NB2,NB1,NB0),
    mult_seq(CL,Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0,
            NQ7,NQ6,NQ5,NQ4,NQ3,NQ2,NQ1,NQ0,
            NA3,NA2,NA1,NA0,
            NB7,NB6,NB5,NB4,NB3,NB2,NB1,NB0).

```

/*

```

mult_init(+A3,...,+A0,+B7,...,+B0,
          -NQ7,...,-NQ0,-NA3,...,-NA0,-NB7,...,-NB0)

```

Multiplier initialization circuit. Initializes the multiplier's A and B registers to A3,...,A0 and B7,...,B0 respectively. Note that only bottom 4 bits of B register are used for multiplication with A register. The top 4 bits should be zero. Q register is initialized to 0. Initialized next states for registers A,B, and Q returned in NQ7,...,NQ0, NA3,...,NA0, NB7,...,NB0 respectively.

*/

```

mult_init(A3,A2,A1,A0,
          B7,B6,B5,B4,B3,B2,B1,B0,
          NQ7,NQ6,NQ5,NQ4,NQ3,NQ2,NQ1,NQ0,
          NA3,NA2,NA1,NA0,
          NB7,NB6,NB5,NB4,NB3,NB2,NB1,NB0) :-
    multiplier(1,1,A3,A2,A1,A0,
              B7,B6,B5,B4,B3,B2,B1,B0,
              0,0,0,0,0,0,0,0, % arbitrarily jam
              0,0,0,0,         % initial state
              0,0,0,0,0,0,0,0, % to zero
              NQ7,NQ6,NQ5,NQ4,NQ3,NQ2,NQ1,NQ0,
              NA3,NA2,NA1,NA0,
              NB7,NB6,NB5,NB4,NB3,NB2,NB1,NB0).

```

/*

```

mult_seq(+CLK,?Q07,...,?Q00,?Q7,...,?Q0,
        +A3,...,+A0,+B7,...,+B0)

```

Multiplier sequencer circuit. For a given list of clock pulses CLK the circuit records the Q register next state outputs for successive clock

204 E. COMPUTING DIAGNOSES FOR A SEQUENTIAL MULTIPLIER CIRCUIT

pulses as lists Q07,...,Q00. Initial states of registers Q, A, and B are Q7,...,Q0, A3,...,A0, B7,...,B0 respectively.

*/

```
mult_seq(0,0,0,0,0,0,0,0,0,0,
  --'-'-'-'-'-'-'-
  --'-'-'-
  --'-'-'-'-'-'-'-).
mult_seq([CL|CLT],
  [Q07|Q07T],[Q06|Q06T],[Q05|Q05T],[Q04|Q04T],
  [Q03|Q03T],[Q02|Q02T],[Q01|Q01T],[Q00|Q00T],
  Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0,
  A3,A2,A1,A0,
  B7,B6,B5,B4,B3,B2,B1,B0) :-
  multiplier(CL,0,
    A3,A2,A1,A0,          % not used
    B7,B6,B5,B4,B3,B2,B1,B0, % not used
    Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0,
    A3,A2,A1,A0,
    B7,B6,B5,B4,B3,B2,B1,B0,
    Q07,Q06,Q05,Q04,Q03,Q02,Q01,Q00,
    A03,A02,A01,A00,
    B07,B06,B05,B04,B03,B02,B01,B00),
  mult_seq(CLT,
    Q07T,Q06T,Q05T,Q04T,Q03T,Q02T,Q01T,Q00T,
    Q07,Q06,Q05,Q04,Q03,Q02,Q01,Q00,
    A03,A02,A01,A00,
    B07,B06,B05,B04,B03,B02,B01,B00)).

/*

multiplier(+CLK,+INIT,+A3,...,+A0,+B7,...,+B0,
  +Q07,...,+Q00,+A03,...,+A00,+B07,...,+B00,
  ?NQ07,...,?NQ00,?NA03,...,?NA00,?NB07,...,?NB00)
```

Multiplier circuit. Possible inputs are a clock pulse CLK, an initialization signal INIT and two values to be multiplied: A3,...,A0 and B7,...,B0. Note that only bits B3,...,B0 are used for multiplication with A3,...,A0. Input bits B7,...,B4 should be zero. Downward clock pulses (represented by 0) have no effect on the state of the circuit. An upward clock pulse (represented by a 1) combined with a high initialization signal (represented by a 1) initialize the A and B registers to the values A3,...,A0 and B7,...,B0 respectively. The Q register is initialized to zero. An upward clock pulse combined with a low initialization signal (represented by a zero) causes the value in the A register to shift down one bit, the value in the B

register to shift up one bit and the value fed back from the adder to the Q register input to be latched. The next state values of the Q, A and B registers are NQ07,...,NQ00, NAO3,...,NA00, NBO7,...,NBO0 respectively.

*/

```
multiplier(CLK,INIT,A3,A2,A1,A0,
           B7,B6,B5,B4,B3,B2,B1,B0,
           Q07,Q06,Q05,Q04,Q03,Q02,Q01,Q00,
           A03,A02,A01,A00,
           B07,B06,B05,B04,B03,B02,B01,B00,
           NQ07,NQ06,NQ05,NQ04,NQ03,NQ02,NQ01,NQ00,
           NAO3,NAO2,NAO1,NAO0,
           NBO7,NBO6,NBO5,NBO4,NBO3,NBO2,NBO1,NBO0) :-
  a_reg(A3,A2,A1,A0,CLK,INIT,
        A03,A02,A01,A00,
        NAO3,NAO2,NAO1,NAO0),
  b_reg(B7,B6,B5,B4,B3,B2,B1,B0,CLK,INIT,
        B07,B06,B05,B04,B03,B02,B01,B00,
        NBO7,NBO6,NBO5,NBO4,NBO3,NBO2,NBO1,NBO0),
  driver(B07,B06,B05,B04,B03,B02,B01,B00,A00,
        D07,D06,D05,D04,D03,D02,D01,D00),
  adder(D07,D06,D05,D04,D03,D02,D01,D00,
        Q07,Q06,Q05,Q04,Q03,Q02,Q01,Q00,
        _,S7,S6,S5,S4,S3,S2,S1,S0),
  q_reg(S7,S6,S5,S4,S3,S2,S1,S0,CLK,INIT,
        Q07,Q06,Q05,Q04,Q03,Q02,Q01,Q00,
        NQ07,NQ06,NQ05,NQ04,NQ03,NQ02,NQ01,NQ00).
```

/*

```
adder(+A7,...,+A0,+B7,...,+B0,?S8,...,?S0)
```

Combinational 8-bit adder circuit. Adds two inputs A7,...,A0 and B7,...,B0. The 9-bit result is S8,...,S0. The adder is composed of 8 full-adders. Note that in call to n_bit_adder the carry-out value from previous full-adder is fed into the carry-in of the next full-adder.

*/

```
adder(A7,A6,A5,A4,A3,A2,A1,A0,
      B7,B6,B5,B4,B3,B2,B1,B0,
      S8,S7,S6,S5,S4,S3,S2,S1,S0) :-
  n_bit_adder([A7,A6,A5,A4,A3,A2,A1,A0],
              [B7,B6,B5,B4,B3,B2,B1,B0],
              [C06,C05,C04,C03,C02,C01,C00,0],
```

206 E. COMPUTING DIAGNOSES FOR A SEQUENTIAL MULTIPLIER CIRCUIT

```
[S7,S6,S5,S4,S3,S2,S1,S0],
[S8,C06,C05,C04,C03,C02,C01,C00]).
```

/*

```
n_bit_adder(+AL,+BL,+CIL,?SL,?COL)
```

Recursive definition of n full-adders. AL is list of n A inputs, BL is list of n B inputs, CIL is list of n carry-in inputs, SL is list of n sum outputs and COL is list of n carry-out outputs.

*/

```
n_bit_adder([],[],[],[],[]).
n_bit_adder([A|AT],[B|BT],[CI|CIT],[S|ST],[CO|COT]) :-
    full_adder(A,B,CI,S,CO),
    n_bit_adder(AT,BT,CIT,ST,COT).
```

/*

```
full_adder(+A,+B,+CI,?S,?CO)
```

Full-adder circuit. A and B are input bits. CI is carry-in bit. S is sum bit and CO is carry-out bit.

*/

```
full_adder(0,0,0,0,0).
full_adder(0,0,1,1,0).
full_adder(0,1,0,1,0).
full_adder(0,1,1,0,1).
full_adder(1,0,0,1,0).
full_adder(1,0,1,0,1).
full_adder(1,1,0,0,1).
full_adder(1,1,1,1,1).
```

/*

```
driver(+I7,...,+I0,+CTL,?O7,...,?O0)
```

Driver circuit. If control bit CTL is 0 then output bits O7,...,O0 are 0. If control bit is 1 then output bits O7,...,O0 are same as input bits I7,...,I0 respectively.

*/

```
driver(I7,I6,I5,I4,I3,I2,I1,I0,CTL,
```

```

07,06,05,04,03,02,01,00) :-
  n_andg([CTL,CTL,CTL,CTL,CTL,CTL,CTL,CTL],
    [I7,I6,I5,I4,I3,I2,I1,I0],
    [O7,O6,O5,O4,O3,O2,O1,O0]).

```

```

/*

```

```

n_andg(+I1L,+I2L,?O1L)

```

Recursive definition of n AND gates. I1L is list of n values to one AND gate port, I2L is list of n values to the other AND gate input port and O1L is list of AND gate output values.

```

*/

```

```

n_andg([],[],[]).
n_andg([I1:I1T],[I2:I2T],[O1:O1T]) :-
  andg(I1 I2,O1),
  n_andg(I1T,I2T,O1T).

```

```

/*

```

```

andg(+I1,+I2,?O1)

```

AND gate circuit. I1 and I2 are input values. O1 is output value.

```

*/

```

```

andg(0,0,0).
andg(0,1,0).
andg(1,0,0).
andg(1,1,1).

```

```

/*

```

```

a_reg(+A3,...,+A0,+CLK,+INIT,+A03,...,+A00,
  ?NA03,...,?NA00)

```

8-bit shift-down register. If CLK=0 then register state unchanged. If CLK=1 and INIT=1 then A3,...,A0 loaded into register. Current register states A03,...,A00 are ignored in this case. If CLK=1 and INIT=0 then current register state A03,...,A00 is shifted down one bit. Input values A3,...,A0 are ignored in this case. NA03,...,NA00 are the next register states. Note that the bit values are ordered [LSB,...,MSB] in the call to n_bit_shift_reg to achieve shift-down effect.

```

*/

```

208 E. COMPUTING DIAGNOSES FOR A SEQUENTIAL MULTIPLIER CIRCUIT

```

a_reg(A3,A2,A1,A0,CLK,INIT,
      AO3,AO2,AO1,AO0,
      NAO3,NAO2,NAO1,NAO0) :-
    n_bit_shift_reg([A0,A1,A2,A3],CLK,INIT,
                    [AO0,AO1,AO2,AO3],
                    [NAO0,NAO1,NAO2,NAO3]).

```

/*

```

b_reg(+B7,...,+B0,+CLK,+INIT,+B07,...,+B00,
      ?NB07,...,?NB00)

```

8-bit shift-up register. If CLK=0 then register state unchanged. If CLK=1 and INIT=1 then B7,...,B0 loaded into register. Current register states B07,...,B00 are ignored in this case. If CLK=1 and INIT=0 then current register state B07,...,B00 is shifted up one bit. Input values B7,...,B0 are ignored in this case. NB07,...,NB00 are the next register states. Note that the bit values are ordered [MSB,...,LSB] in the call to n_bit_shift_reg to achieve shift-up effect.

*/

```

b_reg(B7,B6,B5,B4,B3,B2,B1,B0,CLK,INIT,
      B07,B06,B05,B04,B03,B02,B01,B00,
      NB07,NB06,NB05,NB04,NB03,NB02,NB01,NB00) :-
    n_bit_shift_reg([B7,B6,B5,B4,B3,B2,B1,B0],CLK,INIT,
                    [B07,B06,B05,B04,B03,B02,B01,B00],
                    [NB07,NB06,NB05,NB04,NB03,NB02,NB01,NB00]).

```

/*

```

n_bit_shift_reg(+DL,+CLK,+INIT,+QL,?NQL)

```

Recursive definition of n cascaded D flip-flops. DL is list of n flip-flop input values. CLK is clock pulse. INIT is initialization value. QL is list of n current state flip-flop outputs. NQL is list of n next state flip-flop outputs. If lists are ordered [LSB,...,MSB] then flip-flop interconnectivity achieves effect of a shift-up register. If lists are ordered [MSB,...,LSB] then flip-flop interconnectivity achieves effect of a shift-down register.

*/

```

n_bit_shift_reg([],...[],[]).
% downward clk pulse
n_bit_shift_reg([D|DT],0,...,[Q|QT],[NQ|NQT]) :-

```

```

    dff(D,0,Q,NQ),
    n_bit_shift_reg(DT,0,_,QT,NQT).
% init register
n_bit_shift_reg([D|DT],1,1,[Q|QT],[NQ|NQT]) :-
    dff(D,1,Q,NQ),
    n_bit_shift_reg(DT,1,1,QT,NQT).
% shift
n_bit_shift_reg([_],1,0,[Q],[NQ]) :-
    dff(0,1,Q,NQ).
n_bit_shift_reg([_|DT],1,0,[Q1,Q2|QT],[NQ|NQT]) :-
    dff(Q2,1,Q1,NQ),
    n_bit_shift_reg(DT,1,0,[Q2|QT],NQT).

```

```
/*
```

```

q_reg(+Q7,...,+Q0,+CLK,+INIT,+Q07,...,+Q00,
      ?NQ07,...,?NQ00)

```

8-bit load and store register. If CLK=0 then register state unchanged. If CLK=1 and INIT=1 then register initialized to zero. Current register states Q07,...,Q00 are ignored in this case. If CLK=1 and INIT=0 then input values Q7,...,Q0 are loaded into the register. Current register states Q03,...,Q00 are ignored. NQ03,...,NQ00 are the next register states.

```
*/
```

```

q_reg(Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0,CLK,INIT,
      Q07,Q06,Q05,Q04,Q03,Q02,Q01,Q00,
      NQ07,NQ06,NQ05,NQ04,NQ03,NQ02,NQ01,NQ00) :-
    n_bit_reg([Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0],CLK,INIT,
              [Q07,Q06,Q05,Q04,Q03,Q02,Q01,Q00],
              [NQ07,NQ06,NQ05,NQ04,NQ03,NQ02,NQ01,NQ00]).

```

```
/*
```

```

n_bit_reg(+DL,+CLK,+INIT,+QL,?NQL)

```

Recursive definition of n D flip-flops. DL is list of n flip-flop input values. CLK is clock pulse. INIT is initialization value. QL is list of n current state flip-flop outputs. NQL is list of n next state flip-flop outputs.

```
*/
```

```

n_bit_reg([],...,[],[]).

```


210 E. COMPUTING DIAGNOSES FOR A SEQUENTIAL MULTIPLIER CIRCUIT

```
% downward clk pulse
n_bit_reg([D|DT],0,_,[Q|QT],[NQ|NQT]) :-
    dff(0,0,Q,NQ),
    n_bit_reg(DT,0,_,QT,NQT).
% init register
n_bit_reg([_|DT],1,1,[Q|QT],[NQ|NQT]) :-
    dff(0,1,Q,NQ),
    n_bit_reg(DT,1,1,QT,NQT).
% load and store register
n_bit_reg([D|DT],1,0,[Q|QT],[NQ|NQT]) :-
    dff(D,1,Q,NQ),
    n_bit_reg(DT,1,0,QT,NQT).

/*

dff(+D,+CLK,+Q,?NQ)

D is data value input. CLK is clock input. Q is current state
output. NQ is next state output.

*/

dff(_,0,Q,Q).
dff(D,1,_,D).
```

2. mult-ex.pl

```
% (c) Copyright 1994, Carleton University

% GVBond July 1993
% bond@ace.carleton.ca

% sequential multiplier diagnosis examples

:- ensure_loaded(diagnose).

% example to compute diagnoses for top two levels of the hierarchy

ex1(D) :-
    unloadPi,
    loadPiVerbose('multiplier.pl'),
    abolish(state_components/1),
    assert(state_components([multiplier/54,adder/25,driver/17,
        a_reg/14,b_reg/26,q_reg/26])),
    statistics(runtime,[T0|_]),
    ida([[mult_test(0,1,1,0, % A3,A2,A1,A0
```

```

0,0,0,0,1,0,0,1, % B7,B6,B5,B4,B3,B2,B1,B0
[1,1,1,1], % CLF
[0,0,0,0], % Q7 [t1,t2,t3,t4]
[0,0,0,0], % Q6
[0,0,1,1], % Q5
[0,1,1,1], % Q4
[0,0,1,1], % Q3
[0,1,0,0], % Q2
[0,0,0,1], % Q1
[1,0,1,0], % Q0
],[],D,_),
statistics(runtime, [T1|_]).
T is T1 - T0,
format('*** Diagnosis took ~3d sec.~n', [T]),
abolish(state_components/1),
unloadPi.

```

3. Output of the Example

```
*** Diagnosis took 56.667 sec.
```

```
% this was actual time on a SPARC 2
```

```
D = [1:[1],1:[2],1:[3],1:[6],2:[4,5]]
```

```

% This value for D indicates that 5 minimal diagnoses
% exist. An element of the form n:[m1,...,mn] belonging to D
% indicates that there exists a minimal diagnosis with n
% abnormal components, and that these components are the
% elements located at positions m1,...,mn in the argument list
% of the asserted state_components/1 predicate. Therefore, in
% this example we have 1:[multiplier/54], 1:[adder/25],
% 1:[driver/17], 1:[q_reg/26], and 2:[a_reg/14,b_reg/26].

```

END

2 5-0 5-9 5

FIN