University of New Hampshire
University of New Hampshire Scholars' Repository

Doctoral Dissertations | Student Scholarship

Spring 2002

Diagnosing interoperability problems and debugging models by enhancing constraint satisfaction with case -based reasoning

Mohammed Houssaini Sqalli
University of New Hampshire, Durham

Follow this and additional works at: https://scholars.unh.edu/dissertation

Recommended Citation
Sqalli, Mohammed Houssaini, "Diagnosing interoperability problems and debugging models by enhancing constraint satisfaction with case -based reasoning" (2002). *Doctoral Dissertations*. 79.
https://scholars.unh.edu/dissertation/79

This Dissertation is brought to you for free and open access by the Student Scholarship at University of New Hampshire Scholars' Repository. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of University of New Hampshire Scholars' Repository. For more information, please contact nicole.hentz@unh.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

DIAGNOSING INTEROPERABILITY PROBLEMS AND DEBUGGING MODELS BY ENHANCING CONSTRAINT SATISFACTION WITH CASE-BASED REASONING

BY

Mohammed Houssaini Sqalli
Ingénieur d'Etat, EMI, Rabat, Morocco, 1992
M.S., University of New Hampshire, 1996

DISSERTATION

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Doctor of Philosophy
in
Engineering - Systems Design

May, 2002

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 3045339

UMI®

UMI Microform 3045339
Copyright 2002 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This dissertation has been examined and approved.

Dissertation Director, Eugene C. Freuder, Professor of Computer Science, Former Professor of Computer Science UNH, Director, Cork Constraint Computation Centre Science Foundation Ireland Research Professor

David W. Aha, Head of NCARAI's Intelligent Decision Aids Group, Naval Research Laboratory

Radim Bartos, Assistant Professor of Computer Science

Robert D. Russell, Associate Professor of Computer Science

William H. Lenharth, Associate Professor of Electrical and Computer Engineering

04-29-02

Date

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Dedication

To my mother Maria and

my father Brahim

"Thy Lord hath decreed that ye worship none but Him,

and that ye be kind to parents."

The Holy Quran. 17:23

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgments

All thanks are due to Allah, God Almighty. Because of Him, I was able to complete this work.

I am indebted to my supervisor, Professor Eugene C. Freuder, whose advice and encouragements have inspired and shaped much of my academic work.

I am grateful to Dr. David W. Aha, Professor Radim Bartos, Professor Robert D. Russell, and Dr. William H. Lenharth, the dissertation committee, for their careful review of my dissertation and valuable feedback.

I would like to thank all my colleagues at the UNH Constraint Computation Center (CCC) for discussions and feedback on this dissertation and earlier related research, namely Richard Wallace, Daniel Sabin, Mihaela Sabin, Peggy Eaton, Charles Elfe, and Paul Snow.

Special thanks to Scott Valcourt, Robert Blais, Adrian Stavish, Jonathan H. McKinney, Fred Mansfield, Joshua Bertoulin, and TJ Beach from the University of New Hampshire InterOperability Laboratory (UNH-IOL) for their support and their evaluation of the ADIOP system.

And last but not least, I owe a special debt to my wife Lamiae for her patience, constant support and comforting encouragement, to my son Abdoullah for sacrificing some of his time and joy to allow me to work on my dissertation, and to my little daughter Sarah who just turned one year old and did not get all the attention she deserves.

I am particularly grateful to Siemens Canada Ltd. - Telecom Innovation Centre, for providing me with support and leave of absence periods to complete my dissertation. I am also thankful to the Moroccan-American Commission for Education and Cultural Exchange (MACECE) for supporting me with a four year Fulbright grant. This material is based in part on work supported by MACECE, UNH-IOL and by the National Science Foundation under Grant No. IRI-9504316.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table of Contents

Dedication iii

Acknowledgments iv

Table of Contents v

List of Figures ix

List of Tables xi

Abstract xii

1 Introduction 1
- 1.1 Motivations . . . 3
  - 1.1.1 Interoperability Testing . . . 3
  - 1.1.2 Modeling . . . 4
  - 1.1.3 Diagnosis . . . 5
  - 1.1.4 Model Debugging . . . 7
- 1.2 Constraint Satisfaction Problems . . . 10
  - 1.2.1 Definition . . . 10
  - 1.2.2 Overview . . . 11
- 1.3 Interoperability Testing . . . 14
  - 1.3.1 Problem Statement . . . 15
  - 1.3.2 Environment: ATM Networks . . . 17
  - 1.3.3 Current Problem Solving Techniques . . . 19
  - 1.3.4 Proposed Problem Solving Technique . . . 21
- 1.4 CSP Modeling for Interoperability Testing . . . 21
- 1.5 Diagnosis of Interoperability Problems . . . 25
- 1.6 Debugging CSP Models . . . 27
- 1.7 Evaluation . . . 35
- 1.8 Contributions . . . 36
  - 1.8.1 Interoperability Testing . . . 36
  - 1.8.2 Constraint Satisfaction Problems . . . 37
  - 1.8.3 Case-Based Reasoning . . . 39
- 1.9 Dissertation Outline . . . 40

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2 CSP Modeling Using Object-Oriented Programming 42
2.1 Modeling and Constraint Satisfaction Problems . . . 43
2.2 Modeling Interoperability Testing . . . 44
2.2.1 One Model Architecture . . . 45
2.2.2 Many Models Architecture . . . 47
2.3 Object-Oriented Programming . . . 49
2.4 Description of the CSP Modeling Process . . . 51
2.5 Modeling with Objects . . . 57
2.5.1 Modeling of Packets . . . 57
2.5.2 Class Hierarchy and Inheritance . . . 59
2.5.3 Decoder . . . 61
2.6 Modeling Interface . . . 62
2.6.1 Variables . . . 63
2.6.2 Domains . . . 64
2.6.3 Constraints . . . 64
2.7 Test Cases as Objects . . . 65
2.8 Modeling Language . . . 67
2.9 Example of CSP Modeling for One Test Case . . . 69
2.10 Application of CSP modeling . . . 70
2.11 Evaluation . . . 73
2.11.1 Evaluation Setup . . . 74
2.11.2 ADIOP Modeling Component Evaluation . . . 75
2.11.3 Limitations . . . 80
2.12 Related Work . . . 81
2.13 Summary . . . 86

3 Constraint-Based Diagnosis of Interoperability Problems 87
3.1 Definitions . . . 88
3.2 Modeling, Decoding and Diagnosis . . . 92
3.3 Diagnosis of Interoperability Problems . . . 96
3.4 Algorithms for Diagnosis . . . 98
3.4.1 Constraint Satisfaction Methods . . . 99
3.4.2 Search . . . 102
3.4.3 Inference and Consistency Checking . . . 109
3.5 Explanation . . . 114
3.6 Test Case Execution . . . 120
3.6.1 Automate Menus Creation . . . 120
3.6.2 Reports Generation . . . 121
3.6.3 Algorithms . . . 123
3.7 Algorithms Evaluation . . . 123
3.7.1 Solvability . . . 123
3.7.2 Explanation . . . 127

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.7.3 Complexity 128
3.8 Evaluation Performed by Testers 128
3.8.1 Decoder 129
3.8.2 Diagnoser 133
3.8.3 ADIOP's General Survey Analysis 142
3.8.4 Limitations 145
3.8.5 Conclusion 147
3.9 Related Work 149
3.10 Summary 153

4 Case-Based Reasoning and Model Debugging 154
4.1 Motivations and Contributions 155
4.2 Advantages 157
4.3 Incompleteness and Incorrectness in the CSP model 158
4.4 Taxonomy of Types of Model Incompleteness and Incorrectness 159
4.4.1 Practical Examples of Incompleteness/Incorrectness in Interoperability Testing 160
4.4.2 Types of Incomplete and Incorrect Models 165
4.4.3 One Type of Model Inconsistency 169
4.5 Case-Based Reasoning 170
4.6 CSP/CBR Integration 173
4.7 CBR/CSP Integration Components of ADIOP 175
4.7.1 Advisor 175
4.7.2 Development Process and Case Collection 177
4.7.3 Case Representation 179
4.7.4 Case Retrieval 182
4.7.5 Case Reuse/Adaptation 191
4.7.6 Case Revision 193
4.7.7 Case Retainment - Learning 194
4.8 Updating CSP Models 195
4.9 Improving Explanations 199
4.10 Experiments and Evaluation 201
4.10.1 Experiments 201
4.10.2 Solvability 204
4.10.3 Evaluation of the CBR system 205
4.10.4 Evaluation of Explanation Improvement 207
4.10.5 Model Updates 208
4.11 Related Work 209
4.12 Summary 213

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5 Conclusion **215**
5.1 CSP Modeling 215
5.2 Constraint-Based Diagnosis 217
5.3 CSP Model Debugging 219
5.4 Directions for Future Work 221
5.5 Conclusion 222

A Test Case Layout **230**

B Testers Evaluation Questionnaire of ADIOP **232**

C ADIOP V2.0 User Manual **248**

D Approval of Protocols from the Institutional Review Board (IRB) **279**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Figures

1.1 Map Coloring Problem 11
1.2 Constraint Graph 12
1.3 CSP Solution 13
1.4 CSP for Problem Representation and Problem Solving 13
1.5 Interoperability Testing of Devices A and B 16
1.6 Physical Setup of Interoperability Testing 20
1.7 CSP Modeling and Diagnosis for Interoperability Testing 22
1.8 A Modeling Example 24
1.9 Diagnosis Example using an Explanation Template 28
1.10 Integration of CSP Model and CBR for Interoperability Testing 29
1.11 Initial CSP Model of the Time Variables of Test Case ID: V4302H_005 . . 33
1.12 Retrieved Similar Case Example 34
1.13 Updated Test Case CSP Model 34
1.14 Corrected CSP Model of Figure 1.11 35

2.1 One Model Architecture 46
2.2 Many Models Architecture 47
2.3 CSP Variable Assignment 48
2.4 A Modeling Example 54
2.5 Packet's Parameters List 57
2.6 Directory Structure of the *packet* Package 59
2.7 Class Hierarchy of the **Packet** Class 60
2.8 Protocols List in the Test Suite Builder Window 63
2.9 Packet Types List in the Test Suite Builder Window 64
2.10 The *testsuite* Directory Hierarchy 66
2.11 Test Suite Menu 66
2.12 The Test Suite Builder Window 71

3.1 Diagnosis of Interoperability Problems 90
3.2 Statement of Interoperability Problems 91
3.3 Modeling, Decoding and Diagnosis Components 92
3.4 ADIOP's Main Window 93
3.5 List of Protocol Analyzers Supported 94
3.6 The Decoder/Diagnoser Window 95
3.7 Test Suite Menu 95

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.8 CSP for Problem Representation and Problem Solving 99
3.9 Backtrack Algorithm 103
3.10 Check Function 105
3.11 GetValue Function 106
3.12 Solver Function 106
3.13 The Test Cases (Objects) Hierarchy 107
3.14 The *testsuite* Directory Hierarchy 107
3.15 ADIOP's Result Window of a Successful Test Case 108
3.16 Protocol Preprocess Function 112
3.17 Packet Type Preprocess Function 113
3.18 ADIOP's Result when packets of a packet type are fewer than required . . . 118
3.19 ADIOP's Result a "Pass With Warning" Test Case 119
3.20 ADIOP's Result Window Showing a Test Case Model 121
3.21 ADIOP's Test Cases Report of One Section 122

4.1 Initial CSP model 165
4.2 CSP model updated when variable X becomes optional 166
4.3 CSP model updated when variable X is removed 167
4.4 Initial CSP model 167
4.5 CSP model updated when there is a false constraint 168
4.6 CSP model updated when a constraint is removed 168
4.7 Case-Based Reasoning Cyclical Process 171
4.8 Case-Based Reasoning Process 172
4.9 Integration of CSP Model and CBR for Interoperability Testing 174
4.10 Test Case Result containing an 'Advisor' button 175
4.11 Advisor/CBR Window 176
4.12 A Partial View of the Case Base Table 178
4.13 A case displayed using the ADIOP's GUI 182
4.14 Cases' Types 189
4.15 Retrieve Similar Cases Menu 190
4.16 Similar Cases Table 190
4.17 Case Adaptation Window 191
4.18 Case Adaptation Menu 192
4.19 Window for Case Revision of the Adapted Case 193
4.20 Update Test Case Model Menu 196
4.21 Updated Test Case Model 197
4.22 Run Updated Test Case 198
4.23 Result of Running an Updated Test Case 198
4.24 Explanation Generated for Test Case V4301H_003 199
4.25 Similar Cases for the failure in Test Case V4301H_003 200
4.26 Relevant Retrieved Cases 204

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Tables

3.1 Summary of Explanation Templates 120
3.2 Results of Running Test Cases on Capture capt001 124
3.3 Summary of Results of Running Test Cases on Different Captures 126
3.4 Summary of results for manual vs. ADIOP testing 135

4.1 Results of Advisor on Capture capt001 202
4.2 Results of Running Test Cases on 10 Captured Data 203
4.3 Useful Explanation vs. Relevant Retrieved Cases 208

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

DIAGNOSING INTEROPERABILITY PROBLEMS AND DEBUGGING MODELS BY ENHANCING CONSTRAINT SATISFACTION WITH CASE-BASED REASONING

by

Mohammed Houssaini Sqalli

University of New Hampshire, May, 2002

Modeling, *Diagnosis*, and *Model Debugging* are the three main areas presented in this dissertation to automate the process of *Interoperability Testing* of networking protocols. The dissertation proposes a framework that uses the *Constraint Satisfaction Problem* (CSP) paradigm to define a modeling language and problem solving mechanism for interoperability testing, and uses *Case-Based Reasoning* (CBR) for debugging interoperability test cases.

The dissertation makes three primary contributions:

1. Definition of a new modeling language using CSP and Object-Oriented Programming. This language is simple, declarative, and transparent. It provides a tool for testers to implement models of interoperability test cases. The dissertation introduces the notions of metavariables, metavalues and optional metavariables to improve the modeling language capabilities. It proposes modeling of test cases from test suite specifications

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that are usually used in interoperability testing performed manually by testers. Test suite specifications are written by organizations or individuals and break down the testing into modules of test cases that make diagnosis of problems more meaningful to testers.

2. Diagnosis of interoperability problems using search supplemented by consistency inference methods in a CSP context to support explanations of the problem solving behavior. These methods are adapted to the OO-based CSP context. Testers can then generate reports for individual test cases and for test groups from a test suite specification.

3. Detection and debugging of incompleteness and incorrectness in CSP models of interoperability test cases. This is done through the integration of two modes of reasoning, namely CBR and CSP. CBR manages cases that store information about updating models as well as cases that are related to interoperability problems where diagnosis fails to generate a useful explanation. For the latter cases, CBR recalls previous similar useful explanations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1

Introduction

This dissertation is on *modeling* using Constraint Satisfaction Problems (CSPs), constraint-based *diagnosis*, and CSP model *debugging*. The domain of application used is *interoperability testing* of networking protocols. This dissertation has been motivated by the work done at the University of New Hampshire InterOperability Laboratory (UNH-IOL).

Modeling, diagnosis, and debugging cover the process through which a problem (i.e., an interoperability test case) is implemented, corrected, executed, and its results explained. *Interoperability testing* involves testing whether two or more networking devices connected to each other and implementing the same protocol are operational. This is done by monitoring the data between these devices using analyzers, and then comparing the data observed with what is expected, i.e., what is stated in the specifications of the protocol tested.

This is a proof-of-concept dissertation where we show how CSP is used to successfully model test cases, diagnose interoperability problems, and generate useful explanations for interoperability testing. We also show how Case-Based Reasoning (CBR) supports CSP for debugging CSP models and improving on the explanations generated for interoperability testing. CSP has been proposed as a paradigm for modeling and diagnosing real-world

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

problems. In this dissertation, modeling and diagnosis are enhanced through the use of a simple modeling language based on Object-Oriented Programming (OOP) for modeling, and the generation of human-like explanations for diagnosis. Since CSP models can be incomplete or incorrect, CBR is integrated with CSP to provide the ability to debug these models. CBR is also used to improve on the explanations obtained in the diagnosis phase. CBR is useful here because similar problems tend to recur and have similar solutions.

This dissertation is focused on testing protocols that run over ATM (Asynchronous Transfer Mode) networks, and most of the examples used are taken from the PNNI protocol. In these examples, we have changed the names of some captured data files to remove company names and preserve privacy. We used instead a generic name such as 'capt00x'. We have also modified the real names for analyzers and used generic names such as 'Analyzer X'. ADIOP (Automated Diagnosis of InterOperability Problems) is a tool that was designed and implemented to prove the feasibility of the work presented and claims made in this dissertation. A Graphical User Interface (GUI) is used by the ADIOP system and provides a user-friendly interaction with testers.

In the following, we present the motivations for the topics of this dissertation. Then, the concept of CSP and interoperability testing are introduced, followed by sections for the three main topics of this dissertation. We then present the major contributions and give an outline of the dissertation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.1 Motivations

1.1.1 Interoperability Testing

One of the main challenges in interoperability testing at the UNH-IOL is how to debug and diagnose interoperability problems in a timely manner. At the present this is done manually at the UNH-IOL, which can be an exhausting task since there can be a large amount of data to check.

In summary, there are two major concerns to address:

1. Checking manually a large amount of decoded data to find out where there is a mismatch between what is expected and what is observed.

2. Spending a considerable amount of time in manually diagnosing problems that may have been diagnosed before at the UNH-IOL or in solving problems that are very similar to previous problems solved. Some numbers are provided in the evaluation sections on how long diagnosis takes. Traces of how previous problems have been solved are not usually kept for future reference.

This shows that there is a need to make the process of diagnosing interoperability problems easier, quicker and more efficient. The work we present in this dissertation aims at solving some of these problems by automating the process of running and debugging interoperability test cases and diagnosing interoperability problems through a user friendly interface. Evaluations included in the next chapters measure some of these statements.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.1.2 Modeling

In this dissertation, we are using CSP *modeling* to provide a simple language for creating CSP models, including variables and constraints, of interoperability test cases. This includes a user-friendly interface that is menu-driven and allows testers in the lab to automate test suites and generate reports.

CSP provides a declarative and transparent modeling language that allows for test cases to be modeled without knowledge of the details of the objects involved in this modeling. Only information about the functionality of these objects is needed by testers. This makes it easy to create models for test cases and, when necessary, to correct and update them. This also means that CSP models are concise since they do not include any details of implementation. The language used is easy to learn and use by testers. This language is also very expressive since it is based on CSP. In other formalisms used in similar application domains, there is a need to extend such formalisms to be able to represent fully and adequately all the information in a model. For example, the Finite State Machine (FSM) formalism has been extended and used in combination with CSP to provide a representation for such models (Riese 1993b).

CSP modeling provides testers with the ability to model test cases using interoperability test suite specifications, some of which are approved by standard bodies (e.g., ATM Forum). These are the same test cases used manually to check the interoperability of devices. These test cases are arranged using the same structure provided in the test suite specifications, including individual test cases and test groups, and thus are made easily accessible and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

usable by testers. This is the same structure used by testers for manual testing and it is familiar to them.

CSP modeling is domain-independent, which allows testers to model test cases for many ATM networking protocols. Since CSP is a framework that provides modeling functionality as well as problem solving methods, it is possible to model test cases as CSPs and then diagnose problems using these CSP models.

Modeling test cases means that they can be reused as often as necessary for diagnosing interoperability problems with no extra effort for testers. This makes modeling of test cases very useful for testers in the lab. New testers may also use test cases implemented by others who may have left the UNH-IOL and whose expertise is kept in the form of these automated test cases.

1.1.3 Diagnosis

The motivation for automating *diagnosis* in interoperability testing is to save time, to reduce repetitive manual testing, to store and reuse knowledge, to automate report generation, and in general to make testing easier and more efficient. The main focus in this dissertation is on how to generate a human-like explanation for interoperability testing results because this is the main goal of a tester.

Constraint-based diagnosis takes advantage of the structure of CSP representation in solving and diagnosing interoperability problems. In this dissertation, CSP methods, including search and inference, are improved and adapted to provide solutions to interoperability problems and to generate human-like explanations of test case results. The explanations

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

obtained provide useful information on the causes of success and failure of test cases, and are easy to check and verify by testers.

The algorithms used for diagnosis are of acceptable degree of complexity. Constraint-based diagnosis provides the ability to complete the execution of test cases quickly. This reduces the amount of time needed by testers to do testing. In interoperability testing, usually many test cases pass, and the human tester needs a lot of time to check just for this result. This makes an automated tool even more efficient when testing is successful.

The efficiency of diagnosis is increased by the re-usability of test cases available to testers and the consistent results obtained. If problems that occurred in the past were detected using an automated tool, it would be possible to reproduce the same results and thus the same diagnosis. But, it is possible for a tester to forget how the same problem was resolved in the past, or a tester with that experience may not be in the lab at the time and thus the knowledge of a diagnosis might be lost. In addition, less expertise is needed when using an automated tool to diagnose interoperability problems, and no previous knowledge of test cases is necessary for new testers in the lab to perform an automated diagnosis. Many ATM protocols can be used for diagnosis by the same automated tool. Automated diagnosis also increases efficiency by solving more problems that may not have been solved by already existing methods including manual testing.

A *decoder* is also needed to make this diagnosis possible. A decoder allows testers to get complete, correct and reliable decodes of data captured by different analyzers, and to check and compare specific fields of different packets decoded. This is provided through a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

user friendly interface for testers. The decoder decodes off-line the data captured by other analyzers used by testers at the UNH-IOL.

In summary, testers need a tool that improves interoperability testing through automation, and whose interface and results are accepted by testers as useful. An automated tool allows for less human intervention and hence decisions made about diagnosis are usually more objective. It is possible for testers to automate many tasks through its friendly user interface, including execution and report generation for individual test cases and test groups. The reports produced are understandable by the lab customers (i.e., vendors), because they follow the same structure defined in a test suite specification.

1.1.4 Model Debugging

Our objective is to have a system that detects and *debugs* inconsistencies in CSP models built by testers. These inconsistencies originate from different sources. They may be inconsistencies in the protocol specification document, in the test suite derived from it, or from the modeling of test cases performed by testers. Independently of the origin of these inconsistencies, we want to provide a way of detecting and resolving them. These inconsistencies are manifested as *incompleteness* or *incorrectness* of CSP models built by testers for different test cases.

This leads to another important motivation, and that is to provide a general framework for model acquisition and *debugging*. The idea is to develop automated ways to compensate for incompleteness and incorrectness of models. This is very useful for debugging models. It includes detecting inconsistencies and resolving them by either storing the information

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

about them for later use or by updating the model. Part of this motivation is to find a taxonomy of these inconsistencies. This provides a formal way for addressing different cases of incompleteness and incorrectness. We use both a bottom-up approach where examples from the domain of application are used as a starting point to come up with part of this taxonomy, and a top-down approach where we look at the concept of CSP modeling and how incompleteness and incorrectness can be manifested in these models. The development of a taxonomy of deficiencies and associated fixes was also motivated by a similar architecture in the work of (Winston 1975) on learning.

CBR implements the process of finding similar past occurrences and adapting them to new situations. CBR supports debugging by providing a retrieval function that recalls how previous problems were solved when a similar new problem is encountered. Cases that represent incompleteness and incorrectness models of test cases are stored and include how the debugging of these models is achieved.

The integration of CSP and CBR provides a framework where interoperability test cases are modeled as CSPs and enhanced with debugging capabilities through the use of CBR. In this dissertation, CBR is used with CSP and provides a module for updating and debugging CSP models. CSP models represent the core of the system, and CBR adds the missing elements in this model. CSP models are easier to use at first because of their generalization. The effectiveness of CSP models increases as more problems are solved because these models get updated by CBR as needed.

CSP model debugging enhances the correctness and completeness of test cases imple-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

mented by testers that are represented as CSP models. This allows the refinement of these models to become more robust for future testing. More details are provided in Chapter 4.

A model is debugged and updated through user interaction. This interaction with testers assures the system's accuracy in reporting results. The adaptation of previous cases to new ones is a major part of this interaction. The debugging component advises a tester by retrieving the most similar cases from a case base, and provides the tools to revise the retrieved cases. It also allows testers to make the final decision about how cases will be adapted, reused, and eventually stored.

Testers expect easy access to model debugging and CBR components when a failure of an interoperability test case occurs. For this reason, a friendly user interface that hides many details of the CBR system from the user is also key to the success of this application.

The information on updating CSP models is represented in cases using a CSP language. This assures uniformity of representation between the CSP models and the updating process. The language used for *updating* CSP models is simple and is based on the same syntax of the CSP modeling language. This makes it easy to understand by testers, to integrate with the modeling language, and to use for updating models.

Debugging also improves on diagnosis by generating useful explanations when diagnosis does not. This is achieved through other types of cases that are stored in a case base to represent actual explanations of interoperability problems when the explanation generated by diagnosis is not useful. A new explanation of these problems is also stored in these cases and can be recalled when future similar situations occur. These cases may not be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

related to the incompleteness or incorrectness of CSP models because they may not include information for updating models.

The cases stored in the case base are even more useful when the authors (i.e., testers) of these are not available. Other testers can then make use of these cases. Cases for debugging models from different protocols can be stored and reused.

1.2 Constraint Satisfaction Problems

1.2.1 Definition

Constraint satisfaction is a powerful and extensively used artificial intelligence paradigm (Freuder & Mackworth 1992). It is a natural way of representing problems because the user needs only to state the variables and constraints of the application domain to be modeled. An n-ary constraint is a constraint involving n variables (e.g., a binary constraint involves two variables).

In addition. CSP is applied in many different domains because of its simple but rich representation. *Constraint Satisfaction Problems* (*CSPs*) involve finding values for variables subject-to restrictions on which combinations of values are acceptable. A constraint graph is a representation of the CSP where the vertices are variables of the problem, and the edges are constraints between variables. Each variable has labels that are the potential values it can be assigned. CSPs are solved using search (e.g., backtrack) and inference (e.g., arc consistency) methods. CSP representations and methods can be used for modeling and solving many problems including interoperability testing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.2.2 Overview

One example that shows how CSP works is the coloring problem. A map coloring problem can be stated as follows: 'Given a map with N regions bordering each other and M colors that can be used to color each region. The problem is whether there is an assignment of one of the colors to each region such that no two neighbors (i.e., regions that share at least one border) have the same color.' Figure 1.1 shows a map coloring problem.

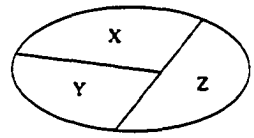

Figure 1.1: Map Coloring Problem

This problem can be represented as a Constraint Satisfaction Problem. The variables of this CSP represent the regions (X, Y and Z), the values are the different colors (red, blue and green), and the constraints are that no neighboring regions have the same color (i.e., no two variables representing two neighboring regions can be assigned the same value).

The constraint graph of this CSP is shown in Figure 1.2. The nodes represent variables, the labels for each node represent the domain of values for the corresponding variable, and the edges represent the constraints between different variables/nodes.

Many other toy problems such as the Queens problem can also be represented and solved using CSP, and these problems have helped in developing methods and tools that are used in real world applications. Many real world applications have used CSP for problem representation and modeling as well as for problem solving. These applications include: design

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

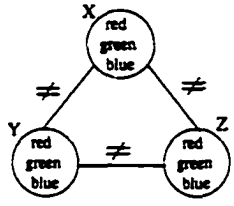

Figure 1.2: Constraint Graph

(Bilgic & Fox 1996) and configuration (Sabin & Freuder 1996) (Weigel & Faltings 1998), diagnosis (Sabin *et al.* 1995a) (Sabin *et al.* 1995b), debugging, verification, graphics, decision support, scheduling, planning (Avesani, Perini, & Ricci 1993), and resource allocation.

Different methods can be used to solve a CSP independently of the context of the application. The main two problem solving techniques are: Search and Inference. There are many algorithms that use search exclusively such as backtracking. Backtracking search may have to explore the entire tree of possibilities to find a solution. Other algorithms make use of inference such as Node Consistency (NC) and Arc Consistency (AC). Please see Section 3.4.3 for more details.

Research and experience have shown that the most successful techniques for solving CSPs are the ones that combine both search and inference. (Wallace 1996) states that arc-consistency techniques and backtrack search have sufficed for a number of practical applications of constraint programming. The question is then how and when do we combine these two to get the best results. That depends on the domain of application, the size of the problem, and the available resources (e.g., memory, etc).

Figure 1.3 shows one solution of the map coloring problem of Figure 1.1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

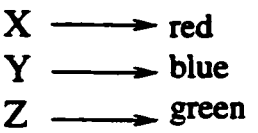

Figure 1.3: CSP Solution

The advantage of CSP is that it is a reasoning mode that provides both modeling and problem solving within the same framework (Figure 1.4). CSP provides a very simple and convenient way of representing problems since it is a natural and declarative approach to modeling. CSP is also domain independent, because it can hide many domain specific issues and be used at a more abstract level. When an application is represented as a CSP, it can be solved independently of the initial context or domain of application. The CSP methods are applied to the CSP representation of the problem, which hides the context used.

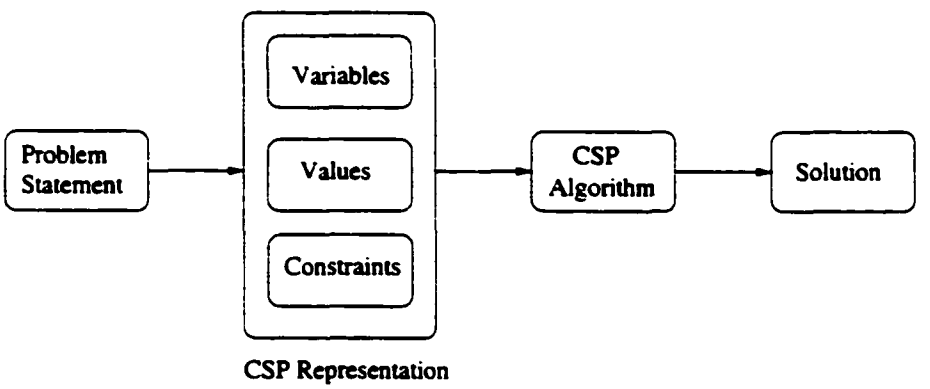

Figure 1.4: CSP for Problem Representation and Problem Solving

CSP provides many advanced algorithms to simplify or solve hard problems. CSP has been used in many real world applications as a modeling and a problem solving tool. In fact

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

commercial constraint programming systems have moved "beyond the black box" (Puget & Leconte 1995) and (Wallace 1996). Examples of problems that can naturally be expressed in terms of constraints include scheduling, configuration, design and diagnosis problems. These applications have improved the CSP paradigm and made it more widely used.

Because of the different applications and domains where the CSP paradigm has been used, there were also some extensions to it such as Partial CSP (Freuder & Wallace 1992), Dynamic CSP (Mittal & Falkenhainer 1990) and Composite CSP (Sabin & Freuder 1996) that enhance CSP capabilities. The CSP has a solution if there is an assignment of values to variables such that all the constraints are satisfied.

1.3 Interoperability Testing

One mission of the University of New Hampshire InterOperability Laboratory (UNH-IOL) is to provide testing services for vendors of computer communications devices. The UNH-IOL is mainly used by a community of over 200 vendors to verify the interoperability and/or conformance of their computer communications products. This service of the UNH-IOL is performed through independent focused interest groups in the lab, namely *consortiums*. The UNH-IOL currently has consortiums in operation to test many computer communications technologies, including Asynchronous Digital Subscriber Line (ADSL), Fast Ethernet (100Base-T), Fibre Channel, Gigabit Ethernet, IPv6, MPLS, SHDSL, Voice over Broadband, Wireless, and others. Check *http://www.iol.unh.edu* for more information on the UNH-IOL.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.3.1 Problem Statement

Networking Protocols are needed to specify how devices should behave in a specific environment. The protocol specification is a standard that many companies agree to implement on their hardware to assure compatibility with other vendors. One would assume that when two vendors implement the same protocol using the same specification, their products supporting this protocol will interoperate without any problems. However, experience has shown that this tends to be a false statement, because two devices that implement the same protocol may not behave in the same way. This can happen for many reasons, some of which are:

- The interpretation of the specification can be different from one vendor to another.
- The hardware used is different. The speed and memory size can affect the interaction between two devices and may cause problems such as delays in sending messages.
- The tools used for implementation can be different. The programming language and the operating system used can be different.
- Human error in coding and development of the implementation.

Interoperability testing is a diagnostic procedure that detects and debugs interoperability problems. An interoperability problem is defined as a problem that occurs because the two or more devices involved implement the same protocol but cannot communicate appropriately. Figure 1.5 shows how interoperability testing is done. Devices A and B are interoperable if the observations match the protocol specifications.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

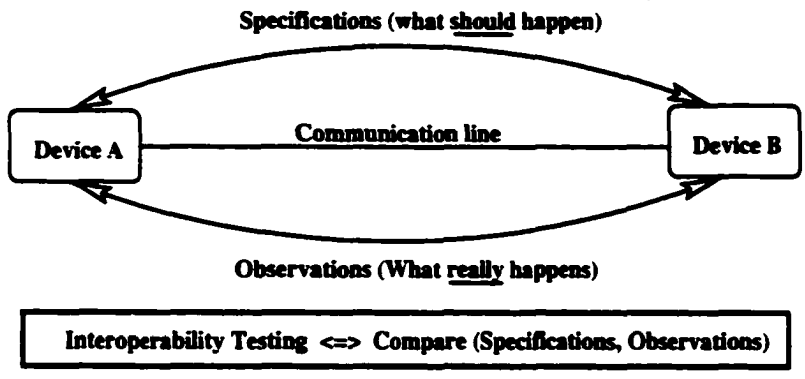

Figure 1.5: Interoperability Testing of Devices A and B

The primary focus of interoperability testing is to monitor the ability of a product to co-exist in a multiple vendor environment and operate with other products. In an industry that has many products from different manufacturers, companies need to ensure that their products are interoperable and remain competitive.

To make interoperability testing easier and more efficient, many organizations and companies develop and maintain interoperability test suites. The test suites are a vehicle by which vendors can verify that their products are interoperable and consistent with other vendors' products for the same technology. A test suite for a specific protocol is based on that protocol specification. It usually breaks down the testing into basic and small tests, each of which allows the testing of a particular issue in the corresponding protocol specification. The idea of a test suite is to make it easier to pinpoint where the problem is without having to test the whole protocol at once.

Another assumption that is frequently made is that the protocol specification and the test suite specification are correct and consistent. However, both of these types of specifica-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tions may be incomplete, inconsistent, ambiguous or incorrect. This may happen because of the following:

- A statement in the specification may be incorrect because of a human error.
- Statements in one section may be inconsistent with statements in another one in the protocol specification.
- Statements may be interpreted incorrectly when developing a test suite.

1.3.2 Environment: ATM Networks

The application domain used in this dissertation is interoperability testing of protocols in ATM networks. The protocol we mainly used is the PNNI (Private Network-Network Interface) protocol. The domain is generalized to many other ATM protocols such as MPOA (Multiple Protocol Over ATM), LANE (Local Area Network Emulation), and others. This shows that the system we developed can be used for other ATM protocols. Some results of the evaluation of data for different protocols is presented in the evaluation sections of different chapters.

- **Asynchronous Transfer Mode**

  Asynchronous Transfer Mode (ATM) has emerged as a networking technology capable of supporting all classes of traffic (e.g., voice, video, data). ATM uses fixed-size cells, each having 5 bytes header and 48 bytes payload. This allows the switching and multiplexing function to be done quickly and easily. ATM is a connection-oriented

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

technology. Thus, for two end systems to communicate, they need to establish a fixed path through which they will send their data. Each connection is called a virtual channel (VC). The virtual path identifier (VPI) and the virtual channel identifier (VCI) are associated with a particular channel. Every cell will have this information (VPI and VCI) in the header. In ATM, the network can guarantee a certain quality of service (QoS) requested by the user.

- **Private Network-Network Interface**

  The PNNI (Private Network-Network Interface) protocol provides dynamic routing, supports QoS, hierarchical routing, and scales to very large networks (PNNI-1.0 1996). Two devices (switches) running PNNI are able to send data to each other either via a direct link or by using a route. More than two devices might be running the PNNI protocol in the same network, but testing is usually performed using only two of these devices. The PNNI protocol is composed of PNNI routing that includes discovery of the topology of the network and becomes ready to route to different points in the network, and PNNI signaling, which is responsible for dynamically establishing, maintaining and clearing ATM connections between two ATM networks or two ATM nodes (PNNI-1.0 1996). The PNNI routing protocol starts when the link is up. Every switch should send HELLO packets (information about itself) during the Hello Protocol phase.

- **Interoperability Testing of the PNNI protocol in ATM Networks**

  Interoperability testing of PNNI allows us to detect problems that arise when two

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

or more devices supporting the PNNI protocol are connected. The network can be large with many devices connected. But, for simplicity we propose to work on a two-device network and perform interoperability testing on them. We assume that the two devices have passed conformance testing to exclude problems that may be detected by testing each device separately. Conformance testing checks whether a device reacts to specific events as it is described in the protocol specifications. We base our work on the ATM Forum document "AF-TEST-CSRA-0111.000" which provides the test suite for performing PNNI interoperability testing (PNNI-IOP 1999).

The monitor gets all the data (observations) necessary to test the interoperability of the devices attached to it. An observation is the data representing an event that occurred. After we get the results of monitoring all the traffic between the two devices, we analyze the data obtained and determine if both devices are interoperable.

1.3.3 Current Problem Solving Techniques

Interoperability testing is done by analyzing the data collected using monitors. These are usually connected to a device being tested. Figure 1.6 shows the physical setup and the steps for interoperability testing.

When two devices that implement the same protocol are being tested, a monitor is placed in the physical link that connects them. These two devices might be connected directly or through a network connecting other devices. This monitor allows the collection of the data that flows between the two devices. The monitor also provides the decoded version of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

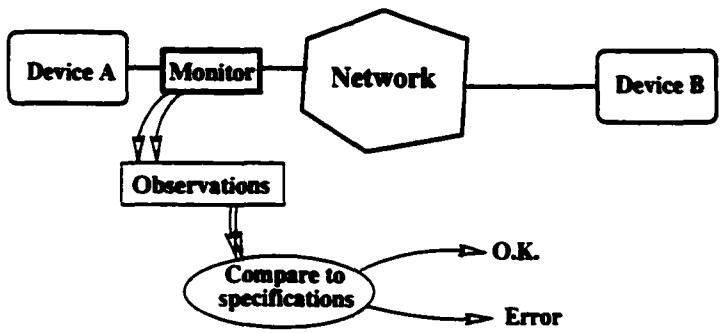

Figure 1.6: Physical Setup of Interoperability Testing

this data using the format of the protocol being tested. This decoded information is then analyzed manually by the tester to check whether the two devices are interoperable. The two devices are interoperable if the data observed by monitoring matches what is expected (what is stated in the specifications of the protocol tested). In the case where they do not match, an interoperability problem is suspected and the tester tries to explain what the problem is, possibly why there is such problem, and how to solve it.

At the present, these tasks are done by the testers who work at the UNH-IOL. Doing these steps manually has many disadvantages such as the large amount of time and effort spent for the analysis of interoperability testing and in solving similar problems if information on how they were solved in the past is not retained.

Test suites have been written to help in diagnosing the interoperability problems. But, using a test suite manually does not solve the above mentioned issues.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.3.4 Proposed Problem Solving Technique

We want to provide a system that automates the process of analyzing data and debugging the protocol and test suite specifications. This system should allow the user to easily state the model of the test case to be performed using a constraint representation.

First, each test case from the test suite is modeled as a Constraint Satisfaction Problem (CSP). CSP provides more flexibility than other formalisms in the representation of the packets and constraints that must be satisfied. In addition, CSP is declarative, meaning that the user can just state the test case packets (i.e., metavariables) and the constraints relating them. Chapter 2 discusses this in more detail.

Second, the diagnosis is done by checking whether all the constraints are satisfied. If a diagnosis of the problem is found, then it is reported. This is discussed in Chapter 3.

When the system is unable to correctly diagnose the problem, CBR is applied to debug what is missing in the model of the test specification, because the model may be incomplete or incorrect. CBR is also used to remember how previous problems were solved. This is the subject of Chapter 4, where we expand more on the debugging component of ADIOP and the integration of CSP and CBR.

1.4 CSP Modeling for Interoperability Testing

In this dissertation we are interested in modeling interoperability testing using CSPs. We developed a simple modeling language that allows testers to build CSP models of interoperability test cases. This is a declarative language based on CSP and Object-Oriented

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Programming (OOP). We use *metavariables* and *metavalues* for the representation of packets and their assignments for a range of ATM protocols. Each field in a packet is a CSP variable that can be used alone or with other variables to define constraints in CSP models.

CSP models are derived from test cases in a test suite based on a protocol specification. A CSP model of one test case can then be used to test the interoperability of one functionality of two devices by checking the observations (i.e., captured data) against this model (Figure 1.7). In this dissertation, each test case is modeled as a CSP. This guarantees that the CSPs obtained are small and can be solved efficiently. This is also closer to how interoperability testing is done in industry since the companies testing their devices prefer to get a report of specific test cases and failures.

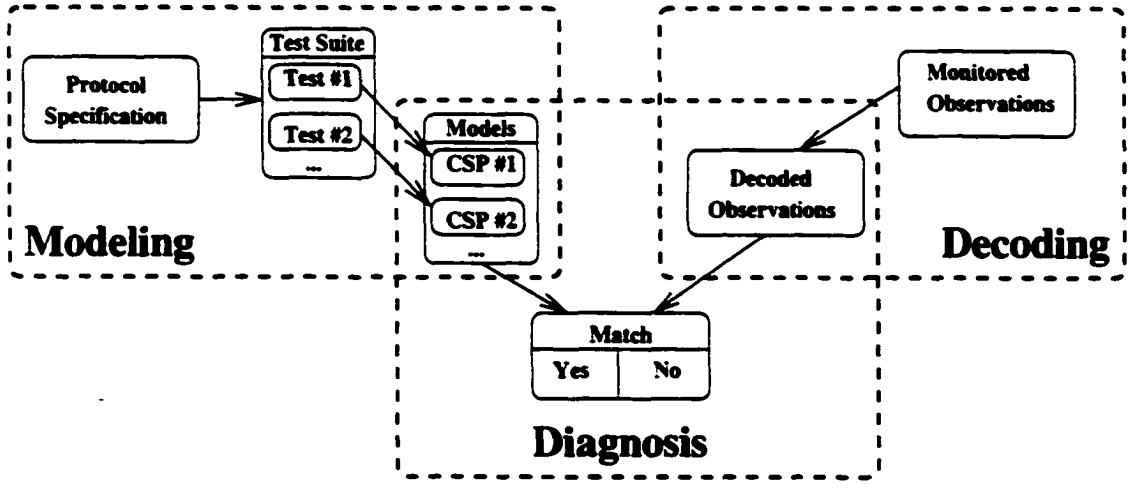

Figure 1.7: CSP Modeling and Diagnosis for Interoperability Testing

We also propose to use the Object-Oriented approach to model these test cases. The choice of this approach for implementation is detailed in Chapter 2. ADIOP is implemented

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

using the Java language that provides the development tools for OOP including GUI-based applications. In this approach, each packet is represented as an object. An object defines a set of variables and implements methods for decoding the packet it represents. For each test case, a CSP model is generated and represented as an object with metavariables and constraints as its parameters and methods respectively.

The modeling interface is a Graphical User Interface (GUI). A user-friendly interface is important for the ADIOP application so the tester can find it easy to use. This also allows us to obtain an evaluation from the testers on this application. The GUI used for modeling allows testers to declare efficiently metavariables, domains, and constraints. The user does not have to know the details of the objects defined in a CSP.

A CSP model is stated in a declarative way. The user needs to define the packets that are expected to be observed for the test case to pass. These packets are represented as objects. An example of a CSP model for test case **V4301H_001** from the PNNI Routing interoperability test suite document (PNNI-IOP 1999) is stated in Figure 1.8 where **1WayIn(A)** and **1WayIn(B)** are the metavariables and **Type**, **Time**, etc. are the variables. The variables presented in this figure are only a subset of all this model's variables. We use here a simple example to be able to show the modeling process without too many details that may prevent the understanding of how this is done in ADIOP. In this example, device A is expected to send a packet of type Hello, namely 1WayInA, to device B. And device B is expected to send a packet of type Hello, namely 1WayInB, to device A. These two devices must be in the same peer group. This is the first step of the PNNI routing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

protocol when the two devices belong to the same peer group. A PNNI network forms a hierarchy where the lowest-level nodes (i.e., devices) are organized into peer groups. A peer group is a collection of nodes, each of which exchanges information with other members of the group, such that all members maintain an identical view of the group. PGID used in Figure 1.8 stands for Peer Group IDentifier. More details can be found in (PNNI-1.0 1996).

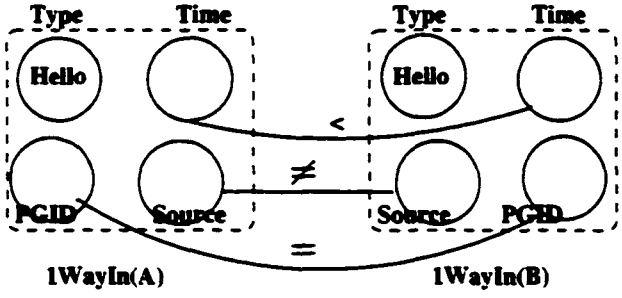

Figure 1.8: A Modeling Example

The following is a CSP representation of this test case using the modeling language defined in this dissertation:

```
$CSP
        $PROTOCOL       PnniRout

        $PACKET         OneWayInA  Hello
        $PACKET         OneWayInB  Hello

        $BINARY_CONSTRAINT      OneWayInA.source != OneWayInB.source
        $BINARY_CONSTRAINT      OneWayInA.time < OneWayInB.time
        $BINARY_CONSTRAINT      OneWayInA.peer_group_id == OneWayInB.peer_group_id

$ENDCSP
```

The constraints may be either unary or binary. The unary constraints are the restrictions

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

on the variable's domain. For example, in the 1WayOut(A) packet, the variable Type can only be assigned the value "Hello". The binary constraints are restrictions on the relation between two variables' domains. For example, there is a $<$ constraint between the Time variable of the 1WayOut(A) packet and the Time variable of the 1WayOut(B) packet.

1.5 Diagnosis of Interoperability Problems

In this dissertation we are interested in how CSP models are used to diagnose interoperability problems. Figure 1.7 shows how diagnosis of interoperability problems interacts with modeling. The use of CSP for modeling allows us to take advantage of methods and algorithms that already exist for solving CSPs. These algorithms are adapted to take advantage of the specialized problem domain structure. This provides a better diagnosis of the interoperability problems including a useful and concise explanation of the testing performed.

The decoding component is responsible for taking the data captured by one analyzer and decoding it to a format that can be used by ADIOP for diagnosis. The outcome of decoding is the decoded observations that represent one input for the diagnosis component. The other input is one CSP model (See Figure 1.7).

The diagnosis component takes the decoded observations from the decoding component and checks if they match the CSP model of the test case being used. In terms of CSP, this means that the decoded observations are metavalues that metavariables can be assigned. The model includes the metavariables that are defined in the test case as well as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the constraints that need to be satisfied.

Different algorithms are being used for this purpose. There are many CSP methods that one can make use of when the problem is represented as a CSP. The problem solving methods in CSP have ranged from pure search (e.g., backtrack) to inference (e.g., arc consistency). The first algorithm we make use of in our application is simple backtracking. This algorithm is adapted to the OO-based CSP presented in this dissertation. Hence, we use metavariables and metavalues instead of variables and values. Section 2.4 describes in more details the concepts of metavariables and metavalues.

We propose to use search supplemented by consistency inference methods in a CSP context to support explanations of the problem solving behavior that are considerably more meaningful than a trace of a search process would be. Constraint satisfaction problems are typically solved using search, augmented by general purpose consistency inference methods.

Our focus in this dissertation is on how to provide testers with a human-like explanation for interoperability testing. When using only search and there is no solution to the test case being executed, the explanation reported for the interoperability problem detected is not very meaningful to the user. We propose to use some specialized inferences, using CSP, that are related to the problem domain structure to generate human-like explanations for the diagnosis of interoperability test cases.

Inference is used mainly to reduce the domains of metavariables. One of these specialized inferences is node consistency at the metavariable level. Node Consistency checks whether constraints involving one variable, (e.g., $V < 3$)) are satisfied. We call this MetaVariable

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Consistency (MVC). The different results that the inference leads to are used as the input to some predefined templates used for different kinds of explanations. The user then gets a useful explanation for the outcome of a test case execution.

After a user runs a test case, a report is generated. Reports are generated for individual test cases and test groups. Both reports can be printed by the user and they provide the information that the customer needs for the interoperability testing of their equipment.

In addition to the reduction of time and effort for solving the problem, the inference process allows for explanations in some cases when no solution is found. The time reduction is even greater when the preprocessing leads to solving the problem since no backtracking is necessary in this case.

An example of the diagnosis and explanation generated for test case **V4301H_007** from the interoperability test suite document (PNNI-IOP 1999) of the PNNI Routing protocol. where an explanation template generated by an inference is used, is shown in Figure 1.9. The purpose of test case V4301H_007 is to verify that after receiving a Hello (1-WayInsideReceived) that the *System Under Test* (SUT) acknowledges the remote identification information. Four packets are required in this test case, but only three were observed in the captured data.

1.6 Debugging CSP Models

A CSP model of a test case can be incomplete or incorrect because:

- The interactions with the external world are unknown,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

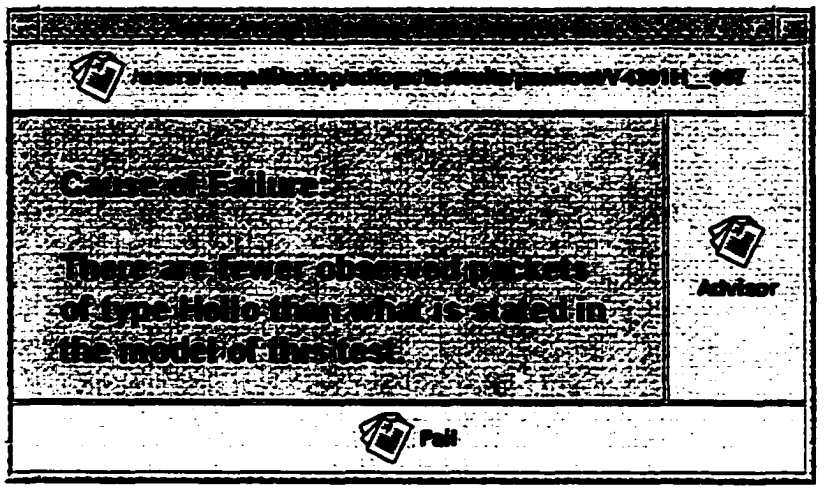

Figure 1.9: Diagnosis Example using an Explanation Template

- The modeling is done by a human being, who may miss or interpret incorrectly some information.

In addition, the protocol and the test suite specifications may be incomplete, inconsistent, ambiguous or incorrect. And if many protocols are running at the same time between two devices, they may cause the wrong behavior of one protocol due to the external interactions with the other. For example, when we enable one ATM protocol on a device, the behavior of another ATM protocol on the same device changes.

We suggest debugging models of interoperability test cases by integrating two modes of reasoning: constraint-based and case-based. The first step is modeling a test case as a CSP. This model may be incomplete or incorrect. We propose to compensate for incompleteness and incorrectness by using the expert's knowledge about this domain, this domain's external interactions, and the flaws it may contain.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We represent interoperability test cases as CSP models supported by a case base to compensate for incompleteness and incorrectness. In Figure 1.10, we show how CBR and CSP are combined to solve these problems. If the results obtained using CSP are inconsistent, CBR is then used to check and debug the CSP model. This model is eventually updated and a new case is stored in the case base.

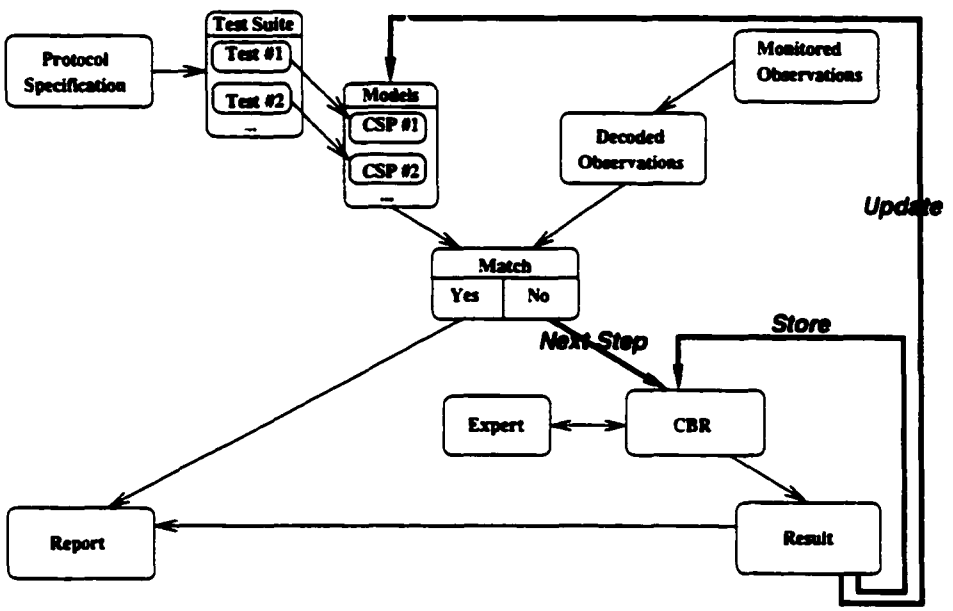

Figure 1.10: Integration of CSP Model and CBR for Interoperability Testing

We are also interested in contributing in terms of the larger CSP domain by acquiring a taxonomy of types of model incompleteness and incorrectness, and associated ways to identify and fix them.

The reliance on past experience that is such an integral part of human problem solving has motivated the use of case-based reasoning (CBR) techniques. A CBR system stores its

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

past problem solving episodes as *cases* which later can be retrieved and reused to help solve a new problem.

The process by which a case-based reasoner operates has been described by (Aamodt & Plaza 1994) as a cyclical process comprised of the *four REs*: *RETRIEVE* the most similar case(s), *REUSE* the case(s) to solve the problem, *REVISE* the proposed solution if necessary, and *RETAIN* the new solution as a new case. The application of this CBR cycle to real problems raises a common set of issues, regardless of the domain of application. These issues include case representation, indexing, storage, retrieval method, and adaptation method. We can abstract the CBR process as one of recalling an old similar problem, and adapting that problem to fit the new situation requirements.

A case is usually composed of a problem description and its solution. Whenever there is a new problem, it is matched to what is already in the case base using similarity metrics such as n-grams for string matching. This will be detailed in Chapter 4. Then the useful cases are retrieved and adapted to the new problem to provide a solution. The new case (problem and its solution) will be stored in the case base if it provides new information.

Some of the cases stored in the case base originate from the incompleteness and incorrectness of the CSP model. In this case, the case stored contains statements for updating the CSP model and making it complete and correct. These cases can then be used in the future to help with similar problems and update other incomplete or incorrect models.

Search and inference methods may fail to generate useful explanations for some interoperability problems. We suggest to store these problems in the case base with an explanation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of the solution provided by experienced testers. Each problem and its solution constitute one case. These cases originate from interoperability problems with a non-useful, incorrect, or incomplete explanation. These cases can then be reused to provide testers with a better, correct, and more complete explanations for future similar interoperability problems.

Different CBR phases are addressed in this dissertation including case retrieval, case adaptation, case revision and case storage.

We use a structural CBR approach for case representation where we decide manually ahead of time what features will be relevant when describing a case, and then we store the cases according to these.

When a new failure occurs, the CBR system (ADIOP's Advisor) constructs a new case and retrieves old cases from the case base that are similar to it. Case retrieval deals with finding ways to match and compare different cases and measure similarity between them, to come up with a solution similar to old ones. This requires the use of algorithms for comparing different features' values and measuring distances between them, defining weights for these features, and methods or formulas for computing the global similarity between old and new cases. We combine both syntactic and semantic similarity measures depending on each feature. For each feature, we provide a distance function. Some features are not used for computing the global similarity and thus have no distance functions associated with them. The distance between two strings is computed using n-grams (Damashek 1995). The weight describes the relative importance of each attribute/feature. We have used different values for the weights. The weights are chosen by an expert using this system and can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be adjusted later as needed. The global similarity is computed using a *Nearest Neighbor Retrieval* equation, which is explained in Chapter 4. If there is a similar case to the new case in the case base, then the user may manually choose this one as the case to be reused and adapted in the new situation.

For adaptation, there are few basic rules that the ADIOP system uses to adapt the case and the user has to confirm this or makes updates to this adaptation. When the user has made all the changes and adapted the new case using a similar case, she/he can revise the adapted case.

If the new revised case is different from old cases in the case base, then the user may choose to retain this case in the case base. If the similarity between all the old cases in the case base and the new one is less than a certain threshold value, then the user should consider adding the new case to the case base. Chapter 4 provides more details on these CBR phases.

ADIOP provides functionality to update the model of a test case that led to a failure caused by incompleteness/incorrectness of this model. The statements on the "Update Model" feature of a case are used for this purpose. These can be either: add, delete, or update statements. A statement can be a constraint, a variable, etc. stated using the same CSP modeling language presented earlier.

Using test case V4302H_005 from the interoperability test suite document (PNNI-IOP 1999) of the PNNI Routing protocol, Figure 1.11 represents the time variables and the constraints between them of the corresponding CSP model. The purpose of test case

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

V4302H_005 is to verify that the SUTs determine that they are in different peer groups. Four packets are required in this test case to check that the two devices are in different peer groups. In this example we want to demonstrate how a CSP model of a test case is updated using the CBR process. We show how we can compensate for an incorrect model.

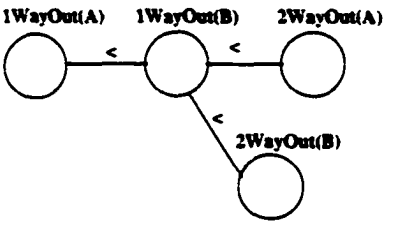

Figure 1.11: Initial CSP Model of the Time Variables of Test Case ID: V4302H_005

For the model in Figure 1.11, the following are some of the results we may observe:

- **Observation X:** Where no packets are observed between the two devices tested. It can be concluded from this observation that the test case fails as we expected to observe four packets.

- **Observation Y:** Where all four packets showing in the CSP model are observed in the captured data. These packets also satisfy all the constraints of this model. Then, it can be concluded from this observation that the test case passes.

- **Observation Z:** Where only three packets of the four showing in the CSP model are observed in the captured data. 1WayOut(B) is not observed. According to this observation, it can be concluded that the test case fails because one packet is missing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The tester wants to check whether the conclusion made about the test case failure in *Observation Z* is correct or whether there is an inconsistency in the model. The tester uses the CBR process in ADIOP to perform this. ADIOP looks in the case base for a previous similar case to check whether the CSP model of this test case is incorrect. The similar case retrieved from the case base and reused for solving this problem is presented in Figure 1.12.

```
Case: 1
   index:              One packet missing
   type:               Incorrect Model
   protocol:           pmsirout
   section:            43028__
   testcase:           V4302H__002
   testpurpose:        Verify that a PNNI version number is agreed upon.-
   testprerequisite:   Both SUTs are SS_B and in different lowest level peer groups.-
   data:               other/PNNI.PRN
   failurecause:       There are fewer observed packets of type Hello than what is stated in the model of this test.
   problem:            The second Hello packet (Hello1B) is missing
   solution:           The second Hello packet (Hello1B) is made optional
   outcome:            Model updated, Warning added, interoperable but not conformant
   modelupdate:        ADD:    $UNARY_CONSTRAINT       Hello1B.status == D_Optional
                       ADD:    $BINARY_CONSTRAINT      Hello1A.time <= Hello2A.time
                       ADD:    $BINARY_CONSTRAINT      Hello1A.time <= Hello2B.time
                       UPD:    Hello1B.peer_group_id   Hello2B.peer_group_id
                       ADD:    $CONSTRAINT     Hello2A.time Hello2B.time D_Mandatory.contains(_Hello1B.status) ||
                               Compare.compare(_Hello2B.time, "<=", _Hello2A.time)
```

Figure 1.12: Retrieved Similar Case Example

CSP/CBR integration is applied to compensate for incompleteness and incorrectness in a CSP model and to debug it. When the example of Figure 1.11 is found to be incorrect, CBR is applied. With the user confirmation, the test case model is updated using the statements from the revised/retained case.

The updated test case CSP model contains the statements of update adapted from Case 1 (Figure 1.12) and shown in Figure 1.13.

```
...
$BINARY_CONSTRAINT  OneWayOutA.peer_group_id != TwoWayOutB.peer_group_id  # Automated Model Update (Statement Update) using Case:
SimCaseNum: 1 #

$UNARY_CONSTRAINT   OneWayOutB.status == D_Optional  # Automated Model Update (Statement Addition) using Case: SimCaseNum: 1 #
$BINARY_CONSTRAINT  OneWayOutA.time <= TwoWayOutA.time  # Automated Model Update (Statement Addition) using Case: SimCaseNum: 1 #
$BINARY_CONSTRAINT  OneWayOutA.time <= TwoWayOutB.time  # Automated Model Update (Statement Addition) using Case: SimCaseNum: 1 #
$CONSTRAINT  TwoWayOutA.time TwoWayOutB.time D_Mandatory.contains(_OneWayOutB.status) || Compare.compare(_TwoWayOutB.time,
"<=", _TwoWayOutA.time) # Automated Model Update (Statement Addition) using Case: SimCaseNum: 1 #
...
```

Figure 1.13: Updated Test Case CSP Model

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Using Case 1, the initial CSP model of Figure 1.11 is updated, and becomes as follows:

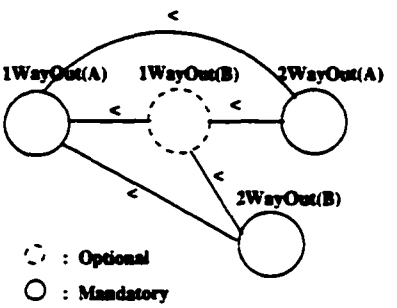

Figure 1.14: Corrected CSP Model of Figure 1.11

This problem happened because of misinterpretation of the specifications that caused an incorrectness in test V4302H_005 of the test suite. When the model was corrected (Figure 1.14). the two observations Y and Z pass this test case.

1.7 Evaluation

The goal of the evaluation is to obtain empirical support for some of the claims made in this dissertation. The evaluation sections of the different chapters present more detailed comparisons of manual versus automated interoperability testing gathered from a questionnaire used by testers (See Appendix B). The testers also provide a survey rating ADIOP on its different aspects. In addition, we present evaluations of the different methods used including the diagnosis algorithms and the CBR system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.8 Contributions

The main contribution of this dissertation is in the applicability of CSP modeling, Constraint-based diagnosis, and CSP model debugging using CBR in the domain of interoperability testing of networking protocols.

1.8.1 Interoperability Testing

- We present a partly automated system, namely ADIOP, to perform interoperability testing. It provides a user friendly interface for testers to create test cases for different types of protocols and to diagnose decoded data captured from different analyzers. This makes it more general than many of the existing tools used in the UNH-IOL. These tools are usually an extension to a specific analyzer and can only work on data captured using it. The addition of more types of protocols and decoders to the ADIOP system is also possible. ADIOP provides the lab with a tool for storing test cases and past experiences. This makes it possible to perform testing and generate reports at any time by testers even if they do not have expertise with the protocol being tested. ADIOP is useful for the UNH-IOL since the experience from testers is kept even after they leave, and can be reused.

- Inconsistencies in the protocol and/or test suite specifications can be detected and debugged through the use of a CSP/CBR integration to update inconsistent models.

- We developed a prototype, namely ADIOP, that uses many types of protocols and analyzers. ADIOP provides a time efficient solution to interoperability testing. It

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

has capabilities to generate human-like explanations of the cause of success or failure using specialized methods and algorithms, and to generate reports for sections of the interoperability test suites as requested by UNH-IOL customers. All the protocols and test cases are dynamically loaded in ADIOP which makes it possible to have more protocols and/or test cases added with minimal changes to the code. We have also performed surveys and evaluations on ADIOP that supports these claims.

1.8.2 Constraint Satisfaction Problems

- We define a new modeling language using CSP and OOP that can be used by testers to implement interoperability test cases. This language is simple, declarative, and transparent. The OO approach provides a natural, concise, scalable and reusable framework for model building. ADIOP provides a GUI for building models with minimal knowledge of the content of protocols and packets being used.

- A major part of the process of model acquisition is automated. Once testers have a high level understanding of the test case description, they can state it in terms of the CSP modeling language. The GUI of ADIOP makes it even easier to state a CSP model since it shows all the different options testers can use including packets to be observed between devices and constraints to be satisfied for interoperability purposes.

- Another contribution in modeling is a novel use of Object-Oriented programming in conjunction with CSP modeling. The notion of *Metavariable* is introduced and allows more flexibility of representation of variables encapsulated in an object. This

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

provides an easier definition of variables within objects that have already been stored in a library of objects. Values are also represented as objects, namely *Metavalues*.

- We introduce the notion of an optional variable and adapted the algorithms used to include the processing of optional variables.

- We improve and adapt the algorithms used in CSP to allow for the diagnosis of interoperability problems and the generation of useful human-like explanations. We use search supplemented by consistency inference methods in an OO-based CSP context to support the generation of explanations of the problem solving behavior that are considerably more meaningful to testers.

- We describe how CBR is used to debug and eventually update CSP models. To our knowledge, previous CBR-CSP integrations do not include this kind of integration. This provides a framework for adding CBR to CSP. This also provides ways of compensating for incompleteness and incorrectness in CSP models. CSP is enhanced by the CBR results. The effectiveness of the model increases as more problems are solved, because the CSP model gets updated as needed.

- We acquire a taxonomy of types of CSP model incompleteness and incorrectness and how to identify and fix one of these types. We describe how CBR is used to update CSP models and debug interoperability test cases.

- The ADIOP system implements and supports many of these claims. From a test case model definition that hides detailed information, a tester can build a CSP model for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

this test case, run it, and generate reports to deliver to UNH-IOL customers. The evaluations included in the different chapters support many of the claims made here.

1.8.3 Case-Based Reasoning

- In this dissertation, CBR supports CSP by providing a module for updating and debugging CSP models. CBR recalls previous cases when a similar problem is encountered. Cases that represent incompleteness and incorrectness in the model are stored in addition to the ways these are solved.

- CBR is also used to store and retrieve cases that are related to interoperability problems where the explanation provided is not complete. Thus, CBR is used to recall similar previous useful explanations.

- The use of CSP provides models for test cases and thus gives a general view of these test cases. If CBR was used exclusively, we would need to gather initial test cases for many situations that then have to be generalized to capture the same information captured in a CSP model. The use of CSP is simpler and models all the information of an interoperability test case in one CSP. CBR captures new interoperability experiences including those for correcting and completing CSP models. This means that CSP is used as first layer of reasoning and CBR as a second one. CBR then takes advantage of the generalities provided by CSP.

- We describe how cases in the case base are defined to include information about updating models. This information is stated using a CSP language similar to the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CSP modeling language used in ADIOP. This provides a uniformity of representation between CBR and CSP to simplify their integration.

- We developed a working prototype as part of ADIOP that uses CBR to generate better explanations and update inconsistent models.

1.9 Dissertation Outline

In Chapter 2, we describe the CSP modeling process in the interoperability testing domain and the use of object-oriented programming. We present a simple modeling language that allows the user to build models of the interoperability test cases. We discuss the use of Object-Oriented Programming (OOP) in conjunction with CSP. Each test case is modeled as a CSP using a many-models architecture and represented as an object.

In Chapter 3, we discuss how we use CSP models to diagnose interoperability problems. CSP algorithms are adapted to take advantage of the specialized problem domain structure. This provides a better diagnosis of the interoperability problems including the generation of human-like explanations of the testing performed.

In Chapter 4, we present a taxonomy of types of incompleteness and incorrectness and how to debug one of them. We discuss the CBR process for debugging and updating models. We describe two types of cases stored in CBR. There are cases that originate from the incompleteness and incorrectness of CSP models. Other cases originate from interoperability problems with a non-useful, incorrect or incomplete explanation. All these cases are reused when future similar situations occur.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5 concludes this dissertation and gives directions for future research.

ADIOP (Automated Diagnosis of InterOperability Problems) is the implementation of a system that includes CSP modeling using OOP, case-based diagnosis and CSP model debugging. A Graphical User Interface (GUI) is used by the ADIOP system and provides a user-friendly interaction with testers. Appendix C includes the User Manual for ADIOP v2.0 including an overview of the different components implemented. In all the main chapters, an evaluation is performed using ADIOP, and its results are presented and analyzed. Appendix B includes the questionnaire used by testers for evaluating ADIOP.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2

CSP Modeling Using Object-Oriented Programming

In this chapter we present a simple modeling language that allows the user to build models of the interoperability test cases. Interoperability testing involves checking the degree of compatibility between two networking devices that implement the same protocol. The Constraint Satisfaction Problem (CSP) paradigm provides a uniform framework based on a declarative language for an accurate representation of the model.

We discuss the use of Object-Oriented Programming (OOP) in conjunction with CSP. The notion of *Metavariable* is introduced and allows increased representational flexibility of variables encapsulated in an object. Values also are represented as objects namely *Metavalues.*

Each test case is modeled as a CSP and represented as an object with metavariables and constraints as its parameters and methods respectively. These objects inherit all the information on how to construct metavariables from a class hierarchy.

ADIOP (Automated Diagnosis of InterOperability Problems) is the implementation of a system that includes CSP modeling using OOP. A modeling interface based on a Graphical User Interface (GUI) is used by the ADIOP system and provides a user-friendly interaction

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with the tester. The diagnosing part of ADIOP is addressed in detail in Chapter 3.

2.1 Modeling and Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) consists of a set of variables, a set of constraints relating these variables and a set of domains of values for the variables. A solution to the CSP is an assignment of domains' values to variables such that all constraints are satisfied.

In our domain of application, CSP is used as a modeling tool and as a problem solving mechanism. One of the main contributions of this dissertation is in modeling interoperability test cases. CSP is useful in modeling because it is declarative and powerful in expressing and describing many application domains. (Wallace 1996) states that "One major contribution of constraints is to problem modeling. It has been claimed that 'constraints are the normal language of discourse for many applications.' Whilst this advantage pays off in all applications, it is central to the design and verification of VLSI circuits and to the specification, development, and verification of control software for electro-mechanical systems."

There are two sides to our modeling work: one is how we model efficiently the problems and second how to make this model a better one by debugging it. In this chapter, we discuss the first part which involves CSP modeling using Object-Oriented Programming (OOP). Model debugging will be discussed in Chapter 4.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2 Modeling Interoperability Testing

A protocol specification is usually written by an organization such as a standardization body (e.g., ISO) and others (e.g., ATM Forum (ATMF)). Most specifications used to implement ATM protocols are taken from the ATMF. From this protocol specification a test suite might be written by one of these organizations. The protocol we are using in this dissertation is PNNI (Private Network-Network Interface), and ATMF provides both the protocol specification and the interoperability test suite documents.

The interoperability test suite is a set of test cases organized into sections. Each section allows for the testing of a part of the protocol. Each section contains a set of interoperability test cases. Each test case tests for a specific issue in the protocol. Each test case is described in detail as to what configuration should be used, what are the steps to follow in testing and what is the verdict criteria to use in deciding whether this test case passes or fails. Creating manually a test suite is a major first step before modeling and automating test cases. In this dissertation, we use a test suite that has been specified and approved by ATMF. A detailed description of a test case layout taken from (PNNI-IOP 1999) can be found in Appendix A.

Each test case's result provides very specific and limited information about the devices being tested. When all the test cases are combined, the result is a detailed interoperability testing of each aspect of the protocol.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2.1 One Model Architecture

In interoperability testing, we want to test whether two devices when connected behave correctly according to the statements in the protocol specification. One way of doing this is by modeling the entire protocol specification as one CSP (Sqalli & Freuder 1996a) (Riese 1993a) (Riese 1993b).

Definition 2.1 (Observations) : *Observations can mean either monitored observations or decoded observations. They represent the same data in different format. Monitored observations are the packets/frames that are captured between two devices using an analyzer. The data flow between two devices is captured by analyzers as binary, converted into Hexadecimal format and then decoded to text according to the protocol specification (e.g., (PNNI-1.0 1996)). Decoded observations are data in text format that is used by testers for checking the interoperability of devices.*

This CSP model can then be used to test the interoperability of two devices by checking the observations against the CSP model (Figure 2.1). One assumption made here is that the observations captured by the different analyzers are correct.

There are advantages and disadvantages to this modeling approach. The advantages are that:

- There is only one model to use
- The model is taken from the protocol specification directly, so there should be fewer inconsistencies in this model than if the model were built from a test suite that is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

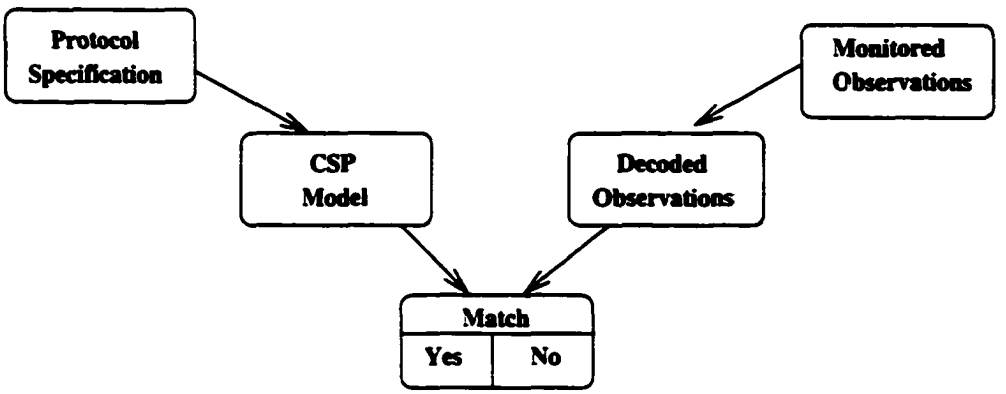

Figure 2.1: One Model Architecture

itself derived from the protocol specification

- There is no duplication of testing as occurs in test suites where two different test cases may have a common testing part

The disadvantages of this approach are:

- The model is too complex to state and to use since it must represent the behavior of all the steps in a protocol
- If there is a problem in the observations, it is hard to pinpoint the cause of the failure
- It is more convenient/preferred by vendors and testers alike to use different test cases for different parts of the same protocol than to have just one large test case
- It is difficult to create interoperability testing reports and to explain what happened in testing
- It is difficult to update the model in case of an error in its statement

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2.2 Many Models Architecture

In this design, the CSP models are derived from the test cases in the test suite written from the protocol specification. We use a test suite that has been specified and approved by ATMF. A test case CSP model can then be used to test the interoperability of one functionality of two devices by checking the observations against this model (Figure 2.2). In this dissertation, we use this form of modeling, where each test case is represented as a CSP.

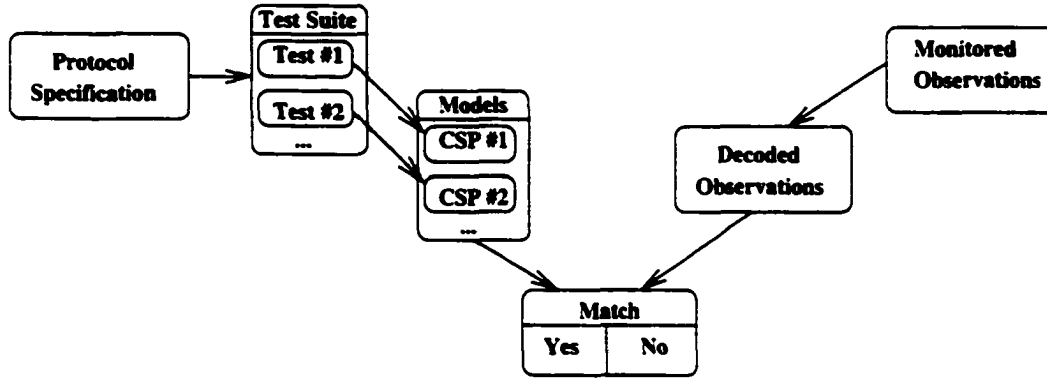

Figure 2.2: Many Models Architecture

The observations represent a set of packets captured. Each packet has many fields as defined in the corresponding protocol specification. The data contained in these fields represent the values that are assigned to the corresponding variables in the model (See example in Figure 2.3). The constraints defined in the CSP model are checked for consistency. If all the constraints are satisfied for an assignment, then the interoperability test case passes. This is then repeated for each test case in the test suite.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

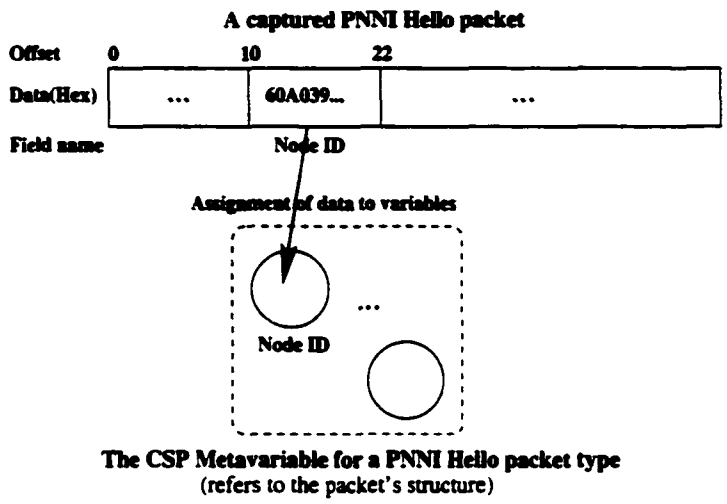

Figure 2.3: CSP Variable Assignment

The advantages of this form of modeling are that:

- It is easy to create models for specific test cases
- Models are easy to work with (i.e., use, debug, etc) because they are small
- It is easier to generate reports for interoperability testing
- This is closer to how interoperability testing is done
- It is easier to give explanations using small models

There are disadvantages to this form of modeling:

- We need to write as many models as there are test cases. This is alleviated in our system ADIOP by providing a tool that makes it easy to create models

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- More inconsistencies might be added to the model, since there are errors that might originate from the protocol specification or from the interoperability testing document. In ADIOP, the debugger, presented in Chapter 4, addresses inconsistencies independently of their origin

- Some parts of testing might be included in more than one test case causing redundant testing. One example of this is that the initial part of a protocol might be included in more than one test case. This is not a major concern since we can copy parts of one model into another one.

2.3 Object-Oriented Programming

Object-Oriented Programming (OOP) has become a very widely used paradigm in software development. Its success can be attributed to its natural way of modeling real-world objects. Many languages are OO such as C++ and Java. Java has combined the benefits of many of its predecessor programming languages. Java also conveniently provides the development tools for GUI-based and web-based software. Our system ADIOP is implemented using Java. In this section, we define some of the OO terms we use in this dissertation.

(Booch 1994) states that: "An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms *instance* and *object* are interchangeable". More programming-oriented definitions of objects and classes are stated in (Campione & Walrath 1998) as the following:

Definition 2.2 *An object is a software bundle of variables and related methods.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition 2.3 *A class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind.*

In the OO terminology, a particular object is called an instance of a class. In the same way we use the term instance variables and instance methods. A class is a set of objects that share a common structure and behavior.

The difference between classes and objects is often the source of confusion. In the real world, it's obvious that classes are not themselves objects they describe: A blueprint of a bicycle is not a bicycle. However, it's a little more difficult to differentiate classes and objects in software. This is partially because software objects are merely electronic models of real-world objects or abstract concepts in the first place. But it's also because the term "object" is sometimes used to refer to both classes and instances (Campione & Walrath 1998).

In this dissertation, we refer to *class* as the implementation of a class of objects, and to *object* as one instance of this class. For example, when we refer to the **Hello** class, we mean the implemented **Hello** class, and when we refer to a **Hello** object, we mean a particular object defined to be from the **Hello** class, which may have a name such as **OneWayInA**. We also use the name **"parameter"** to refer to an object's variable so that there is no confusion between CSP variables and object's variables.

There are many properties in OOP that make modeling more efficient. Two of which we are interested in here are: Encapsulation and Inheritance. (Coad & Yourdon 1991) defines: "Encapsulation (Information Hiding). A principle, used when developing an overall

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

program structure, that each component of a program should encapsulate or hide a single design decision... The interface to each module is defined in such a way as to reveal as little as possible about its inner workings. [Oxford, 1986]"

Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two primary benefits to software developers: modularity and information hiding (Campione & Walrath 1998). Modularity means that objects can be created and maintained independently of other objects. This makes it easy to use the same object by different components of the system. Information hiding means that an object can have private information that other objects cannot access but they can still use its functionality.

Inheritance is the ability to define classes in terms of other classes. A subclass inherits variables and methods from a superclass. Subclasses can add variables and methods of their own to the ones they inherit, and they can override inherited methods. This is called specialization. Superclasses can be of abstract nature. An abstract class defines the behavior that subclasses can inherit. Inheritance can be of many levels to constitute a class hierarchy.

2.4 Description of the CSP Modeling Process

In terms of modeling, we propose to model each test case from the test suite as a CSP. This guarantees that the CSPs obtained are small and can be solved efficiently. This is also closer to how interoperability testing is done in the real world since the companies testing their devices prefer to get a report of specific tests and failures. The breakdown of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

interoperability testing into small test cases provided by a test suite, written manually by experts, allows us to do incremental testing and to easily detect problems at each level of this testing.

We also propose to use the Object-Oriented methodology to model these test cases. In interoperability testing, an analyzer is usually used to collect data between the two devices being tested. The data collected is then decoded as packets. Hence, it is natural to represent the CSP in term of packets. Each packet contains many fields that should be checked against other packets' fields to test for interoperability. Since the constraints exist between the packets' fields, we represent each field as a variable in the CSP. The constraints represent restrictions on these variables.

However, it is tedious work to state each one of these variables separately because a packet may contain a large number of fields and a tester may not remember all of these for each type of packet. The idea is then to represent a packet definition as a metavariable in the CSP representation and each observed packet as a metavalue. A metavariable or a metavalue is respectively an object or instantiation of an object representing a packet.

For each packet type, a class of objects is defined. Each packet is an object of one of these classes that corresponds to its type. Each class of objects includes parameters, some of which are the packets' fields, and methods needed by these objects to manipulate the packets' data.

Definition 2.4 (Metavariable) : *A metavariable in the CSP model refers to the representation of a packet that encompasses many variables. Some of these variables are the*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

packet' fields describing the content of the packet. Four other variables are taken from the captured data and added to the metavariable structure are: ***time, source, protocol,*** *and* ***status****. A variable of this metavariable can be itself a metavariable encompassing many other variables. This can be expanded down hierarchically.*

Definition 2.5 (Metavalue) : *A metavalue in the CSP model refers to the data captured in a packet. This data is used to instantiate a metavariable.*

Definition 2.6 (Metaconstraint) : *A metaconstraint is a set of constraints relating variables belonging to one or more metavariables. The concept of metaconstraint is an abstract one for representation and design purposes. Constraints are defined using variables as their arguments.*

In this dissertation, we use only unary and binary constraints. A unary metaconstraint is a set of unary constraints belonging to the same metavariable. A binary metaconstraint is a set of binary constraints relating variables belonging to two metavariables. The concept of metaconstraint is an abstract one for representation and design purposes only.

There has been some work combining OO and Constraint Satisfaction (Roy & Pachet 1997) (Paltrinieri 1994a) (Paltrinieri 1994b) (Stone 1995). To our knowledge, no one has used this integration in the same way we present it in this dissertation. The closest work to ours is what has been done in (Paltrinieri 1994a) (Paltrinieri 1994b). More details on this can be found in the related work section of this chapter.

Another advantage of this CSP representation, besides its declarative nature, is that one can state an object in the model without having to know all the fields of that object. This

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

allows for a very concise CSP model statement. From this CSP model statement, the ADIOP system generates an object corresponding to this CSP model with CSP metavariables as its parameters and constraints as its methods. We use here the name "model" rather than "object" to distinguish between the different types of objects used in ADIOP. This model is then integrated into the system and used for testing.

The CSP model is stated in a declarative way. The user needs to specify the packets that are expected to be observed for the test case to pass. These packets are represented as objects (i.e., metavariables). An example of a CSP model is shown in (Figure 2.4), where **1WayIn(A)** and **1WayIn(B)** are the metavariables and **Type**, **Time**, etc are the variables. The variables presented in this figure are only a subset of all the variables.

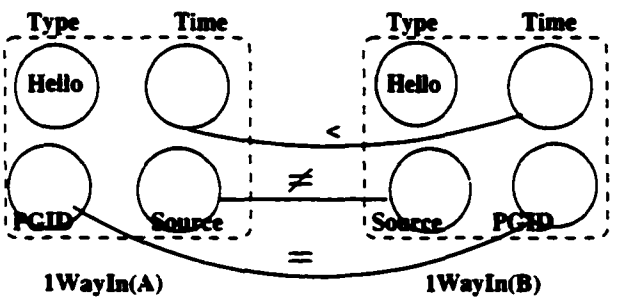

Figure 2.4: A Modeling Example

The following steps show how the CSP modeling of interoperability testing is performed:

1. Identify uniquely each packet using the packet's type. In the case where we have more than one packet with the same type, other parameters (e.g., Source (**A** or **B**)) can be used to identify each of them. Each packet is represented as a metavariable (i.e., an object or a set of variables).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. Represent the packets into a constraint graph where the variables are fields (e.g., Node_ID, Status, Time, ...) of the packets (e.g., 1WayIn, ...). The variable labels are the values that may be assigned to these variables (e.g., **A** or **B** for variable Source), and the edges are the constraints we need to satisfy such as the order of captured packets, or the value a field must have.

3. Get the input data by monitoring the traffic between the devices tested (e.g., **A** and **B**). These are called Observations. These are stored as metavalues using the same structure as for the metavariables. (See example in Figure 2.3).

4. Use the packet identifier (e.g., 1WayIn(A)) to map the packets' fields into variables, and assign values to them (i.e.. assign metavalues to metavariables).

5. Test if all the constraints are satisfied after instantiating all the variables.

6. Report the results (Pass/Fail).

The following is an example of the modeling language:

- $PROTOCOL PnniRout: states that this CSP model implements a test case of the PNNI Routing protocol.

- $PACKET OneWayInA Hello: This states that the model contains a packet (metavariable) of type **Hello** named **OneWayInA**. The Hello class is created and stored in ADIOP as part of the decoder. This is done by a tester at the UNH-IOL. Java is the language used to create the decoder and the classes for different types of packets.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Each metavariable in the model is a representation of one packet with all its fields. When a metavalue (i.e., actual packet observed between devices) is assigned to this metavariable, all the fields of this packet are assigned the respective values from the observed packet. Since the **Hello** class is already stored and contains all the information about this type of packet, this statement creates all the necessary parameters (variables) for the metavariable **OneWayInA**, including the *Time*, *Source*, *Status*, and *Type* variables.

- The domains are declared in a similar fashion: $DOMAIN D_Source DTE DCE. This declares two sources of where the data can be sent from. These represent the two devices being tested.

- Unary constraints state the name of the variable and the domain of values or one value restricting this variable: $UNARY_CONSTRAINT OneWayInA.source == D_Source

- Binary constraints are declared as relations between two variables: $BINARY_CONSTRAINT OneWayInA.time < OneWayInB.time

- General constraints allow for a larger scope of constraint declaration. They can be either unary or binary: $CONSTRAINT OneWayInA.time OneWayInB.time f(OneWayInA.time,OneWayInB.time) where f(x,y) is a Java statement that returns a boolean and has x and y as its parameters.

More details of the modeling language are provided in a later section of this chapter.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

After defining packets using the $PACKET statement, there is no need to state each variable (packet's field) separately. When a packet is defined, the ADIOP application provides a list generated dynamically from the packet's fields, listing all the variables belonging to this packet (Figure 2.5). This list can be used for stating constraints between these different variables.

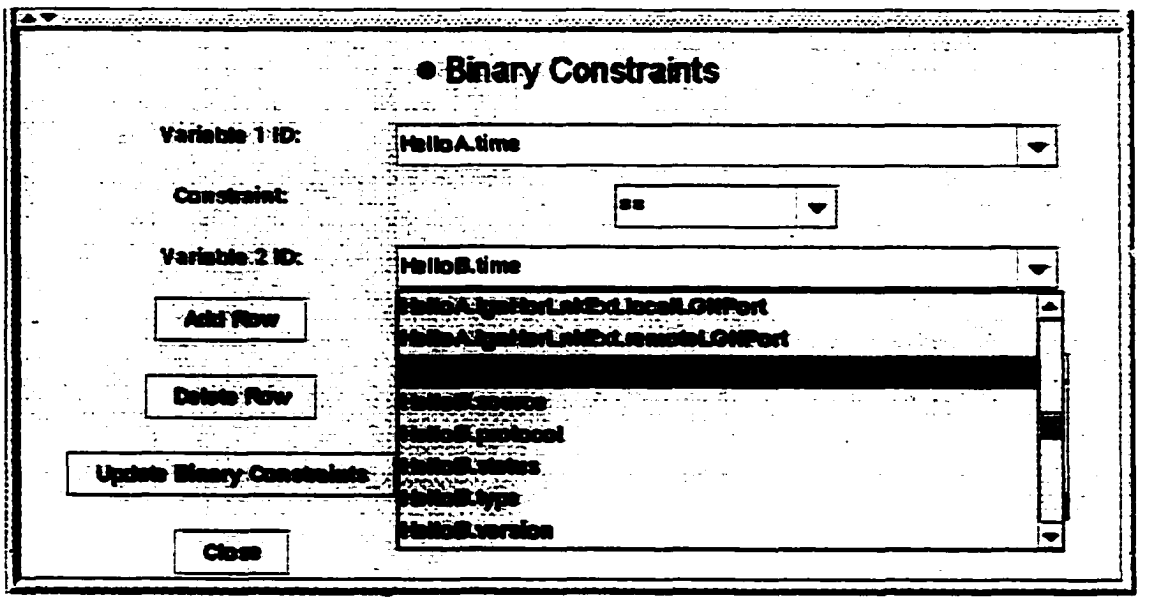

Figure 2.5: Packet's Parameters List

2.5 Modeling with Objects

2.5.1 Modeling of Packets

Interoperability testing of equipment uses packets captured for a specific protocol to determine if a test case passes or fails. These packets contain a number of fields. The values of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

these fields are checked against some constants or against fields' values from other packets to determine if the test case passes. It is natural to represent this problem using the OO approach, where a packet is represented as an object.

Because fields are used to state constraints, it is natural to represent these as variables. This way, an object defines a set of variables. The object also implements methods for decoding the packet it represents.

The use of OO gives us many advantages:

- Each object is a separate entity with its own functionality
- Information hiding of the objects definition as the users do not need to know the details but only the functionality of these objects.
- Inheritance between object allows for a hierarchical definition of packets that matches the way protocols are specified
- The CSP model obtained using objects is concise and expressive

The implementation of these objects as classes uses packages. *adiopx* is the main package in the ADIOP application, and thus it is the name of the root of the whole ADIOP directory structure. Under this directory there is one subdirectory called *packet* that includes all the classes needed for representing packets. One of these is the class **Packet**, which is the parent of all other classes under the **packet**'s directory. This class implements the common parameters and methods for all types of packets. Figure 2.6 shows a representation of the directory structure of the *packet* package. This is created, using the Java language, by a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tester at the UNH-IOL; and this hierarchy uses the structure of protocols as defined in their respective specification documents.

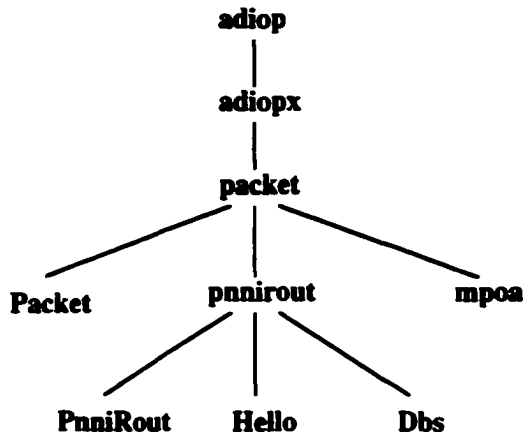

Figure 2.6: Directory Structure of the *packet* Package

One advantage of this representation is that each class is used for decoding and CSP modeling, which saves us resources and provides a clean implementation (no redundancy of functionality).

2.5.2 Class Hierarchy and Inheritance

The classes are stored under the packages as described earlier. These classes are defined in a hierarchical manner to allow for more flexibility of extension and scalability of protocols and packet types being used by the application. The class **Packet** defines the common parameters and methods of all types of ATM packets. As shown in Figure (Figure 2.7), the class **Packet** is the parent of all the classes included in the package *packet.*

In the next level of hierarchy, classes represent a particular protocol type, e.g., PnniRout

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

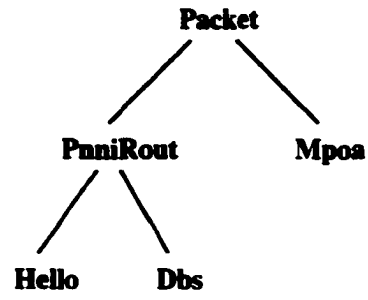

Figure 2.7: Class Hierarchy of the **Packet** Class

which stands for PNNI Routing protocol. The class **PnniRout** defines the common parameters and methods of packets of type PNNI Routing. The class **PnniRout** is a subclass of the class **Packet**. This level inherits from the class **Packet** parameters and methods used by ADIOP.

The classes that are children of this protocol type class are the leaves of the class hierarchy and represent the packet types within this protocol (e.g., Hello, Dbs). They inherit parameters and methods that are common to all these protocol packets from their parent **PnniRout**. Each one of these classes implements specific parameters and methods for its own type.

The parameters can be of a more complex definition if they are themselves classes. Examples of such parameters are *OneWayInA.aggregToken.length* and *OneWayInA.aggregToken.status*. This is an example of a metavariable (aggregToken) within another metavariable (OneWayInA).

This hierarchy makes it easy to add/remove classes. We can add more protocols and more packet types within protocols. We only need to add the decoder for each one of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

these packet types to have them available for use by the decoder and the CSP modeling component. Testers can do this using the Java language. Actually, parts of the hierarchy that are already in ADIOP have been implemented by testers at the UNH-IOL.

When all the hierarchy of packets is defined, including parameters and methods, the user can declare an expected observation in a test case model to be one of these types and does not need to know or specify all the details of these packets. These details are defined as part of the decoder written for this type of packets.

2.5.3 Decoder

The Decoder uses the same hierarchy of classes defined in the previous subsection. Adding the decoding functionality of a new packet type to ADIOP is a matter of adding one class to the hierarchy.

This can be made even simpler if a generator of these objects is implemented, because all these objects have the same general functionality. The main difference between these objects is in the parameters being used and the way the packets are decoded from Hexadecimal format to text format. The Hexadecimal format is provided by the different analyzers used at the UNH-IOL.

This decoder is used with the monitored observations between two devices to generate the decoded observations, which is a set of packets. Each packet is an instantiation of one of the classes in the bottom of hierarchy (leaves). The same classes are used to state the CSP models. A packet is defined in the CSP model by its type, which is a leaf in the class hierarchy.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Each decoding class contains parameters that represent the specific fields of one type of packets. It also inherits fields from parent decoding classes. This class also contains methods that perform different decoding functions. The advantage of this representation is that the classes used for decoding are also used for modeling, and it provides a concise representation of CSP via objects.

2.6 Modeling Interface

The modeling interface is a Graphical User Interface (GUI). A user-friendly interface is important for the ADIOP application so the tester can find it easy to use. This allows us as well to obtain an evaluation from the tester on this application.

The Test Suite Builder (TSB) component of ADIOP provides the functionality for modeling a test case as a CSP. The GUI used for modeling allows the user to declare metavariables, domains, and constraints in a very efficient manner.

From the main menu of the TSB window, the user can choose which protocol they want to use. The list of protocols as shown in Figure 2.8 is constructed from the structure of ADIOP directories implemented by testers as part of the decoder. If the decoder for a new protocol is added by testers to this directory, this protocol type will be dynamically loaded and shown in this menu.

Each test case object is built as a file with the *.iop* extension. This file may contain a description of the test case taken usually from the interoperability specification document. This file's main section is the CSP model defining the variables and constraints for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

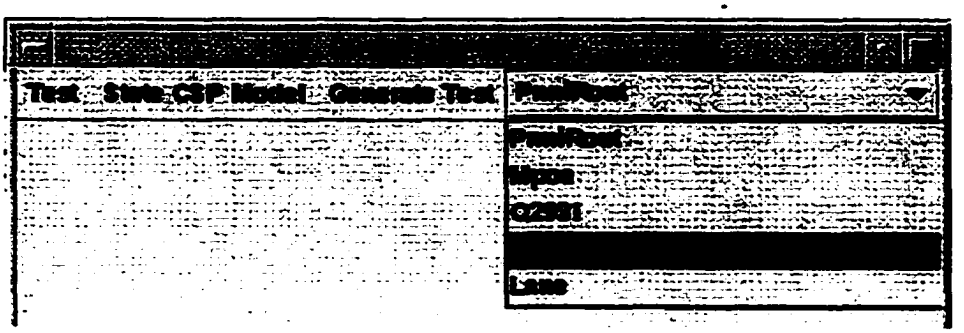

Figure 2.8: Protocols List in the Test Suite Builder Window

this test case. The CSP model is defined between two ADIOP keywords (i.e., **$CSP** and **$ENDCSP**).

2.6.1 Variables

Variables are not declared individually, but rather when a packet is declared using the **$PACKET** statement, a metavariable representing this packet is created and with it all the corresponding variables. Hence, the declaration of a metavariable is sufficient for defining all the variables within. ADIOP provides a functionality to automatically update the *.iop* file with the variable declaration using the appropriate format when the user presses the corresponding button in the GUI.

The packet types shown in Figure 2.9 are also dynamically loaded from the protocol directory structure. For example, if we choose **PnniRout** as the protocol to be used, the packet types list will show: **Dbs**, **Hello**, etc. But, if we choose **Mpoa** instead to be the protocol, then the packet types list will show: **Cache_Imp_Req**, **Cache_Imp_Rpl**, etc.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

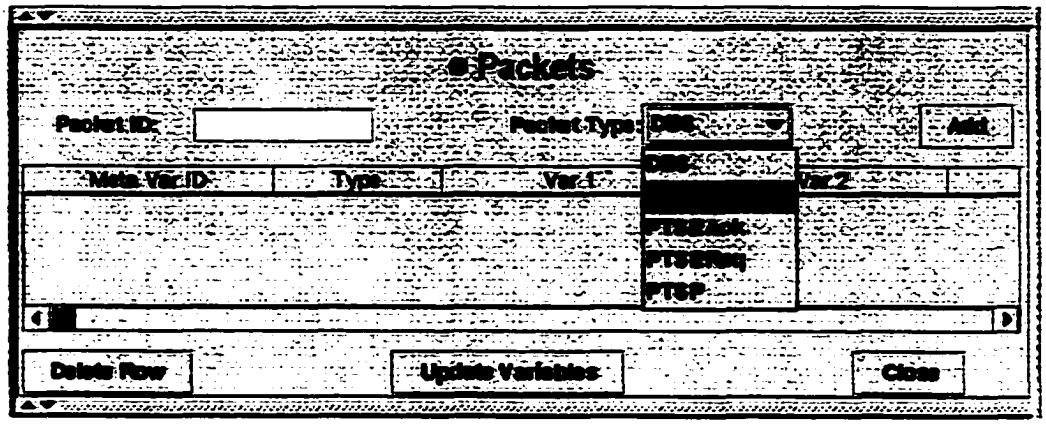

Figure 2.9: Packet Types List in the Test Suite Builder Window

2.6.2 Domains

The domains can be declared as a set of discrete values. These are used to declare unary constraints.

2.6.3 Constraints

A window is provided to add constraints by choosing from existing lists of variables and constraint operations. Constraints can be declared as unary or binary. ADIOP provides a list with all the variables that can be used for this purpose (Figure 2.5). These variables are dynamically loaded using the structure of the metavariable (packet) they are part of. ADIOP also provides a flexible way to declare general constraints. These are unary or binary constraints that can be of a more complex definition than what is provided in the GUI through the list of available constraint operations. The constraint in this case can be any Java function using one or two variables as its arguments. The constraints can be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

added to the CSP model definition using the **update** function.

In addition, this GUI is used to decode packets from Hexadecimal format generated by different analyzers to readable text format. It also provides the tools for running interoperability test cases that have been implemented and generating reports of testing results. We will expand on this in Chapter 3.

2.7 Test Cases as Objects

When the model definition is completed and the *.iop* file is stored, the user can generate an object from this file. The parameters of this object are the CSP metavariables and the methods are the constraints. This object represents the CSP model of the test case declared, and it will be dynamically added to the Decoder/Diagnoser window menu. By choosing this item from the menu, the user is able to execute this test case on any decoded observations shown on the main Decoder/Diagnoser window. More details on this are presented in Chapter 3.

The set of objects representing test cases are stored under the *testsuite* directory under the appropriate protocol name using a test suite hierarchy (See Figure 2.10).

ADIOP constructs a menu in the Decoder/Diagnoser window from the structure of the directories under the *testsuite* directory. If a new protocol is added or more test cases are generated, the menu will get updated. Figure 2.11 shows the menu generated in the Decoder/Diagnoser window.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

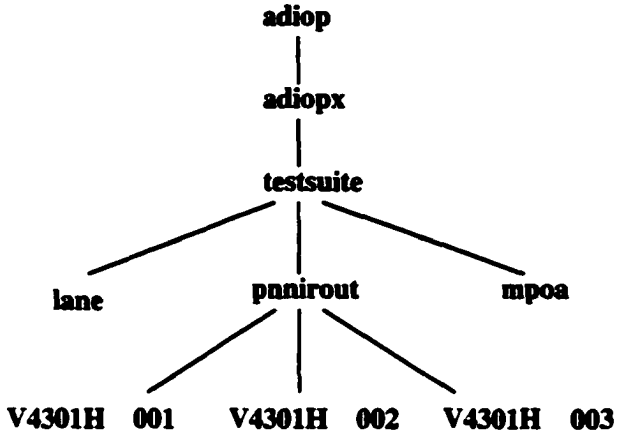

Figure 2.10: The *testsuite* Directory Hierarchy

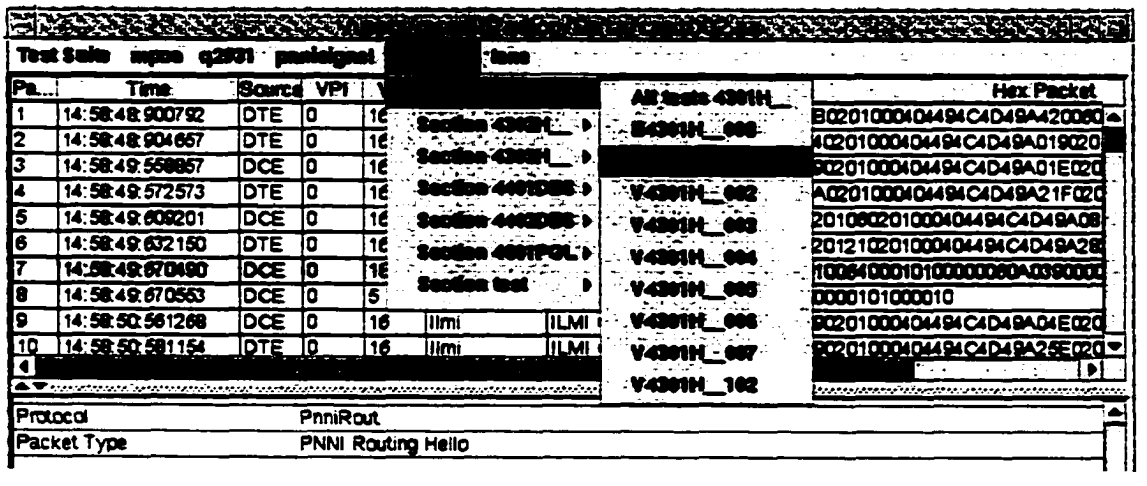

Figure 2.11: Test Suite Menu

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.8 Modeling Language

The model is stated in a very simple language. The following syntax including keywords and their meaning is used:

- **$CSP**: This states that the CSP model declaration starts at this point. This statement is added automatically when a test case is created.
- **$ENDCSP**: An optional statement that means that the CSP model declaration ends at this point. If not used, the EOF is used to detect the end of the model declaration. This statement is added automatically when a test case is created.
- **$PROTOCOL protocolTested**: states that this CSP model implements a test case of the 'protocolTested' protocol. This statement is added automatically when a test case is created.
- **$PACKET packet_name packet_type**: This statement states that this test case being modeled contains a packet of type **packet_type** which was given the name **packet_name**. The **packet_type** has to be a leaf of the class hierarchy. (e.g., *Hello*, *DBS*). This statement generates an object of type packet_type and name packet_name. From the way objects are implemented, there is no need to know details about packet types when they are being used in the CSP model. The declaration of one packet in the CSP model using **$PACKET packet_name packet_type** implicitly defines all the parameters and methods that belong to this packet including its fields (CSP variables) that can be used for stating constraints.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **$DOMAIN domain_name value_1 value_2 ... value_n**: This states that a domain is declared with name **domain_name** and contains values: value_1, ... value_n. All these values are declared as strings.

- **$UNARY_CONSTRAINT variable_name operation domain_name #print_statement#**: This states that the value that can be assigned to **variable_name** must satisfy the **operation** constraint on the domain_name. For example if the operation is ==, then the value assigned to this variable must be in the domain_name set. variable_name must be one of the parameters in one of the objects declared by **$PACKET**. domain_name must have been declared in **$DOMAIN** or in one of the predefined domains in ADIOP. The predefined domains are domains that are always included in all the models and cannot be modified (e.g., *D_Optional* and *D_Mandatory* to state that the existence of a packet in the captured data is optional or mandatory). Alternatively, the user can use a single value instead of a domain_name. **operation** can be one of the following operations: ==, ! =, <=, >=, < or > if a single value is used, and only == or ! = if a domain_name is used. **print_statement** is a statement which will be printed as part of the diagnosis report if this constraint is violated when this test case is used.

- **$BINARY_CONSTRAINT variable1_name operation variable2_name #print_statement#**: variable1_name and variable2_name must be different and both have to be parameters in one or two of the objects declared with $PACKET. **operation** can be one of the following operations: ==, ! =, <=, >=, <, *or* >.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **$CONSTRAINT variable1_name variable2_name f(variable1_name,variable2_name) #print_statement#**: f(v1,v2) is a Java statement that returns a boolean and it is a function with two arguments: v1 and v2, where v1 may be the same as v2. This means that this can be a unary or binary constraint. The idea behind this kind of declaration is to allow for broader constraint statements. The function used here can be made reusable by storing it under the *"util"* directory of ADIOP. This also allows for the use of more complex functions.

- Comments can be included using the "// comments"

2.9 Example of CSP Modeling for One Test Case

The following is an example of a test case (Test Case ID: V4301H_001) from the PNNI (Private Network-Network Interface) Interoperability Test Suite document (PNNI-IOP 1999):

```
Test Case ID:   V4301H__001
Update Version: 0
Test Description:
        Test Case ID:   V4301H__001
        Test Purpose:   Verify that the Hello Protocol is running
                        on an operational physical link.
        Reference: 5.6
        Pre-requisite: Both SUTs are SS_M and in the same lowest
                       level peer group.
        Test Configuration: #1
        Test Set-up:
                1. Connect the two SUTs with one physical link.
        Test Procedure:
                1. Monitor the PNNI (VPI/VCI=0/18) between SUT A
                and SUT B.
        Verdict Criteria: Hello packets shall be observed in both
                directions on the PNNI.
        Consequence of Failure: The PNNI protocol cannot operate.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The following is a CSP representation of this test case using the language presented in the previous section and created using the TSB window from the GUI presented earlier:

```
$CSP
        $PROTOCOL       PnniRout

        $PACKET         HelloA  Hello
        $PACKET         HelloB  Hello

        $BINARY_CONSTRAINT      HelloA.source != HelloB.source
        $BINARY_CONSTRAINT      HelloA.time <= HelloB.time
        $BINARY_CONSTRAINT      HelloA.peer_group_id == HelloB.peer_group_id

$ENDCSP
```

Figure 2.12 shows the actual TSB window including the CSP modeling example.

ADIOP generates an object representing this test case with HelloA and HelloB metavariables as its parameters and the three binary constraints as its methods. A menu item with the name of this test case is added to the Decoder/Diagnoser window. This menu item is used to execute this test case by calling its corresponding object. Chapter 3 explains how the captured data between two devices is used with a CSP model to perform interoperability testing.

2.10 Application of CSP modeling

The CSP models are used to diagnose and solve interoperability problems (Figure 2.2). All the test cases implemented using the ADIOP's modeling component are accessible through the menu in the Decoder/Diagnoser window of ADIOP (Figure 2.11).

The diagnosis component takes the decoded observations from the decoding component and checks if they match the CSP model of the test case being used. In terms of CSP, this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

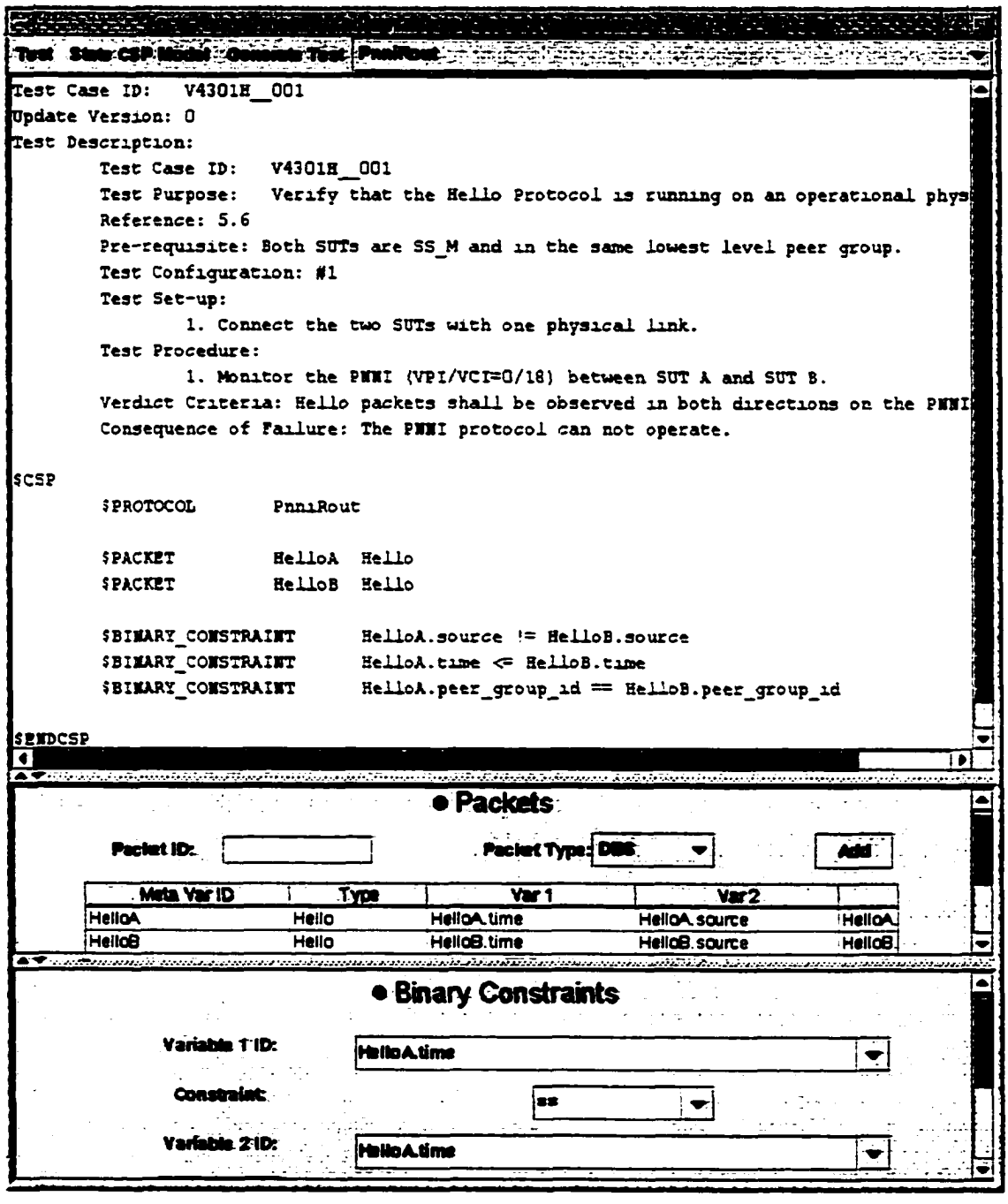

Figure 2.12: The Test Suite Builder Window

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

means that the decoded observations are metavalues that metavariables can be assigned. The model provides the metavariables that are defined in the test case as well as the constraints that need to be satisfied.

Our motivation for automating the diagnosis of interoperability testing is to save time, reduce repetitive testing, store and reuse knowledge, automate reports generation, and in general to make testing easier and more efficient. Our focus is on how to get a "good" explanation to the problem we are solving. This is detailed in Chapter 3.

The advantage of CSP is that it is a reasoning mode that provides both modeling and problem solving within the same framework. Chapter 3 discusses the problem solving part of CSP. The use of CSP for modeling allows us to take advantage of methods and algorithms that already exist for solving CSPs including search and inference. These algorithms are adapted to take advantage of the specialized problem domain structure. This provides a better diagnosis of the interoperability problems including an accurate and concise human-like explanation of the testing performed.

ADIOP uses search supplemented by consistency inference methods in a CSP context to support explanations of the problem solving behavior that are considerably more meaningful than a trace of a search process would be. Constraint satisfaction problems are typically solved using search, augmented by general purpose consistency inference methods.

More details about the Diagnoser component of ADIOP, including its evaluation, are presented in Chapter 3.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.11 Evaluation

The purpose of this evaluation is to support our claims about the performance of ADIOP, and to show that it is doing what it was intended for. This evaluation includes 3 ADIOP components: Decoder, Diagnoser, and Test Suite Builder. The Test Suite Builder is the CSP modeling component and the Diagnoser is the CSP solver component. Appendix B contains the questionnaire that the testers used for the evaluation of these three components.

We are interested in the evaluation of different aspects of these components. Some of the common aspects are that a component does what it is intended to do, is user-friendly, flexible, reusable, useful, and fast. And for each component, there are specific aspects we are interested in, such as accuracy of test case execution results, clarity of explanation, report generation of results, and execution time for the Diagnoser. As for the Test Suite Builder, we are more interested in the modeling language ease-of-use and the correctness of the models generated. In addition, we collect from testers a general survey on the overall performance of ADIOP, how it enhances the way interoperability testing is done, and how it can be improved.

For each component, there are two types of evaluation. One is the evaluation of the component behavior, whether it matches the intended description, and how it compares to other methods including manual testing. The second evaluation type is a survey of each component capabilities and performance.

The evaluation is performed using different data sets and test cases. The testers were given the same data sets and test cases to work with. These were chosen in a way to provide

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

an evaluation of different situations.

The outcome of this evaluation allows us to confirm our claims about ADIOP, its success and its contribution. This includes that ADIOP's Diagnoser outperforms manual diagnosis and other tools, ADIOP's Test Suite Builder outperforms other tools, the CSP modeling language capabilities and ease-of-use, and the ability to generate correct and useful explanations.

In this section, we present the results obtained for the Modeling component of ADIOP. The evaluation results of the other two components are presented in Chapter 3.

2.11.1 Evaluation Setup

Three testers performed the evaluation of ADIOP. According to the survey collected from them, their knowledge of the protocol used in this evaluation ranges from moderate (2) to high (1). Their knowledge of interoperability testing ranges from moderate (1) to high (2). They all had a moderate knowledge of the interoperability test cases of the protocol they used in this evaluation. None of the testers rated low on any of the above questions. This shows that the choice of testers was appropriate. Actually, there were not many testers in the lab with such knowledge and we believe that the most knowledgeable among them in this area were involved in this evaluation. The testers that participated in this evaluation were also chosen by the lab manager as the ones that have the most experience with the protocol and the test cases we used in this evaluation.

All of them had no knowledge of the ADIOP system and its functionality when they started this evaluation. This was their first encounter with this system. They were provided

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with an ADIOP user manual to help them in the evaluation. This manual is presented in Appendix C. This shows that the testers did not have any bias towards ADIOP because the tool was new to them. It also shows that it is not so difficult to use by new testers. This will be further supported by the results of this evaluation.

They were all experienced testers and two of them had over 2.5 years of experience. They have been at the UNH-IOL, respectively, for periods of 10 months, 2.5 years, and more than 3.5 years.

This evaluation included practical use and surveys of the 3 components of ADIOP, namely the Decoder, the Diagnoser and the Modeling component (Test Suite Builder). In this section, we report the results obtained for the Modeling component. The results of the evaluation of the other two are reported in Chapter 3.

The evaluation also included a general survey of the ADIOP performance including all three components. The results of this will be reported in the conclusion chapter of this dissertation.

2.11.2 ADIOP Modeling Component Evaluation

Test Case Modeling Analysis

Each tester was given the task to implement test case objects using the Test Suite Builder of ADIOP. One tester used 4 test cases to do this. The other two testers used only 2. This leads to a total of 8 $(4 * 1 + 2 * 2)$ test case objects used for this evaluation. These test case objects were from 4 different protocols: 3 from PNNI Routing, 3 from LANE, 1 from

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

MPOA, and 1 from Q2931. All these test cases were chosen by the testers themselves, and each tester has chosen the same set of test cases as the other testers.

The one test case used from the PNNI Routing protocol was the only one implemented successfully by all three testers. This was counted as three test cases. The testers reported that it took 6 seconds to open the test suite builder window, open a new test case, define the CSP model using ADIOP menus, generate the test case including its compilation, open the test case and run it successfully. This test case did not require any debugging. However, 6 seconds seems too short for this. I suspect that it may have taken 6 minutes which is more realistic for all the tasks above. or the 6 seconds may have referred to the final task which is the execution of the test case.

As for the other three protocols, the test case objects were created using the CSP modeling language and the ADIOP GUI. But they all failed to compile and generate runnable test case objects.

The data provided by the testers does not allow for a thorough investigation of what the problem was. But what we can suspect is the fact that the decoders for the three protocols (LANE, MPOA, and Q2931), which are also used by the test suite builder, were not completely implemented and not fully debugged. I was, however, able to create sample test cases using these protocols that are working.

CSP Modeling Survey Analysis

Each tester also answered questions of a survey on rating the Modeling component, that is the Test Suite Builder (TSB). The survey contained 13 questions. The questionnaire was

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

built based on a likert scale (Likert 1932) that ranges from 1 to-5, with 5 being "Strongly Agree" and 1 being "Strongly Disagree". (Trochim 2000) states that "... to use your Likert scale. Each respondent is asked to rate each item on some response scale. For instance, they could rate each item on a 1-to-5 response scale where:

1. = strongly disagree
2. = disagree
3. = undecided
4. = agree
5. = strongly agree"

The ADIOP attributes that got the higher marks according to the respondents were that:

- ADIOP TSB is friendly: this had three subquestions on whether it has an easy GUI interaction, it is easy to use, and easy to build test cases with. They were all answered with an average score of 4.67 out of 5. This shows that the ADIOP TSB is easy to use and interact with, and that building test cases is simplified for the tester.
- The ADIOP language: this had two subquestions on whether it is easy to model a test case using ADIOP and easy to understand the CSP model definition of a test case. They were all answered with an average score of 4.67 out of 5. The learning curve usually represents an issue when new languages and tools are introduced to testers. But the score obtained here shows that the ADIOP modeling language is easy to learn and use by testers and that the learning period can be very short.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- ADIOP is flexible: that it is possible to correct a test case's definition. The score is also at 4.67. Testers will usually need to update their test cases and correct them. If this means that the tester has to redo everything from scratch, it will mean a lot of work to them. But ADIOP has proven to testers that it is flexible and that it allows them to update their test cases without creating new ones.

- ADIOP is easier to use for automating test cases than other languages (e.g., TCL/TK, C, etc) got a score of 4.67 as well. It can be argued that testers can write their test cases with many other programs. That is true but it is not always easy to do so and it takes time and experience. ADIOP makes this task very easy for the testers who do not need to know much about programming languages to create their test cases and run them.

- ADIOP TSB is a useful tool for the lab also got a score 4.67. It is not enough to just show that ADIOP works. But it is equally important to find out whether it can be deployed and used regularly by testers.

Other statements about ADIOP TSB are:

- The ADIOP TSB will help testers do more interesting work got a score of 4.33. One of the benefits of test automation is the fact that testers can finish their testing quickly and have time to do more interesting work, and this was also confirmed by the testers.

- ADIOP TSB is fast, that the test case objects are built in a reasonable amount of time got also a score of 4.33. One of the things that makes a tool useful is its ability

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to do the task in a reasonable amount of time and this was the case for ADIOP TSB.

- ADIOP TSB generates a correct test case object (i.e., you can execute the test case object and it reports the correct diagnosis) scored 4. This is very important because there is no usefulness of a tool that builds test cases generating false reports.

- Re-usability of ADIOP TSB scored 4 as well, and it states that it is useful to have these test cases stored so there is no need for the testers to know all the details. Testers come to the lab and leave but test cases stay. So, capturing the knowledge of testers leaving is crucial to the lab. ADIOP TSB allows the lab manager and testers to create and store test cases that can be used by new testers even if their knowledge of these test cases is minimal.

The item that scored the least was:

- The new test case is added to the menu on the Diagnoser/Decoder window under the appropriate protocol scored 3.67. I was not able to reproduce this problem. This could be a bug in the program that needs to be fixed, but it does not affect the functionality of ADIOP.

One of the positive points in this evaluation is that the testers' evaluations were all on the positive side, except three that were neutral and one that was negative. The neutral scores were all from only one tester on different questions, and were supported by positive marks from the two other testers. For the only negative mark where one tester responded to the question of re-usability with "Disagree" (i.e., score of 2), the two other testers responded

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with a "Strongly Agree" and the average score for this question was 4.

The overall average score of ADIOP TSB was at about 4.44.

2.11.3 Limitations

This evaluation included only three testers and they only had to use ADIOP in brief sessions. Thus, we cannot generalize the results obtained. But this was the extent of the help we could get from the UNH-IOL to perform this evaluation due to the limited number of available testers and the tight schedule at the lab. However, the fact that all three testers consistently rated ADIOP high on its different attributes strengthens the results even though the sample is tiny.

The test cases used were few in number because of time constraints in the lab. We would have better analysis with more test cases being used. However, another tester, not involved in this evaluation, and I have used ADIOP to implement more than 60 test cases of 5 different ATM protocols: PNNI Routing, PNNI Signaling, LANE, MPOA, Q2931. These test cases contained between 1 and 25 constraints. The evaluation in Chapter 3 gives detailed numbers for different test cases. All of these test cases were compiled and executed.

The original plan for the evaluation included a second step involving judges who will perform a blind review and compare results obtained from manual testing to the ones obtained using ADIOP. But since the evaluation we obtained from the three testers did not include a substantial amount of subjective information to be reviewed by independent judges, we opted for the analysis of this information as it was provided by the testers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.12 Related Work

There has been some related work on using the Object-Oriented approach with CSP.

- (Stone 1995) presents an Object-Oriented Constraint Satisfaction Planning for whole farm management. A whole-farm planning system (CROPS: Crop ROtation Planning System) has been developed and tested on Virginia farms. The implementation is object-oriented and employs partial arc-consistency algorithms, variable ordering, and constraint relaxation. The paper describes the constraint-based scheduler (CBS), its representation, and how it handles constraint relaxation. This system implements classes for CSPSolver, ConstraintManager, and a hierarchy of constraint objects all descended from Constraint, CSPNode, and CSPSolution. The CSPSolver object includes methods to solve a CSP by several different algorithms. The nodes are represented by CSPNode objects. Each CSPNode object has a domain and a list of unary constraints. Each CSPSolver includes a ConstraintManager that keeps track of all the constraints in the CSP: unary, binary, and n-ary. The Constraint class has many subclasses: UnaryConstraint, BinaryConstraint, NAryConstraint which themselves can be superclasses for other classes.

  The difference between this and our work is that variables are represented as objects in the former and as object's parameters in ours. Constraints also are represented as objects while in our work they are methods of the objects. As explained in this chapter, our application uses packets containing many fields and it will not be practical to use variables (i.e., fields) as objects in this case like what is done in this work. So,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

depending on the domain of application and structure of objects used, one way would be more suitable than the other.

- (Puget & Leconte 1995) propose to give access to the constraints as first class citizens of the CLP (Constraint Logic Programming) language. They implemented their approach into an OO language, where constraints are explicitly represented by objects. Their implementation, ILOG Solver, used an abstract machine that is implemented in an object oriented programming language, namely C++. Each finite domain variable, each constraint, and even each non deterministic goal is represented by a C++ object.

  This work represents variables and constraints as objects while in ours variables and constraints are respectively represented as the parameters and methods of the objects. As explained in the previous related work, this is suitable for different applications than the one we used in this dissertation.

- (Roy & Pachet 1997) discuss the problem of representing constraints in an object-oriented programming language. They present a class library that integrates constraints within an object-oriented language. The library is based on the systematic reification of variables, constraints, problems, and algorithms. The library is implemented in Smalltalk, and is used to state and efficiently solve complex constraint satisfaction problems involving Smalltalk structures. BackTalk is a constraint solver written in Smalltalk-80. BackTalk can be seen as a class library for stating and solving constraint satisfaction problems in Smalltalk. The implementation is based on the systematic reification of the main concepts of constraint satisfaction program-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ming: *Domains* are implemented using a proprietary representation. *Variables* are organized into a hierarchy of classes, including integer, Boolean and general purpose constrained variables. *Constraints* are also organized into a hierarchy of classes. Many predefined constraints are provided and user-defined constraints are supported. *Problems* are also represented as objects, thus allowing one to state, and to solve, several problems simultaneously.

This work also represents variables and constraints as objects while in ours variables and constraints are respectively represented as the parameters and methods of the objects. As explained in the previous related work, this is suitable for different applications than the one we used in this dissertation.

- (Paltrinieri 1994b) has abstracted both variables and constraints as defined in the classical CSP to a new, more compact model, called an object-oriented constraint satisfaction problem (OOCSP), by introducing several notions, such as attribute, object, class, inheritance, and association. A visual environment for constraint programming based on the OOCSP model has been developed. An *attribute* is a feature taking values from a domain. An *object* is a collection of attributes. Object attributes correspond to variables in CSP's. The set of attributes of an object defines the *structure* of the object. Objects sharing the same structure are grouped into *classes*. Classes are organized into a *hierarchy*. Constraints can be defined both on object and class attributes. A *solution* to an OOCSP is an assignment of domain values to object attributes such that all the constraints are satisfied. The OOCSP is converted into an

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

equivalent CSP, which is then solved through a traditional constraint-programming language.

The definition of CSP is enhanced through concepts derived from the object-oriented paradigm. The main difference is that here objects do not have methods (but just data members) since their state is updated by the constraints (Paltrinieri 1994a).

This work is the most closely related to ours as it models a set of variables as an object. However, objects do not include methods while in our work, there are objects that are used for decoding and stating models and these include decoding methods. We also present objects that represent test cases and have constraints as their methods. Another difference is that this work converts an OOCSP into an equivalent CSP, while we use OOP for defining CSP models and for generating them.

In this work, objects do not have methods and this is not sufficient in our domain because methods are needed for decoding different types of packets and packets' fields. Test cases are also represented as objects and methods are also needed here to represent constraints. So our approach is more suitable for the domain used in this dissertation. In addition, it provides the ability to include any required functionality within the different objects through the inclusion of methods.

There has been some related work on modeling protocol testing as well.

- In (Marrero, Clarke, & Jha 1997), Model Checking is used for verifying hardware designs, security protocols, etc. By modeling circuits or protocols as finite-state ma-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

chines, and examining all possible execution traces, model checking is used to find errors in real world designs. This work uses finite-state machines for representation.

A model checker accepts a logic model and a logic formula, and through exhaustive analysis determines whether or not the formula holds in the model (Atlee 1992). Security protocols can have bugs that are difficult to find. By examining all possible execution traces of a security protocol in the presence of a malicious intruder with well defined capabilities, it can be determined if a protocol does indeed enforce its security guarantees. If not, a sample trace of an attack on the protocol can be provided.

This presents model checking using exhaustive search to check whether there is an inconsistent instance before using this model. In this dissertation, model checking is done after a test case model execution using non-exhaustive search. We take an instance. that is the model with observed data, and check whether it is consistent.

- A model-based approach has been used in (Riese 1993a) for interpreting observations and diagnosis. The model, called the system description *SD*, includes (possibly extended) finite-state machine (FSM) rules or constraints modeling agent communication behavior. In (Riese 1993b), a protocol is represented as a set of constraints derived from an Extended FSM (EFSM). Several existing approaches to protocol diagnosis and testing are characterized in terms of the EFSM and the CSP formulation.

  In these contributions FSM is still used to represent the protocol specification. Also, the CSP techniques are an extension to this approach, which may carry some disadvantages. Our work shows how to model the protocol specifications from a test suite

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

as CSP models in a declarative way. This represents a new contribution in this field. The CSP formulation of a protocol is derived from the test cases' specifications rather than from some other formalism.

2.13 Summary

In this chapter we discussed CSP modeling of interoperability testing using Object-Oriented Programming. CSP modeling was introduced in Section 2.1. The different modeling architectures were presented in Section 2.2 and our motivation for using a many-models architecture. The CSP modeling process using OOP was outlined in Section 2.4. A more detailed description of how objects are used in modeling is provided in Section 2.5. In Section 2.5.2, the class hierarchy and inheritance that we used in CSP modeling is presented. The modeling GUI is covered in Section 2.6. Section 2.7 described how the test cases that are modeled as CSPs are converted into usable objects with metavariables and constraints, respectively representing their parameters and methods. The more detailed language specification was the subject of Section 2.8. A full example of CSP modeling of an interoperability test case was shown in Section 2.9. Section 2.10 presented an overview of the applications of CSP modeling. The evaluation results of the ADIOP modeling component were presented in Section 2.11. We covered related work in Section 2.12 including work on the integration of CSP and OO as well as work on modeling of protocol testing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3

Constraint-Based Diagnosis of Interoperability Problems

As shown in Chapter 2, the ADIOP (Automated Diagnosis of InterOperability Problems) system provides a modeling language based on CSP that allows the user to implement test cases. We use Object-Oriented Programming (OOP) to implement ADIOP, and we explained why we have chosen this approach and how it is being used. Each test case is represented as an object that is the corresponding CSP model. We have also shown how this is used in a many-models architecture.

In this chapter, we discuss how we use these models to diagnose interoperability problems. The use of CSP for modeling allows us to take advantage of methods and algorithms that already exist for solving CSPs. These algorithms are adapted to exploit the specialized problem domain structure. This provides a better diagnosis for interoperability problems, including developing an accurate and concise explanation of the testing being performed.

In the following sections, we give some definitions related to diagnosis, we define and demonstrate the diagnosis of interoperability problems, and what are the algorithms and methods used for diagnosis, including search and inference. One section is dedicated to explanation and what templates are being used. We then discuss test case execution including

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reports generation. An evaluation of the performance of the different algorithms used is then presented, followed by an analysis of results obtained from an evaluation performed by the testers of ADIOP.

The main contributions to CSP presented in this chapter include solving CSPs that are represented using OOP, using some specialized inference methods that are related to the problem domain structure and generation of human-like explanations.

3.1 Definitions

In the literature, there are many definitions to diagnosis as well as many ways to perform this diagnosis. Diagnosis has been applied to physical devices, software testing, networking protocols, etc. For each one of these fields, there are many methods for diagnosing problems, representing problems, diagnosis steps, and reports. In the related work section, we talk about some of these issues.

Diagnosis in general can mean many things. The diagnosis we are presenting in this dissertation addresses one area of diagnosis related to testing of networking protocols. Although others have dealt with the area of network diagnosis, such as (Riese 1993a), (Riese 1993b), (Sabin *et al.* 1995b), and (Leckie 1995), their motivation and goals differ from ours.

The following definitions are taken from (ATMF-TestSpec 1994):

Definition 3.1 (Implementation Under Test (IUT)) : *The part of the system that is to be tested.*

Definition 3.2 (System Under Test (SUT)) : *The system in which the IUT resides.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition 3.3 (Conformance Testing) : *Testing the extent to which an IUT conforms to a specification.*

Definition 3.4 (Interoperability Testing) : *Testing the degree of compatibility between two different implementations based on features that both have implemented.*

Definition 3.5 (Test Event) : *An indivisible unit of test specification (e.g., sending or receiving a single PDU (Protocol Data Unit)).*

Definition 3.6 (Test Step) : *A named subdivision of a test case, constructed from test events and/or test steps.*

Definition 3.7 (Test Case) : *A series of test steps needed to put an IUT into a given state to observe and describe its behavior.*

Definition 3.8 (Test Group) : *A named set of related test cases.*

Definition 3.9 (Test Suite) : *A complete set of test cases, possibly combined into nested test groups, that is necessary to perform conformance testing or interoperability testing for an IUT-or a protocol within an IUT.*

In summary, conformance testing attempts to evaluate an implementation against a specific protocol specification, and interoperability testing attempts to evaluate an implementation against other implementations; regardless of how well it meets the protocol specification (ATMF-TestSpec 1994).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In this dissertation, we are interested in diagnosing interoperability problems in networking devices. Figure 3.1 shows the physical setup for performing interoperability testing. Two devices A and B are said to be interoperable according to a networking protocol P if they both implement protocol P and if the data flow between the two devices conforms to the specifications of protocol P. Usually an analyzer is used as a monitoring device between the two devices to capture the data flow sent and received by both. This data flow is a set of observations or packets that are decoded by the analyzer and provided to the user in a readable format. These observations are compared to the protocol specifications to check if they conform. The two devices pass the interoperability testing if the observations and specifications match.

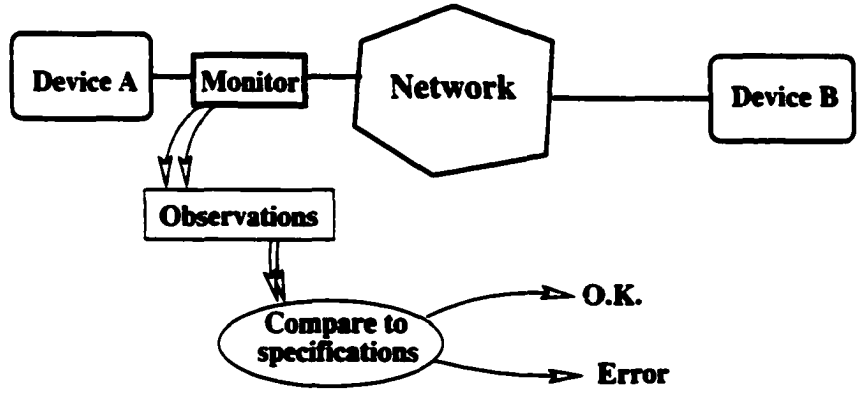

Figure 3.1: Diagnosis of Interoperability Problems

Usually an interoperability test specification (test suite) derived from the protocol specification is used for interoperability testing. Test suites are written manually by testers or organizations and it takes a considerable amount of effort to create them. ADIOP provides

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

testers with a tool to implement test cases included in these test suites. In this dissertation, we use the words 'protocol specification' or 'specification' to mean the interoperability test specification for that particular protocol, unless we make a distinction.

The logical setup of interoperability testing is shown in Figure 3.2. More details on how the diagnosis of interoperability problems is performed are provided in the next section.

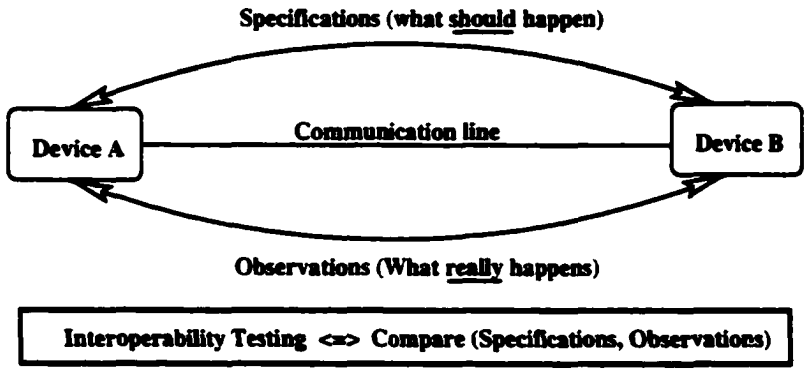

Figure 3.2: Statement of Interoperability Problems

The following is the definition we use for the word **Diagnosis** in the context of this dissertation.

Definition 3.10 (Diagnosis) : *of interoperability problems is the detection of problems that occur when two devices running the same networking protocol are connected to each other through a network. The devices are assumed to have passed conformance testing, which states that each device by itself is conformant in its behavior to the protocol specification. Otherwise, a problem that is due to conformance of one device to the protocol specification may show up as an interoperability problem between two devices, which would be misleading.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2 Modeling, Decoding and Diagnosis

Figure 3.3 shows the three components of ADIOP, namely the modeling, decoding, and diagnosis components. Modeling and decoding are two steps that are needed before diagnosis takes place. The Figure shows the interrelation between these three components.

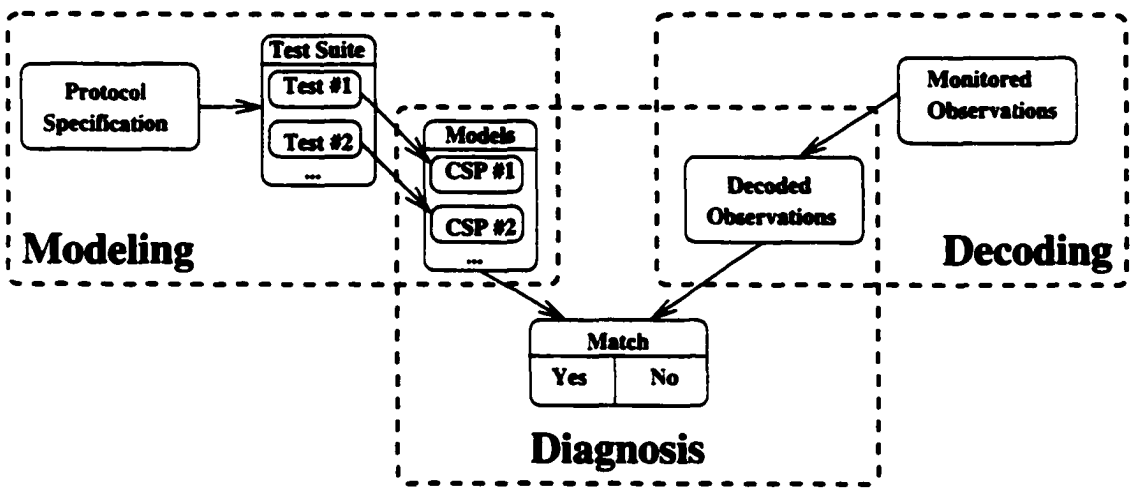

Figure 3.3: Modeling, Decoding and Diagnosis Components

Since we plan to use the test suite specification for diagnosis instead of the protocol specification, we need to diagnose problems using test cases taken from the test suite specification. This is why we have chosen for modeling the many-models architecture instead of the one-model architecture (refer to Section 2.2 for more details).

The modeling component was detailed in Chapter 2. ADIOP provides a Graphical User Interface (GUI) that uses a simple modeling language allowing the user to build a CSP model for each test case. We also discussed the use of Object-Oriented Programming (OOP) in conjunction with CSP. Objects are used to define CSP models as well as representing test

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cases. The outcome of this modeling is a number of objects each representing one test case with packets (metavariables) as its parameters and constraints as its methods. All the test case objects built using this component are accessible through the menu in the Decoder/Diagnoser window of ADIOP. One input for the diagnosis component is the set of objects (CSP models) representing test cases.

The decoding component is responsible for taking the data captured by one analyzer and decoding it into a format that can be used by ADIOP for diagnosis. Figure 3.4 shows the main ADIOP window with an example of the data captured.

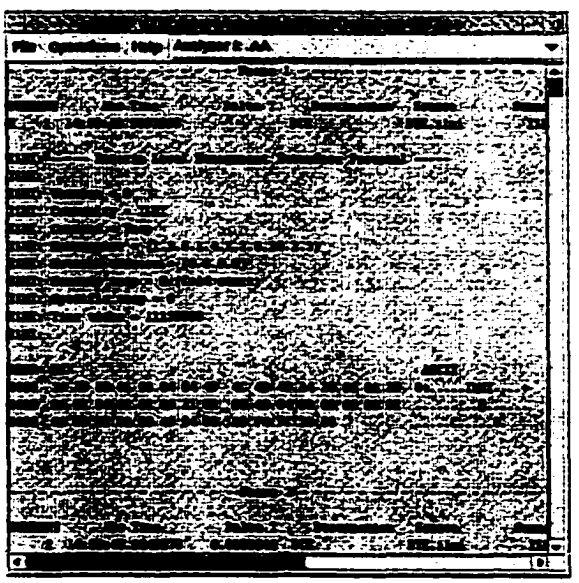

Figure 3.4: ADIOP's Main Window

Figure 3.5 shows the protocol analyzers supported by ADIOP. The names shown here are different from the real names used in ADIOP to maintain the anonymity of vendors. This list can be extended as needed if more analyzers are being used in the lab by adding

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the corresponding decoder. Testers do not need to know what decoder to use. ADIOP picks the appropriate decoder type according to the extension of the file containing the captured data.

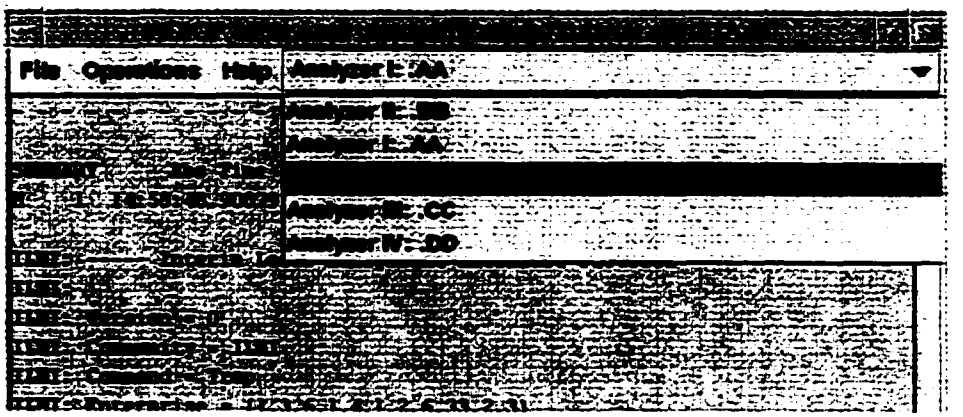

Figure 3.5: List of Protocol Analyzers Supported

Figure 3.6 shows the Decoder/Diagnoser window of ADIOP. It shows the summary of decoded observations as its data, and the test case objects that were built by the modeling component as its menu list (see Figure 3.7). The lower panel in the window shows the details of the packet highlighted in the upper panel that contains the summary of the packets observed.

Figure 3.7 shows the menu generated in the Decoder/Diagnoser window that includes all the test case objects built using the modeling component. This figure shows the menu of test cases for Section 4301H of the PNNI Routing Protocol.

The diagnosis component takes the decoded observations from the decoding component and checks if they match the CSP model of the test case being used. In terms of CSP, this means that the decoded observations are metavalues that metavariables can be assigned.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

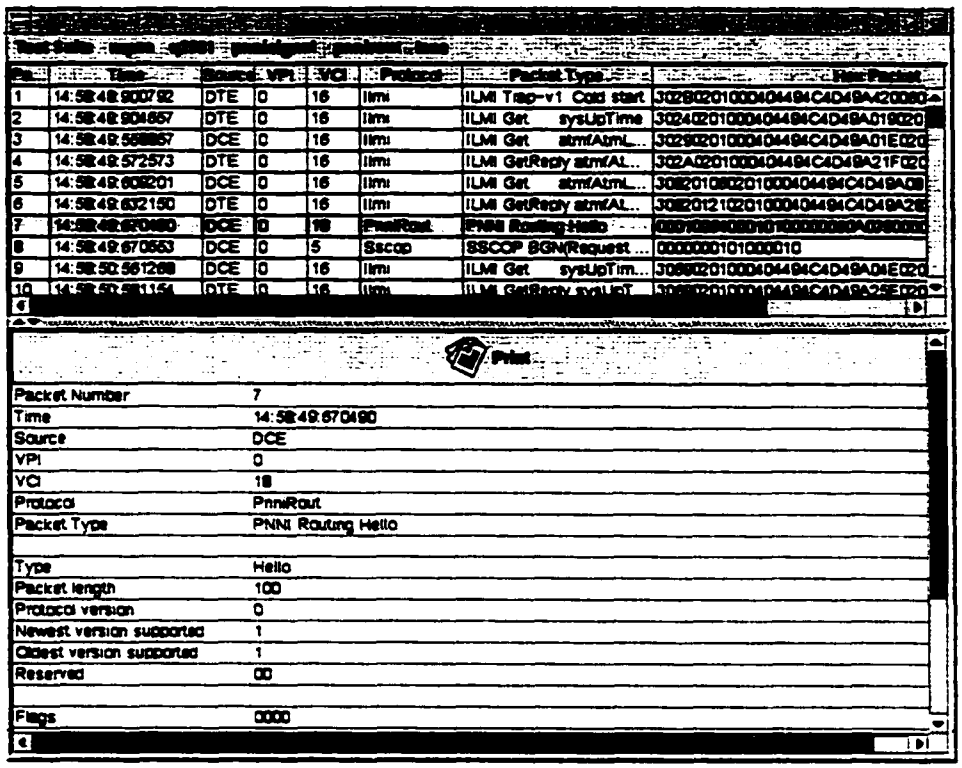

Figure 3.6: The Decoder/Diagnoser Window

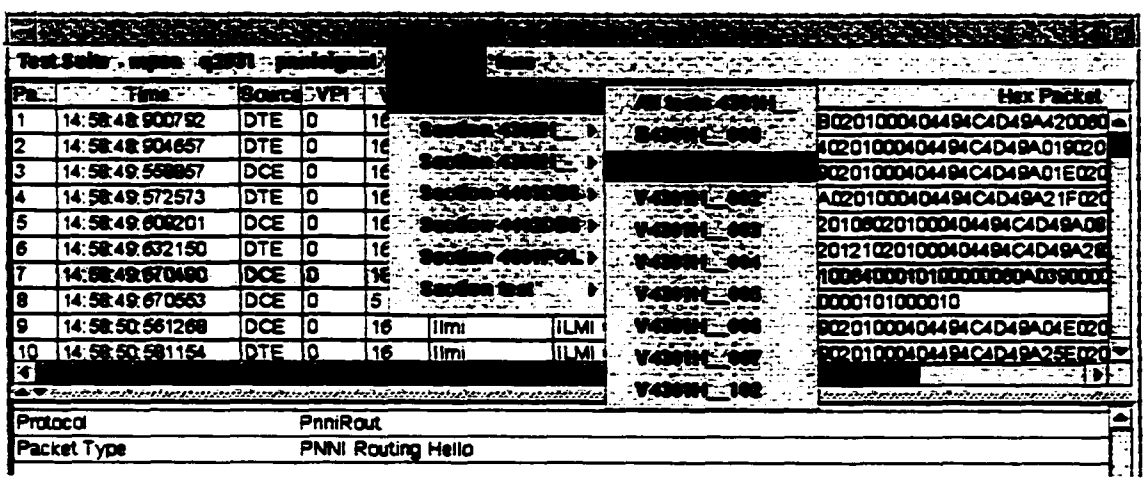

Figure 3.7: Test Suite Menu

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The model provides the metavariables that are defined in the test case as well as the constraints that need to be satisfied.

Different algorithms are being used for this purpose and these are presented in Section 3.4. ADIOP also generates an explanation of the diagnosis, which is the subject of Section 3.5. A report can be generated for one test case or a set of test cases from the same test group (section), and this will be detailed in Section 3.6.2.

3.3 Diagnosis of Interoperability Problems

In (ATMF-TestSpec 1994), it is stated that: "The problem of interoperability arises when end-users need to interconnect equipment from different manufacturers and to have a certain confidence level that these pieces of equipment can interoperate. The purpose of interoperability testing is to confirm the degree of interoperability. Interoperability testing is used to measure the condition under which two or more systems with separate and different implementations will interoperate and produce the expected behavior. Interoperability testing can be bound to specific protocols within the stack. It involves testing both the capabilities and the behavior of an implementation in an interconnected environment and checking whether an implementation can communicate with another implementation of the same or of a different type."

In this dissertation, the scope of interoperability testing involves detecting and analyzing problems of non-interoperability that exist between two devices. Other methods and tools have been used for checking the interoperability of devices, including manual/visual

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

testing using monitors and expertise. There are some proprietary software packages that implement some test cases of certain protocols. These are usually implemented using known programming languages but they do not attempt to provide the tester with the flexibility to easily implement more test cases. In contrast, we have shown in Chapter 2 how ADIOP provides a user-friendly interface so that the tester can easily implement and run new test cases.

Our contribution to diagnosis has two parts. First, we provide a modeling language that allows the user to implement test cases to perform different steps of interoperability testing. Chapter 2 expands on this subject. Second, since we use CSP for modeling, this allows us to take advantage of methods and algorithms that already exist for solving CSPs. Section 3.4 tackles this issue in more detail. Some of these algorithms can be tuned to respond to some specificities of this problem domain, and allow for a better diagnosis of the interoperability problems, including to accurately and concisely explain the detected problems. Section 3.5 is devoted to this subject.

ADIOP automates part of the process of interoperability testing by providing a useful tool for the user to create reports of interoperability testing for different devices. These reports are now created manually by looking at the monitored data through an analyzer. Our goal is to simplify this task and provide an easy-to-use user interface that supports decoding and the analysis/diagnosis of the observed data, as well as reports generation. First, each test case from the test suite is modeled as a Constraint Satisfaction Problem (CSP). Second, the diagnosis is done by checking whether all the constraints are satisfied.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The evaluation section (3.8) of this chapter presents a more detailed comparison of manual versus automated diagnosis of interoperability testing gathered from a questionnaire used by testers (See Appendix B).

3.4 Algorithms for Diagnosis

The advantage of CSP is that it is a reasoning mode that provides both modeling and problem solving within the same framework. As mentioned earlier, there are many CSP methods that one can make use of when the problem is represented as a CSP. The problem solving methods in CSP have ranged from pure search (e.g., backtrack) to inference (e.g., arc consistency). While each has its advantages and shortcomings, they both have evolved and depending on what applications we are dealing with and what our goals are, one or the other or a combination of both would be more advantageous.

Our focus in this dissertation is on how to get a "good" explanation to the problem we are solving. As we show later, there is a limited concern on the time it takes to solve the problem even when only search is used to find a solution. This is somehow obvious if we look at the small size of the problems we are dealing with. As we explained in the modeling section, each test case is represented as a separate CSP model. The number of metavariables is usually very limited in each test case. The captured data can be very large, but it is easy to prune many of its metavalues by a simple and fast preprocessing of packets. This will be discussed in more detail later in this section.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.4.1 Constraint Satisfaction Methods

When we have the CSP representation of a system, we can use different methods to solve it independently of the context of the application. Figure 3.8 shows how CSP is used for problem representation and problem solving. The main two problem solving techniques are: Search and Inference. There are many algorithms that use Search exclusively such as backtracking. (Kumar 1992) states that: "Backtracking involves instantiating each variable (that is, giving it a value from its domain) sequentially, and checking to see if the set of instantiated variables satisfies all constraints involving the variables instantiated so far. In other words, it behaves as a depth first search in the space of potential CSP solutions. Backtracking is still an imperfect search method, as it suffers from a phenomena called *thrashing*, in which search in different parts of the problem space keep failing for the same reasons." Backtracking search may also have to explore the entire tree of possibilities to find a solution.

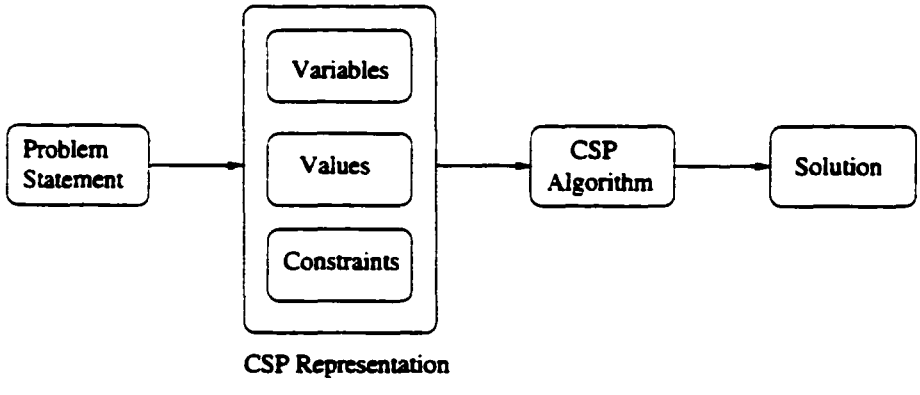

Figure 3.8: CSP for Problem Representation and Problem Solving

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Other algorithms make use of inference such as Node Consistency (NC) and Arc Consistency (AC). AC checks whether the values of two variables are consistent with each other, and deletes a value from the domain of the first variable if it has no support from any value of the second variable. i.e., there is no value in the second variable that is consistent with the value from the first variable. This value is deleted because it cannot participate in any solution. Node consistency is the lowest type of consistency. It checks whether unary constraints (constraints involving one variable) are satisfied.

Arc Consistency makes all the values of every 2 variables consistent. Path consistency makes all the values of every three variables consistent. As the consistency level gets higher we get closer to the solution, but it gets more complex to perform.

In general, a graph is *K-consistent* if the following is true: Choose values of any K-1 variables that satisfy all the constraints among these variables and choose any K^{th} variable. Then there exists a value for this K^{th} variable that satisfies all the constraints among these K variables. A graph is strongly K-consistent if it is J-consistent for all $J \leq K$. Node consistency discussed earlier is equivalent to strong 1-consistency and arc-consistency is equivalent to strong 2-consistency (Kumar 1992).

It has long been known (Freuder 1978) that CSPs can be solved by pure inference, involving higher and higher order consistency processing. For some problem structures it has been proven that limited higher order consistency processing is sufficient to leave only backtrack-free search (Dechter & van Beek 1995). However, the efficiency of obtaining even moderately high order consistency processing can be problematic in practice.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The drawback of search is that the time to explore all the possibilities grows exponentially with the number of variables. The drawback of inference is that lower consistency checking, such as NC and AC, is usually not enough to solve the entire problem. For solving the entire problem using only inference, one needs to perform higher order consistency checking that includes many variables, but that leads again to the same problem of higher complexity as with search.

Research and experience have shown that the most successful techniques for solving CSPs are the ones that combine both search and inference. Nevertheless, arc-consistency techniques and backtracking search have sufficed for a number of practical applications of constraint programming (Wallace 1996). The question is then how and when do we combine these two to get the best results. That depends on the domain of application, the size of the problem, and the available resources (e.g., memory, etc).

CSP provides many advanced algorithms to simplify or solve hard problems. Some surprising successes have been achieved by the simple combination of constraint propagation and search. For example, constraint propagation techniques have recently enabled interval reasoning to achieve some spectacular results (Van Hentenryck, McAllester, & Kapur 1995). Constraint reasoning takes advantage of many mathematical methods and algorithms that were improved to work on CSPs. CSP has been used in many real world applications as a modeling and a problem solving tool. In fact commercial constraint programming systems have moved "beyond the black box" (Puget & Leconte 1995) (Wallace 1996). These applications have improved the CSP paradigm and made it more widely used.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Because of the different applications and domains where the CSP paradigm has been used, there were also some extensions to it. Partial CSP (Freuder & Wallace 1992) involves finding values for a subset of the variables that satisfy a subset of the constraints. PCSP can be used to solve over-constrained problems by allowing the violation of some constraints. Dynamic CSP (Mittal & Falkenhainer 1990) can be applied in domains where the set of constraints and variables involved in the problem evolves with time. Composite CSP (Sabin & Freuder 1996) unifies several CSP extensions, providing a more comprehensive and efficient basis for formulating and solving configuration problems. An example of these extensions is the hierarchical domain CSP where a value may itself be another CSP.

The CSP has a solution if there is an assignment of values to variables such that all the constraints are satisfied. A solution in CSP can mean different things depending on the context and the goal to be achieved. The goal can be to find any solution, an optimal solution, a solution with specific characteristics, to find whether there is a solution, how many solutions the problem has, or why a solution cannot be found. In this dissertation, we are interested in finding whether one solution exists. The first solution found, if there is one, is presented to testers as an explanation of the successful result of interoperability testing. If no solution exists, ADIOP uses other methods, which will be discussed later in this dissertation, to provide testers with a useful explanation.

3.4.2 Search

The first algorithm we make use of in our application is simple backtracking. This algorithm is adapted to the OO-based CSP we are using. Hence, we use metavariables and metavalues

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instead of variables and values. Figure 3.9 shows the algorithm for backtracking to find the first solution if it exists.

```
Backtrack (MetaVariables, MetaValues, Assignment, Solution)
1.  Begin
2.    If MetaVariables is empty
3.    |  Solution <- Solution + Assignment;
4.    |  Solutions_Nbre++;
5.    |  return;
6.    EndIf
7.    For i <- 0 to MetaValues.size() - 1
8.    |  If (Check (Assignment, MetaVariables[0], MetaValues[i]))
9.    |  |  MetaVariables <- MetaVariables - {MetaVariables[0]};
10.   |  |  MetaValues <- MetaValues - {MetaValues[i]};
11.   |  |  Assignment <- Assignment + {[MetaVariables[0], MetaValues[i]]};
12.   |  |  Backtrack (MetaVariables, MetaValues, Assignment, Solution);
13.   |  |  If (Solutions_Nbre != 0)
14.   |  |  |  return; // This returns after the first solution is found
15.   |  |  EndIf
16.   |  EndIf
17.   EndFor
18. End
```

Figure 3.9: Backtrack Algorithm

Solutions_Nbre is a variable that stores the number of solutions found. This variable is declared in the class "Model", and initialized in the **Solver** function (see Figure 3.12). When this variable has a value of 1, the backtrack function exits because it searches for the first solution. The parameter **Solution** stores the solution found. It contains a set of assignments of metavalues to metavariables. The parameter **Assignment** contains the set of assignments of metavalues to metavariables made at each step of the search algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The algorithm uses the metavariables declared in the model of the test case being used. The metavalues are the packets contained in the decoded observations. The algorithm searches the tree of possibilities for a solution where there is an assignment of one metavalue to each metavariable so that all the constraints are satisfied.

The **Check** function checks if one metavalue assigned to one metavariable is consistent by itself and whether it is consistent with the previous assignments made to other metavariables from the same model. The algorithm is presented in Figure 3.10.

mv is a local array to store the metavalues (i.e., packets captured). **type** is a local variable that stores the packet type and is used with inference methods. The **CONS** array stores names of the constraints as they appear in methods of the class corresponding to a test case model generated by ADIOP.

Lines 6-9 check whether a metavariable is optional and whether it can be assigned an empty packet (i.e., no packet). If the metavariable is optional, line 8 returns true. If no constraint is defined in the CSP model about the status of a metavariable, then it is assumed that it is mandatory and thus these statements return false. An optional metavariable can be assigned either an observed packet or no packet at all.

Lines 10-12 check that no two metavariables are assigned the same observed packet. Lines 13-15 check unary constraints. Lines 16-20 check binary constraints within the same metavariable. Lines 21-29 check binary constraints that involve this metavariable and other metavariables already assigned.

The **CONS** array stores the names of constraints defined in a test case as methods of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Boolean Check (Assignment, metaVariable, metaValue)
1.  Begin
2.    index <- Assignment.size();
3.    mv[index] <- metaValue;
4.    type <- fillMetaVar (index, mv[index]);
5.    If (type == null) return false;
6.    If (mv[index].status == ABSENT)
7.    |  If (CONS[index][STATUS][index][STATUS] == null) return false;
8.    |  Else return getValue(CONS[index][STATUS][index][STATUS]);
9.    EndIf
10.   For i <- 0 to Assignment.size() - 1
11.   |  If (Assignment.mv[i] == mv[index]) return false;
12.   EndFor
13.   For i <- 0 to maxVariables
14.   |  If (getValue(CONS[index][i][index][i]) == false) return false;
15.   EndFor
16.   For i <- 0 to maxVariables
17.   |  For j <- i+1 to maxVariables
18.   |  |  If (getValue(CONS[index][i][index][j]) == false) return false;
19.   |  EndFor
20.   EndFor
21.   For i <- 0 to index
22.   |  fillMetaVar(i, mv[i]);
23.   |  If (mv[i].status == ABSENT) continue;
24.   |  For j <- 0 to maxVariables
25.   |  |  For k <- 0 to maxVariables
26.   |  |  |  If (getValue(CONS[i][j][index][k]) == false) return false;
27.   |  |  EndFor
28.   |  EndFor
29.   EndFor
30.   return True;
31. End
```

Figure 3.10: Check Function

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the corresponding object. These methods are invoked using the **getValue** function that shows in Figure 3.11.

```
Boolean getValue (Constraint_Function)
1.  Begin
2.    If (Constraint_Function == null) return True;
3.    constraintChecks++;
4.    return invoke_method(Constraint_Function);
5.  End
```

Figure 3.11: GetValue Function

The **fillMetaVar(index, mv)** method is implemented in each test case object. This method assigns a 'packet' from the observations to the metavariable of position 'index' in the test case model.

Before we make use of the **Backtrack** function, there are some initializations to be made. The **Solver** function makes these initializations as shown in Figure 3.12.

```
Integer Solver (MetaValues)
1.  Begin
2.    For each mv in MetaValues
3.    |  mv.status <- PRESENT;
4.    EndFor
5.    Solutions_Nbre <- 0;
6.    emptyMetavalue.status <- ABSENT;
7.    MetaVariables // Get assigned by each test case
8.    MetaValues <- MetaValues + emptyMetavalue;
9.    Assignment <- empty
10.   Solution <- empty
11.   Backtrack(MetaVariables, MetaValues, Assignment, Solution);
12. End
```

Figure 3.12: Solver Function

Lines 2-4 state that all observed packets are packets that can be assigned to a mandatory

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

or an optional metavariable. Line 6 creates one empty packet (i.e., no packet). Line 8 adds it to the set of observed packets to constitute the set of metavalues. This empty packet can be assigned to an optional metavariable if there is no observed packet to be assigned to it.

All these methods are part of the object **Model** which is the parent of all objects modeling test cases. All the test cases are children of the **Model** object and thus they inherit all the methods mentioned above for solving the CSPs (see Figure 3.13).

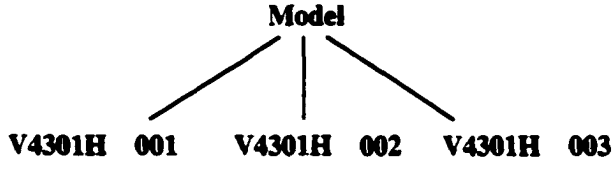

Figure 3.13: The Test Cases (Objects) Hierarchy

The set of objects representing test cases are stored under the *testsuite* directory under the appropriate protocol name using a test suite directory hierarchy (See Figure 3.14).

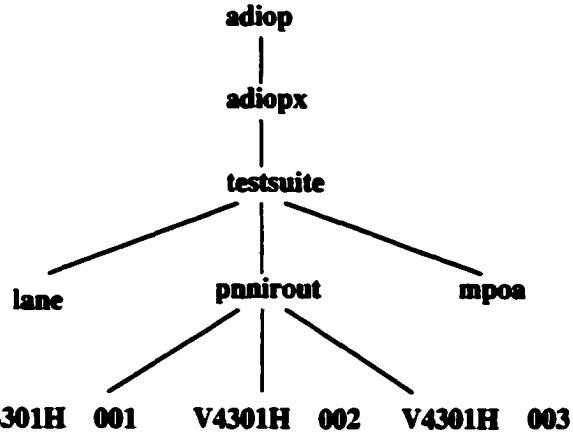

Figure 3.14: The *testsuite* Directory Hierarchy

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since the problems are small, search returns very quickly with a solution if one exists. If there is a solution, it is reported to the user with an explanation of which packets satisfy the constraints of the test case. One example of this is shown in Figure 3.15.

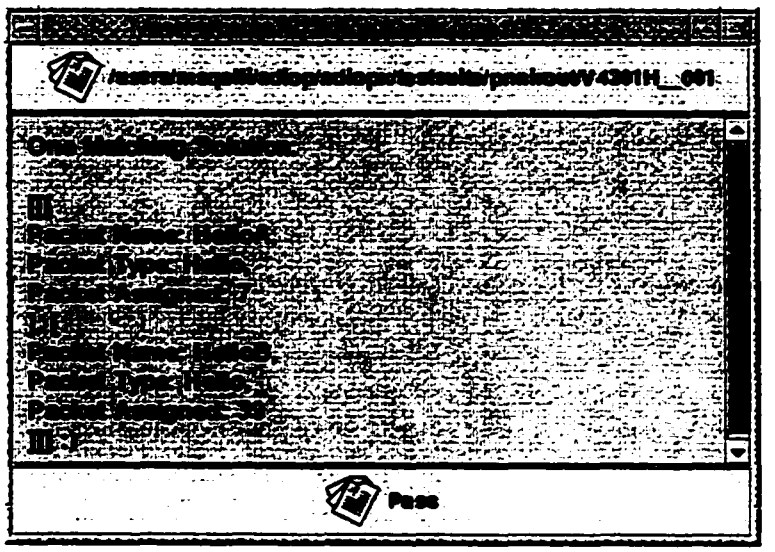

Figure 3.15: ADIOP's Result Window of a Successful Test Case

The user is not usually interested in learning how ADIOP found the solution. However the user may want to have the metavalues (observed packets) that were assigned to metavariables to be able to understand the presented solution. Multiple solutions are not important for the user because the outcome of diagnosis is based on whether there is a solution (i.e., whether the test case passes or fails) rather than the number of solutions found. This is why, when a test case fails and no solution is found, diagnosis and explanation becomes more crucial for the tester and requires more investigation by ADIOP.

When using only search and there is no solution to the test case being executed, it fails

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and the solution reported is not very meaningful to the user because it just states what constraints have been violated. While this may give some hints to the tester if the number of constraints violated is small, it is not very useful.

One way to provide a better explanation is the use of inference or some specialized methods that check for certain conditions, allowing the system to report some meaningful explanation of the diagnosis of interoperability problems.

3.4.3 Inference and Consistency Checking

In interacting with human users it may not be enough to simply supply a solution to a problem. The user may want an "explanation": how was the solution obtained or how is it justified? The computer may be functioning as a tutor or as a colleague. The user may want to consider alternative solutions, or need to relax constraints to permit a more complete solution. In these situations it is helpful if the computer can think more like a person, and people tend to use inference to avoid massive search (Sqalli & Freuder 1996b).

Tracing through the search process of backtracking would result in an explanation of the form: "I tried this and then I tried that, and it didn't work, then I backed up and tried the other, ...". A more useful explanation to the user is an inference-based one with statements of the form: "X cannot be v because ..."

The use of "pure inference" problem solving in the domain of logic puzzles is presented in (Sqalli & Freuder 1996b). The focus of that work was on the support that inference methods provide for explanation. It was also demonstrated how surprisingly powerful the inference methods can be.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We propose to use search supplemented by consistency inference methods in a CSP context to support explanations of the problem solving behavior that are considerably more meaningful than a trace of a search process. Constraint satisfaction problems are typically solved using search, augmented by general purpose consistency inference methods.

Even when inference does not provide a complete solution, it can still be used as a preprocessing step and the results obtained from this can be then be given to a search engine. If a combination of inference methods fails to completely solve a problem, the progress made in the form of domain reductions might be exploited by subsequent search.

Node Consistency

Node consistency (NC) is the lowest type of consistency. It checks whether unary constraints (constraints involving one variable, (e.g., $V < 3$)) are satisfied. Node consistency is equivalent to strong 1-consistency.

As stated in Chapter 2, ADIOP provides a way of defining unary constraints. These constraints are stated using variables and not metavariables. For example we can state that the variable **source** of the metavariable **1WayInA** has to be equal to **D_Source**, where **D_Source** is a domain containing **DCE** and **DTE**. We are not interested in the NC where a node is a variable because these inferences are not useful for explanation in this domain. Instead, we are interested in the NC where a node is a metavariable.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Specialized Node Consistency Inference for an OO-based CSP

We propose to use Node Consistency at the metavariable level. We call this MetaVariable Consistency (MVC) as it differs from the NC presented earlier. A metavariable represents a packet with many fields. Each field is a variable in the CSP. The CSP model we use is defined in term of metavariables. The observed packets are the metavalues that represent the domains for the metavariables. For each metavariable we have to assign a metavalue satisfying all the constraints to obtain a solution. All metavariables have the same domains of metavalues initially. This set of metavalues is the domain of all observed packets.

There are some variables that can be used to reduce the domain of metavalues for metavariables. We use two of these to perform some preprocessing and obtain some useful explanation in addition to problem solving time reduction. These are **protocol** and **packetType**. We make use of some inferences that ADIOP can check by looking at these two variables. The **protocol** variable is set to the same value for all metavariables of a CSP model representing one test case because a test case belongs to a test suite written for the same protocol.

Our first inference is that if there are no packets observed that match the protocol defined in the CSP model of a test case, then there is no solution to the problem. The algorithm for domain reduction using the **protocol** variable is shown in Figure 3.16.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
ProtocolPreprocess (protocolFromModel, MetaValues)
1.  Begin
2.    For each mv in MetaValues
3.    |  If (protocolFromModel != mv.protocol)
4.    |  |  MetaValues <- MetaValues - mv
5.    |  EndIf
6.    EndFor
7.    If (MetaValues.size() == 0)
8.    |  Report that there are no observed packets
      |  of the protocol type used in the model
9.    EndIf
10. End
```

Figure 3.16: Protocol Preprocess Function

In addition, the value of the **packetType** is used to reduce the domains of metavalues for the metavariables. For instance, all the observed packets of types different than the ones defined in the CSP model can be deleted from the domain of metavalues. The algorithm for this preprocessing is shown in Figure 3.17.

The algorithm is made more efficient by assigning the same domain of metavalues to all the metavariables of the same type (Lines 10-16). **domain[i][0]** stores the packet type for the metavariable **MV[i]**. The **allDomainsEmpty** indicates whether all the domains for all the metavariables are empty. This would be one explanation to the failure of a test case. If only one metavariable's domain is reduced to become empty with this preprocessing, then the explanation given to the user would state that this metavariable cannot be assigned any metavalue (packet).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
PacketTypePreprocess (MetaVariables, MetaValues, Domains)
1.  Begin
2.    allDomainsEmpty = True;
3.    For each MV[i] in MetaVariables
4.    |  domain[i] <- null;
5.    |  For each mv[j] in MetaValues
6.    |  |  type = fillMetaVar(i, mv[j]);
7.    |  |  If (type != null)
8.    |  |  |  domain[i] <- domain[i] + {mv[j]};
9.    |  |  |  allDomainsEmpty = False;
10.   |  |  |  For k <- 0 to i-1
11.   |  |  |  |  If (domain[k].size() != 0 && domain[k][0] == domain[i][0])
12.   |  |  |  |  |  domain[i] = domain[k];
13.   |  |  |  |  |  done = True;
14.   |  |  |  |  |  break;
15.   |  |  |  |  EndIf
16.   |  |  |  EndFor
17.   |  |  EndIf
18.   |  |  If (done)
19.   |  |  |  done = False;
20.   |  |  |  break;
21.   |  |  EndIf
22.   |  EndFor
23.   EndFor
24. End
```

Figure 3.17: Packet Type Preprocess Function

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A more interesting situation is when the size of the domain shared by n metavariables is reduced to r with $r < n$, then there is no possible solution. This can be seen as a clique of metavariables where all of them have to be assigned a different metavalue but the number of metavalues is not sufficient.

Appropriate explanations are generated for these situations through the use of templates. Templates for the different kinds of explanations used in ADIOP are stated in Section 3.5.

Limitations

The inference methods we use in this dissertation show that we can obtain better and more human-like explanations. However, we do not claim that these methods cover all of the cases we may have.

Partial CSP can be used to relax some of the constraints. This would yield a better explanation if only one constraint is violated. In this case the system can report to the user that this constraint is the one that caused an interoperability problem.

3.5 Explanation

As stated earlier, (Sqalli & Freuder 1996b) present some specialized inferences that are useful for explanation. It was obvious that a trace of standard search techniques would not produce anything like a satisfactory explanation. This led to the use of inference methods, and the transformation of individual inferences into bits of explanation.

Inference is used mainly to reduce the domains of metavariables. The different results

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that inference leads to are used as the input to some templates so that the user receives a useful explanation for the outcome of a test case execution.

Some of these templates are used with the search algorithm. Although some of these are not very useful explanations for the user when a test case fails, we include them here (see template 7) for a complete list of all types of explanation that the user may receive. The previous section discussed the different algorithms used to obtain an explanation of the execution of an interoperability test case. Inference methods are used first to check whether the captured data fulfill certain conditions that allow ADIOP to get an explanation of the interoperability problem without using search. We showed the algorithms used with the different inference methods. The outcome of these methods determines which template is to be used, and what specific values are to be included with this template to generate an explanation. For example, when there is an over-constrained clique ADIOP gets the template to be used and the **packetType** value that caused the over-constraint (see template 4). These two are merged together to form an explanation of the interoperability problem. If inference methods are not successful, search is used. If there is a solution, search generates the assigned observed packets to the metavariables in the test case CSP model. This is reported as an explanation of the interoperability test case. If there is no solution, all the constraints violated during search are reported in the explanation. This last one is not useful to testers, and that is why we use CBR, as detailed in Chapter 4, to provide better explanations in this case.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Explanation Templates

The results obtained using search and inference are used with the templates presented in this section to provide the user with an explanation of the results of running a test case. These templates are specific to this domain, but the inferences used to generate them are not. For example the use of inference with an over-constrained clique to generate explanations is not domain specific, but the template we use in this section is. It will be possible to write templates for other domains using the same inference methods. We have done more work on this in (Sqalli & Freuder 1996b) where we used the same generic templates to obtain explanations for different domains using logic puzzles. In this dissertation, these templates are used with different ATM protocols. One way to improve on this is to have two levels of templates. The first level generates generic templates from the inference, and the second level takes these generic templates and applies them to a certain domain by using specific keywords to obtain a final explanation.

1. *There is no observed packet from the* ***protocolTested*** *protocol*: this template is used when the value assigned to the variable **protocol** does not match the protocol type of any of the packets observed. As stated in Chapter 2, there is a statement in the CSP model of each test case that indicates what protocol is being tested. This statement is of the form "$PROTOCOL *protocolTested*". The **protocol** variable for each metavariable is assigned this **protocolTested** value.

2. *There are no observed packets matching any type of the ones stated in the model of this test*: this template is used when domain reduction from metavariable con-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sistency (MVC) using the **packetType** variable leads to empty domains for all the metavariables defined in the CSP model of this test case. It means that for all these metavariables there are no packets to be assigned from the ones observed.

3. *There is no observed packet matching the type **packetType** as it is stated in the model of this test*: this is used when domain reduction from MVC using the **packetType** variable leads to empty domains for all the metavariables of one type of packets defined in the CSP model of this test case. It means that there are no packets observed that have the type **packetType** and so there is no assignment possible for all the metavariables of this type.

4. *There are fewer observed packets of type **packetType** than what is stated in the model of this test*: this template is used when the number of metavariables of one type of packet stated in the model of a test case is less than the number of packets observed of this type. It means that there are not enough packets observed of such type to be assigned to all the metavariables of the same type. This is equivalent to a clique of metavariables with the same domain of metavalues that has a size smaller than the number of metavariables in the clique.

   An example of this explanation is presented in Figure 3.18.

5. *One Matching Solution: [[Packet Name: **packetName**, Packet Type: **packetType**, Packet Assigned: **packetObservedNumber**], ..., [...]]*: this is used when the search algorithm is executed and there is a solution to the execution of this test case.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

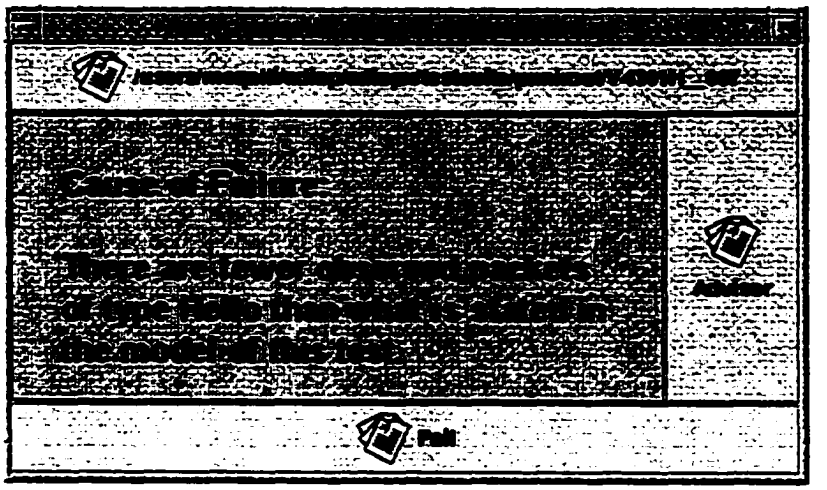

Figure 3.18: ADIOP's Result when packets of a packet type are fewer than required

An example of this explanation is presented in Figure 3.15.

6. *One Matching Solution: [WARNING: There is a missing packet, [packet Name:* ***packetName****, Packet Type:* ***packetType****, Packet Assigned:* ***packetObservedNumber****], [packet Name:* ***packetName****, Packet Type:* ***packetType****, Packet Assigned: None (Optional)], ..., [...]]*: this is used when the search algorithm is executed and there is a solution to the execution of this test case. However, the solution found contains an optional metavariable that was not assigned a packet.

   An example of this explanation is presented in Figure 3.19.

7. *One or more of these constraints declared in the model of this test is/are violated:* ***violatedConstraints***: this is used when the search algorithm is executed after the preprocessing has not lead to one of the explanations mentioned earlier, and there is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

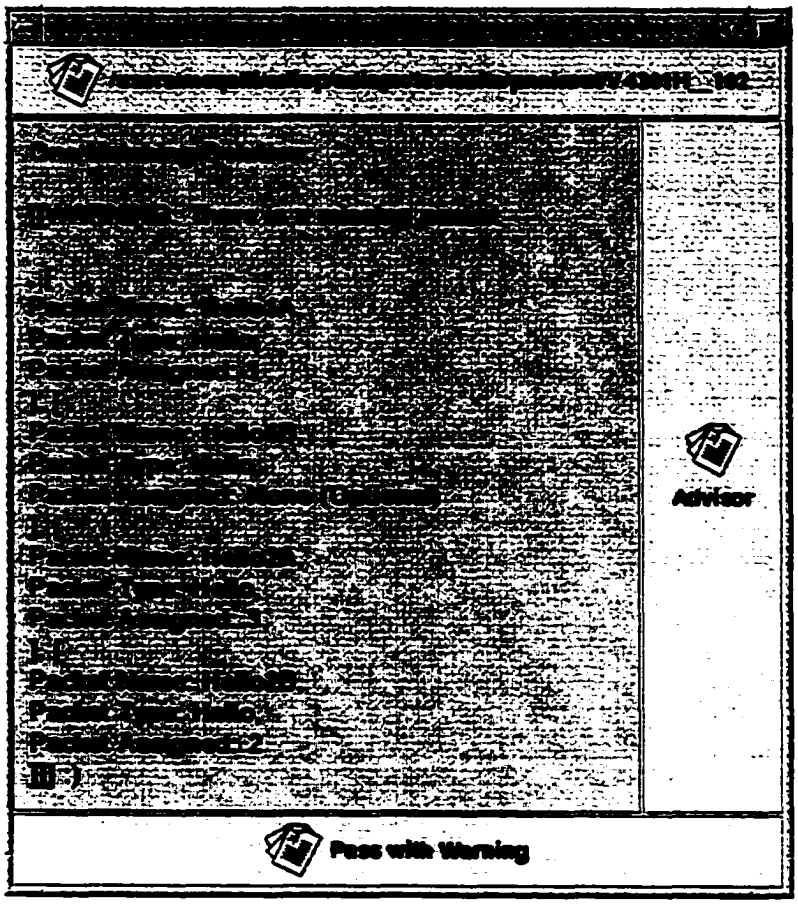

Figure 3.19: ADIOP's Result a "Pass With Warning" Test Case

no solution to the execution of this test case. This is the least useful explanation for testers because it only presents the constraints violated when search is executed. This may not provide a meaningful explanation to testers, but Chapter 4 presents one way to resolve this in some cases and help testers find a useful explanation to this test case execution.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A summary of the explanation templates used here is shown in Table 3.1.

Table 3.1: Summary of Explanation Templates

	Summary of Explanation Templates
1	There is no observed packet from the **protocolTested** protocol
2	There are no observed packets matching any type of the ones stated in the model of this test
3	There is no observed packet matching the type **packetType** as it is stated in the model of this test
4	There are fewer observed packets of type **packetType** than what is stated in the model of this test
5	One Matching Solution: [[Packet Name: **packetName**, Packet Type: **packetType**, Packet Assigned: **packetObservedNumber**], ..., [...]]
6	One Matching Solution: [WARNING: There is a missing packet, [packet Name: **packetName**, Packet Type: **packetType**, Packet Assigned: **packetObservedNumber**], [packet Name: **packetName**, Packet Type: **packetType**, Packet Assigned: None (Optional)], ..., [...]]
7	One or more of these constraints declared in the model of this test is/are violated: **violatedConstraints**

3.6 Test Case Execution

3.6.1 Automate Menus Creation

The Decoder/Diagnoser window contains menus that are generated from the directory of implemented test cases. Figure 3.14 shows this structure and Figure 3.7 shows the menus generated for the lowest level in the hierarchy.

The menus used for testing the interoperability are automatically generated. The user does not need to search for test cases to be able to run them. ADIOP stores test case objects that are built using its Test Suite Builder component under one directory according to the protocol they belong to. This makes it possible for ADIOP to extract dynamically

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the names of these test cases and construct menus that the user can use for their testing. More details of the Test Suite Builder and some of these issues were presented in Chapter 2.

3.6.2 Reports Generation

After the user runs a test case, a report is generated. There are two kinds of reports: individual test case reports and section test cases reports.

- Individual test case reports are displayed when the user runs one test case. The report contains a window that provides the user with the option to show the test case model or the result of its execution. (See examples in Figures 3.15 and 3.20.)

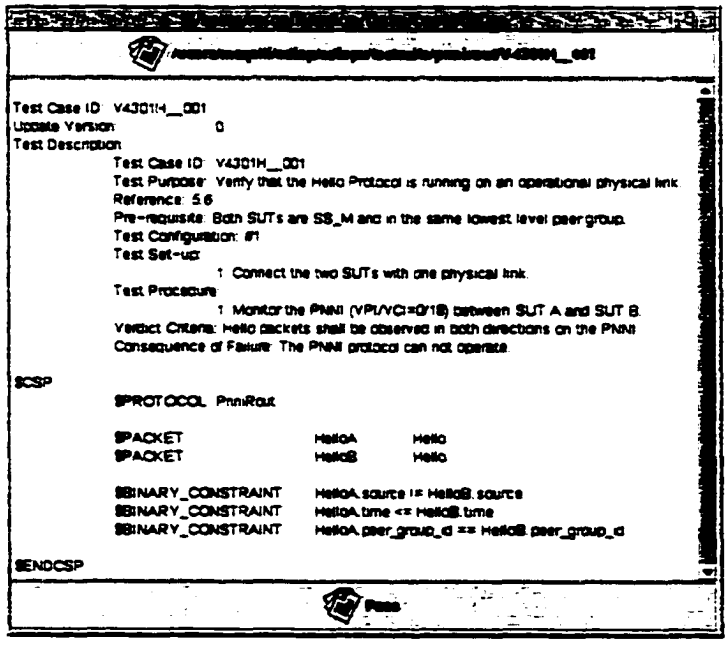

Figure 3.20: ADIOP's Result Window Showing a Test Case Model

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Section test cases reports (i.e., Test Group Reports) are generated when the user runs a batch of test cases that belong to the same section in one action. This report shows the test cases that were run, the result and the explanation of the diagnosis obtained.(See example in Figure 3.21.)

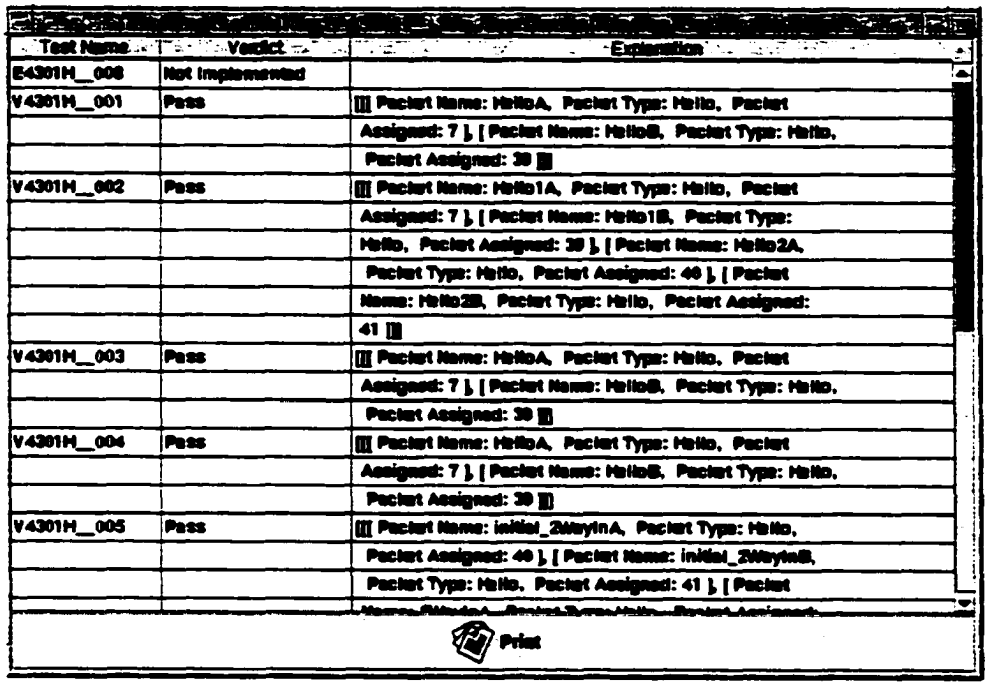

Figure 3.21: ADIOP's Test Cases Report of One Section

Both reports can be printed by the user and they provide the information that the customer needs about the interoperability testing of their equipment.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.6.3 Algorithms

We have shown earlier in this chapter how the test cases that were implemented using the ADIOP modeling component can be run and how the reports are obtained. When a test case is implemented, its corresponding object is added to the library of objects. Each one of these objects models one test case and is a descendent of the object **Model**, which includes all the algorithms for solving the CSP models. The different algorithms using search and inference are used and the result is a solution to the CSP, if it does exist. If no solution is found, it indicates that the test case fails and inference may provide an explanation to this failure. In addition, the reports generated provide an explanation of the diagnosis.

In Section 3.7, we gathered some data from multiple execution of different test cases. The comparison factor in these runs is the execution time for finding a diagnosis of the problem with and without preprocessing.

3.7 Algorithms Evaluation

3.7.1 Solvability

In this section, we present an evaluation of the algorithms presented earlier in this chapter. We used four captured data sets (observations). Three observations are for the PNNI Routing protocol and one is for the LANE protocol. We have used captures from real-world data obtained at the UNH-IOL. We run only test cases that belong to the protocol used when capturing the observations. We compare the time it takes to solve the problem using preprocessing then backtracking to the time it takes to solve the problem using backtracking

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

only.

Table 3.2 shows the results obtained for one captured data set. Each test case execution is repeated 5 times so the numbers shown in the table averages 5 runs. The test cases are taken from (PNNI-IOP 1999) using their actual names in the document. The result of the execution of each test case is shown in the "Res" column. The "MV#" column shows the number of metavariables defined in each test case. In all the test cases used in this table the maximum number of variables in metavariables is 41. The "Con#" column shows the number of constraints defined in each test case.

Table 3.2: Results of Running Test Cases on Capture capt001

			Captured Data: capt001.aa					
Test Case	**Res**	**MV #**	**Con #**	**Pre T(ms)**	**BtPre T(ms)**	**Pre+BtPre T(ms)**	**Bt T(ms)**	**Ratio (Pre+ BtPre/Bt)**
V4301H_001	Fail	2	3	20	101.2	121.2	105.6	115%
V4301H_002	Fail	4	13	7	96	103	108.6	**95%**
V4301H_003	Fail	2	7	4.4	134.8	139.2	147	**95%**
V4301H_004	Fail	2	7	4.2	33.4	37.6	51.4	**73%**
V4301H_005	Fail	4	19	4.4	74.8	79.2	71.2	111%
V4301H_006	Fail	4	9	6.2	91.2	97.4	102	**95%**
V4301H_007	Fail	4	11	4.4	90.8	95.2	104.8	**91%**
V4302H_001	Pass	2	17	21.8	6.6	28.4	3.2	888%
V4302H_002	Pass	4	25	9.4	11.4	20.8	19.2	108%
V4302H_003	Pass	2	24	4.6	4.2	8.8	4.8	183%
V4302H_004	Pass	2	3	4.4	3.2	7.6	4	190%
V4302H_006	Pass	4	13	4.6	10.4	15	14	107%
V4302H_007	Pass	4	7	4.2	44	48.2	81.6	**59%**
V4302H_008	Pass	4	7	4.4	20.6	25	15.8	158%
V4401DBS001	Fail	6	11	23	0	23	107	**21%**
V4401DBS002	Fail	7	19	6.6	0	6.6	93.2	**7%**
V4401DBS003	Fail	7	8	5.6	0	5.6	96.2	**6%**
Total				139.2	722.6	**861.8**	**1129.6**	**76%**
Total without V4302H_001				117.4	716	**833.4**	**1126.4**	**74%**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The "Pre" column shows the preprocessing time. Preprocessing uses the inferences we introduced earlier in this chapter. The "BtPre" column shows the time it takes to find a solution or none using backtracking after preprocessing. These two numbers are summed up in the next column. The "Bt" column shows the time it takes to find a solution or none using backtracking from scratch. The last column "Ratio" shows the percentage of time that a preprocessing+backtracking uses compared to backtracking alone. A ratio of less than 100% shows that time was saved using preprocessing.

The results obtained for the ratio are between 6% and 190% except for one where it is of 888%. If we exclude this one from the overall statistics, because it may bias the results, the total ratio is 74%. The total ratio for executing all the test cases is 76% when all results are included. We can see that this exclusion did not greatly affect the final result. This means that using both inference and search to perform diagnosis took only 76% of the time it took to do the same using only search for these test cases.

However, only half of the test cases presented in Table 3.3 give a clearcut improvement due to preprocessing. An open question here is why improvement was so sporadic and apparently unpredictable. The results are significant anyway because, in addition, there is an extra benefit of generating useful explanations, which is certainly worth the added cost sometimes incurred.

Table 3.3 shows the summary of the results obtained for many captures. The "Dom Size" column shows the number of packets observed in each capture. The protocol tested in each captured data is shown in the "Protocol Tested" column. If we look into the result for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

capture "capt002", excluding one test case makes the overall result jump from 232% down to 67%. We were not able to find out why we obtained such poor results for this test case.

Table 3.3: Summary of Results of Running Test Cases on Different Captures

Summary of results for different captures								
Captured Data	**Dom Size**	**Protocol Tested**	**TCs Run**	**Pre (ms)**	**BtPre (ms)**	**Pre+BtPre (ms)**	**Bt (ms)**	**Ratio(Pre+ BtPre/Bt)**
capt001	72	PnniRout	17	139.2	722.6	861.8	1129.6	**76%**
capt002	91	PnniRout	17	49	1212.6	1261.6	543.2	232%
capt002 excluding v4401dbs002	91	PnniRout	16	42.2	291.2	333.4	495.6	**67%**
PNNI	103	PnniRout	17	101.4	87.4	188.8	452.2	**42%**
Lane	77	Lane	18	285.8	174.4	460.2	998.4	**46%**

The overall results show a reduction of effort to between 42% and 76% for each captured data set, except for capt002, which yields 232%..

The test cases we ran were all related to the protocol of the data captured. If the user runs test cases on observations that do not contain any packets belonging to this protocol, the preprocessing will solve this and the explanation will be straight forward. We did not execute these test cases because we knew the protocol used for each capture, and because the users would usually make sure that the test cases being run are for a capture of the same protocol. This preprocessing using the "Protocol" value is useful when the user is not sure of what data she/he is testing, and it will save a lot of time not having to search the whole tree space for a solution. For instance, we tried one test case from the "Pnni Signaling" protocol on a capture from the "PNNI Routing" protocol and the savings were about 100%. This is explained by the fact that preprocessing time is linear and there is no

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

backtracking performed, while when backtracking alone is used a search of the whole tree of possibilities occurs because no solution exists.

3.7.2 Explanation

In addition to the reduction of time for solving the problem, preprocessing allows for better explanations in some cases with no solution. The time reduction is even greater when preprocessing yields solutions because no backtracking is necessary in this case. Three examples of this are shown in Table 3.2 with test cases V4401DBS001, V4401DBS002, and V4401DBS003. In the execution of these test cases, no packets of type "DBS" were found.

In the PNNI capture shown in Table 3.3 we had many test cases where preprocessing was enough to solve the problems because there were test cases requiring 4 packets (MetaVariables) of type "Hello" but only three were found in the captured data. In all these cases where preprocessing was enough to solve the problem, the explanation that was produced was more meaningful to the users.

Out of the 69 test cases we ran, 36 passed and 33 failed. The ones that passed produced a meaningful explanation for the user. Out of the 33 that failed, 14 were solved by preprocessing alone, thus producing a meaningful explanation for the user. In summary, 50 test cases out of 69 produced a meaningful explanation, which makes about 73% of the test cases.

For the other 27% of the test cases, which resulted in failures with no meaningful explanation, ADIOP's Advisor was used to further reduce this percentage, as discussed in Chapter 4.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.7.3 Complexity

Let s be the size of the domain of metavalues (observations), r be the reduced size of the domain of metavalues after preprocessing, and n be the number of metavariables defined in the model. In the worst case, r is equal to s.

The complexity of "Backtrack" is $O(s^n)$. Thus when search is performed alone the complexity is exponential.

The complexity of "ProtocolPreprocess" is $O(s)$, and that of "PacketTypePreprocess" is $O(n^2s)$. Thus, the complexity for performing the preprocessing is $O(n^2s)$. The complexity of search after preprocessing is $O(r^n)$ with $r \leq s$. When preprocessing solves the problem, then the complexity is of $O(n^2s)$. When it does not, then the complexity is of $O(r^n)$ with $r \leq s$. So, in the worst case, the complexity is the same $O(s^n)$ whether we use preprocessing or not. But in other cases, complexity can be reduced with successful preprocessing and may even become linear.

3.8 Evaluation Performed by Testers

An overview of the evaluation performed by testers is included in the evaluation section of Chapter 2. In this chapter, we analyze the results collected from testers for the Decoder and Diagnoser components of ADIOP. Appendix B contains the questionnaire that the testers used for the evaluation of these components. Testers were also provided with an ADIOP User Manual (Appendix C).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.8.1 Decoder

The evaluation of this component includes one part for the practical use of the Decoder and another part for a survey of the Decoder's performance. Both parts were performed by 3 testers. 86% of the data sets used in this evaluation were predefined in the questionnaire. We opted for predefined data sets to be able to test different types of decoders and to be able to combine the results obtained from testers from the same set of data.

Analysis of Data Decoding

Each tester used ADIOP for decoding 4 to 5 observations (captured data). These observations are all obtained from 4 different analyzers widely used in the lab. One tester stated in the questionnaire that: "Definitions for 'a particular' Analyzer captures would be useful." Data captured using this analyzer was not used in this evaluation since the appropriate decoder for it was not implemented in ADIOP. Some of the decoders were implemented in ADIOP by other testers in the lab and some by myself. One tester used 4 predefined observations for the 4 different analyzers. The other 2 testers used an additional observation they captured on their own. This yields a total of 14 (4*1+5*2) decodes performed in this evaluation.

For all predefined observations, the captured data were successfully and correctly decoded by all three testers using ADIOP for the protocol being tested and using the correct type of decoder.

As for data that was captured and used by two testers, ADIOP recognized the appropri-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ate decoder type for this data but did not decode it correctly. The problem might be that the data was not saved in the proper format from the analyzer so that ADIOP could decode it. Another explanation is that the ADIOP decoder for this particular type of captures has bugs in its implementation.

Decoder vs. Analyzers Analysis

The testers were asked to compare the decodes obtained from ADIOP vs. those obtained from each analyzer. For the analyzers, the complete decodes were obtained 6 out of 14 times, and they include all that is needed for the protocol being tested. As for ADIOP, the same was achieved 12 out 14 times. Even though the analyzers capture all the packets between devices, the decodes they provide to testers, for example through a GUI, may not be complete. ADIOP uses the Hexadecimal format and provides more complete decodes. This was confirmed by the survey results obtained from testers. ADIOP also provides a functionality that allow testers to open the decodes of many packets, each in a different window for an easy comparison of their contents.

We also wanted to find out whether the decodes contained all the information needed. The testers were asked about whether the decodes lack information that might be needed by the protocol being tested but should not affect the diagnosis, or may affect the diagnosis, or lack all the information needed by the protocol being tested. The analyzers did not lack any such information. For ADIOP, 12 out of 14 times it did not lack any kind of information, and in 2 out of 14 it lacked all three kinds of information. This was because the data could not be decoded for these, as stated earlier in this section.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As for whether the decodes obtained are usable, the testers' response shows that the analyzers provide decodes that are usable only 8 out of 14 times, compared to 12 out of 14 for ADIOP. Again ADIOP failed for the data that could not be decoded.

We can conclude that, on average, ADIOP outperformed the analyzers in providing the complete decodes and all that was needed for the protocol being tested, as well as in providing decodes that are usable. However, ADIOP performed worse than the analyzers when captures were not decoded. As explained earlier, this was caused by the fact that two captures were not decoded at all by ADIOP so it was not possible to get any information out of these captures.

The Decoder Survey Analysis

Each tester also answered questions in a survey on rating the Decoder component of ADIOP. The survey contained 8 questions. The questionnaire was built based on a likert scale (Likert 1932) that ranges from 1 to 5, with 5 being "Strongly Agree" and 1 being "Strongly Disagree". More information on the likert scale used here can be found in the evaluation section of Chapter 2.

All ADIOP's Decoder attributes received the same marks from the respondents (testers) for an average score of 4.13. For each attribute, the testers answered all the questions with "Strongly Agree" for 12 decodes and "Strongly Disagree" for 2 decodes. This shows that the responses were all affected by whether ADIOP was able to decode the captured data.

The ADIOP's Decoder attributes used in this survey are the following:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- ADIOP Decoder includes decodes of all the packets needed for the protocol being tested.
- Decodes given are correct for the packets needed for the protocol being tested.
- Decodes given are complete for the packets needed for the protocol being tested.
- ADIOP Decoder is friendly: this had three subquestions on whether it had an easy GUI interaction, it was easy to read the decoded information, and it was easy to compare two or more decoded packets. This shows that the ADIOP Decoder is easy to use by testers, and makes it easy to read and compare decoded data.
- ADIOP Decoder is a useful tool for the lab.
- Fast - the data is decoded in a reasonable amount of time.

The overall average score of ADIOP Decoder was 4.13. Again this score was mainly affected by the two captures that ADIOP failed to decode. This can be avoided by debugging the problem of the decoder that caused these failures.

Notwithstanding this, we can say that the Decoder received a score of 5 on all the attributes for the data that was decoded and this is a very positive result. This is also supported by the tester with the longest experience at the UNH-IOL, who stated in the general comments section of the questionnaire that: "**ADIOP is the 1st tool we have had that can reliably analyze decodes** and produce reports".

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.8.2 Diagnoser

The evaluation of this component includes one part for the practical use of the Diagnoser where the testers compare manual interoperability testing with the automated testing using ADIOP. The second part includes a survey of the Diagnoser's performance. Both parts were performed by 3 testers (see questionnaire in Appendix B).

Manual vs. ADIOP Automated Testing

The testers were given 36 predefined data sets in addition to some optional ones to evaluate. A total of 11 data sets were received. Only 6 out of the 36 predefined data sets and 1 from the optional ones were received from testers. In addition, there were 3 responses from non predefined data sets and one response where the data set was not clearly stated. 2 out of the 11 responses did not include any information about the manual testing and thus cannot be used. One of these 2 was because the tester was unable to decode the trace file using ADIOP. Another 2 responses have conflicting information between test cases executed and ADIOP results and cannot be used.

As a result only 7 data sets could be used in this evaluation, and of these 4 are predefined, 1 optional, and 2 not predefined. If we consider the optional one as predefined and add it to the set of the total predefined data sets, then 5 out of 7 data sets, that is 71%, used in this evaluation were predefined. And the predefined data sets used by testers represent only about 14% (5 out of 37) of all the predefined data sets included in the questionnaire. These were test cases from the PNNI Routing and LANE protocols. We opted for more

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

predefined data sets to be able to test different test cases from different protocols using different decoder (analyzer) types, and to be able to test and evaluate different situations of pass and fail results that ADIOP can handle. This makes it also possible to combine the results obtained from testers about the same set of data tested.

These data sets can be regrouped into 5 groups of different data used. Two test cases were performed by 2 testers and 3 by one tester (7=2*2+3*1). Since the results obtained from 2 testers about the same test case were similar in terms of result status and timing ratios, we include the average of these results into one group. Table 3.4 presents the results obtained from the questionnaire. All the information in this table was provided by the testers. **MV <- mv** in the "Explanation" column of this table means that, in the solution found, the observed packet mv is assigned to the metavariable MV of the test case CSP model.

The "Data sets" column contains the name of the file containing the captured data, the protocol, and the test case being used. All of the captured data were obtained using the same analyzer, namely, "Network General". The "Actors" column states whether a test case was executed manually by testers or using ADIOP. The "Rslt" column indicates the results of a test case execution. The "Explanation" column states the explanation that was provided as the outcome of a test case execution and diagnosis. The "Time" column states the time in seconds it took to execute one test case. The "# TCs" column shows the number of test cases in the section in which this test case belongs. The"Rep" column states the time in seconds it took to generate a report of the execution of all test cases in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 3.4: Summary of results for manual vs. ADIOP testing

Summary of results for manual vs. ADIOP testing

Data sets	Actors	1 Test Case			TCs Section		Pre
		Rslt	Explanation	Time (s)	# TCs	Rep (s)	
capt002 PnniRout V4301H_001	2 Testers	Pass	Hello packets observed on link	31	8	1350	Yes
	ADIOP	Pass	HelloA <- 7 HelloB <- 39 (Figure 3.15)	5	8	42	
PNNI PnniRout V4301H_002	2 Testers	Pass	-	300	9	1800	Yes
	ADIOP	Fail	Less Hello Packets (Figure 3.18)	5	9	60	
PNNI PnniRout V4302H_001	1 Tester	Pass	Hello packets sent in both directions	25	11	900	No
	ADIOP	Pass	HelloA <- 41 HelloB <- 77	5	11	46	
capt003 PnniRout V4301H_004	1 Tester	Pass	-	20	8	900	No
	ADIOP	Pass	HelloA <- 1 HelloB <- 8	5	8	42	
LANE Lane V100_LEC_Configure_Request_003	1 Tester	Pass	LAN Destination field is 16 bytes	15	4	900	Yes (Optional)
	ADIOP	Pass	Frame1 <- 36	5	4	40	

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

this section. The "Pre" column shows whether the data set was predefined or not in the questionnaire.

To avoid contamination effects due to learning, the first thing the testers do is the manual testing and then they use ADIOP. The timing and results of ADIOP will not depend on the knowledge the testers have of the data. If the testers use ADIOP first, which is not the case, they may learn things about the problems being diagnosed and their results from manual testing might be affected and thus biased.

The number of test cases in a section and the predefinition of data sets did not have any influence on the results obtained in this table and thus will not be included in the analysis that follows.

Comparison of Results

In one out of the five data sets, the testers' results ("Rslt" column) and ADIOP results do not match. There was no explanation provided by the testers about this test case. As for ADIOP, the explanation provided is similar to the one shown in Figure 3.18 where ADIOP finds that there are fewer packets of type Hello than what is declared in the model of the test case used. To analyze this mismatch, we have checked the description of the test case V4301H_002 in which it is stated that the verdict criteria is to "Observe that the value in the version field in the next exchanged Hello packets is the same in both directions ...". This means that there are at least two initial Hello packets exchanged between the two devices tested, and there are two more Hello packets (one from each device) with the version field value set as specified in the verdict criteria. By looking into the PNNI capture, we found

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that there are only 3 Hello packets out of a total of 103 packets in this capture. This means that the test case should fail.

The explanation to this can be that the testers checked the last 2 Hello packets of this capture for the information on the version field and that is why it passed. But this may not be always the case with other captures. And, if the test case is modeled as it is defined in (PNNI-IOP 1999), then it must fail.

For all the other 4 data sets, the results obtained were the same between testers and ADIOP. We can say that 80% of the time ADIOP and the testers reported the same results. And ADIOP successfully reported the correct diagnosis 100% of the time.

Comparison of Explanations

The testers provided an explanation in 3 out of 5 data sets. ADIOP provided an explanation for all data sets. The content of this explanation was different between the testers and ADIOP. We found that the explanations provided by testers is similar to the "Test Purpose" defined within the test case used (see Appendix A). In ADIOP, however, the explanation states which packets in the captured data were used to verify this test case when it passes, or what is the cause of the problem encountered when it fails.

With ADIOP, it is still possible for the tester to check what is the "Test Purpose" of this test case just by clicking one button on the same window showing the result of its execution (see example in Figure 3.20).

In summary, ADIOP has provided a more detailed explanation on the diagnosis in all cases while, with manual testing, either no explanation or only a general explanation (i.e.,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

"Test Purpose" of a test case) was provided. Other testers cannot check the validity of this explanation unless they go through the whole capture, while with ADIOP's explanation, any tester can easily determine and check manually why a test case passed or failed.

We can still make the explanation of ADIOP much more useful and easy to understand by the tester by pasting the "Test Purpose" of a test case with the result provided.

Comparison of the Execution Times for 1 Test Case

As expected, ADIOP was faster in providing the result of a test case execution. There was one value that can be considered as an outlier compared to the other values and that is 300 seconds to perform manual testing for V4301H_002. However, this value seems to be more realistic than the others for a tester to go through a capture and get the result. If this value is included, then testers have taken on average 78 seconds to perform 1 test case. If the outlier is excluded, then the average is 22.75 seconds. ADIOP took 5 seconds to perform the same task. This means that ADIOP took 22% of the time it took a tester to diagnose one test case, a savings of 78% of the time. In other words, it will take the same amount of time for a tester to diagnose 1 test case as it takes for ADIOP to diagnose more than 4. Thus, ADIOP is more than four times faster than the UNH-IOL testers in diagnosing these test cases.

Comparison of the Times to Generate a Report for a Section of Test Cases

The time shown for ADIOP in the "Rep" column comprises the time to generate and print a report for the execution results of test cases of one section. For the testers, it represents

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the time to write a similar report. All the sections used in this evaluation included between 4 and 11 test cases and on average 8 test cases.

For ADIOP, it took an average time of 46 seconds to generate a report for one section. For the testers, it took an average of 1179 seconds to create and write a similar report.

This means that ADIOP took 3.9 % of the time it took a tester to generate a report, a saving of 96.1%. In other words, it took the same amount of time for a tester to generate a report for one section as it takes for ADIOP to generate more than 25 reports.

As for the quality of the report, all the testers agree that the reports generated by ADIOP are useful for the lab. This will be further discussed in the Diagnoser survey analysis section.

The tester with the longest experience at the UNH-IOL stated, when asked about the usefulness of ADIOP for the lab, that: "ADIOP would allow for faster completion of tests and vendors could obtain a report while still in the lab." Another tester, when asked about how much better is ADIOP than what we had before, stated that: "ADIOP is the first test tool that can generate reports."

The Diagnoser Survey Analysis

Each tester also answered questions from a survey on rating the Diagnoser component of ADIOP. The survey contained 19 questions. The questionnaire was built based on a likert scale (Likert 1932), as discussed in Chapter 2.

The ADIOP Diagnoser attributes that received the highest marks according to the respondents were:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- The Diagnoser is friendly got an average score of 4.33. This contained 4 subquestions: easy to execute test cases individually scored 5, easy to execute the test cases of one section scored 5, easy GUI interaction scored 4.33 and easy to read the diagnosis scored 3. This means that diagnosing test cases is made easy for the tester with a friendly interface.

- The Diagnoser is flexible scored on average 4.33. This contained 2 subquestions. It is possible to diagnose data from different analyzers using ADIOP scored 4.66, and it is possible to diagnose data for different protocols scored 4. This shows that the testers agree that ADIOP successfully diagnoses data captured from different analyzers and for different types of protocols.

- The Diagnoser is fast, meaning the data is diagnosed in a reasonable amount of time scored 4.33. The analysis of this was detailed in a previous section where we compared ADIOP to manual testing.

Three other attributes scored an average of 4 (i.e.. Agree):

- The reports generated by ADIOP are useful for the lab.

- Re-usability - the storage of the diagnosis obtained is useful.

- ADIOP diagnoser is a useful tool for the lab.

As for the quality of the explanations provided by ADIOP, the following was gathered from the survey:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- "The explanation given by ADIOP is correct" scored 4.33 for test cases that pass and 4 for test cases that fail.
- "The explanation given by ADIOP is complete" scored 3 for test cases that pass and 3 for test cases that fail.
- "The explanation given by ADIOP is useful" scored 2.66 for test cases that pass and 3.33 for test cases that fail.

This shows that the testers agree that the explanation provided by the Diagnoser is correct. However, they were undecided on whether this explanation is complete or not. The usefulness of the explanation was also undecided on average, and that there is more usefulness of the explanation for the testers when a test case fails than when it passes. This can be explained by the fact that ADIOP provides explanations that do not include the "Test Purpose" statement of the test case description, which represents for the testers the explanation they expect. This can be seen as a useful feedback from testers as to what to include in the diagnosis of a test case. ADIOP includes this information as part of the test case description, which can be viewed by the testers from the result's window (e.g., Figure 3.15).

To resolve this issue we can either state how to access this information in the ADIOP user manual (Appendix C) or add the "Test Purpose" statement to the result. This last resolution means that there will be redundant information accessible from the same window, but will be considered by the testers as more useful for a better understanding of the testing results.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In case of failures, the explanation provided by ADIOP can be very useful when inference is successful and less useful when it is not. That could explain why testers agree more on the usefulness of the explanation when test cases fail. The testers were also undecided about whether ADIOP's Diagnoser generates the correct result (Pass/Fail).

Again one positive point in this evaluation is that the individual scores of testers were negative in only two statements. The usefulness of ADIOP's explanation when test cases pass scored 2 (i.e., Disagree) by one tester and scored 2.66 on average by all testers. There was also a score of 2 by one tester on whether it is possible to diagnose data for different protocols but the other two testers responded with a score of 5 (i.e., Strongly agree).

The overall average score of ADIOP's Diagnoser was at about 3.86.

3.8.3 ADIOP's General Survey Analysis

Each tester also answered questions of a general survey on rating ADIOP including Modeling, Decoding, and Diagnosing components. This survey did not include the debugging component of ADIOP. Model debugging is the subject of Chapter 4. The survey contained 24 questions (Appendix B). The questionnaire was built based on a likert scale (Likert 1932), as discussed in Chapter 2.

The ADIOP attributes that received the highest marks according to the testers were:

- Re-usability scored 4.67. It states that ADIOP provides a good way to store test cases and re-use them later.

- "Fast" scored 4.67. It states that ADIOP provides solutions in a reasonable amount

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of time.

Other statement that scored 4 (i.e., Agree) or more and thus were in average agreed upon by the testers are:

- Testers can find problems quickly using ADIOP compared to manual diagnosis scored 4.33.
- ADIOP is friendly scored 4.33 as well. This contained four subquestions all of which scored 4.33 in average. These are: an easy GUI interaction in ADIOP, easy to learn ADIOP, easy to use ADIOP, and easy to find what we are looking for in ADIOP.
- Test cases can be accessible and executed by anyone without much knowledge on how they were created scored 4.33.
- Testers expect ADIOP to be even more useful for large data sets with hundred of frames (packets) scored 4.33.
- Testers can automate interoperability test cases using ADIOP scored 4.
- ADIOP saves time for testers scored 4.
- Testers recommend using ADIOP in the lab wherever applicable scored 4.

Other statements scored more than 3, and thus were still toward the agreement side of the likert scale:

- Testers can diagnose more interoperability problems using ADIOP scored 3.67.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Testers know more about protocols when using ADIOP scored 3.67.
- ADIOP is a useful tool for the lab scored 3.67.
- Testers prefer to work with ADIOP rather than without it for interoperability testing scored 3.67.
- ADIOP is flexible scored 3.58 in average. This contained 4 subquestions, namely, it is possible to use many decodes on different windows at the same time, it is possible to perform many diagnoses on different windows at the same time, it is possible to create many test cases on different windows at the same time, and it is possible to do all these tasks with no problem of conflicts in the application. Three of these subquestions scored 3.67, and one scored 3.33.
- It is better to remember how old problems are diagnosed using ADIOP than manually scored 3.33.
- Testers know more about interoperability testing when using ADIOP scored 3.33.
- ADIOP will help testers do more interesting work scored 3.33.

The statement that scored the lowest with a value of 3 and thus it was undecided by testers whether they agree or not is that: "The explanation generated by ADIOP is useful."

The overall average score of ADIOP was at about 3.90.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.8.4 Limitations

This evaluation included only three testers and thus we cannot generalize the results obtained because the sample is very small and thus no significant statistical inference can be used. But this was the extent of the help we could get from the UNH-IOL to perform this evaluation due to the limited number of available testers and the tight schedule in the lab. However, the fact that all three testers consistently rated the ADIOP components high on their different attributes strengthens the results even though the sample is small.

For the decoder, the captured data used were predefined. The purpose of this was to obtain results from different decoders and to be able to compare results obtained from different testers for the same set of data. For the Diagnoser, however, some data sets were predefined and others were not.

The data sets used were few in number and that is because of time constraints in the lab. We would have obtained a better analysis with more data sets involving all different decoders, using different protocols, and for different testing situations. However, we have used ADIOP's Decoder to decode more than 20 captures of 4 different analyzers. More than 10 of these captures were used in the evaluation of the other components of ADIOP. We have also used the Diagnoser for testing more than 60 test cases. Some of the results obtained with these test cases can be found in Section 3.7 on the evaluation of algorithms and in Table 3.3.

One tester stated that "ADIOP in general is a great tool. But in the lab we would have to implement entire test suites before it could be used which takes too much time." It is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

expected that some time will be spent to create test cases and implement test suites, but this is only needed once for each test case. After that the same test case can be run as many times as the tester wants with no need to do manual testing or create reports manually. So, the savings with the ADIOP tool are clearer when we think of the long term impact on testing.

He also stated that "ADIOP can be made better by implementing more test suites," and that "The ADIOP tool works well but needs to be implemented more completely before it proves useful. The concept and functionality are great. Once fully implemented, ADIOP could prove to be extremely useful." This is true and so far we have implemented few test suites to evaluate ADIOP's performance. And the first feedback on this prototype has been positive and suggests that ADIOP can be used in the lab but needs further improvements.

Another tester stated that "ADIOP can be made better and more useful if it is made of a client/server model so that test cases can be accessed from any workstation." This can be investigated further as needed by testers, and the fact that ADIOP was implemented using Java makes it easy to upgrade it to a client/server version either by using a CORBA interface or some other architecture. In any case, it is feasible if deemed to be useful.

He also stated that: "on various tasks ADIOP would crash." This is true because there was no quality assurance performed on ADIOP as we did not have enough resources for that and the main goal was to support the ideas presented in this dissertation.

Another tester stated that: "ADIOP can be made better and more useful if it allows test cases to be generated for LANE and Q2931. It was unable to generate test cases for protocols

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

other than PNNI", and that "generating some test cases for PNNI caused ADIOP to crash." As stated earlier, there are some test cases already implemented using these protocols but it is possible that not all the functionality of the protocol is implemented and thus it was not possible for testers to create some test cases. A complete implementation of the different protocols would allow for better results as was proven through the PNNI Routing protocol, where testers had fewer problems in creating test cases and executing them.

3.8.5 Conclusion

As stated at the beginning of this chapter, our motivation for automating the diagnosis of interoperability testing is to save time, reduce repetitive testing, store and reuse knowledge, automate reports generation, and in general to make testing easier and more efficient. Many of the claims we made about modeling and diagnosing using ADIOP were confirmed by the testers. Testers agreed that ADIOP is user-friendly, flexible, fast, saves time, provides reusable diagnosis and useful reports, that it is a useful tool for the lab, and that it helps testers know more about the protocols. They also agreed that the knowledge required to run test cases is minimal, that testers can automate test cases using ADIOP, and that more interoperability problems can be diagnosed using ADIOP. ADIOP was also recommended by testers to be used in the lab, and that testers prefer to work with ADIOP than without it.

The scores obtained by all the components also show that ADIOP is a tool testers would want to use because of all its attributes presented earlier. We evaluated each component by itself as well as the overall performance of three ADIOP components. In summary,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the Modeling component presented in Chapter 2 scored 4.44, the Decoder scored 4.13, and the Diagnoser scored 3.86, All these three ADIOP components together scored 3.90 in the ADIOP's general survey. All of these scored above or close to Agree (i.e., score of 4) which supports our earlier statement about ADIOP. This also shows that the components' behavior matches their intended functionality.

Some of the positive statements provided by the testers in the questionnaire are the following:

- One tester stated that: "ADIOP would allow for faster completion of tests and vendors could obtain a report while still in the lab." and that: "ADIOP is the 1st test tool we have had that can reliably analyze decodes and produce reports."
- Another tester stated that: "ADIOP is the first test tool that can generate reports."
- The third tester stated that: " Testing PNNI using the ATM Forum's Test Suite is much easier to complete using ADIOP.", and that: "Overall, a very friendly interface. It will make PNNI testing much more efficient and easier."

The above statements also confirm the thesis we support in that CSP modeling and methods are suitable for many applications including interoperability testing. In addition, the testers in the general survey on the overall performance of ADIOP agree that it enhances the way interoperability testing is done, and that they recommend it as a useful tool for the lab. They also stated that ADIOP is a better tool than what testers had before.

There are also some areas of improvements that were pinpointed by the testers, some of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

which were mentioned in the limitations section. There was also the explanation provided by the Diagnoser that testers found not to be very useful and we explained how it is possible with a small modification in ADIOP to solve this issue.

The outcome of this evaluation confirms our claims about ADIOP, its success and its contribution. This includes the statements that ADIOP Diagnoser outperforms manual diagnosis and other tools, and that ADIOP Test Suite Builder outperforms other tools. This includes also the CSP modeling language capabilities and ease-of-use, and the ability to generate correct and useful explanations.

3.9 Related Work

- (Abu-Hakima 1993) argues that causal explanations in diagnostic tasks are more easily obtained using fault-based or failure-driven reasoning versus model-based reasoning. Fault-based or failure-driven diagnosis is more of a contextual task and can more easily be used to support user interaction through explanation than model-based diagnosis. The diagnostic hierarchy (classification tree) branches into more specific hypotheses that explain the more detailed symptoms provided by the user. As the system is used, the diagnostic hierarchy forms the basis for a dynamically generated explanation hierarchy that holds both successful and failed branches of the reasoning tree. Her paper elaborates on explanations in RATIONALE, a fault-based diagnostic system. RATIONALE is a workstation diagnosis system that establishes context in reasoning so that it may support the user with sophisticated explanations of diagnoses that help

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

justify system behavior and clarify reasoning (Abu-Hakima 1988). It uses template-based explanations. Templates connect pieces of text to variables that are instantiated from the knowledge in the system. This allows explanation templates to be domain independent. Templates also simplify the task of generating dynamic explanations according to the current context.

(Abu-Hakima 1994) states that fault-based reasoning (FBR) is used in many diagnostic systems. Knowledge in FBR is largely based on maintenance manuals and interviews with experts intended to capture heuristic knowledge about the maintenance and repair of a device or process. In the same paper, Abu-Hakima also states that model-based reasoning (MBR) for diagnosis concentrates on reasoning about the expected and correct functioning of a device. A device is modeled based on its components and their expected behavior (Hamscher & Struss 1990). (Abu-Hakima 1994) presents the DR (Diagnostic Remodeler) algorithm for automating model acquisition for diagnosis. She states that humans use failure-driven reasoning for successful device diagnosis and repair, argues that MBR for diagnosis can detect novel faults but can lead to a combinatorial explosion in producing a diagnosis, and FBR uses the faults in the functioning of a device rather than its actual behavior but cannot detect novel faults. The DR algorithm was implemented to combine model-based diagnosis (MBD) and fault-based diagnosis (FBD) by automating the generation of a model of a device by the re-use of its fault knowledge. This implies the automated generation of MBR knowledge from FBR knowledge.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In the MBR type of diagnosis, once a device model is stabilized then a device's observed behavior can be predicted from the model. If a discrepancy in behavior is detected then possible fault candidates are generated based on assumptions that describe correct model behavior. In MBR, the definition of models range from causal models to numerical simulations. In this work, a device's behavior is modeled and used for diagnosis. In interoperability testing, we need to model and diagnose the interaction between devices. Thus, modeling a device's behavior is not suitable for the domain of interoperability testing. The application of each approach is different according to the goal to be achieved in diagnosis.

- One approach to model-based diagnosis has taken diagnosis to be a constraint satisfaction problem (CSP) (Fattah & Dechter 1992). (Sabin *et al.* 1994) implement a refinement of this approach using Partial Constraint Satisfaction Problem (PCSP) to diagnose distributed software systems. PCSPs were introduced for applications that settle for partial solutions that leave some of the constraints unsatisfied (Freuder & Wallace 1992). Regarding components as constraints, and faulty components as failed constraints, minimal diagnoses naturally correspond to PCSP solutions that leave minimal sets of constraints unsatisfied.

  The same technique is applied and extended to the diagnosis of some configuration problems involving FTP and DNS network software (Sabin *et al.* 1995a), where diagnosis is considered as a Dynamic Partial Constraint Satisfaction Problem (DPCSP). The finite-state machine specification of a protocol is translated to a standard CSP

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

representation and configuration tasks are modeled as dynamic CSPs (DCSP) (Sabin *et al.* 1995b). We take Diagnosis a step further by fixing CSP models and improving explanation of diagnosis using CBR. The integration of CBR and CSP improves on the CSP modeling by debugging and updating models and improves on explanations generated by CSP.

- (Leckie 1995) presents an application designed to automate the tasks of performance monitoring and fault diagnosis of transmission equipment for a special purpose telephone network. The large volume of data collected daily made it impossible for the experts to check every aspect of the network. They developed a connectionist data filter that could quickly detect abnormalities in large volumes of raw data. Then, they introduced a rule-based expert system to perform more detailed diagnosis based on the output of the data filter. In interoperability testing, there are many test cases to model, and they differ from each other in terms of packets and types of constraints to be checked. Also, we need to have the flexibility to implement test cases for different protocols. A rule-based system is not adequate for this type of application because it does not provide this flexibility. We have shown in this dissertation how the integration of CSP and CBR allow this flexibility of modeling and diagnosing in addition to model debugging.

- (Novak *et al.* 1993) extend the DFT (Design-for-Test) methodology by using CLP(R) to model analog circuits and by a model-based diagnosis approach to implement a diagnostic algorithm. CLP(R) is a constraint logic programming language which

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

combines symbolic and numeric computation.

3.10 Summary

In this chapter, we discussed how we use CSP models to diagnose interoperability problems. We showed how the use of CSP for modeling allows us to take advantage of methods and algorithms that already exist for solving CSPs. These algorithms are adapted to take advantage of the specialized problem domain structure. This provides a better diagnosis of the interoperability problems including an accurate and concise explanation of the testing performed.

We gave some definitions related to diagnosis, we presented the diagnosis process for interoperability problems, and the algorithms and methods used for diagnosis, including search and inference. Section 3.5 is dedicated to explanation and explanation templates. We then discussed test case execution including menus and reports generation.

An evaluation of the performance of the different algorithms used was then presented and showed that some of the specialized algorithms can lead to better results in terms of time and explanation. Then an evaluation performed by testers supported some of the claims we made about ADIOP such as its usefulness, friendliness, flexibility, time saving, re-usability. Testers also recommended the use of ADIOP in the lab. These attributes of ADIOP were agreed upon by testers in the evaluation and most of them scored high. There are also some areas of improvements that were pinpointed by the testers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4

Case-Based Reasoning and Model Debugging

In the previous chapters, we discussed how interoperability testing is performed in ADIOP using CSP for modeling test cases and for diagnosing problems. We showed that the diagnosis of interoperability problems using search and inference generates useful explanations in most cases. However, ADIOP's search and inference methods may fail to generate useful explanations for some problems/failures. In this case, we suggest having a case base where these problems are stored, along with an explanation of the solution provided by the experts. Each problem and its solution constitute one case. These cases can then be reused in the future to help the testers with similar problems.

Some of these cases originate from the incompleteness and incorrectness of the CSP model. In this case, the ADIOP system provides the functionality to store the case in addition to statements for updating the CSP model and making it complete and correct. These cases can then be used in the future to help with similar problems and update other incomplete or incorrect models.

Other cases originate from interoperability problems with a non-useful, incorrect or incomplete explanation. These cases are reused to provide better, correct, and complete

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

explanation for future problems.

4.1 Motivations and Contributions

Some of the motivating issues for the work presented in this chapter are:

- Learning from previous cases
- Debugging: Compensate for incompleteness and incorrectness
- User Interaction and Advising

In summary, we want to have a system that detects and debugs inconsistencies in the CSP model built by the user. These inconsistencies may originate from different sources. They may be inconsistencies in the protocol specification document, in the test suite derived from it, or in the modeling of these tests by the user. Independently of the origin of these inconsistencies, we want to provide a way of detecting and resolving them.

This leads to another important motivation, and that is to provide a general framework for model acquisition and debugging. The idea is to develop automated ways to compensate for incompleteness and incorrectness of models. This can be very useful for debugging models. It includes detecting inconsistencies and resolving them by either storing the information about them for later use or by updating the model. Part of this motivation is to find a taxonomy of these inconsistencies. This provides a formal way for addressing different cases of incompleteness and incorrectness.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We used both a bottom-up approach, where we collected the examples from the application as a starting point to come up with part of this taxonomy, and a top-down approach, where we started looking at the concept of CSP modeling and how incompleteness and incorrectness can be manifested in these models.

Our approach is to debug the model when some of these inconsistencies are present. The model is debugged through user interaction, and CBR is used as the learning tool.

The integration of CSP and CBR in the way presented in this dissertation is novel. CBR is not part of the CSP solving mechanism but rather is an addition to it. So there is less interaction between the two than in other integrations. CBR is used to remember old cases when a similar problem is encountered. Cases that represent incompleteness and incorrectness in the model are stored along with the ways these are solved.

Other types of cases are also stored that represent failures and explanation of these failures, and can be recalled if a test case fails. They may not be related to the incompleteness or incorrectness of the CSP models.

Another important contribution is the use of CBR to debug and eventually update the model. To our knowledge CBR-CSP integrations that were researched and implemented do not include this kind of integration. We wrote a survey on CBR-CSP integrations in (Sqalli, Purvis, & Freuder 1999), where more information can be found about these. For some domains, updating the model is not an option because that may add inconsistencies to the model and make it inadequate for use. In contrast, this is a useful task in our application because it allows us to obtain more robust models.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In our approach CBR adds to CSP in that it allows the use of CBR as an addition and a learning tool with CSP, and it also provides a good module for updating and debugging the CSP model.

4.2 Advantages

The claimed advantages of this approach are as follows:

- The modeling of the protocol specifications as a CSP is easier to start with than gathering a set of cases. If we use only CBR then we will need to store many cases. Instead, we choose to reduce the number of cases by using the CSP model. The CSP model represents the core of the system, and CBR adds the missing elements in this model.

- There is no need for CBR use at first but only after CSP fails. The CSP model is easier to use at first because of its generality.

- CSP is enhanced by the CBR results. The effectiveness of the model increases as more problems are solved, because the CSP model gets updated as needed.

- CSP is used to represent the information on updating models in cases. This assures uniformity of representation between the CSP models and the updating process.

- The system is open to new expertise and easily updated. The expert can add cases as needed by the system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.3 Incompleteness and Incorrectness in the CSP model

A model is *incomplete* if it is missing some knowledge about the system's behavior. This means that this incomplete model suffices to answer questions not involving the missing knowledge. Otherwise, the behavior will be unpredictable. A model is *incorrect* if it represents wrong knowledge. This model will be sufficient for and will answer correctly questions that do not involve the incorrect knowledge. Otherwise, the answer given might be wrong. The problem in all these scenarios is that it is hard to know where the missing or the incorrect information is, so it may not be possible to tell whether the answer provided by these models is correct. An example of an incomplete model is a CSP problem where a constraint or a variable is missing. An example of an incorrect model is a CSP problem where a constraint is incorrect.

A model can be incomplete or incorrect because:

- The interactions with the external world are unknown,
- The modeling is done by a human being, who may miss or interpret incorrectly some information.

An assumption that is frequently made is that the protocol specification and the test suite specification are correct and consistent. However, both of these types of specifications may be incomplete, inconsistent, ambiguous, or incorrect. This may happen because of the following:

- A statement in the specification may be incorrect because of a human error.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Statements of one section may be inconsistent with statements in another one in the protocol specification.

- Statements may be interpreted incorrectly when developing a test suite.

In addition, if many protocols are running at the same time between two devices, they may cause the wrong behavior of one protocol due to the external interactions with the other. Specifications can be incomplete because they represent the behavior of a specific domain application and may not include all the interactions with the external world.

We are interested in contributing and evaluating in terms of the larger CSP domain by acquiring a taxonomy of types of model incompleteness and incorrectness. and associated ways to identify and fix them.

4.4 Taxonomy of Types of Model Incompleteness and Incorrectness

The first step in acquiring this taxonomy involves collecting some of these inconsistencies in the domain of interoperability testing of networking protocols. We use a bottom up approach to collect some examples, gathered in the lab, from the real world application of interoperability testing to different protocols in ATM, and these are shown in the next section. This allows us to gather types of incompleteness and incorrectness found in this domain. From these, we identify the first part of this taxonomy.

The second step consists of a top down approach to derive some more model incom-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

pleteness and incorrectness that my occur in this domain. This allows us to analyze the CSP models in a broader sense and identify types of incompleteness and incorrectness that may appear in other situations not included in the first part.

The third step consists of debugging these models by associating different procedures for fixing each problem of incompleteness or incorrectness. We provide more details on one type of these problems that was implemented and tested, and more empirical results on it are presented in this dissertation.

We used only one type of what will be presented in this section in the evaluation of the Advisor component. This is due to the fact that:

- Available captures do not include all the different cases presented here.
- Decoders are not implemented in ADIOP for all protocols.

4.4.1 Practical Examples of Incompleteness/Incorrectness in Interoperability Testing

This section contains examples of problems encountered when testing different ATM protocols. The purpose is to show why there may be incomplete/incorrect models.

1. **LANE 1.0:**

   The LANE 1.0 Specification (LANE-1.0 1995) states that:

   5.3.1.1 Configure Request (p61): *The requester MUST issue an LE_CONFIGURE_REQUEST to the LE Configuration Server containing at least the*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

primary ATM Address of the prospective LE Client in the SOURCE-ATM-ADDRESS field. Other information **MAY** *be included ...*

5.3.1.3 Successful Configure Response (p62): *... In this case, the ..., and ELAN-NAME parameters MUST be copied to the prospective LE Client ...*

According to the specification, when no ELAN-NAME is specified in the 'Configure Request', the LECS (LE Configuration Server) should send a 'Successful Configure Response' with the ELAN-NAME set.

But if there are many ELANs in the same ATM network and there is no ELAN-NAME specified in the 'Configure Request' packet, then the LECS has no way of determining which ELAN-NAME to send in the 'Configure Response'. To avoid this situation, some devices implemented this protocol in a way that the LECS rejects the 'Configure Request' if it does not contain an ELAN-NAME. Some vendors want to have this feature of rejecting requests with no ELAN-NAME for security reasons, so that if the requester does not know which ELAN to connect to, she/he will not get a successful response.

These devices are not conformant to the specification but are interoperable with other devices.

2. **MPOA 1.0:**

In the MPOA 1.0 Specification (MPOA-1.0 1997), page 61 (Section 5.3.2.4) states that:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

MPOA Control messages **may** *have the same Extensions as an NHRP packet.*

In the NHRP Specification (NHRP 1998), page 37 (Section 5.3.0) reads:

When extensions exist, the extensions list is terminated by the End of Extensions/Null TLV

NHRP does require the End of Extensions/Null TLV when extensions are used, but MPOA does not specifically make any such requirement. Some MPCs (MPOA Clients) will not terminate the extensions list with the End of Extensions/Null TLV (claiming they are conformant to the specification). And the MPS (MPOA Server) may reject it because it expects the last four bytes of the extensions list to be the End of Extensions/Null extension (in conformance with the specification).

In this case, the two devices are conformant to the specification but are not interoperable with each other.

3. **MPOA 1.0:** (MPOA-1.0 1997)

In the MPOA 1.0 Specification (MPOA-1.0 1997), it is stated in page 70 (Section 5.3.9) about Extensions: *An MPOA Keep-Alive Lifetime Extension must be added as follows:*

Type: 0x1003

Length: Two Octets

Keep-Alive Lifetime [Value]: duration of time ...

The following is what we see in an example of testing SUT A and SUT B (values are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in HEX):

SUT A: Type = 1003, Length = 0002, Value= 0067

SUT B: Type = 1003, Length = 0004, Value= 09000035

SUT A assumes that the Length is always 2 Octets since the maximum value "Value" can take can fit in 2 bytes. Because of this, SUT A decodes "Value" from SUT B as 0000 (not 0035), which is a bad value and rejects it.

SUT B uses 4 bytes to represent a value that always fits in 2 bytes. (Implementation choice)

The issue that caused the problem here is the decision to represent "Value" as 2 or 4 bytes. Both were valid interpretations from the vendors of the protocol specification, since the protocol specification does not state how to represent "Value". But this caused the non interoperability of the two devices.

The two devices are conformant to the specification but are not interoperable with each other.

4. **UNI 4.0:** (UNI-4.0 1996)

   Information Elements like the 'Minimum Traffic Descriptor' are necessary to negotiate traffic parameters. Hence, some devices require that they be present. Other devices may not supply them and reject the call if they are present. (UNI4.0 8.1.1.1 p61)

   These devices are neither conformant nor interoperable.

5. **PNNI 1.0:** (PNNI-1.0 1996)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A switch may not send 1WayIn (required event in PNNI) at all, but still interoperate with other switches because it sends the messages that follow 1WayIn.

This switch is not conformant to the specification but is interoperable with other switches.

6. **PNNI 1.0:** (PNNI-1.0 1996)

   Because of a race condition at the monitoring point, 1WayIn may be captured by the analyzer instead of the expected 2WayIn.

   The devices in this case are conformant and interoperable.

7. **PNNI 1.0:** (PNNI-1.0 1996)

   Some devices that are running PNNI do not work when ILMI (Interim Local Management Interface) (UNI-3.1 1994) is enabled on one switch and disabled on the other. In the PNNI specification no such requirement is made.

   These devices are conformant but are not interoperable.

Summary

As a summary to this section, we can have one of four scenarios when testing two devices:

1. They are conformant and interoperable
2. They are conformant but not interoperable
3. They are not conformant but interoperable
4. They are neither conformant nor interoperable

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.2 Types of Incomplete and Incorrect Models

In this section, we use a top-down approach to derive more ideas on model incompleteness and incorrectness by looking at the structure of some CSP models.

Figure 4.1 represents an initial CSP model. This model is an illustrative example used here in order to provide insights on the different types of model incompleteness and incorrectness. This section is not intended to provide a detailed analysis of these types but rather an introduction for future work on this subject. In the following examples, we state some initial actions that can be taken by a tester to make changes in a CSP model, and we look at what impact this would have on the whole model.

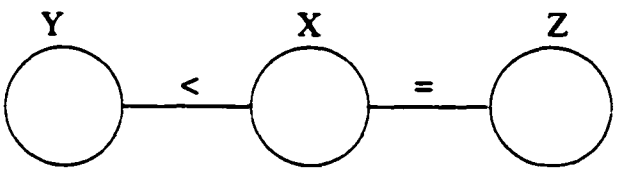

Figure 4.1: Initial CSP model

Note: Adding transitivity constraints involving a variable X means that, for every variable pair Y and Z that have constraints with X, a constraint between Y and Z is added. Examples of this are shown in the following:

- if Y<X and X<Z then Y<Z is added.

  if Y<X and X=Z then Y<Z is added.

  if Y=X and X=Z then Y=Z is added.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- if Domain(X) = Domain(Y) = Domain(Z) = {a,b}

  and if Y≠X and X≠Z then Y=Z is added.

We have identified two main categories of model inconsistencies:

1. Deletion of a constraint/event or event becoming optional,

2. Addition or modification of a constraint

We have broken these up into sub-categories presented in the following:

1. **Variable becomes optional: missing value, missing constraint**

   If a variable X becomes optional, then we need to add the transitivity constraints. The idea here is that if X is not observed, which is OK (because X is optional), then the constraint between Y and Z should be preserved. Figure 4.2 shows the result of updating the CSP model of Figure 4.1 in this case.

   One real example related to this category is the one presented in item 5 of the previous section.

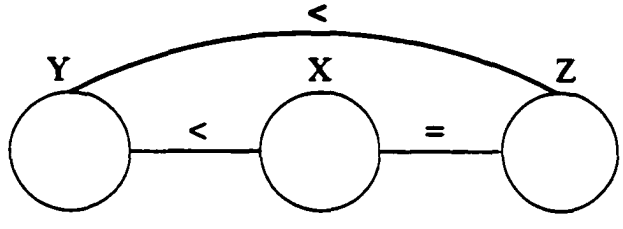

Figure 4.2: CSP model updated when variable X becomes optional

2. **Extra variable: variable removed**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If a variable X has to be removed, first we need to add the transitivity constraints then remove it. The idea here is that variable X is not involved in this model, but some constraints may have been initially omitted because they are captured in other constraints involving X. Figure 4.3 shows how the model of Figure 4.1 is updated to account for this problem.

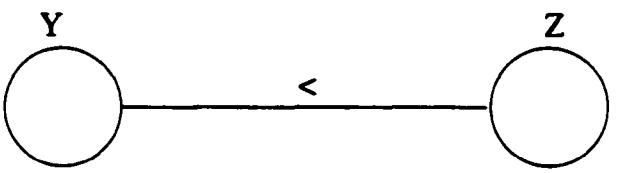

Figure 4.3: CSP model updated when variable X is removed

3. **False constraint: constraint updated**

Figure 4.4 shows a CSP model involving three variables and three constraints.

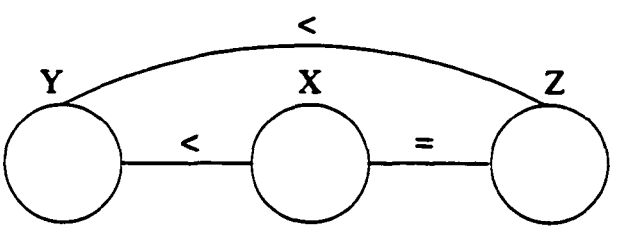

Figure 4.4: Initial CSP model

If there is a false constraint in the CSP model, we need to update the model so that there is no inconsistency in it. If the CSP model in Figure 4.4 has a false constraint between X and Y and that the constraint should be Y>X, then the constraint between Y and Z has to be updated too by changing Y<Z to Y>Z. Figure 4.5 shows the result

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of this update.

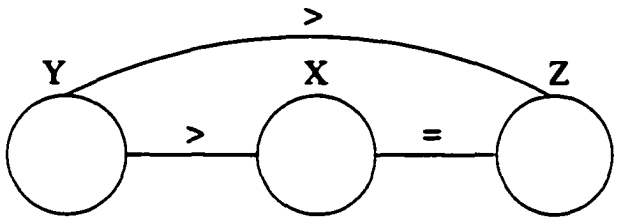

Figure 4.5: CSP model updated when there is a false constraint

4. **Extra constraint: constraint removed**

If a constraint is removed, some information may have to be captured. If the constraint between X and Z does not exist in Figure 4.1, then the constraint between Y and Z should be added. Figure 4.6 shows how this is done.

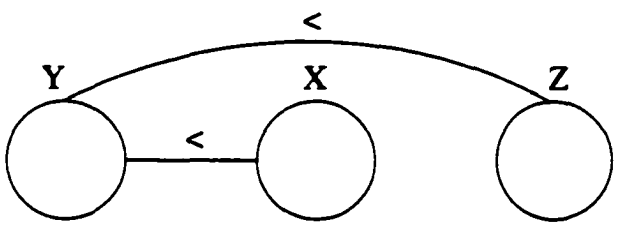

Figure 4.6: CSP model updated when a constraint is removed

5. **Missing constraint: constraint added**

If a constraint is added, we may need to add the transitivity constraints and check for the model consistency, because this new constraint may conflict with another existing one.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.3 One Type of Model Inconsistency

We present in this section one type of model inconsistency that is implemented and tested in this dissertation. This is the type presented in sub-category 1 of the previous section and is related to item 5 of the examples section.

The following states the problem presented by this type of model inconsistency and the actions taken, as implemented using CBR, to debug this problem. This problem is found the first time by a tester at the UNH-IOL, and by CBR in future similar cases. This is detailed throughout the rest of this chapter. The "Actions" are taken by the tester and may be stored as a case in the case base.

Problem: Event not mandatory (Missing event)

Actions taken:

1. Update the status of the variable of the missing event to become 'Optional'

2. Add transitivity constraints involving the time variable of this event

In this chapter, we will show how this type of inconsistency (i.e., Event should not be mandatory) is fixed and debugged using CBR. The expert's approval is needed to confirm that this model debugging is valid, and more actions may be added by the expert. The expert also checks that correct solutions are not lost in the new updated model by executing the updated test cases with other data sets. The actions taken by ADIOP may not be sufficient, and manual checking by an expert might be needed. More empirical results are provided in the evaluation section showing that these actions taken are justified for this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

type.

We worked with this type of inconsistency to prove that model debugging works in the interoperability testing domain using a CBR-CSP integration. We defined a framework for CBR and its integration with CSP. This is detailed throughout the rest of this chapter. The implementation of this framework also led to the use of CBR for improving problem solving and explanation even when models are consistent. CBR has been implemented and tested with a case base containing these two types of cases: a case related to model inconsistency and cases for improving problem solving and explanation. This chapter covers more about how to use CBR in different types of cases, but it does not cover many types of model inconsistencies. However, it exemplifies how the CBR process can work in both the domain of debugging models and in the domain of improving problem solving and explanation, and that similar conclusions can be drawn for both domains.

Further work is needed in this area to include more types of inconsistencies and identify ways to fix them, and to generalize these findings to more model debugging cases.

4.5 Case-Based Reasoning

The reliance on past experience that is such an integral part of human problem solving has motivated the use of case-based reasoning (CBR) techniques. A CBR system stores its past problem solving episodes as cases, which later can be retrieved and used to help solve a new problem. CBR is based on two observations about the nature of the world: that the world is regular, and therefore similar problems have similar solutions, and that the types of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

problems encountered tend to recur (Leake 1996). When these two observations hold true, it is worthwhile to solve new problems by reusing prior reasoning. Much of the original inspiration for the CBR approach came from the role of reminding in human reasoning (Schank 1982).

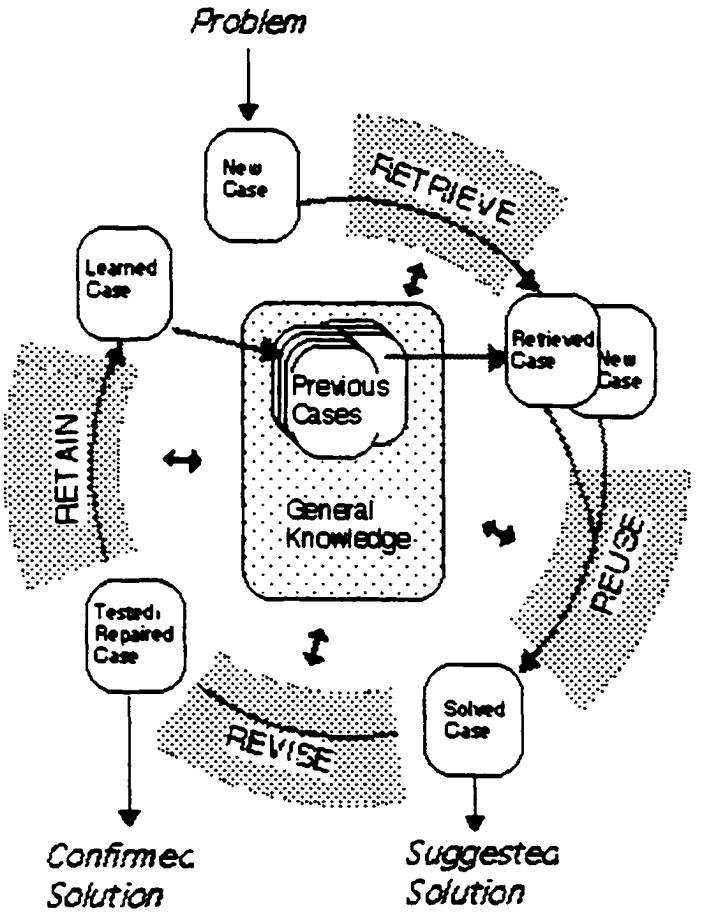

Figure 4.7: Case-Based Reasoning Cyclical Process

The process by which a case-based reasoner operates has been described by (Aamodt & Plaza 1994) as a cyclical process comprised of the *four REs*: ***RETRIEVE*** the most

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

similar case(s), *REUSE* the case(s) to solve the problem, *REVISE* the proposed solution if necessary, and *RETAIN* the new solution as a new case. This process in shown in Figure 4.7 taken from (Aamodt & Plaza 1994). The application of this CBR cycle to real problems raises a common set of issues, regardless of the domain of application. These issues include case representation, indexing, storage, retrieval method, and adaptation method. We can abstract the CBR process as one of recalling an old similar problem, and adapting that problem to fit the new situation requirements, as shown in Figure 4.8, taken from (Maher, Balachandran, & Zhang 1995).

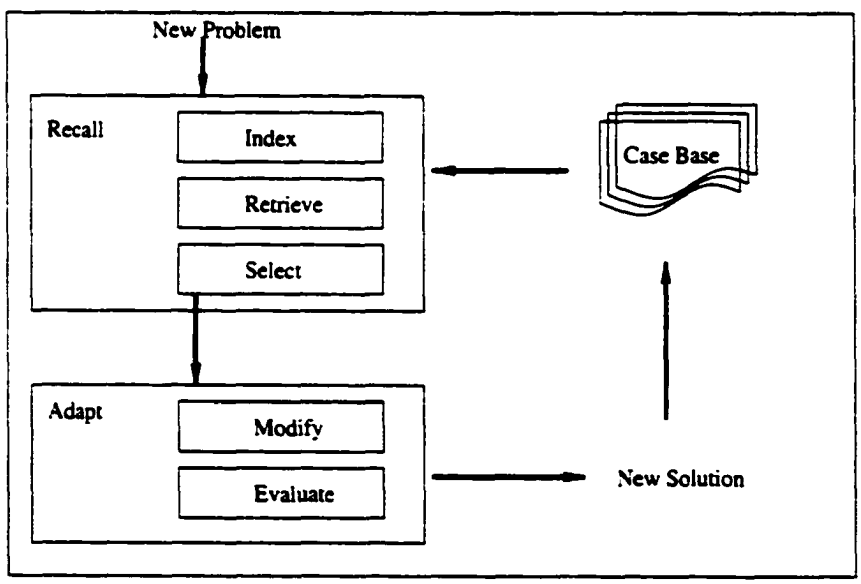

Figure 4.8: Case-Based Reasoning Process

A case is usually composed of a problem description and its solution. Whenever there is a new problem, it is matched to what is already in the case base using similarity metrics to determine how close an old case in the case base is to the new problem. Then the useful

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cases are retrieved and adapted to the new problem to provide a solution. The new case (problem and its solution) will be stored in the case base if it provides new information.

4.6 CSP/CBR Integration

We have presented in (Sqalli, Purvis, & Freuder 1999) a survey of the various applications integrating constraint satisfaction and case-based reasoning. This exploration provides insight into how CSP can be enhanced by combining it with CBR, thereby enabling its usage in an even broader spectrum of applications. Although there has been a lot of work done combining CBR and MBR (Model-Based Reasoning) including CBR with CSP, our approach to this integration is novel in the way the two paradigms are integrated. We propose to represent our system as a CSP model supported by a case base to compensate for incompleteness and incorrectness. This section focuses more on the CBR/CSP interface and how CBR is used to compensate for the incompleteness and incorrectness of a CSP model. In Figure 4.9, we show how CBR and CSP are combined to solve these problems.

Cases are represented using a flat table with feature/value pairs. Cases that are used for updating CSP models have a feature for this purpose that is represented using CSP. This allows ADIOP to easily update CSP models using the value stored in this feature. In the example that follows in this section we show how this representation is done for one case.

CBR checks if there is a similar case in the case base. If one or many similar cases are found, then they are retrieved and adapted to solve the new problem. The adaptation process is simple in many cases because of the tests' similarity within the same test suite.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

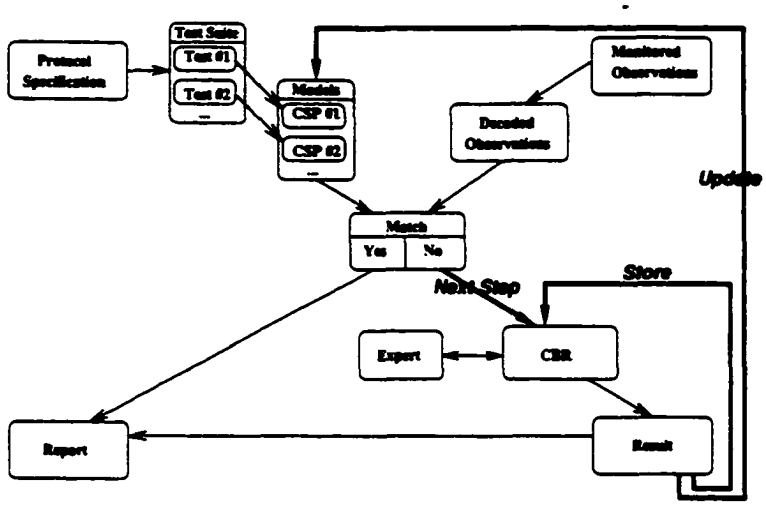

Figure 4.9: Integration of CSP Model and CBR for Interoperability Testing

It is based on some simple rules that will be described later in this chapter. Finally, the user checks whether the adaptation is appropriate.

The new case, consisting of the problem and solution, is eventually stored in the case base if the tester decides to do so. The new solution can also be used to update the CSP model, and make it more adaptable to new situations. The process of updating the model is done manually for the first case. As the system learns more cases, there is less interaction with the user on the model update process. A set of general rules are used to update the model from a case. Some examples of these rules are:

1. Add or remove the constraints from the case's UpdateModel feature to the model.

2. Modify variables in the model using the case's UpdateModel feature to make the constraints consistent.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In the next section, more explanation will be given on how to apply this reasoning in a practical example.

4.7 CBR/CSP Integration Components of ADIOP

4.7.1 Advisor

If the diagnosis result of a test case is a failure, then an "Advisor" button is shown in the test case result's window (Figure 4.10). More on the ADIOP's Diagnosis component can be found in the previous chapter.

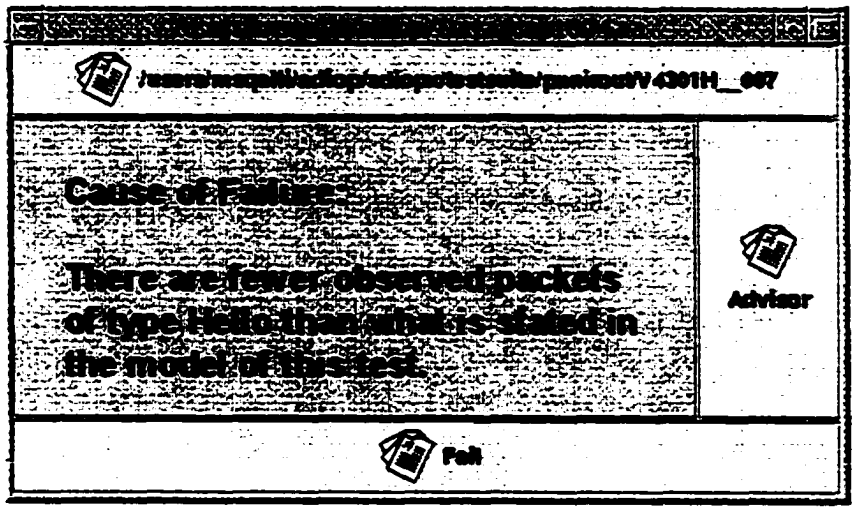

Figure 4.10: Test Case Result containing an 'Advisor' button

The Advisor is the CBR component of the ADIOP system (Figure 4.11). It provides the user with the functionality to recall previous similar problems and reuse them to solve new problems. This is done through user interaction and advising. More details will be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

provided later in this chapter.

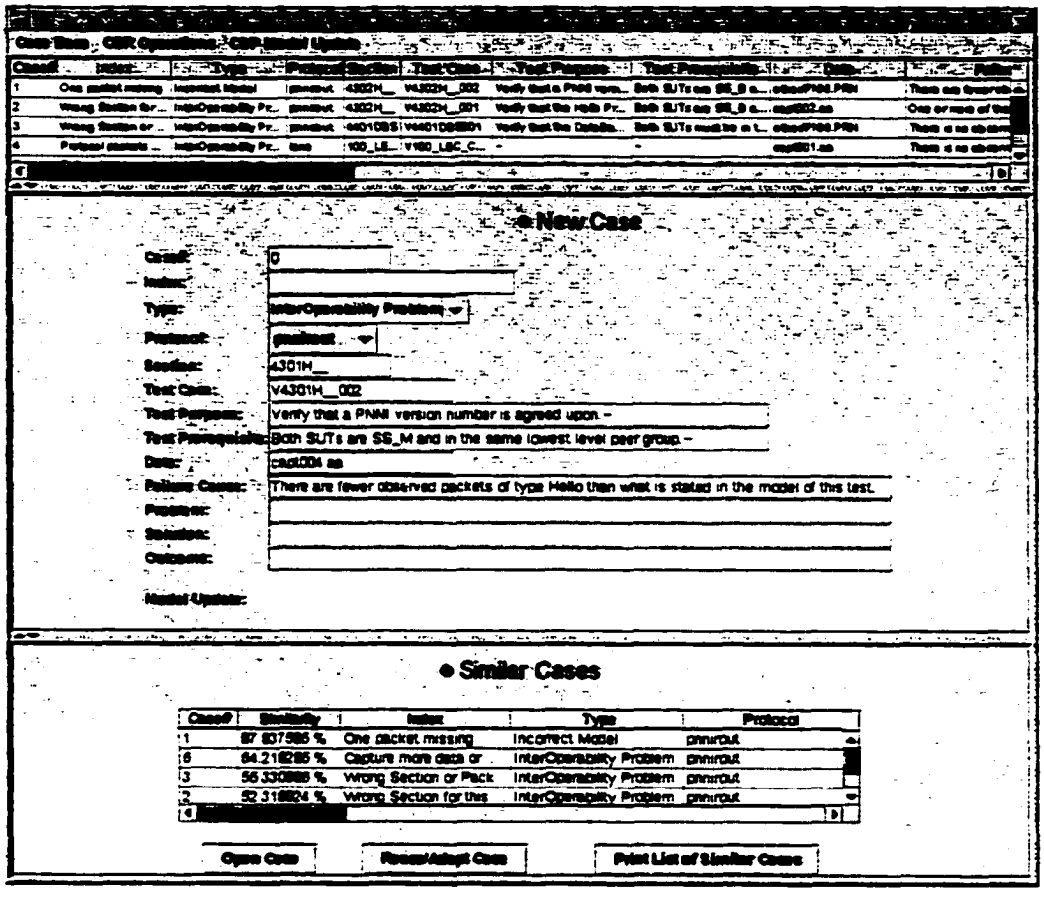

Figure 4.11: Advisor/CBR Window

The Advisor is used only when a test case fails. If a test case passes, the user gets a matching solution as was explained in detail in the previous chapter, and there is no need for the Advisor.

Another way to access the Advisor/CBR window is from the main ADIOP window. This is useful for checking the CBR component of ADIOP, adding cases, updating cases,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

etc. without having to execute test cases and debug failures.

The Advisor window shows all the cases stored in the case base in the top panel (Figure 4.11). The middle panel shows the information about a new case. If the Advisor was called from the result's window of a failed test case, then the 'New Case' will contain information generated from this test case's result. If the Advisor window is called directly from the main ADIOP window, then the new case will have empty fields/values. The third panel in the Advisor window contains the cases stored in the case base sorted by their similarity to the new case. More details are provided later in this chapter.

There are two kinds of failures stored in the case base. The first is related to incomplete or incorrect CSP models. These CSP models are eventually updated through the Advisor component. The second kind of failure are those due to interoperability problems in the devices being tested. The purpose of using the Advisor here is to get a useful, correct, and complete explanation of the cause of a test case's failure. This case base is stored in a file as a flat table (Figure 4.12).

4.7.2 Development Process and Case Collection

The general development process of a case-based system as described in (Bergmann *et al.* 1999) page 17 is to:

- Build and maintain a case base
- Customize the user interface
- Tune the way the information system operates.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Case: 1
   index:              One packet missing
   type:               Incorrect Model
   protocol:           pnnirout
   section:            4302H__
   testcase:           V4302H__002
   testpurpose:        Verify that a PNNI version number is agreed upon.-
   testprerequisite:   Both SUTs are SS_B and in different lowest level peer groups.-
   data:               other/PNNI.PRX
   failurecause:       There are fewer observed packets of type Hello than what is stated in the model of this test.
   problem:            The second Hello packet (Hello1B) is missing
   solution:           The second Hello packet (Hello1B) is made optional
   outcome:            Model updated, Warnming added, interoperable but not conformant
   modelupdate:        ADD:    $UNARY_CONSTRAINT       Hello1B.status == D_Optional
                       ADD:    $BINARY_CONSTRAINT      Hello1A.time <= Hello2A.time
                       ADD:    $BINARY_CONSTRAINT      Hello1A.time <= Hello2B.time
                       UPD:    Hello1B.peer_group_id   Hello2B.peer_group_id
                       ADD:    $CONSTRAINT     Hello2A.time Hello2B.time D_Mandatory.contains(_Hello1B.status) ||
                               Compare.compare(_Hello2B.time, "<=", _Hello2A.time)

Case: 2
   index:              Wrong Section for this captured data
   type:               Interoperability Problem
   protocol:           pnnirout
   section:            4302H__
   testcase:           V4302H__001
   testpurpose:        Verify that the Hello Protocol is running on an operational physical link.-
   testprerequisite:   Both SUTs are SS_B and in different lowest level peer groups.-
   data:               capt002.aa
   failurecause:       One or more of these constraints declared in the model of this test is/are violated:
                       -[Hello4.peer_group_id != Hello8.peer_group_id :: , Hello4.source != Hello8.source :: ,
                       Hello4.time <= Hello8.time :: ]-with the following respective occurrences:-[9, 6, 15]
   problem:            The data captured is from devices in the same peer group (Section 4301H__). But, the test
                       case run is for devices in different peer groups (Section 4302H__).
   solution:           Use test cases from another section (i.e., 4302H__) to run with this captured data.
   outcome:            The problem is solved.
   modelupdate:
```

Figure 4.12: A Partial View of the Case Base Table

ADIOP follows loosely this process, as it is meant to be a prototype. The case base is organized inside the computer memory using a well known CBR format that is a flat database (Figure 4.12). We will discuss case representation in the next section. To improve the system, we need to look at the maintenance issue as well as the information system where this operates.

As for case collection, ADIOP allows the user to add new cases from scratch. The new case is given a new number to be used if stored in the case base. ADIOP also provides the functionality for the tester to get all cases from the case base file displayed in the GUI.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.7.3 Case Representation

The representation problem in CBR is primarily the problem of deciding what to store in a case, finding an appropriate structure for describing case contents, and deciding how the case memory should be organized and indexed for effective retrieval and reuse (Aamodt & Plaza 1994).

What is a case? A case is a contextualized piece of knowledge representing an experience that teaches a lesson fundamental to achieving the goals of the reasoner (Leake 1996).

"The first step in building a case-based application is to decide how to represent a case inside the computer... In commercially available systems, there are different approaches to case representation and, related to that, different techniques for case-based reasoning: the textual CBR approach, the conversational CBR approach, and the structural approach ... In the structural CBR approach, the developer of the case-based solution decides ahead of time what features will be relevant when describing a case and then stores the cases according to these." (Bergmann *et al.* 1999), page 19.

"In different structural CBR systems, attributes may be organized as flat tables, or as sets of tables with relations, or they may be structured in an object-oriented manner ... This approach always gives better results than the two others, but it requires an initial investment to produce the domain model ... The domain model specifies a set of attributes (also called features) that are used to represent a case." (Bergmann *et al.* 1999), page 21.

The Advisor uses the structural CBR approach described above. The way the cases are represented and stored is very important because these cases will be reused in future similar

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

occurrences and we need to capture the main features of these cases.

In ADIOP, each case includes a set of features and their associated values (e.g., Protocol: pnnirout). There is a total of 14 features per case:

1. **Case:** this is a sequential number assigned to cases in the case base.

2. **Index:** this is a short English text description of this case.

3. **Type:** describes the type of this case. It can be either "Interoperability problem" for test case failures or "Incomplete Model" or "Incorrect Model" for test case bugs. The user may add other types to the three provided by ADIOP.

4. **Protocol:** states which protocol was used when this case was generated.

5. **Section:** states the section of the test case that caused a failure and the generation of this case.

6. **Test Case:** states the name of the test case that caused a failure and the generation of this case.

7. **Test Purpose:** states the purpose of the test case that caused a failure and the generation of this case. This is taken from the test case description stored in the Modeling component of ADIOP.

8. **Test Prerequisite:** states the prerequisites needed to run the test case that caused a failure and the generation of this case. This is also taken from the test case description stored in the Modeling component of ADIOP.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

9. **Data:** this is the name of the data captured that was being diagnosed when this case was generated.

10. **Failure Cause:** this is the message generated by ADIOP's Diagnosis component when running the test case stated in the "Test Case" feature of this case.

11. **Problem:** this is a description of the stored problem as viewed by the tester who detected it when using ADIOP's Diagnosis.

12. **Solution:** this is a description of how this problem was dealt with by the tester.

13. **Outcome:** this is a description of what is the outcome of this case such as "The model has been updated using this case", general advice from the tester who solved this problem, etc.

14. **Model Update:** contains statements on how to update the CSP model of the test case stated in the "Test Case" feature using this case. It is a set of statements (mainly constraints) to be added, removed, or updated in the CSP model. More about the language used in this field will be detailed later in this chapter.

Figure 4.13 shows the GUI used for displaying one case. We will use this case as an example throughout the rest of this chapter.

The case base in ADIOP is expected to remain small since CSP is sufficient for test cases that pass and usually in interoperability a high percentage of test cases pass. The case base is used only in case of failures and many of these will be similar. For the case base storage we use a flat table format. (Kitano & Shimazu 1996) state that in CBR, one

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

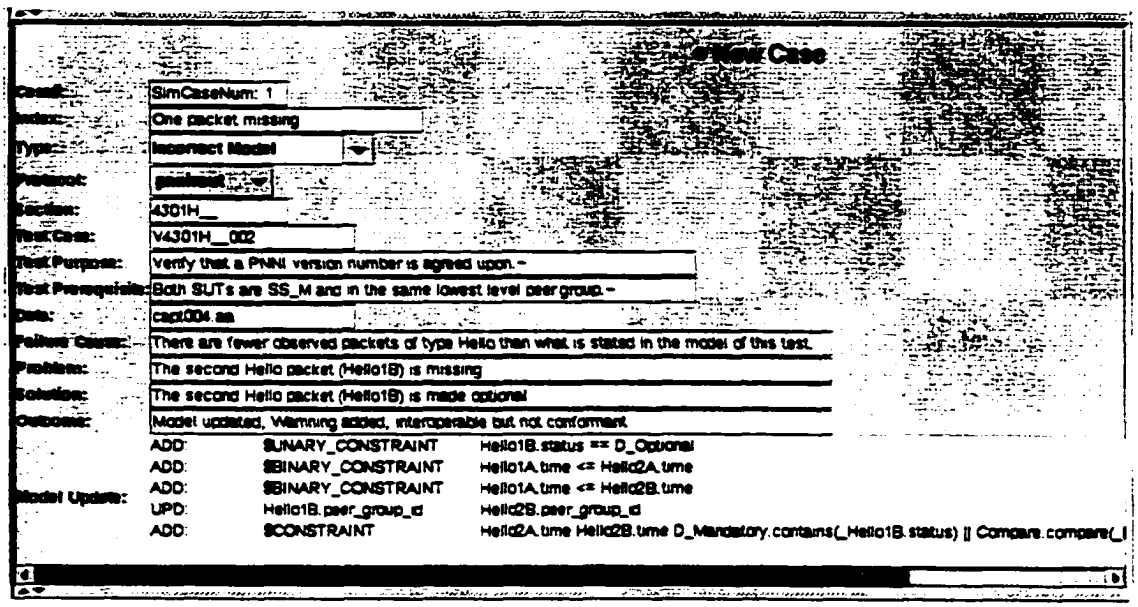

Figure 4.13: A case displayed using the ADIOP's GUI

of two methods are usually used for case base storage, structured indexing or a flat-record style database. We plan to use the latter.

4.7.4 Case Retrieval

When a new failure occurs, the CBR system (ADIOP's Advisor) constructs a new case and retrieves old cases from the case base that are similar to it. As (Leake 1996) states. "Similar problems have similar solutions."

Case retrieval deals with finding ways to match and compare different cases and measure similarity between them, to derive a solution similar to old ones. This requires the use of algorithms for comparing different features' values and measuring distances between them, defining weights for these features, and methods or formulas for computing the global

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

similarity between old and new cases.

As stated in (Aamodt & Plaza 1994), while some case-based approaches retrieve a previous case largely based on superficial syntactic similarities among problem descriptors (e.g., CYRUS (Kolodner 1983), ARC (Plaza & Lopez de Mantaras 1990), and PATDEX-1 (Richter & Weiss 1991)), some approaches retrieve cases based on features that have deeper, semantic similarities (e.g., the PROTOS (Bareiss 1988), CASEY (Koton 1989), GREBE (Branting 1991), CREEK (Aamodt 1991), and MMA (Plaza & Arcos 1993)). We combine both syntactic and semantic similarity measures depending on which feature is being compared.

Comparing Feature's Values: Distance and Local similarity

The goal is to be able to assign distances between individual values of the same feature. Some features are not used for computing the global similarity and thus have no distance functions associated to them.

The following is a description of how distances are computed for different features.

If both values are empty (null), then the distance is 1. If both values are equal then the distance is 0.

- **Case:** no distance is computed for this feature.
- **Protocol:** Semantic similarity is used for this feature. If the two values are the same (same protocol), then the distance is 0 (=0/4). If they are of the same type of protocol (e.g., signaling protocols: UNI and PnniSignal) then the distance is 1/4.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If there are many common packets between the protocols (e.g., LANE and MPOA) then the distance is 2/4. If there are few common packets between the protocols (e.g., LANE and UNI) then the distance is 3/4. If there are no common packets between the protocols (e.g., PnniRout and UNI) then the distance is 1 (=4/4). We used only ATM protocols in this dissertation, but one can update these numbers if other types of protocols are defined.

- For the other 12 features, we use syntactic similarity by computing the distance between two strings (i.e., distance between values of the same feature for the old and new case).

For computing the distance between two strings, we use a widely known method based on n-grams that is used for computing similarities between documents.

"Instead of representing documents as sets of index terms, CBR EXPRESS uses an even simpler matching based on n-grams of common characters for comparing documents. More precisely, the text contained in a document is cut into sequences of n subsequent letters (most often n=3) and the set of all the sequences is used as a representation of the original document ... Compared to [some] models of IR [(Information Retrieval)], this is firstly less computationally expensive and secondly appears to be very robust against minor changes in the test as, for example, grammatical variations and misspellings. As with the IR models, this kind of document matching does not permit the integration of additional knowledge sources, such as domain specific thesauri, glossaries, etc." (Lenz, Hubner, & Kunze 1998)

This method also takes partial matching of words into account (e.g., packet vs. packets).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As for n, the more commonly used values found in the literature are 3 and 5. We use the value 5 in ADIOP.

N-grams Distance

The distance between two strings is computed using n-grams. Each string is first transformed into a set of n-grams. The exhaustive set of constituent n-grams comprises all n-character sequences produced by an n-character-wide window displaced along the text one character at a time, and contains many duplications (Damashek 1995).

A reference n-grams vector, we call it REF, represents the set of n-grams of the union of both strings. Each string is then represented with a vector of relative frequencies of its distinct constituent n-grams using REF as its baseline.

Let REF contain k distinct n-grams, with m_i occurrences of the i^{th} n-gram. Then the value (weight) associated to the i^{th} vector component as stated in (Damashek 1995) is:

$x_i = m_i / \Sigma_{j=1}^{k} m_j$ where $\Sigma_{j=1}^{k} x_j = 1$ [Eq 4.1]

For example, if REF = "Hello World Hello", then we obtain 13 5-grams: ('Hello', 'ello ', 'llo W', ..., ' Hell', 'Hello'). The 5-gram 'Hello' is the only one that is repeated twice. The weight associated with this 5-gram is 2/13, and that associated with all the other 5-grams is 1/13.

Hash-tables are used for storing these vectors. Each element of the table is represented with:

- A hash key which is an n-gram from the REF vector, and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- the frequency of occurrence of this n-gram in the string.

In the above example, the hash table will contain 12 elements as stated in the following: {(2/13, 'Hello'), (1/13, 'ello '), ... (1/13, ' Hell')}

The following is a summary of n-grams representation of a string taken from (Damashek 1995):

(i) Step the n-gram window through the document, one character at a time.

(ii) Convert each n-gram into an indexing key.

(iii) Concatenate all such keys into a list and note its length.

(iv) Order the list by key value [efficient algorithms will do this in linear time].

(v) Count and store the number of occurrences of each distinct key while removing duplicates from the list.

(vi) Divide the number of occurrences of each distinct key by the length of the original list.

In ADIOP, step (iv) is skipped, because the list is usually small and will not be affected by the order of its elements.

The number of distinct n-grams will initially closely track the document size in characters (Damashek 1995). This is true because each character (with the exception of the last n-1 characters) is the initial character of some n-gram.

The following equation is used in (Damashek 1995) to compute the similarity between two strings represented by two n-grams.

$$S_{mn} = \Sigma_{j=1}^{k} x_{m_j} x_{n_j} / (\Sigma_{j=1}^{k} x_{m_j}^2 \Sigma_{j=1}^{k} x_{n_j}^2)^{1/2} = \text{Cos}\ \theta_{mn} \qquad \text{[Eq 4.2]}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

x_{m_j} is the relative frequency with which key j (out of a total of k possibilities) occurs in document m. The score given by [Eq 4.2] is the cosine of the angle θ_{mn} between two vectors in the high-dimensional document space as viewed from the absolute origin.

(Damashek 1995) improves this algorithm by translating the origin of the vector space to a location that characterizes the information one wishes to ignore, so that words such as "is the", "and the" ... do not influence the similarity. This is not implemented in ADIOP.

A cosine value of 1.0 indicates that the document and reference vectors are perfectly correlated (or identical), a value of minus 1.0 that they are perfectly anti-correlated (or antithetical), and a measure of 0.0, that they are uncorrelated (or orthogonal) ... (Huffman 1995)

Another way to compute the distance is by using the Cluster Euclidean distance: $d(x_m, x_n) = \sqrt{\Sigma_{j=1}^{k}(x_{m_j} - x_{n_j})^2}$. However, this is not widely used, and the results we obtained using this method were worse than the ones obtained using [Eq 4.2].

Weights

The weight describes the relative importance of each attribute/feature. We have used different values for the weights that are manually set. The following weight values are what worked better for us after few trials.

W(Case)= 0, W(Index)= 0, W(Type)= **1**, W(Protocol)= **3**, W(Section)= 0, W(Test Case)= **3**, W(Test Purpose)= **3**, W(Test Prerequisite)= **3**, W(Data)= **1**, W(Failure Cause)= **15**, W(Problem)= 0, W(Solution)= 0, W(Outcome)= 0, W(Model Update)= 0.

In this configuration, only 7 attributes are used. But, we retain the other 5 attributes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for an eventual future improvement of ADIOP, because we believe they provide useful information about cases. More empirical studies are needed to derive the most suitable weights for this application. However, the evaluation section of this chapter shows that these values lead to good similarity measures.

Global Similarity

The global similarity is computed using the following Nearest Neighbor Retrieval equation:

$S(oc, nq) = \Sigma_{i=1}^{n} w_i S_{oc_i nq_i} / \Sigma_{i=1}^{n} w_i$ [Eq 4.3]

where:

- oc: old case. This contains a problem and its solution.
- nq: new query. This contains the current problem with no solution.
- w_i: weight of the i^{th} feature.
- $S_{oc_i nq_i}$ is obtained from [Eq 4.2] to compute the similarity between the same features of both cases: oc and nq.

We take the example when Case 2 is retrieved as the most similar case when using test case V4301H_003. Section 4.9 discusses in detail this example.

In this example, the values obtained for similarity for different features are: S(Type) = 1, S(Protocol) = 1, S(Test Case)= 0.55, S(Test Purpose)=0.33, S(Test Prerequisite)= 0.75, S(Data) = 0, S(Failure Cause) = 0.97.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Using [Eq 4.3] and the weights introduced above, the global-similarity is computed as follows: S = W(Type)*S(Type) + ... + W(Failure Cause)*S(Failure Cause) = 1*1 + 3*1 + 3*0.55 + 3*0.33 + 3*0.75 + 1*0 + 15*0.97 = 0.81

Retrieval Process

By default, ADIOP uses the new case's values to match against existing cases in order to compute their respective similarity values. The user may change the values of certain features and then retrieve similar cases using these changed values. One feature that can be very useful to the user is the "Type" feature. The user can try the different types of cases including the ones related to model incorrectness/incompleteness or to interoperability problems and failures (Figure 4.14). The similarity scores in using different values of this feature will allow the user to better judge what type best matches the new case.

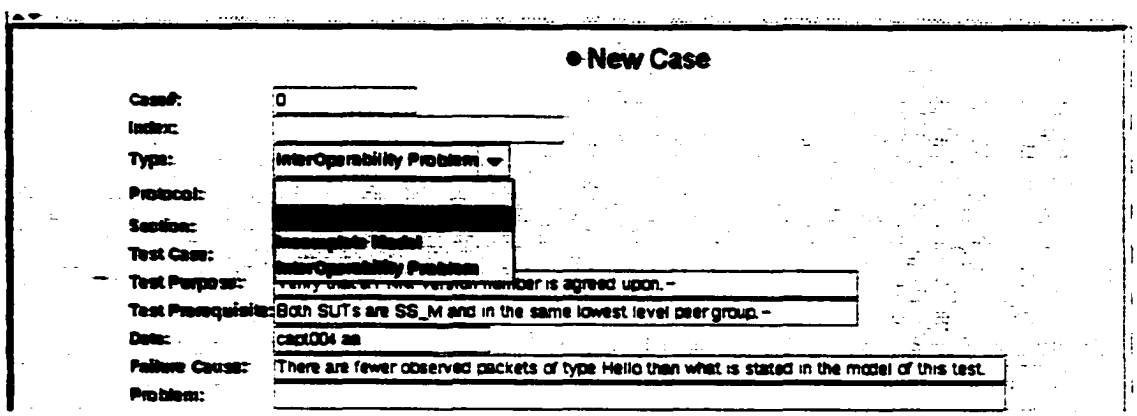

Figure 4.14: Cases' Types

The user may retrieve similar cases through the GUI (Figures 4.15 and 4.16). The cases

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

are displayed according to their similarity. The lower panel of ADIOP's Advisor window shows this list. The similarity value is a percentage representing how close the new case is to the old cases.

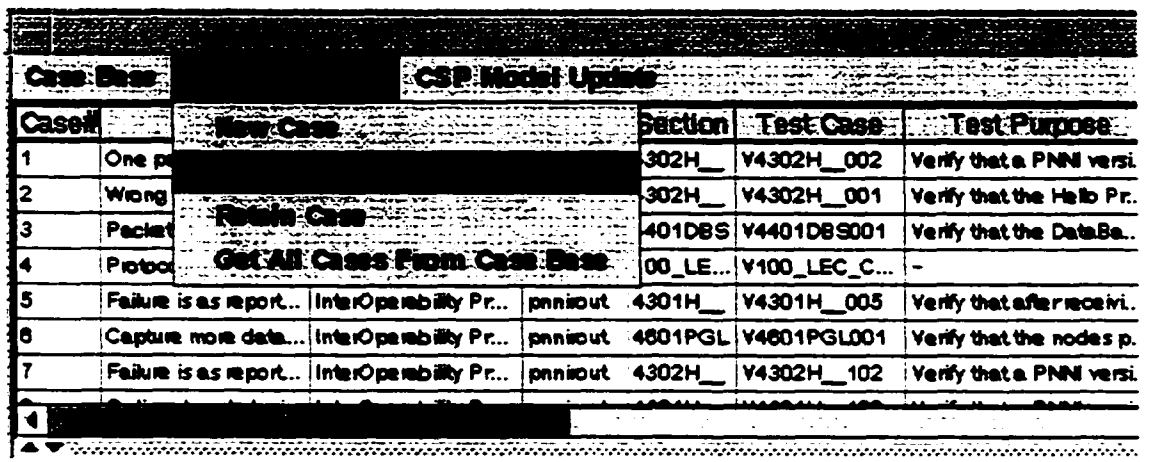

Figure 4.15: Retrieve Similar Cases Menu

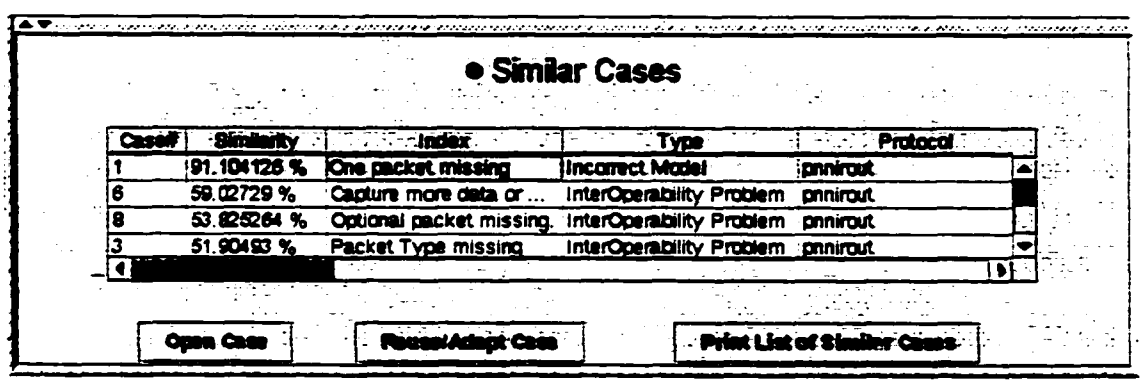

Figure 4.16: Similar Cases Table

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.7.5 Case Reuse/Adaptation

"The reuse of the retrieved case solution in the context of the new case focuses on two aspects: (a) the differences among the past and the current case and (b) what part of a retrieved case can be transferred to the new one" (Aamodt & Plaza 1994).

If there is a similar case to the new case in the case base, then the user may choose this one as the case to be reused and adapted in the new situation. For adaptation, ADIOP uses few basic rules to adapt the case and the user has to confirm this or make updates to this adaptation (Case Revision).

The "Case Adaptation" window in ADIOP (Figure 4.17) shows features of the new case. the similar case chosen for reuse, and the adapted case generated in addition to the similarity value for each feature between the new and similar case and the weight used for each feature to compute the global similarity between the two cases.

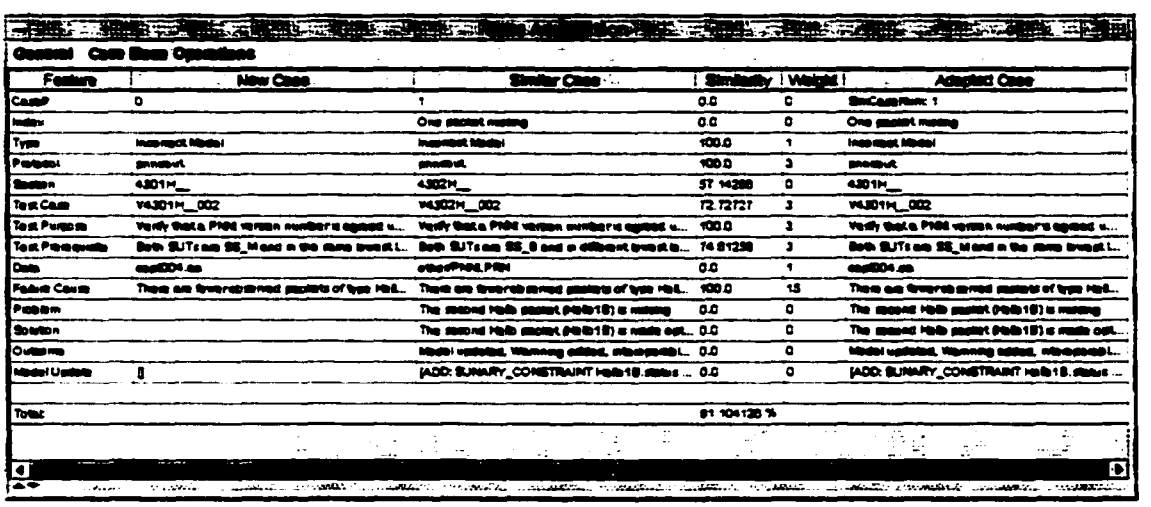

Figure 4.17: Case Adaptation Window

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The user has the option to make changes to values of two columns: the 'New Case' column and the 'Adapted Case' column. The user may then invoke the "Compute Similarity" function to compute the new similarity values or call the "Reuse/Adapt Case" functionality to appropriately update the 'Adapted Case' values (Figure 4.18).

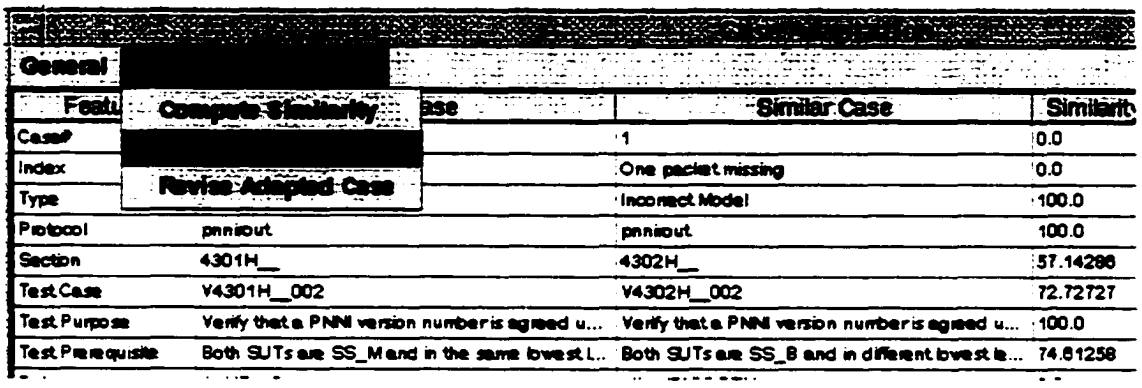

Figure 4.18: Case Adaptation Menu

The adaptation rules used in ADIOP are the following: for the Case number, if the new case does not have a case number then the value "SimCaseNum: " + <similarCaseNumber> is temporarily assigned to it until the user decides whether to add it to the case base. If so, a sequence number is assigned to it. For all other features, if a value of the new case is empty(null) then the most similar case's value is assigned to the adapted case, otherwise it is the new case value that is assigned in the adapted case.

The adaptation rules can be improved further to make this phase in ADIOP more efficient and to derive a more useful adapted case that will need less effort during the revision phase. With the rules we have implemented in ADIOP, the user has to spend more effort during case revision.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.7.6 Case Revision

This phase is called *case revision* and consists of two tasks: (1) evaluate the case solution generated in the case adaptation phase. If successful, record the success (case retainment, see next section). Otherwise, (2) repair the case solution using domain-specific knowledge (Aamodt & Plaza 1994).

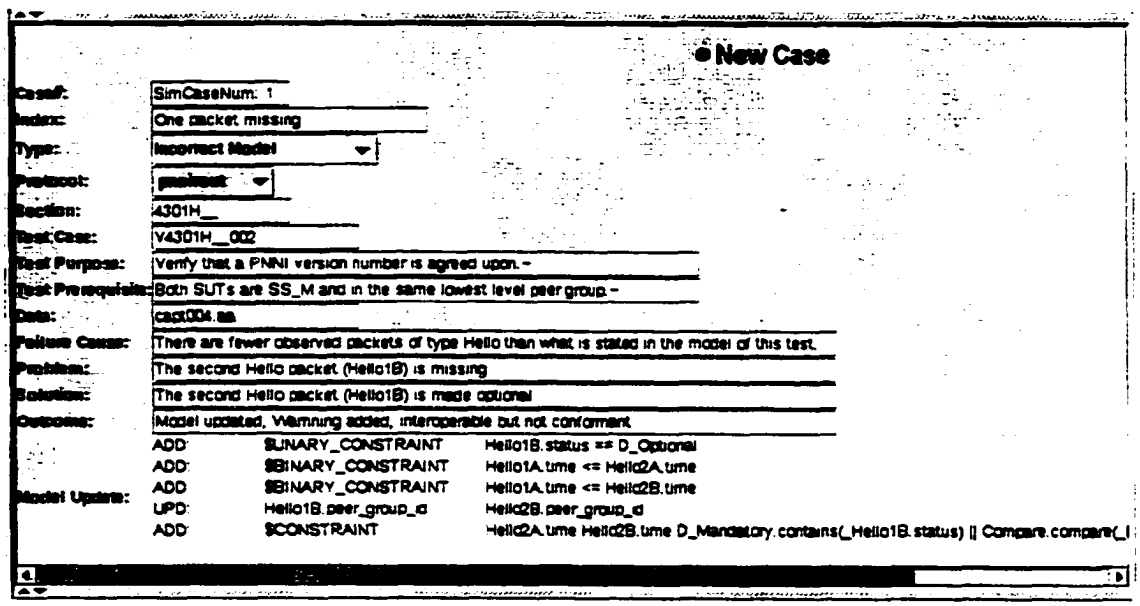

Figure 4.19: Window for Case Revision of the Adapted Case

When the user has made the initial changes and adapted the new case using a similar case, she/he can revise the adapted case (Figure 4.19). This will update the features' values of the new case in the "Advisor" window. At this stage, the user has to evaluate this new adapted case by making sure that the case is adapted correctly and checking that the "Model Update" value is set to the right statements if the test case model is to be updated

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

using this case. This can be done through applying this new adapted case to a real problem. If the outcome is successful, then the user can step to the next phase. Otherwise, the tester may repair the case solution in the "New Case" panel of the Advisor window (Figure 4.19).

Although it is always possible to correct these statements later, it is recommended to do it earlier in this process.

4.7.7 Case Retainment - Learning

This is the process of incorporating what is useful to retain from the new problem solving episode into the existing knowledge. The learning from success or failure of the proposed solution is triggered by the outcome of the evaluation and possible repair (Aamodt & Plaza 1994).

If the new revised case is different from old cases in the case base, then the user may choose to retain this case in the case base (Figures 4.12 and 4.19). Usually, if the similarity between all the cases in the case base and the query is less than a certain threshold value, then the user should consider retaining the new case. In ADIOP, we suggest using 70% as an initial threshold value, because we need more statistical data to come up with a more meaningful number. This is not in the scope of this research. However, we show in the evaluation section that the 70% value gives good results.

The user then can fill any empty features with values (e.g., the "Update Model" feature). The "Index" feature should give a summary of what the case is all about so as to make it easy to understand in future uses of this case. The user can then store this case in the case base. ADIOP will assign a new sequential number to this case and add it to the case base

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

file.

4.8 Updating CSP Models

ADIOP provides functionality to update the model of a test case that led to a failure caused by incompleteness/incorrectness of this model. The statements on the "Update Model" feature of a case are used for this purpose.

These can be either: 'ADD', 'DEL', or 'UPD' statements: (Figure 4.19)

- An 'ADD' statement adds a new statement (usually a constraint definition) to the CSP model.

  Example: *ADD: $BINARY_CONSTRAINT Hello1A.time <= Hello2A.time.*

  This statement will add the following line in the CSP model:

  $BINARY_CONSTRAINT Hello1A.time <= Hello2A.time # Automated Model Update (statement Addition) using Case: "Case Number" #

- A 'DEL' statement deletes a statement (usually a constraint) from the CSP model if it does exist. This statement is simply commented out so it is easier for the user to know which statements have been deleted.

  Example: *DEL: $BINARY_CONSTRAINT Hello1A.time <= Hello2A.time*

  This statement will comment out the following line in the CSP model:

  // $BINARY_CONSTRAINT Hello1A.time <= Hello2A.time # Automated Model Update (statement Deletion) using Case: "Case Number" #

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- An 'UPD' statement replaces one variable with another in all occurrences an a test case.

  Example: *UPD: Hello1B.peer_group_id Hello2B.peer_group_id.*

  This statement will update all occurrences of Hello1B.peer_group_id by Hello2B.peer_group_id in the CSP model:

  $BINARY_CONSTRAINT Hello1A.peer_group_id == Hello2B.peer_group_id # Automated Model Update (statement Update) using Case: "Case Number" #

For all these statements, a comment using the # Comment # format as stated above is inserted in the CSP model so the user can track down these updates. When updating the model, these statements are applied in the order they are defined. These three statements are sufficient to cover all kinds of updates.

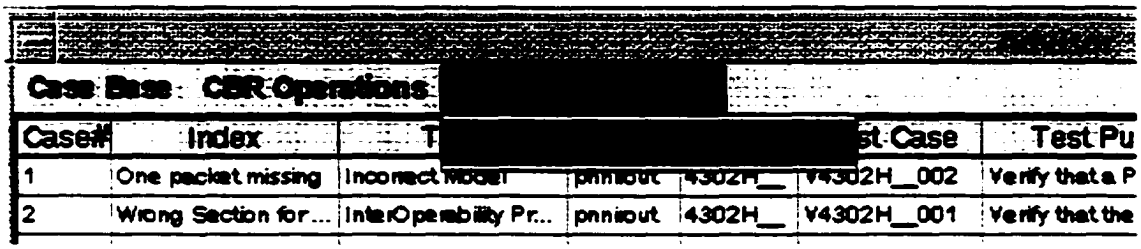

Figure 4.20: Update Test Case Model Menu

When the user chooses to update a CSP model using a case (Figure 4.20), the test suite builder window will appear with the test case model updated using the statements from the "Update Model" feature of the revised/retained case (Figure 4.21).

The user may then update the 'Test Case ID' and the 'Update Version' of the new

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

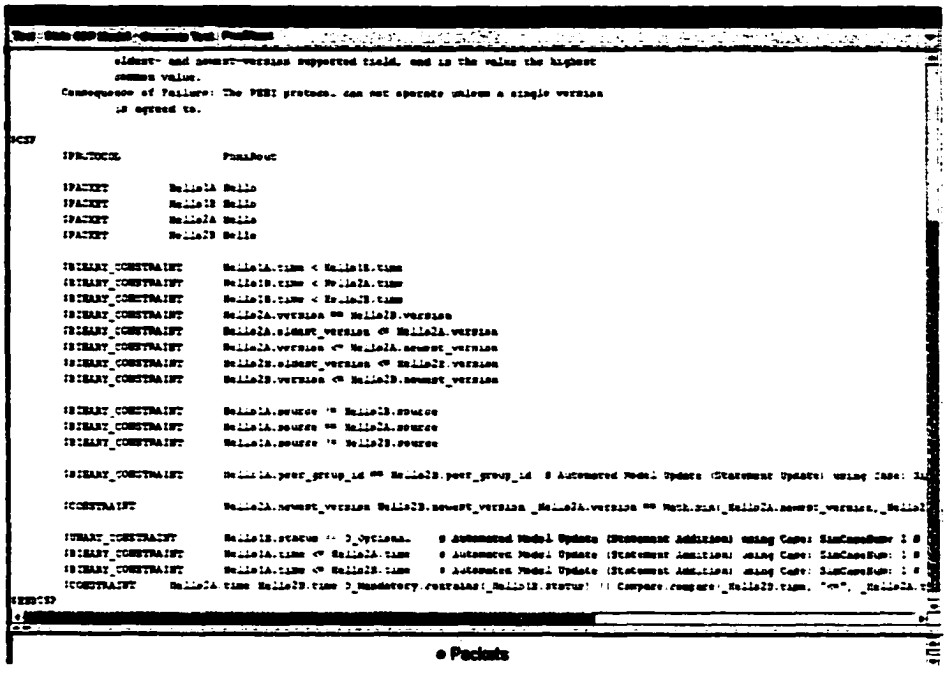

Figure 4.21: Updated Test Case Model

updated test case. The user can then save this new test case, usually in a different file, so that it can be tested for some time before becoming part of the set of test cases that are frequently used. Then the user can generate the test case as an object corresponding to this new updated CSP model by using the "Generate Test From CSP Model" menu item in the Test Suite Builder Window. The new test case will then be available through ADIOP's decoder window (see Chapter 3 for more details on Diagnosis using generated test cases) (Figure 4.22). The outcome of the execution of this updated test case model is shown in Figure 4.23).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

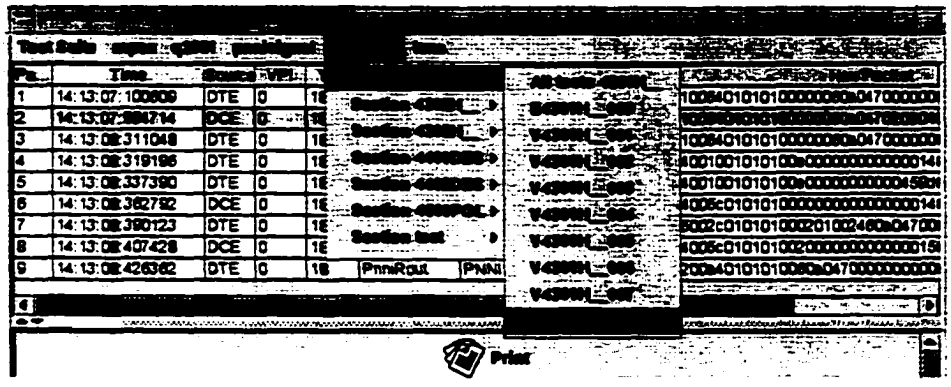

Figure 4.22: Run Updated Test Case

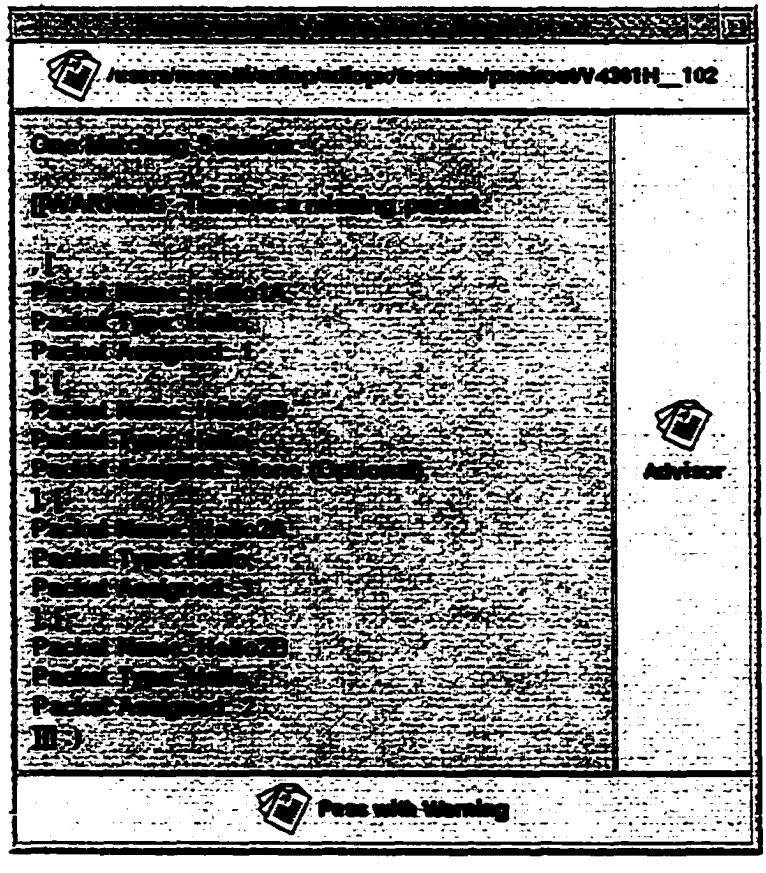

Figure 4.23: Result of Running an Updated Test Case

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.9 Improving Explanations

Explanations generated by ADIOP's Diagnoser may be incorrect, incomplete, or not useful. This can happen because:

- Incorrect CSP models generate *incorrect* explanations
- If inference does not lead to an explanation, the explanation provided by search in case of failure contains the violated constraints and is *not useful*. An example of this is shown in Figure 4.24.
- The explanation provided by ADIOP's Diagnoser may be *incomplete* when only the problem is diagnosed, but no remedy is suggested. Cases can store information about how to resolve the interoperability problem found.

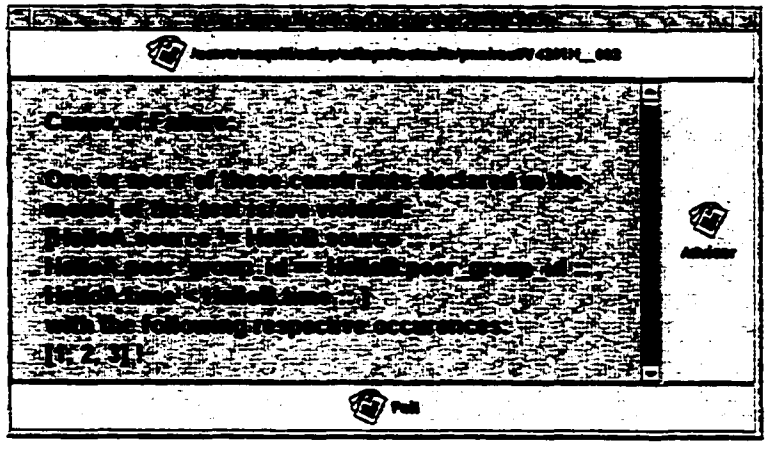

Figure 4.24: Explanation Generated for Test Case V4301H_003

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ADIOP's Advisor provides useful explanations in these cases by retrieving similar previous situations. First, Advisor retrieves similar cases for the above failure as shown in Figure 4.25. This figure shows that Advisor retrieves Case 2 as the most similar case to the new problem with a similarity value of about 81%. Case 2 is stored in the case base shown in Figure 4.12.

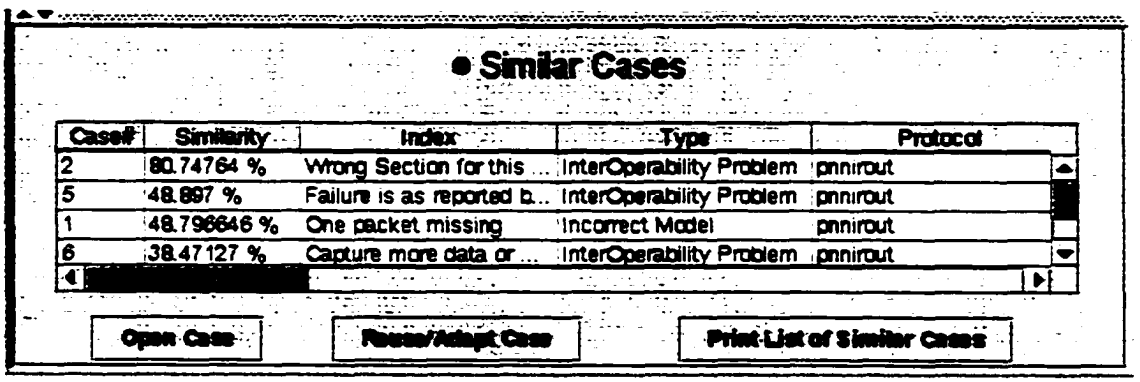

Figure 4.25: Similar Cases for the failure in Test Case V4301H_003

Case 2 is used in this situation, and the new explanation generated for this problem is that **"the data captured is from devices in the same peer group (Section 4301H__). But, the test case run is for devices in different peer groups (Section 4302H__)."** The solution proposed here by ADIOP is to "use test cases from another section (i.e., 4302H__) to run with this captured data."

In this case ADIOP provides an explanation of the interoperability problem and a solution to the problem. Likewise, there are other cases where even when the explanation provided by ADIOP's Diagnoser (through inference methods) is useful, using Advisor allows testers to obtain a solution to the problem by providing them with the actions to be taken

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

after a failure is encountered.

4.10 Experiments and Evaluation

4.10.1 Experiments

In this section, we present an evaluation of the Advisor component of ADIOP. The Advisor is the ADIOP component that integrates the CSP and CBR. We used 10 captured data (observations) to perform this evaluation. These captures are from real-world data obtained at the UNH-IOL. All of these are for the Pnni Routing and Pnni Signaling protocols. We only run test cases that belong to the protocol used when capturing the observations.

All Pnni Routing test cases are taken from (PNNI-IOP 1999) using their actual names in the document. In addition, we used one Pnni Signaling test case to check the basic functionality of this protocol. Only test cases that fail are being used in this evaluation since the Advisor is only called when there is a failure, and we have shown that the Diagnoser is sufficient when test cases pass.

There are a total of 202 test cases that we could run if we used all of the test cases available for all captured data. We have chosen to run only once test cases that produce the same diagnosis in two or more different captures because these will have exactly the same results generated by the Advisor. This leads to running a total of 90 test cases instead.

The case base contains a total of 6 cases learned from running 6 different test cases. We ran the 90 test cases using 1 case (i.e., case 1) in the case base and collected results about how well the Advisor (CBR) performs. Case 1 was selected first because it involves updating

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

models, and we were interested in the performance of this case in debugging models. This, however, should not affect the final results as we use all 6 cases in the final experiment to check the overall performance of Advisor. Then we reran the same test cases while we have 2 cases in the case base, and collected results. Finally, we reran the same test cases while we have all 6 cases in the case base.

Only case 1 is related to model incompleteness and incorrectness. The other cases were learned by the Advisor to correct, complete or confirm the explanation provided by the Diagnoser.

Table 4.1 shows the results obtained for one captured data set. The test cases are taken from (PNNI-IOP 1999) using their actual names in the document. The result of the execution of each test case is shown in the "Res" column.

Table 4.1: Results of Advisor on Capture capt001

Captured Data: capt001.aa					
Test Case	**Res**	**Similar Case**	**Similarity Score**	**Useful Explanation**	**Relevant SimCase**
V4301H_001	Fail	Case 2	**92%**	No	**Yes**
V4301H_002	Fail	Case 2	**77%**	No	**Yes**
V4301H_003	Fail	Case 2	**83%**	No	**Yes**
V4301H_004	Fail	Case 2	**71%**	No	**Yes**
V4301H_005	Fail	Case 5	**94%**	No	No
V4301H_006	Fail	Case 2	**71%**	No	**Yes**
V4301H_007	Fail	Case 2	69%	No	**Yes**
V4401DBS001	Fail	Case 3	**97%**	**Yes**	**Yes**
V4401DBS002	Fail	Case 3	**93%**	**Yes**	**Yes**
V4401DBS003	Fail	Case 3	**94%**	**Yes**	**Yes**
V4601PGL001	Fail	Case 6	**100%**	**Yes**	**Yes**
Vtest001(Signal)	Fail	Case 4	**77%**	**Yes**	**Yes**

The "Similar Case" column contains the case most similar to the actual problem (ac-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cording to ADIOP's Advisor) retrieved from the case base. The "Similarity Score" column states the similarity percentage generated by ADIOP that shows how similar the retrieved case is to the actual problem. The "Useful Explanation" column states whether the explanation provided by the Diagnoser of ADIOP is useful to the tester. This is determined by checking whether the explanation provides the correct diagnosis and is easy to understand by testers. The "Relevant SimCase" column states whether the Similar Case retrieved is relevant to the actual problem. A retrieved case is relevant for the new problem if it provides the correct solution.

Table 4.2 shows the results obtained for 10 captured data sets using 90 test cases. The first column lists the six training cases. Each of the next three columns represents the number of relevant retrieved cases for each case in the case base. The 'Case 1' column shows this number when the case base contains only 'Case 1'. The 'Case 1&2' column shows this number when the case base contains 'Case 1' and 'Case 2'. The 'Cases 1-6' column shows this number when the case base contains all 6 cases.

Table 4.2: Results of Running Test Cases on 10 Captured Data

Summary of 10 Captured Data			
Similar Case	**Case 1**	**Cases 1&2**	**Cases 1-6**
Case 1	12	12	12
Case 2	-	31	27
Case 3	-	-	8
Case 4	-	-	6
Case 5	-	-	6
Case 6	-	-	5
Total	12	43	64
Percentage	13.3%	48%	71%

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 4.26 shows a graph that summarizes the above table.

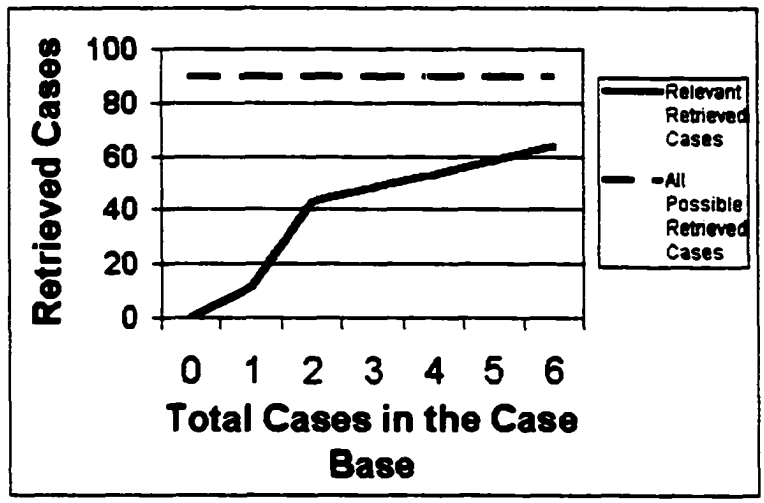

Figure 4.26: Relevant Retrieved Cases

In this dissertation, we have used a single training set. For a thorough evaluation of our system, more training sets should have been used. One issue we had was the limited availability of data from the lab, as we had to rely on testers in the lab to send us such data. Another issue is that testers did not have time to perform an evaluation of this part of ADIOP which could have helped us in getting a better information on the performance of the system.

4.10.2 Solvability

The results obtained above show that as more cases are learned the ADIOP system is able to retrieve more relevant similar cases from the case base. This increases the solvability of problems that were not solved by CSP alone or where the explanation generated by CSP

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

may have been incomplete or incorrect.

Although only one case of the case base is used for debugging models, the results of this section show that if more cases are added to the case base, then the solvability increases including the solvability for debugging models.

The main difference between cases related to model debugging and cases related to the explanation of interoperability problems is the 'Update Model' field, which allows the updating of a test case.

4.10.3 Evaluation of the CBR system

Precision and Recall

(Daniels & Rissland 1997) state that most retrieval systems are judged on the basis of precision and recall. These measure what percentage of the retrieved items are relevant (precision) and what percentage of the relevant items are retrieved (recall), respectively. A retrieved case is relevant for the new problem if it provides the correct solution. Recall is computed by dividing the relevant retrieved cases by the relevant cases. Precision is computed by dividing the relevant retrieved cases by the retrieved cases. The higher the values of precision and recall, the better the system is in retrieving relevant cases.

For the experiments we conducted (90 test cases executed, and 6 cases in the case base), the following results were obtained:

- Relevant retrieved cases = 64
- Relevant cases = 73

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Retrieved cases = 90
- **Recall = (Relevant retrieved cases)/(Relevant cases) = 64/73 = 88%**
- **Precision = (Relevant retrieved cases)/(Retrieved cases) = 64/90 = 71%**

These numbers show that the CBR is retrieving relevant cases in most situations. We do not claim that these numbers are sufficient to give us a final answer on the performance of our system, but it is an indication that the pattern of behavior is the one we expected. More evaluation has to be done in addition to what we have performed on our prototype to validate this claim.

Similarity Measures

We have used 70% as the threshold for the tester to decide whether the case retrieved by the Advisor as the most relevant is a correct assertion. Out of the 64 relevant retrieved cases, 51 had a similarity percentage of more than 70%. 13 had a similarity percentage of less than 70%. The average similarity percentage value of these 13 cases is 61%. This shows that for 80% of the cases, the Advisor made the correct decision for the tester, and in 20% of cases, it made the incorrect decision and was in average off by less than 10% of the expected 70%.

Out of the 26 non-relevant retrieved cases, 14 had a similarity percentage of less than 70%, and 12 had a similarity percentage of more than 70%. The average similarity percentage value of these 12 cases is 79%. This shows that for 54% of cases, the Advisor made the correct decision for the tester, and in 46% of cases, it made the incorrect decision and was

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

on average off by less than 10% of the expected 70%.

If we combine both numbers from all the retrieved cases (relevant and non-relevant), then out of 90 total retrieved cases, the Advisor made the correct decision for the tester in 65 cases, that is 72%, and made the incorrect decision for the tester in 25 cases, that is 28% but was only off by 10% on average.

So in summary, the value of 70% seems to be realistic and the fact that it was off by plus 10% in one case and minus 10% in the other case shows that it is the closest value to a real threshold. More evaluation can be performed, including the analysis of more data, to be able to make a more informed decision on this threshold. For the purpose of our prototype it shows that the idea presented in this dissertation is viable.

4.10.4 Evaluation of Explanation Improvement

In this section, we investigate the performance of Advisor and how it improves performance compared with using only the ADIOP Diagnoser.

Table 4.3 shows that, out of 54 test cases with non-useful explanations, 33 can be explained using the Advisor by retrieving a relevant case from the case base. This means that more than in 60% of test cases for which the Diagnoser did not give a useful explanation the Advisor provided an explanation. This is an improvement over the Diagnoser in generating useful explanations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 4.3: Useful Explanation vs. Relevant Retrieved Cases

Useful Explanation vs. Relevant Retrieved Cases for 90 test cases			
Relevant Retrieved Cases / Useful Explanation	No	Yes	Total
No	21	33	54
Yes	5	31	36
Total	26	64	90

4.10.5 Model Updates

Case 1 was used to update 12 test case models. Most of the problems we found were related to interoperability problems where the explanation given is incorrect or insufficient. The case base was then used to store cases that are used to complete and correct these explanations in addition to a case related to model debugging.

We found only one case that is related to model incompleteness and incorrectness. Our analysis of the problems found is not that of an expert and a more in-depth analysis of the problems has to be done to extract the real problem of incompleteness and incorrectness in models. However. our experiment with one type of incompleteness and incorrectness has proven to be successful.

The evaluation of the Advisor was performed using different kinds of cases, one of which is related to model debugging. The types of cases used does not affect the results of this evaluation but rather these results are influenced by the CBR structure and implementation of the components. This is true because the evaluation shows that there is no difference between case 1 and other cases in terms of performance and outcome of the Advisor. It

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is also true because the main difference between the two types of cases presented in this dissertation is the 'Model Update' feature.

However, more evaluation needs to be done for a longer period of time involving many captures, protocols, etc, to validate these claims with a stronger assertion.

4.11 Related Work

(Karamouzis & Feyock 1992) show that the integration of CBR and MBR enhances CBR by the addition of a model that aids the processes of matching and adaptation, and it enhances MBR by the CBR capacity to contribute new links into the causality model. In this dissertation, the result obtained from the CBR process is used to update models. This is similar to what has been done in (Karamouzis & Feyock 1992) for integrating CBR and MBR to update causality models. The difference is that we are using CSP models, taking advantage of the CSP representation and applying that to the interoperability testing domain.

In (Huang & Miles 1996), CBR was used to enhance CSP in problems characterized by large cardinality, and heavy database searches. In this paper, CBR was mainly used to reduce the search space. In this dissertation, the integration is used to debug CSP models and improve explanations. In our case, the CSP models are small and the search space is manageable through search and inference. Our main concern however was to generate useful explanations for interoperability testing.

(Bartsch-Sporl 1995) presents a way to bridge CBR and MBR by using schema-based

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reasoning. A case is enhanced by adding to it generic knowledge (rules and constraints). In this dissertation, cases include information about updating models using a CSP representation. We can imply from both findings that constraints can improve on the CBR representation of cases.

In (Purvis & Pu 1995), case adaptation in assembly planning problems was formalized as a CSP. Each case is represented as a primitive CSP, and then a CSP algorithm is applied to combine these primitive CSPs into a globally consistent solution for the new problem. CBR is used to fill in the values of the problem, then CSP is used to make the problem consistent. In this work, CBR is supported by CSP while in ours CBR supports CSP. In both, CSP was used in the representation of cases. Depending on the domain of application, CBR or CSP will be more appropriate to start with. In our case, CSP provides models for test cases that we have shown are easy to create and use for diagnosis. These models and their results are then improved by the use of CBR.

(Bilgic & Fox 1996) present the case-based retrieval for engineering design as a set of constraints. They state that knowledge, constraints and goals change over time. In this work also, CSP supports CBR by using constraints for case-based retrieval.

(Portinale & Torasso 1995) stated that approaches combining MBR and CBR can be roughly classified into two categories: approaches considering CBR as a speed-up and/or heuristic component for MBR, and approaches viewing CBR as a way to recall past experience in order to account for potential errors in the device model. Their proposal was in the first category by means of the development of ADAPtER, a diagnostic system integrating

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the model-based inference engine to AID (a pure model-based diagnostic system), with a case-based component intended to provide a guide to the abductive reasoning performed by AID. In this work, the CBR supports MBR, which is used for modeling device behavior. In our case, CBR supports CSP, which is used for modeling the interaction between devices. The choice between MBR and CSP depends on what type of diagnosis is performed. In (Van Someren, Surma, & Torasso 1997), CBR is used as a form of "caching" solved problems to speedup later problem solving. The approach taken is to construct a "cost model" of a system that can be used to predict the effect of changes to the system. Their CBR-MBR architecture is essentially the one used previously in ADAPtER. They state that in general model-based diagnosis is very expensive from a computational point of view since the search space is very large.

(Lee *et al.* 1997) developed a case and constraint based expert system for project planning of an apartment domain. This large scale, case-based, and mixed initiative planning system integrated with intensive constraint-based adaptation utilizes semantic level meta-constraints and human decisions in order to compensate for incomplete cases embedding specific planning knowledge. The case and constraint based architecture inherently supports cross-checking cases with constraints during the system development and maintenance. In this work, CSP supports CBR by compensating for incomplete cases. In our work, CBR compensate for incomplete CSP models. The choice of the type of integration is again driven by the application and the structure of the problem.

(Hastings, Branting, & Lockwood 1995) describe a technique for integrating CBR and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

MBR to predict the behavior of biological systems characterized both by incomplete models and insufficient empirical data for accurate induction. They suggest using multiple, individually incomplete, knowledge sources to accurately predict the behavior of such systems. They state that precise models exist for the behavior of many simple physical systems. However, models of biological, ecological, and other natural systems are often incomplete, either because a complete state description for such systems cannot be determined or because the number and type of interactions between system elements are poorly understood. In their paper, MBR is mainly used to determine values for variables in cases, and to compute new values from old cases' values. MBR is used to adapt cases (MBR is used within the CBR formalism). In this work MBR supports CBR in the adaptation of cases, which is different from what drives our application where CBR supports CSP.

In (Marrero, Clarke, & Jha 1997), Model Checking is used for verifying hardware designs, security protocols, and other components. By modeling circuits or protocols as finite-state machines, and examining all possible execution traces, model checking is used to find errors in real world designs. This work uses finite-state machines for representation, which we have shown in (Sqalli & Freuder 1996a) to be less expressive than CSPs. The way the model is checked is also different from what we do; we take an instance and check whether it is consistent, while in model checking the whole space is searched to check if there is an inconsistent instance.

Our focus is to automate interoperability testing and show how we can get better results by enhancing the CSP model with the case-base reasoner. First, CSP is used to solve the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

problem. If the CSP model is insufficient, then CBR is used. This way CBR will not be used unless CSP fails. The result obtained from the CBR process is then used to update the model.

In many of the publications reviewed in this section, we found that CSP supports CBR. This dissertation covers the other type of integration where CBR supports CSP and which did not receive as much attention as the other type. CBR provides support for CSP model debugging and explanation improvement. In addition, we have used CSP to partially support CBR in the representation of cases.

4.12 Summary

In this chapter we presented a taxonomy of types of model incompleteness and incorrectness and how to fix and debug one of these types. We then presented the CBR system and its integration with CSP to debug and update test case models and compensate for incompleteness and incorrectness.

We presented an example throughout the different sections to show how this works. An evaluation of the Advisor component of ADIOP, which integrates CBR and CSP was performed. The results show that this improves on the Diagnoser component of ADIOP.

Even though our original goal for the Advisor component was mainly to debug models, we were able to achieve more through the integration of CBR and CSP. We showed that models can be updated efficiently by the Advisor. The Advisor helps the testers identify more incorrect and incomplete models. It also improves on the Diagnoser performance and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

generates better explanations.

The cases used in the CBR system of the Advisor are of different types and may include any information that can be stored to help the tester learn from experience in new similar situations.

Actually, there are two kinds of failures stored in the case base. The first is related to incomplete or incorrect CSP models. These CSP models are eventually updated through the Advisor component. The second kind of failures are those due to interoperability problems in the devices being tested. The purpose of using the Advisor here is to get a useful, correct and complete explanation of the cause of failure of a test case.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5

Conclusion

In this chapter, we conclude this dissertation by highlighting the contributions made and outline directions for future research for the three main chapters. In this dissertation, we presented a proof-of-concept of how CSP is used to successfully model test cases and diagnose interoperability problems, and how CBR supports debugging of CSP models and the generation of useful explanations for interoperability testing.

5.1 CSP Modeling

The main contribution in Chapter 2 is the definition of a new modeling language using CSP and OOP. This language is simple, declarative, transparent. It provides an automated tool for testers to implement interoperability test cases. We introduced the notions of metavariables, metavalues and optional metavariables to improve the modeling language capabilities. We proposed to model test cases defined in test suite specifications. These test suites are manually written by individuals or organizations. They break down testing into modules and make diagnosis of problems more meaningful to testers and lab customers. We have used this break down in ADIOP to benefit from the advantages it provides.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Open Research Issues:

- The modeling language defined in this dissertation includes how to state metavariables and constraints. One issue that is raised here is the level of completeness of this language to model different test cases' requirements. We use the notion of 'general constraint', which allows testers to state any constraint that they are not able to state using the other predefined utilities for Unary and Binary constraints. This can be improved by looking at the general constraints used in practice by testers and add them to the predefined utilities to provide a simpler definition for similar constraints definition. This will increase the capabilities of the modeling language and will provide testers with an even easier interface for modeling test cases.

- Modeling is domain-independent as it is possible to model test cases using different types of packets from different ATM protocols. This can be advanced further to cover new domains such as planning and scheduling where tasks, subtasks and optional tasks can be defined as metavariables, variables, and optional metavariables respectively.

- Many test cases are defined in an incremental fashion. So, some test cases can be modeled starting from others instead of starting from scratch. This introduces the idea of hierarchy in model definition. The use of this type of hierarchy to model test cases will save time and space since new test cases can be modeled based on other existing ones.

- The use of OOP adds many advantages to CSP modeling. We need to investigate

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

whether there are other aspects of OOP that were not discussed in this dissertation that can benefit CSP modeling.

5.2 Constraint-Based Diagnosis

The major contribution of Chapter 3 is to diagnose interoperability problems using search supplemented by consistency inference methods in a CSP context to explain problem solving behavior. These methods were also adapted to the OO-based CSP context. Testers can then generate reports for individual test cases and for test groups, from a test suite specification, that are useful for UNH-IOL customers. We also presented a decoder that provides utilities for decoding data captured on different analyzers. This makes the diagnosis available for a range of analyzers.

Open Research Issues:

- We use specialized inference methods to provide more meaningful explanations to testers. Testers stated that the explanation generated by ADIOP are not always useful. We can explore the use of other types of inferences by looking at the structure of test cases and types of explanations for interoperability problems that testers generate manually.

- The explanation generated in ADIOP uses templates that we developed from some interoperability problems. We can look at the possibility to generate these templates from some cases stored in the case base where a tester may have added a new type of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

explanation.

- We have adapted CSP algorithms to OOP. We may further investigate how to benefit from the structure of test cases modeled using OO-based CSP to develop new algorithms for improving problem solving mechanisms and the generation of human-like explanations.

- Partial CSP (PCSP) (Freuder & Wallace 1992) can be used to solve over-constrained problems by allowing the violation of some constraints. In the case when search and inference fails to provide a solution, PCSP can be used to detect, in some instances, the constraint that is violated and that may be the cause of failure of the test case being used. This will provide an explanation to testers and is mainly useful even when CBR does not provide this explanation.

- For decoding, we have to implement decoders for the different packets being used for automated testing. To minimize the overhead of this task, it will be more efficient to define a language where a high level definition of the packets used is provided and from which decoders for these packets are generated. Natural language processing can make this more interesting by using the parts of the protocol specifications that define the different packets as an input for generating decoders for different packets of this protocol.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.3 CSP Model Debugging

The major contribution of Chapter 4 is to detect and debug incompleteness and incorrectness in CSP models of interoperability test cases. This is done through the integration of two modes of reasoning, namely CBR and CSP. CBR manages cases that store information about updating models as well as cases that are related to interoperability problems where diagnosis fails to generate a useful explanation. In the latter, CBR recalls previous similar useful explanations.

Open Research Issues:

- The adaptation of cases is mainly done manually by testers. This can be improved by automating or semi-automating the adaptation phase of CBR. The tester can then get more useful solutions to the problem they have with less intervention into the process. This is a first step into automating the model updating process.
- Similarity metrics including distance measurements and weights can be improved further by checking results obtained from testers using different values.
- The integration of CSP and CBR was used in a specific way. There is a need to investigate other possibilities of integration in this domain that will allow us to make more informed decisions about the best way to integrate these two paradigms.
- The case base used in this dissertation stores two types of cases, the ones related to model debugging and the ones related to the explanation of interoperability problems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We can take further advantage of this case base by storing other types of cases that testers may want to recall later.

- The updating of models is done on a one-on-one basis when an inconsistency is detected in a model. We want to investigate the possibility to debug and update models that have not been used yet for testing but that we suspect of being inconsistent because of their similarity to other models that were found to be inconsistent. This will be a more pro-active form of model debugging compared to the reactive model debugging that we present in this dissertation.

- The case base used in this dissertation was small but improved considerably the results obtained. If testers decide to store many more cases in this case base without control of what is being stored, then how this will affect the CBR results, including updating models?

- We have shown that the structure of CSP/CBR integration we used is valid for the interoperability domain investigated in this dissertation. One question is to what extent this is valid for other domains of application, and what updating of models means in other contexts.

- We have not explored the synergies that may exist in the area of CBR-OOP integration in this dissertation. More work needs to be done in this area.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.4 Directions for Future Work

There are some challenges for making the three main areas presented in this dissertation accessible to other applications. We need to show the usefulness of these challenges in the interoperability testing domain as well as in other real-world applications.

- **Automated Model Acquisition:** In this dissertation, we automated a major part of the process of model acquisition. Once testers have a high level understanding of the test case description, they can state it in term of CSP modeling language. However, testers have to read test cases from test suite specifications and define models of these before using the modeling language to model test cases. This modeling is partly subjective since testers have to define what they understand about a test case. This can be improved by automating further the process of model acquisition. A graphical interface that shows the CSP graphs (i.e., nodes and vertices) of models generated can provide testers with a more useful tool for building these models.

- **Inference-Based Explanation:** We have used a few inferences to generate useful explanations to testers. There is a need to investigate further the inference-based algorithms and to come up with a general framework for detecting different types of inferences that will generate useful explanations.

- **Model Debugging and Learning:** The automation of more CBR tasks including adaptation can further automate the model debugging process. CSP models can then be debugged and updated with less human interaction. The goal is to obtain robust

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CSP models that have fewer inconsistencies through this learning process.

- **On-line Diagnosis:** In this dissertation, we investigated off-line diagnosis since data is captured on-line through different analyzers, saved into a file and then decoded and diagnosed by ADIOP. We can investigate the possibility of performing on-line diagnosis by integrating ADIOP with the different analyzers. This will provide real-time diagnosis of interoperability problems.

- **Client/Server Architecture:** We developed ADIOP as a stand-alone application using the Java language. One tester who evaluated ADIOP suggested that the use of the client/server architecture will make interoperability testing using ADIOP even more useful for the UNH-IOL. Because Java provides the framework for client/server architecture, we believe that it can be adapted to this environment. Also, we need to investigate the implication of using this architecture on all the areas presented in this dissertation.

5.5 Conclusion

Modeling, Diagnosis, and Model Debugging are the three main areas presented in this dissertation to automate the process of interoperability testing of networking protocols. In this dissertation, we presented a framework that uses CSP to define a modeling language and problem solving mechanism for interoperability testing, and uses CBR for debugging models of interoperability test cases.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We defined a new modeling language using CSP and OOP. It provides an automated tool for testers to implement models of interoperability test cases.

We presented how to diagnose interoperability problems using search supplemented by consistency inference methods in a CSP context to generate explanations of the problem solving behavior.

We discussed how detecting and debugging incompleteness and incorrectness in CSP models is performed using an integration of two modes of reasoning, namely CBR and CSP.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

References

Aamodt, A., and Plaza, E. 1994. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications* 7(i):39–59.

Aamodt, A. 1991. A Knowledge-Intensive Approach to Problem Solving and Sustained Learning. Technical Report PUB 92-08460, Ph.D. Dissertation, May 1991, University of Trondheim, Norwegian Institute of Technology.

Abu-Hakima, S. 1988. *RATIONALE: A Tool for Developing Knowledge-Based Systems that Explain by Reasoning Explicitly.* Ottawa, Canada: Masters Thesis, Carleton University.

Abu-Hakima, S. 1993. A perspective on explanation in diagnosis. In *Proceedings of the IJCAI Workshop on Explanation and Problem Solving*, 77–87.

Abu-Hakima, S. 1994. Automating Model Acquisition by Fault Knowledge Re-use, DR, the Diagnostic Remodeler Algorithm. In *International Workshop on the Principles of Diagnosis*, 1–6.

Atlee, J. M. 1992. *Automated Analysis of Software Requirements.* Ph.D. Thesis, Department of Computer Science, University of Maryland.

ATMF-TestSpec. 1994. *Introduction to ATM Forum Test Specifications.* The ATM Forum. Technical Committee. AF-TEST-0022.000.

Avesani. P.; Perini, A.; and Ricci, F. 1993. Combining CBR and Constraint Reasoning in Planning Forest Fire Fighting. In *In Proceedings of 1st European Workshop on Case-Based Reasoning, Kaiserslautern.*

Bareiss, R. 1988. PROTOS; a Unified Approach to Concept Representation, Classification and Learning. Technical Report AI88-83, Ph.D. Dissertation, University of Texas at Austin, Dep. of Computer Sciences.

Bartsch-Sporl, B. 1995. Towards the Integration of Case-Based, Schema-Based and Model-Based Reasoning for Supporting Complex Design Tasks. In Veloso, M., and Aamodt, A., eds., *Topics in Case Based Reasoning, Proceedings of the First International Conference on Case Based Reasoning, LNAI Series*, 145–156. Springer Verlag.

Bergmann, R.; Breen, S.; Goeker, M.; Manago, M.; and Wess, S. 1999. *Developing Industrial Case-Based Reasoning Applications: The INRECA-Methodology. (LNAI-1612).* Springer.

Bilgic, T., and Fox, M. S. 1996. Constraint-Based Retrieval of Engineering Design Cases: Context as constraints. *Artificial Intelligence in Design* 269–288.

Booch, G. 1994. *Object-Oriented Analysis and Design with Applications, 2nd Ed.* Benjamin Cummings.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Branting, K. 1991. Exploiting the Complementarity of Rules and Precedents with Reciprocity and Fairness. In *Proceedings from the Case-Based Reasoning Workshop. Sponsored by DARPA*, 39–50. Washington DC, USA: Morgan Kaufmann.

Campione, M., and Walrath, K. 1998. *The Java Tutorial, Second Edition: Object-Oriented Programming for the Internet (Java Series)*. Addison-Wesley Pub Co.

Coad, P., and Yourdon, E. 1991. *Object-Oriented Analysis, 2nd ed.* Prentice Hall.

Damashek, M. 1995. Gauging Similarity with n-Grams: Language-Independent Categorization of Text. *Science* 267:843–848.

Daniels, J. J., and Rissland, E. L. 1997. What You Saw Is What You Want: Using Cases to Seed Information Retrieval. In Leake, D. B., and Plaza, E., eds., *Case-Based Reasoning Research and Development: Second International Conference on Case-Based Reasoning, ICCBR-97 (LNAI-1266)*. 325–336.

Dechter, R., and van Beek, P. 1995. Local and global relational consistency. In *Principles and Practice of Constraint Programming - CP '95, Montanari and Rossi, eds. LNCS 976*, 240–257. Springer.

Fattah, Y. E., and Dechter, R. 1992. Empirical Evaluation of Diagnosis as Optimization in Constraint Networks. In *Working Papers of the Third International Workshop on Principles of Diagnosis (DX-92)*.

Freuder, E., and Mackworth, A. 1992. Constraint-Based Reasoning, Special Volume. *Artificial Intelligence* 58.

Freuder, E., and Wallace, R. 1992. Partial Constraint Satisfaction. *Artificial Intelligence* 58:21–70.

Freuder, E. 1978. Synthesizing constraint expressions. *Communications of the ACM* 21:958–966.

Hamscher, W., and Struss, P. 1990. Model-Based Diagnosis. In *AAAI-90 Tutorial Notes, Eighth National Conference of Artificial Intelligence*. 1–179.

Hastings, J. D.; Branting, L. K.; and Lockwood, J. A. 1995. Case Adaptation Using an Incomplete Causal Model. In Veloso, M., and Aamodt, A., eds., *Topics in Case Based Reasoning, Proceedings of the First International Conference on Case Based Reasoning, LNAI Series*. 181–192. Springer Verlag.

Huang, Y., and Miles, R. 1996. Using Case-Based Techniques to Enhance Constraint Satisfaction Problem Solving. *Applied Artificial Intelligence, an International Journal* 10(4):307–328.

Huffman, S. 1995. Acquaintance: Language-Independent Document Categorization by N-Grams. In Harman, D. K., and Voorhees, E. M., eds., *Proceedings of TREC-4, 4th Text Retrieval Conference*, 359–371.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Karamouzis, S. T., and Feyock, S. 1992. An Integration of Case-Based and Model-Based Reasoning and its Application to Physical System Faults. In Belli, F., and (Eds.), F. R., eds., *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Lecture Notes in Artificial Intelligence 604*. Springer-Verlag.

Kitano, H., and Shimazu, H. 1996. The Experience-Sharing Architecture: A Case Study in Corporate-Wide Case-Based Software Quality Control. In Leake, D. B., ed., *Case-Based Reasoning: Experiences, Lessons and Future Directions*, 235–268.

Kolodner, J. 1983. Maintaining Organization in a Dynamic Long-term Memory. *Cognitive Science* 7:243–280.

Koton, P. 1989. Using Experience in Learning and Problem Solving. Technical Report MIT/LCS/TR-441, Ph.D. Dissertation, October 1988, Massachusetts Institute of Technology, Laboratory of Computer Science.

Kumar, V. 1992. Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine* 13(4):32–44.

LANE-1.0. 1995. *LAN Emulation Over ATM, Version 1.0.* The ATM Forum, Technical Committee. af-lane-0021.000.

Leake, D. B. 1996. *Case-Based Reasoning: Experiences, Lessons, and Future Directions.* AAAI Press.

Leckie, C. 1995. Experience and Trends in AI for Network Monitoring and Diagnosis. In *Proceedings IJCAI-95 Workshop on AI in Distributed Information Networks.*

Lee, K. J.; Kim, H. W.; Lee, J. K.; Kim, T. H.; Kim, C. G.; Yoon, M. K.; Hwang, E. J.; and Park, H. J. 1997. Case and Constraint Based Apartment Construction Project Planning System: FASTrak-APT. In *Proceedings of IAAI-97.*

Lenz, M.; Hubner, A.; and Kunze, M. 1998. Textual CBR. In Lenz, M.; Bartsch-Sporl, B.; Burkhard, H.-D.; and Wess, S., eds., *Case-Based Reasoning Technology: From Foundations to Applications (LNAI)*, volume 1400, 115–137. Springer.

Likert, R. 1932. A Technique for the Measurement of Attitudes. *Archives of Psychology* 140(June).

Maher, M. L.; Balachandran, M. B.; and Zhang, D. M. 1995. *Case-Based Reasoning in Design.* Lawrence Erlbaum.

Marrero, W.; Clarke, E.; and Jha, S. 1997. Model Checking for Security Protocols. Technical Report CMU-CS-97-139, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213.

Mittal, S., and Falkenhainer, B. 1990. Dynamic Constraint Satisfaction Problems. In *AAAI90*, 25–32.

MPOA-1.0. 1997. *Multi-Protocol Over ATM, Version 1.0.* The ATM Forum, Technical Committee. af-mpoa-0087.000.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NHRP. 1998. *NBMA Next Hop Resolution Protocol.* Network Working Group, Request for Comment: 2332. RFC 2332.

Novak, F.; Mozetic, I.; Santo-Zarnik, M.; and Biasizzo, A. 1993. Enhancing Design-for-Test for Active Analog Filters by Using CLP(R). *Journal of Electronic Testing: Theory and Applications* 4:315–329.

Paltrinieri, M. 1994a. On the Design of Constraint Satisfaction Problems. In *Principles and Practice of Constraint Programming, Second International Workshop (PPCP94) - Lecture Notes in Computer Science Vol. 874: Alan Borning (Ed.)*, 299–311. Rosario, Orcas Island, Washington, USA: Springer.

Paltrinieri, M. 1994b. Visual Environment for Constraint Programming. In *11th International Symposium on Visual Languages*, 118–119.

Plaza, E., and Arcos, J. L. 1993. Reflection and Analogy in Memory-Based Learning. In *Proc. Multistrategy Learning Workshop*, 42–49.

Plaza, E., and Lopez de Mantaras, R. 1990. A Case-Based Apprentice that Learns from Fuzzy Examples. In Ras, Z.; Zemankova, M.; and Emrich, M. L., eds., *Methodologies for Intelligent Systems*, 420–427.

PNNI-1.0. 1996. *Private Network-Network Interface Specification Version 1.0 (PNNI 1.0).* The ATM Forum, Technical Committee. af-pnni-0055.000.

PNNI-IOP. 1999. *Interoperability Test for PNNI Version 1.0.* The ATM Forum, Technical Committee. AF-TEST-CSRA-0111.000.

Portinale, L., and Torasso, P. 1995. ADAPtER: An Integrated Diagnostic System Combining Case-Based and Abductive Reasoning. In Veloso, M., and Aamodt, A., eds., *Topics in Case Based Reasoning, Proceedings of the First International Conference on Case Based Reasoning, LNAI Series*, 277–288. Springer Verlag.

Puget, J.-F., and Leconte, M. 1995. Beyond the Glass Box: Constraints as Objects. In *Logic Programming, Proceedings of the 1995 International Symposium (ILPS): John W. Lloyd (Ed.)*, 513–527. Portland, Oregon: MIT Press.

Purvis, L., and Pu, P. 1995. Adaptation Using Constraint Satisfaction Techniques. In Veloso, M., and Aamodt, A., eds., *Topics in Case Based Reasoning, Proceedings of the First International Conference on Case Based Reasoning, LNAI Series*, 289–300. Springer Verlag.

Richter, A. M., and Weiss, S. 1991. Similarity, Uncertainty and Case-Based Reasoning in PATDEX. In Boyer, R. S., ed., *Automated Reasoning, Essays in Honour of Woody Bledsoe*, 249–265. Kluwer.

Riese, M. 1993a. Diagnosis of Communicating Systems: Dealing with Incompleteness and Uncertainty. In *Proceedings IJCAI-93*, 1480–1485.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Riese, M. 1993b. Diagnosis of Extended Finite Automata as a Constraint Satisfaction Problem. In *Proceedings of the Fourth International Workshop on Principles of Diagnosis (DX-93)*, 60–73.

Roy, P., and Pachet, F. 1997. Reifying Constraint Satisfaction in Smalltalk. *Journal of Object-Oriented Programming* 10(4):51–63.

Sabin, D., and Freuder, E. 1996. Configuration as Composite Constraint Satisfaction. In *Proceedings of the AI and Manufacturing Research Workshop.*

Sabin, D.; Sabin, M.; Russell, R.; and Freuder, E. 1994. A constraint-based approach to diagnosing distributed software systems. In *Proceedings of the Fifth International Workshop on Principles of Diagnosis (DX-94).*

Sabin, D.; Sabin, M.; Russell, R.; and Freuder, E. 1995a. A constraint-based approach to diagnosing configuration problems. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-95).*

Sabin, D.; Sabin, M.; Russell, R.; and Freuder, E. 1995b. A constraint-based approach to diagnosing software problems in computer networks. In *Proceedings of Principles and Practice of Constraint Programming (CP-95).*

Schank, R. 1982. *Dynamic Memory: A Theory of Learning in Computers and People.* New York: Cambridge University Press.

Sqalli, M., and Freuder, E. 1996a. A Constraint Satisfaction Model for Testing Emulated LANs in ATM Networks. In *Proceedings of the 7th International Workshop on Principles of Diagnosis (DX-96).* 206–213.

Sqalli, M., and Freuder, E. 1996b. Inference-Based Constraint Satisfaction Supports Explanation. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, 318–325.

Sqalli, M., and Freuder, E. 1998. Diagnosing InterOperability Problems by Enhancing Constraint Satisfaction with Case-Based Reasoning. In *Working Papers of the Ninth International Workshop on Principles of Diagnosis (DX-98)*, 266–273.

Sqalli, M., and Freuder, E. 2001a. Constraint-Based Modeling of InterOperability Problems using an Object-Oriented Approach. In *Proceedings of the Thirteenth Annual Conference on Innovative Applications of Artificial Intelligence (IAAI-01).*

Sqalli, M., and Freuder, E. 2001b. Solving InterOperability Problems Using Object-Oriented CSP Modeling. Technical report, IJCAI, Seattle, Washington, USA.

Sqalli, M.; Purvis, L.; and Freuder, E. 1999. Survey of Applications Integrating Constraint Satisfaction and Case-Based Reasoning. In *PACLP99: The First International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming.*

Stone, N. D. 1995. Object-Oriented Constraint Satisfaction Planning for Whole Farm Management. *AI Applications* 9(1).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Trochim, W. M. 2000. *The Research Methods Knowledge Base.* Atomic Dog Publishing, Cincinnati, OH. or Internet WWW page, at URL: <http://trochim.himan.cornell.edu/kb/index.htm>, 2nd edition.

UNI-3.1. 1994. *User-Network Interface (UNI) Specification, Version 3.1.* The ATM Forum, Technical Committee.

UNI-4.0. 1996. *ATM User-Network Interface (UNI), Signalling Specification, Version 4.0.* The ATM Forum, Technical Committee. af-sig-0061.000.

Van Hentenryck, P.; McAllester, D.; and Kapur, D. 1995. Solving Polynomial Systems using a Branch and Prune Approach. *SIAM Journal on Numerical Analysis.*

Van Someren, M.; Surma, J.; and Torasso, P. 1997. A Utility-based Approach to Learning in a Mixed Case-Based and Model-Based Architecture. In *Proceedings of the Second International Conference on Case Based Reasoning.*

Wallace, M. 1996. Practical Applications of Constraint Programming. *Constraints - An International Journal* 1(1-2):139–168.

Weigel, R., and Faltings, B. V. 1998. Interchangeability for Case Adaptation in Configuration Problems. Technical Report SS-98-04, AAAI, Stanford University.

Winston, P. 1975. Learning Structural Descriptions from Examples. In *The Psychology of Computer Vision.* McGraw-Hill.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix A

Test Case Layout

This test case layout is taken from (PNNI-IOP 1999)):

Each Test case has the following parts: Test Case ID, Test Purpose, Reference, Prerequisite, Test Configuration, Test Set- up, Test Procedure, Verdict Criteria, and Consequence of Failure*.

Test Case ID:
This is the test case identifier. Layout is ABBBBCCCDDD. The following table provides detailed information.

Test Case Identification Layout and Description		
Positions	**Meaning**	**Current Values**
A	Type of test	V=Valid or E=error
BBBB	Section number in this document	See this document
CCC	Abbreviated description of the protocol or part of protocol being tested	H__= Hello, DBS= DataBase Synchronization, FLD= Flooding, PGL= Peer Group Leader Election, BPI= Border Node PGL Interactions, LGN= Logical Group Node, EST= Call Establishment, REL= Release call, CRK= Crankback, DTL= Designated Transit List, RST= Restart
DDD	Number of the test case within the particular section	See this document

Test Purpose:
Defines the reason for running the test.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reference:
The section from the PNNI v1.0 Specification that supports this test case.

Pre-requisite:
Listed is the information that must be known before the test can be run.

Test Configuration:
Lists which of the test configurations should be used.

Test Set-up:
Describes the devices and physical connections needed for this test.
NOTE: The term connect two devices does not necessarily imply that the systems are on and operational when the physical connection is made. The test being run will determine the situation.

Test Procedure:
Lists the steps necessary to carry out this test. Items in parenthesis, "()", mean that the item occurs at either the A or C monitoring point. Items in brackets, "[ ]", provide necessary information on coding of messages or information elements.

Verdict Criteria:
Lists the observations that must occur in order for this test case results to be successful (i.e. satisfy the Test Purpose). Items in parenthesis, "()", mean that the item occurs at either the A or C monitoring point. It is given here as additional information, but is not required for determination of pass or failure of the test case.

Consequence of Failure*:
Reason for including this test case as a necessary part of this interoperability test suite.

*Note - not all test cases have this, at this time

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix B

Testers Evaluation Questionnaire of ADIOP

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Tester Evaluation of ADIOP

(Automated Diagnosis of Interoperability Problems)
Version 1.0

Mohammed Sqalli

January 30, 2001

Table of Contents

1. Setup **234**

2. Survey of the Tester **234**

3. Comparison and Survey of ADIOP performance **236**

3.1. DECODER 236
3.1.1. Steps *236*
3.1.2. Survey *236*
3.1.3. Data sets *237*
3.2. DIAGNOSER 238
3.2.1. Steps *238*
3.2.2. Survey *240*
3.2.3. Data sets *241*
3.2.4. Optional Data sets *241*
3.3. TEST SUITE BUILDER 242
3.3.1. Steps *242*
3.3.2. Survey *242*
3.3.3. Data sets *244*
3.3.4. Optional Data sets *244*

4. General Survey **245**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. Setup

Before you start the evaluation, please make sure you have the following:

- ✓ ADIOP manual.
- ✓ ADIOP program. Run "clean" and then "make" to build ADIOP (See Manual for more details).
- ✓ Data/capture(s) to be used is available on ADIOP. The path is "***./adiop/Data/***" The Informed Consent Format (ICF) document to sign.

In addition you may have to check this if you have any problems decoding or diagnosing data:

- ✓ Protocol(s) to be used are available on ADIOP in the path "***./adiop/packet/***"
- ✓ Test Cases to run are available on ADIOP in the path "***./adiop/testsuite/***"
- ✓ Analyzers' types to be used are available on the main menu on the main ADIOP window.

The Data sets (not including the optional ones) provided here are the optimal number we want to achieve. The optional data sets are additional for people willing to perform a more extended evaluation.

If you have any comments on any question in this document, please feel free to include it in as part of your evaluation. Please record any issues/problems you have with ADIOP and report them at the end of this form. If you need any help, please contact Mohammed Sqalli at msqalli@cs.unh.edu

When you finish the evaluation, please fill electronically this document, email it to Mohammed Sqalli (msqalli@cs.unh.edu)

Thank you for taking the time to complete this evaluation.

2. Survey of the Tester

Please answer the following questions to the best of your knowledge:

- ✓ How would you rate your knowledge of the protocol you are testing?
  ____ Low ____ Moderate ____ High

- ✓ How would you rate your knowledge of interoperability testing?
  ____ Low ____ Moderate ____ High

- ✓ How would you rate your knowledge of the interoperability test cases of the protocol you are testing?
  ____ Low ____ Moderate ____ High

- ✓ How much do you know about ADIOP?
  ____ Nothing ____ Familiar ____ Very Familiar

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- ✓ Any other information you would like to add.

- ✓ For how long you have been at IOL?

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. Comparison and Survey of ADIOP performance

3.1. Decoder

3.1.1. Steps

Please fill the following information for each captured data: (make copies of this page as needed)

- Data: _____
- Protocol: _____
- Analyzer type: _____

Tasks	Yes	No
Open a captured data using ADIOP. Was the task completed?		
Check if the analyzer type in the main ADIOP window menu matches the capture type. Does it match?		
Decode the data. Was it decoded?		
Was the decoding done correctly for the protocol being tested?		

Evaluate the ADIOP decode and the one given by the analyzer/sniffer (please state which statement is correct for each one of them): [Add your comments]

Statements	Analyzer		ADIOP	
	Correct	Not Correct	Correct	Not Correct
Includes the complete decodes.				
Includes all what is needed for the protocol being tested.				
Lacks information that might be needed by the protocol being tested but should not affect the diagnosis.				
Lacks information that might be needed by the protocol being tested, and it can affect the diagnosis.				
Lacks all the information needed by the protocol being tested.				
Decode not usable.				

3.1.2. Survey

How do you rate the Decoder?
(Please answer with: Strongly Agree, Agree, Disagree, or Strongly Disagree) [Add your comments]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Statements		Strongly Agree (5)	Agree (4)	(3)	Disagree (2)	Strongly Disagree (1)
Includes decodes of all the packets needed for the protocol being tested.						
Decodes given is correct for the packets needed for the protocol being tested.						
Decodes given is complete for the packets needed for the protocol being tested.						
Friendly	Easy GUI interaction.					
	Easy to read the decoded information.					
	Easy to compare two or more decoded packets. (Use the double-click feature on a decode row).					
ADIOP decoder is a useful tool for the lab. Explain why?						
Fast - The data is decoded in a reasonable amount of time.						

3.1.3. Data sets

Data: ./adiop/Data/capt002.aa
Protocol: PNNI Routing
Analyzer type: Analyzer I

Data: ./adiop/Data/PNNI.bb
Protocol: PNNI Routing
Analyzer type: Analyzer II

Data: ./adiop/Data/other/capt005.ee
Protocol: PNNI Routing
Analyzer type: Analyzer V

Data: ./adiop/Data/dir/capt006.cc
Protocol: Lane
Analyzer type: Analyzer III

Data: "One that you captured yourself"
Protocol: "One that is implemented in ADIOP"
Analyzer type: "One of the types defined by ADIOP"

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2. Diagnoser

3.2.1. Steps

Please fill the following information for every test case (make copies of this page as needed):

- Data: _____
- Protocol: _____
- Analyzer Type: _____
- Test Case: _____

Analyze manually or using a sniffer the captured data.	**Done ____**	**Not Done ____**
Check manually if a test case passes or fails using this data.	**Pass ____**	**Fail ____**
Please give an explanation of why it passed or failed:		
Record the time (in seconds) it took to analyze and explain the results for every test case.		
Write a report of all the test cases from one section and record the time (in seconds) it took to complete this report. The report must include the section name and three columns (Test Case Name, Result, Explanation).		

Repeat the same steps above using ADIOP. In addition, run all test cases of one section (when applicable) in one step and record the time it took to finish the whole section.

Analyze the captured data using ADIOP.	**Done ____**	**Not Done ____**
Check using ADIOP whether a test case passes or fails using this data.	**Pass ____**	**Fail ____**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Print the explanation generated by ADIOP of why it passed or failed:	
Record the time (in seconds) it took to analyze and explain the results for every test case.	
Record the time (in seconds) it took to analyze and explain the whole section to which this test case belongs.	
Record the number of test cases that exist in this section	
Print the report generated by ADIOP of the results of all the test cases in this section, and record the time (in seconds) it took to get this report.	

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2.2. Survey

How do you rate the Diagnoser?
(Please answer with: Strongly Agree, Agree, Disagree, or Strongly Disagree) [Add your comments]

Statements		Strongly Agree (5)	Agree (4)	(3)	Disagree (2)	Strongly Disagree (1)
Generates the correct result [Pass/Fail]						
If the result is PASS	The diagnosis (explanation) given is correct.					
	The explanation given by ADIOP is complete. There is no need to investigate the problem further.					
	The explanation given by ADIOP is useful, but there is a need to investigate the problem further.					
	The explanation given by ADIOP is not useful.					
If the result is FAIL	The diagnosis (explanation) given is correct.					
	The explanation given by ADIOP is complete. There is no need to investigate the problem further.					
	The explanation given by ADIOP is useful, but there is a need to investigate the problem further.					
	The explanation given by ADIOP is not useful.					
Friendly	Easy GUI interaction.					
	Easy to execute the test cases individually.					
	Easy to execute the test cases in batch mode (one section).					
	Easy to read the diagnosis.					
Flexibility	It is possible to diagnose data from different analyzers.					
	It is possible to diagnose data for different protocols.					
The reports generated by ADIOP are useful for the lab.						
Reusability - The storage of the diagnosis obtained is useful						

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ADIOP diagnoser is a useful tool for the lab. Explain why?					
Fast - The data is diagnosed in a reasonable amount of time.					

3.2.3. Data sets

Data: ./adiop/Data/capt002.aa
Protocol: PNNI Routing
Analyzer type: Analyzer I
Individual Test Cases: V4301H__001, V4301H__002, V4301H__003, V4301H__004, V4301H__005, V4301H__006, V4301H__007, and V4401DBS001.
Section Test Cases: V4301H__

Data: ./adiop/Data/other/PNNI.PRN
Protocol: PNNI Routing
Analyzer type: Analyzer I
Individual Test Cases: V4301H__002, V4401DBS001, and V100_LEC_Configure_Request_001 (from LANE test suite)

Data: ./adiop/Data/capt003.aa
Protocol: PNNI Routing
Analyzer type: Analyzer I
Individual Test Cases: V4301H__005

Data: "One that you captured yourself"
Protocol: "One that is implemented in ADIOP"
Analyzer type: "One of the types defined by ADIOP"
Test Cases: "Your choice"

3.2.4. Optional Data sets

Data: ./adiop/Data/other/LANE.PRN
Protocol: Lane
Analyzer type: Analyzer I
Section Test Cases: V100_LEC_Configure_Request_, V200_LEC_Configure_Request_

Data: ./adiop/Data/capt001.aa
Protocol: PNNI Routing
Analyzer type: Analyzer I
Section Test Cases: V4302H__

Data: ./adiop/Data/mpoa_csp.aa
Protocol: MPOA
Analyzer type: Analyzer I
Section Test Cases: test

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3. Test Suite Builder

3.3.1. Steps

Please fill the following information for every test case to be created (make copies of this page as needed):

- Protocol: _____
- Test Case: _____
- Document: _____

Tasks	Yes	No
Open the Test Suite Builder Window. Was the task completed?		
Open a new test case. Was the task completed?		
Define the CSP model of this test case using "State CSP Model" menu. Was the task completed?		
Generate the test case. Was the test case compiled correctly?		
Open the Decoder/Diagnoser window and check if the test case run correctly. Did the test case run as expected? If not, please explain:		
Debug the test case using the Test Suite Builder (if needed). Did you need any debugging? Please explain.....		
Record the time (in seconds) it took to do this for every test case.		

3.3.2. Survey

How do you rate the Test Suite Builder (TSB)?
(Please answer with: Strongly Agree, Agree, Disagree, or Strongly Disagree) [Add your comments]

Statements	Strongly Agree (5)	Agree (4)	(3)	Disagree (2)	Strongly Disagree (1)
It is easier to automate a test case using ADIOP than using other programs (e.g., TCL/TK, C, etc)					
Generates a correct test case, (i.e., you can					

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

execute the test case and it reports the correct diagnosis)						
The new test case is added to the menu on the Diagnoser/Decoder window under the appropriate protocol.						
Friendly	Easy GUI interaction.					
	Easy to use					
	Easy to build a test case.					
Flexible - It is possible to correct test case definition						
Reusability - It is useful to have these test cases stored so there is no need for the testers to know all the details.						
The TSB will help the testers do more interesting work						
Language	Easy to model a test case using ADIOP					
	Easy to understand the CSP model definition of a test case					
ADIOP TSB is a useful tool for the lab. Explain why?						
Fast - The test cases are built in a reasonable amount of time.						

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3.3. Data sets

Protocol: PNNI Routing
Test Cases: V4301H_001.
Document: ./adiop/adiopx/testsuite/pnnirout/af-test-csra-0111.000.txt

Protocol: "One that is implemented in ADIOP"
Test Cases: "Your choice"
Document: "Interoperability document for the protocol chosen"

3.3.4. Optional Data sets

Protocol: PNNI Routing
Test Cases: V4401DBS001.
Document: ./adiop/adiopx/testsuite/pnnirout/af-test-csra-0111.000.txt

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. General Survey

How do you rate the ADIOP system in general?
(Please answer with: Strongly Agree, Agree, Disagree, or Strongly Disagree) [Add your comments]

Statements		Strongly Agree (5)	Agree (4)	(3)	Disagree (2)	Strongly Disagree (1)
We can automate interoperability test cases using ADIOP						
We can diagnose more interoperability problems using ADIOP						
We can find problems quickly using ADIOP compared to manual diagnosis						
Reusability – ADIOP provides a good way to store test cases and re-use them later.						
It is better to remember how we diagnosed old problems using ADIOP than manually						
ADIOP saves time for testers						
ADIOP will help the testers do more interesting work						
We know more about protocols when using ADIOP						
We know more about interoperability testing when using ADIOP						
Friendly	Easy GUI interaction in ADIOP					
	Easy to learn ADIOP					
	Easy to use ADIOP					
	Easy to find what you are looking for in ADIOP					
Flexible	It is possible to use many decodes on different windows at the same time					
	It is possible to perform many diagnoses on different windows at the same time.					
	It is possible to create many test cases on different windows at the same time					
	It is possible to do all the above tasks with no problem of conflicts in the application.					

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Test cases can be accessible and executed by anyone without much knowledge on how they were created.					
The explanation generated by ADIOP is useful.					
Fast – ADIOP provides solutions in a reasonable amount of time.					
ADIOP is a useful tool for the lab. Explain why?					
I prefer to work with ADIOP rather than without it for interoperability testing					
I recommend using ADIOP in the lab wherever applicable					
I expect ADIOP to be even more useful for large data sets with hundreds of frames.					

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

How much better is ADIOP than what we had before?

How can we make ADIOP better and more useful?

Is ADIOP useful for other types of activity that we have not mentioned here? Can you think of other problems we can resolve or be helped with using ADIOP?

What are some of the issues/problems you had when using ADIOP?

Final Comments:

Thank you for taking the time to evaluate ADIOP.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix C

ADIOP V2.0 User Manual

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Manual for using ADIOP

(Automated Diagnosis of Interoperability Problems)

Version 2.0

Mohammed Sqalli

March 14, 2001

Table of Contents

1. INSTALLATION GUIDE 251

2. ADIOP V2.0 ADDITIONS OVER ADIOP V1.0 252

3. ADIOP IMPLEMENTATION 253

4. USERS GUIDE 254

4.1. DECODER 254
4.1.1. Choose the Analyzer type 254
4.1.2. Open captured data 255
4.1.3. Decoding data 256
4.2. DIAGNOSER 257
4.2.1. Choosing the protocol being tested 257
4.2.2. Running one test case 258
4.2.3. Running all test cases of one section 258
4.2.4. Close Decoder/Diagnoser 259
4.3. DEBUGGER 259
4.3.1. Open Advisor/CBR window 259
4.3.2. Retrieve Similar Cases 261
4.3.3. Reuse/Adaptation of a Case 262
4.3.4. Revise Adapted Case 263
4.3.5. Update Test Case Model 264
4.3.6. Retain a New Revised Case 267
4.3.7. Other Advisor menus 267
4.4. TEST SUITE BUILDER 268
4.4.1. Open Test Suite Builder Window 268
4.4.2. Choose the protocol 268
Open a test case 269
4.4.4. State the CSP Model 269
4.4.4.1. Start CSP Model 269
4.4.4.2. Declaring Packets 270
4.4.4.3. Domains 271
4.4.4.4. Unary Constraints 272
4.4.4.5. Binary Constraints 273
4.4.4.6. General Constraints 275
4.4.4.7. End CSP Model 275
4.4.5. Save a test case 275
Get CSP Model 276
4.4.7. Generate Test from the CSP Model 276
4.4.8. Close a test case 277
ANNEX: DESCRIPTION OF THE CSP MODELLING PROCESS 277

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Figures

FIGURE 1 - ADIOP V2.0 IMPLEMENTATION 253
FIGURE 2 - MAIN ADIOP WINDOW 254
FIGURE 3 - PROTOCOL ANALYZERS 255
FIGURE 4 – MAIN MENU 255
FIGURE 5 - OPEN TEST CASE 255
FIGURE 6 – DECODER/DIAGNOSER WINDOW 256
FIGURE 7 - ONE FRAME DETAILED DECODE 257
FIGURE 8 - RUNNING TEST CASES 257
FIGURE 9 - ONE TEST CASE EXECUTION RESULT 258
FIGURE 10 - RESULTS OF TEST CASES EXECUTION FROM ONE SECTION 259
FIGURE 11 - TEST CASE RESULT CONTAINING AN 'ADVISOR' BUTTON 259
FIGURE 12 - ADVISOR/CBR WINDOW 260
FIGURE 13 - ACCESS TO ADVISOR FROM MAIN ADIOP WINDOW 260
FIGURE 14 - CASES' TYPES 261
FIGURE 15 - RETRIEVE SIMILAR CASES MENU 261
FIGURE 16 - SIMILAR CASES TABLE 262
FIGURE 17 - CASE ADAPTATION WINDOW 262
FIGURE 18 - CASE ADAPTATION MENU 263
FIGURE 19 - REVISE ADAPTED CASE 264
FIGURE 20 - UPDATE TEST CASE MODEL MENU 264
FIGURE 21 - UPDATE TEST CASE MODEL 265
FIGURE 22 - VERSION OF UPDATED TEST CASE 265
FIGURE 23 - SAVE UPDATED MODEL 266
FIGURE 24 - GENERATE UPDATED TEST CASE 266
FIGURE 25 - RUN UPDATED TEST CASE 266
FIGURE 26 - ADVISOR MENU. 267
FIGURE 27 - NEW EMPTY CASE 268
FIGURE 28 - TEST SUITE BUILDER PROTOCOLS 268
FIGURE 29 - AN EMPTY TEST CASE 269
FIGURE 30 - INITIAL TEST CASE DECLARATION 270
FIGURE 31 - PACKET TYPES 270
FIGURE 32 - LIST OF PACKETS TO ADD 271
FIGURE 33 - PACKETS ADDED TO CSP DECLARATION 271
FIGURE 34 - DOMAIN DECLARATION 272
FIGURE 35 - DOMAINS AVAILABLE FOR UNARY CONSTRAINTS 273
FIGURE 36 - UNARY CONSTRAINTS DECLARATION 273
FIGURE 37 – VARIABLES FOR BINARY CONSTRAINTS DECLARATION 274
FIGURE 38 - CONSTRAINTS ADDED TO THE CSP MODEL 274
FIGURE 39 - BINARY CONSTRAINTS DECLARATION 274
FIGURE 40 - GENERAL CONSTRAINTS DECLARATION 275
FIGURE 41 - GET CSP MODEL 276
FIGURE 42 - RESULT OF GENERATE TEST FROM CSP MODEL 276

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. Installation Guide

- ✓ Get the file adiop.tar.gz.
  In Unix, type "**gzip adiop.tar.gz**". Then type "**tar xvf adiop.tar**".
  In Windows, use WinZip to unzip the file adiop.tar.gz.

- ✓ A directory called "**adiop**" will be created under your actual working directory. It contains all the files needed to generate documentation for backup, compile and run ADIOP.

- ✓ The Home directory for ADIOP is "**./adiop/**"

- ✓ To delete all the ".class" files:
  In Unix, run **clean**.
  In Windows, run **clean.bat**.

- ✓ To delete all temporary files:
  In Unix, run **clean_temp**.

- ✓ To compile ADIOP:
  In Unix, run **make**. This uses the "makefile" file. You can also use the "**compile-testsuite**" command to compile all test cases created using the Test Suite Builder (See **4.4**).
  In Windows, run **make.bat**.

- ✓ To generate documentation:
  Make necessary changes in the "javadocgen" file to where you want to put documentation. Then run **javadocgen**.

- ✓ To make a backup:
  Make changes to the file "backup". Then, run **backup**. This command deletes all "**.class**" files under "**./adiop/**", makes a backup using tar and gzip in a file called "adiop_*today*.tar.gz" which is stored in "**./adiop-backups/**", then run the make command to re-build adiop. This process takes about 3 minutes.

- ✓ To get the tree of all ".java" files in ADIOP:
  Run **gentree**.

- ✓ "**./adiop/Data/**" contains all the captures that we want to test using ADIOP. This directory is the default captured data directory for ADIOP.

- ✓ "**./adiop/adiop.java**" is the file that contains the main java function for ADIOP.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- ✓ "**./adiop/adiopx/**" is the directory that contains all the ".java" files needed to build and run ADIOP. More information about the implementation of ADIOP can be found under "www.cs.unh.edu/~msqalli/adiopx.docs/".

2. ADIOP V2.0 additions over ADIOP V1.0

- ✓ There is a constant **DEBUG** defined in adiopx/util/Constants.java that can be set to true or false. If set to true, there will be more debugging messages printed on the screen when running ADIOP. It is being used so far only for the java files added in ADIOP-V2.0. The plan is to have it for all debugging messages.

- ✓ All the files under adiopx/debug are new for version 2.0 and they all deal with Case-Based Reasoning and CSP model debugging.

- ✓ The whole Debugger section (See **4.3**) is new in this version of the manual.

- ✓ The initial case-base contains 8 cases in the file: "./adiop/adiopx/debug/casebase" defined as **CASEBASE_FILE** in "./adiop/adiopx/util/Constants.java" file.

- ✓ The ADIOP Implementation figure is updated to reflect the new implementations. All planned modules have been implemented as of version 2.0.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. ADIOP Implementation

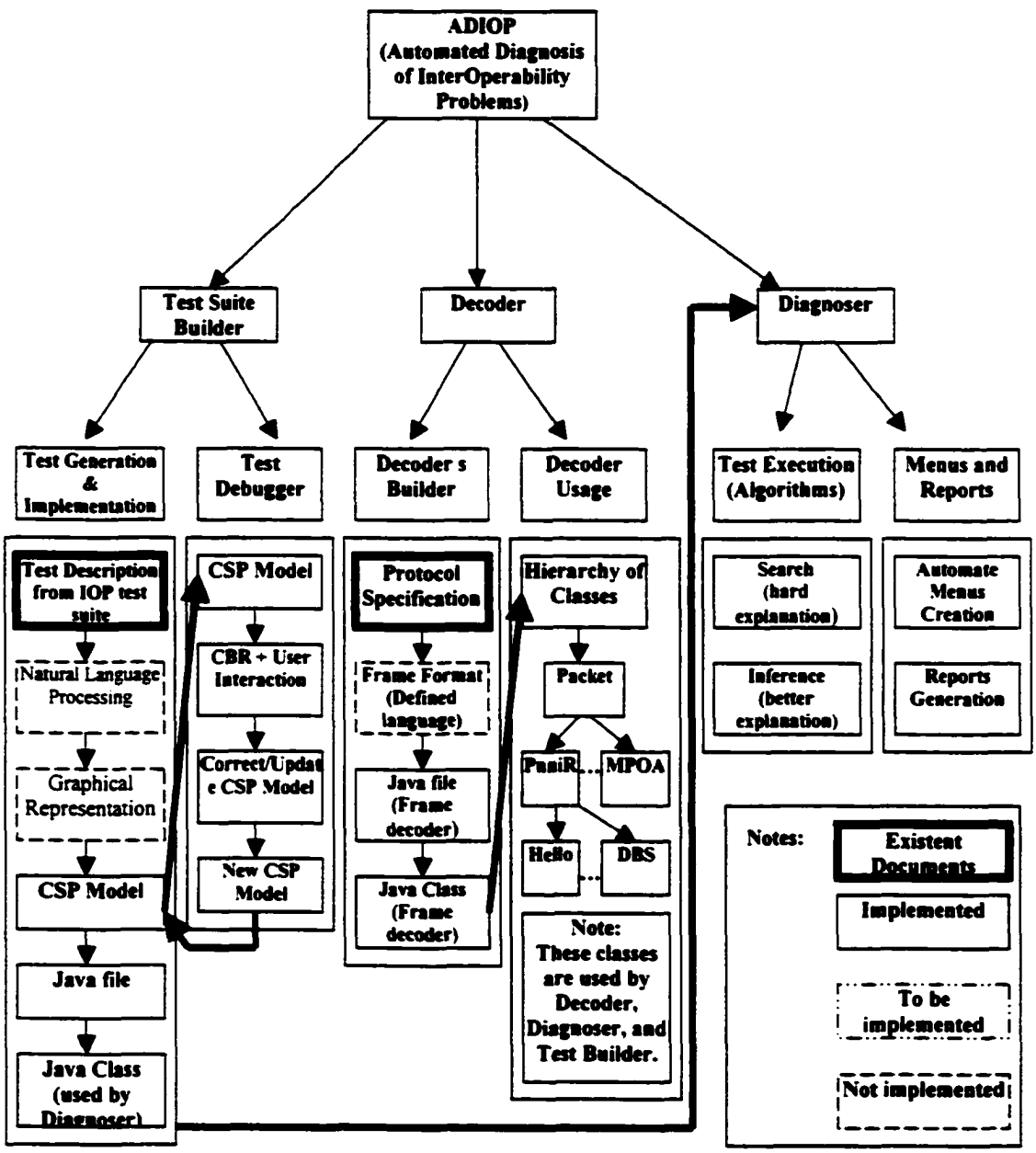

Figure 1 - ADIOP V2.0 Implementation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. Users Guide

You must have Java (JDK1.2) or newer version installed. After compiling ADIOP using the make command, type "**java adiop**" from "**./adiop/**" directory to start the ADIOP application.

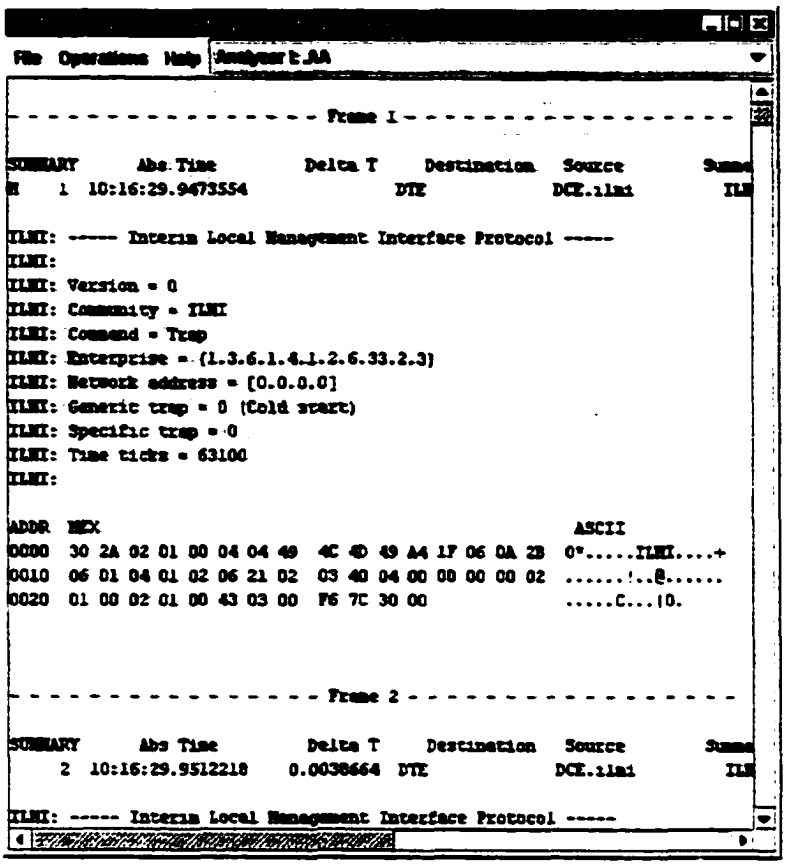

Figure 2 - Main ADIOP Window

4.1. Decoder

4.1.1. Choose the Analyzer type

In the main menu, you may choose from a list box the analyzer type that was used to capture the data. This is an optional step because ADIOP uses the extension of the file when it is opened to update this field automatically.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

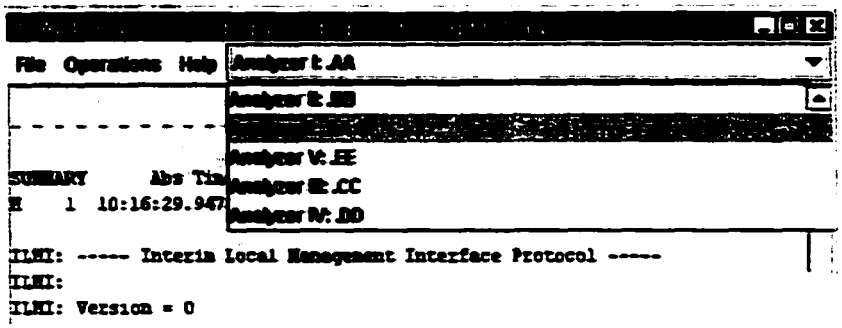

Figure 3 - Protocol Analyzers

4.1.2. Open captured data

From the main menu, choose **File → Open**. By default, ADIOP opens a file dialog box with a default directory of "**./adiop/Data/**". This can be changed if the data is stored in a different location. Click on the file you want and press the "**Open**" button. You should see the data captured in the text area of the main window.

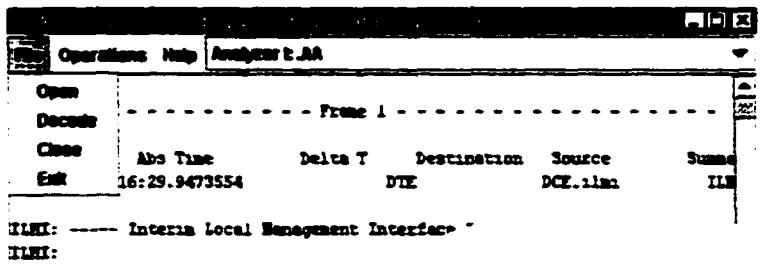

Figure 4 – Main Menu

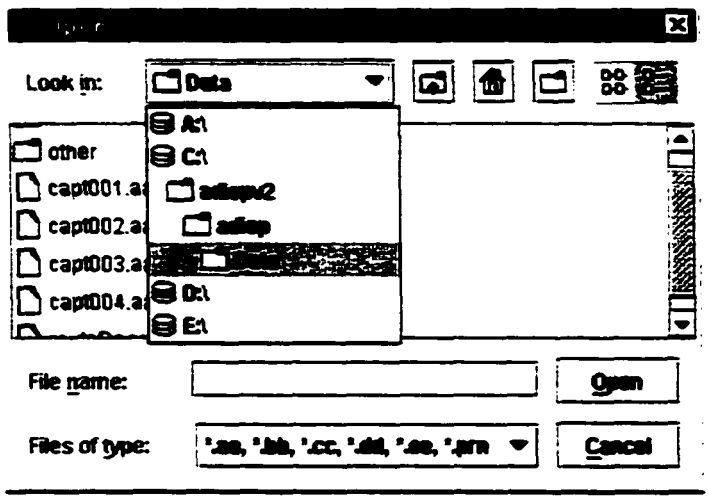

Figure 5 - Open Test Case

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Note: Data has to be initially captured using the text format and not the internal analyzer's format, and it should include the Hex bytes. ADIOP uses some of the information from the header and the Hex bytes to decode the frames. For more information, check some of the captured data under "./adiop/Data/".

4.1.3. Decoding data

From the main menu, choose **File → Decode**. A new window will open showing all the frames decoded. You can click on any of the frames to see more details of its decoded data. If the decode is not implemented for some types of frames, you will only see the header. You can press the "**Print**" button in the same panel as the detailed decoded data to print the content of this frame to a printer or a file. You can also print the summary of all the frames decoded by using the menu **Test Suite → Print**.

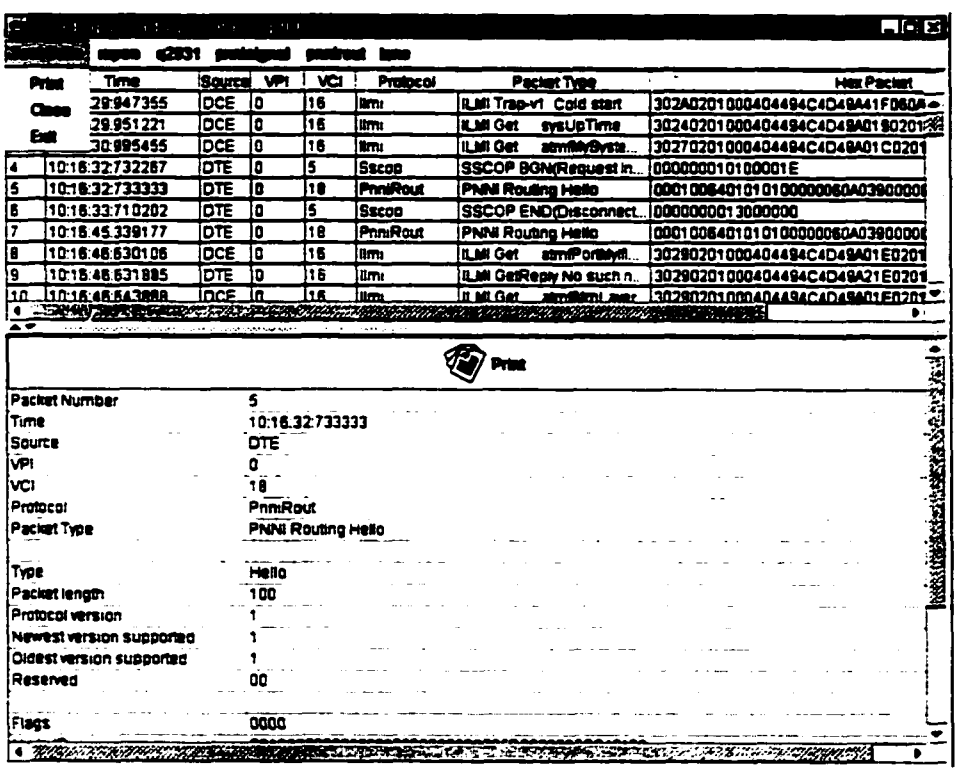

Figure 6 – Decoder/Diagnoser Window

You can also double-click on one frame to get the decoded data of this frame in a new window.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

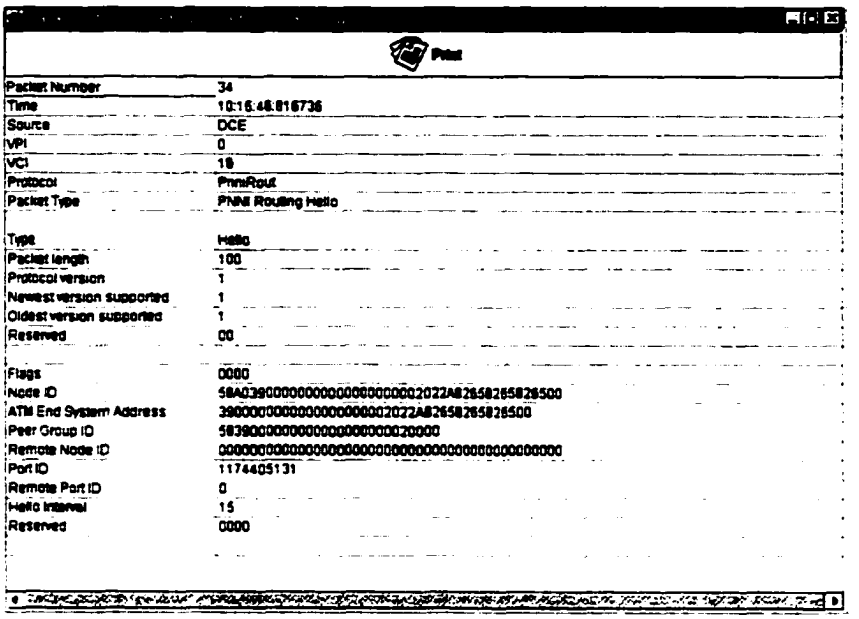

Figure 7 - One Frame Detailed Decode

This is useful if you want to see the decoded data of more than one frame at the same time.

4.2. Diagnoser

4.2.1. Choosing the protocol being tested

In the window showing the decoded frames, there are menus for running different test cases either in one at a time or all test cases of one section at once. For example, if you want to test PNNI Routing, you can go to the menu "**pnnirout**" and choose from one section the test case(s) you want to run. You can choose a specific test case or all the test cases of one section.

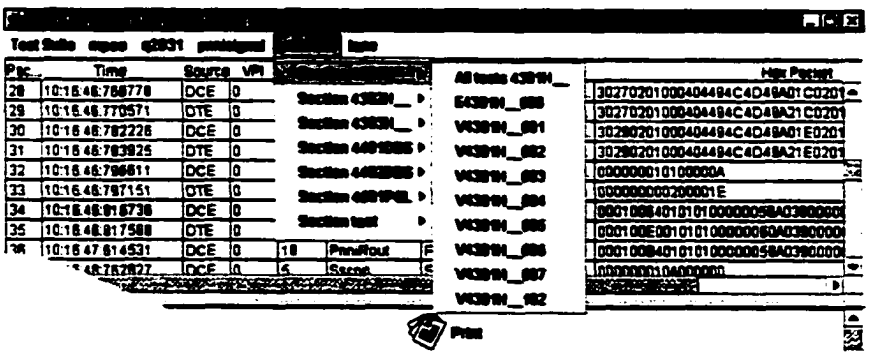

Figure 8 - Running Test Cases

ADIOP constructs this menu from the structure of the directories under "**./adiop/testsuite/**". So, if a new protocol is added or more test cases are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

added/deleted, the menu will get updated when this window is closed and opened again.

4.2.2. Running one test case

If you run one test case, a new window appears with the result of executing this test case. It has one text box showing the detailed solution colored in blue (pass), green (pass with warning), and red (fail).

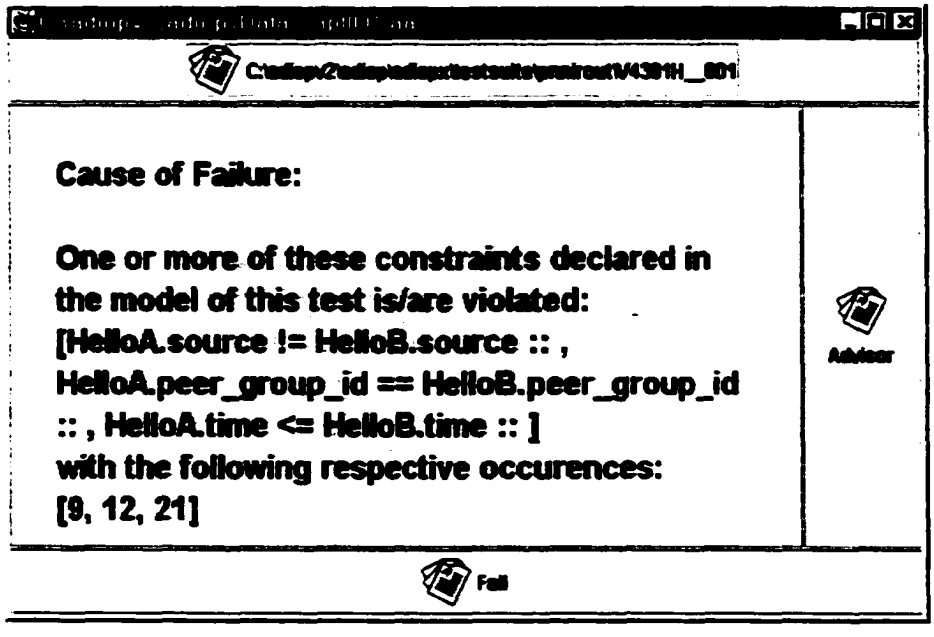

Figure 9 - One Test Case Execution Result

The window also contains up to three buttons. The first one shows the name of the test case, and if pressed the test case specification plus its CSP model will be shown in the text box. The second button shows the overall result: Pass, Pass with warning, or Fail, and if pressed the detailed result will be shown in the text box. A third button appears only when the result is "Fail". This is called "Advisor" and it is used for debugging the problem. "Advisor" is not implemented yet.

4.2.3. Running all test cases of one section

Choose "All tests ..." from the section submenu. A new window showing the test results of all test cases of this section appears. It has three columns "TestName", "Verdict", and "Explanation". "Verdict" shows the result of a test case execution. It can be either Pass, Pass with warning, or Fail. "Explanation" shows the detailed solution of a test case. This window also has a "Print" button for printing the result of all test cases of one section.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

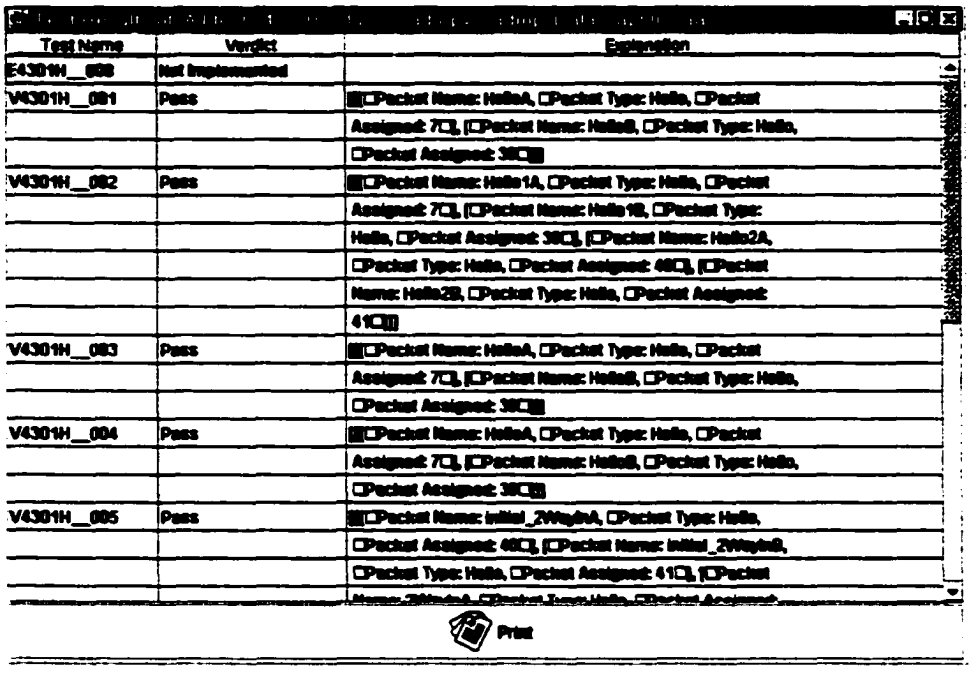

Figure 10 - Results of Test Cases Execution from One Section

4.2.4. Close Decoder/Diagnoser

From the main menu, choose **Test Suite → Close**

Note: The menu **Test Suite → Exit** will close all the windows of the application ADIOP.

4.3. Debugger

4.3.1. Open Advisor/CBR window

If the result of a test case is a failure, then an "Advisor" button will be shown in the result's window (see **4.2.2.**) (Figure 11).

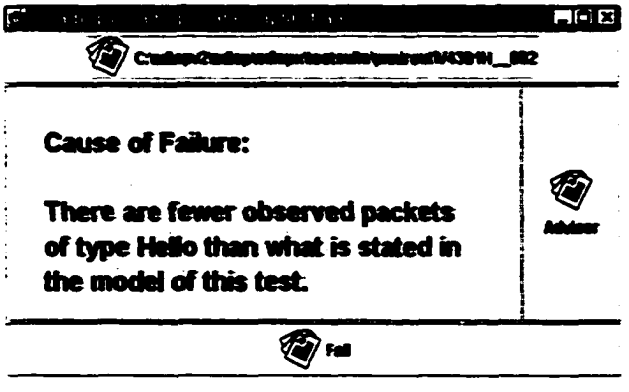

Figure 11 - Test Case Result containing an 'Advisor' button

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

You can click on this button to get more help from the ADIOP system on the cause of this failure. The Advisor/CBR window appears (Figure 12).

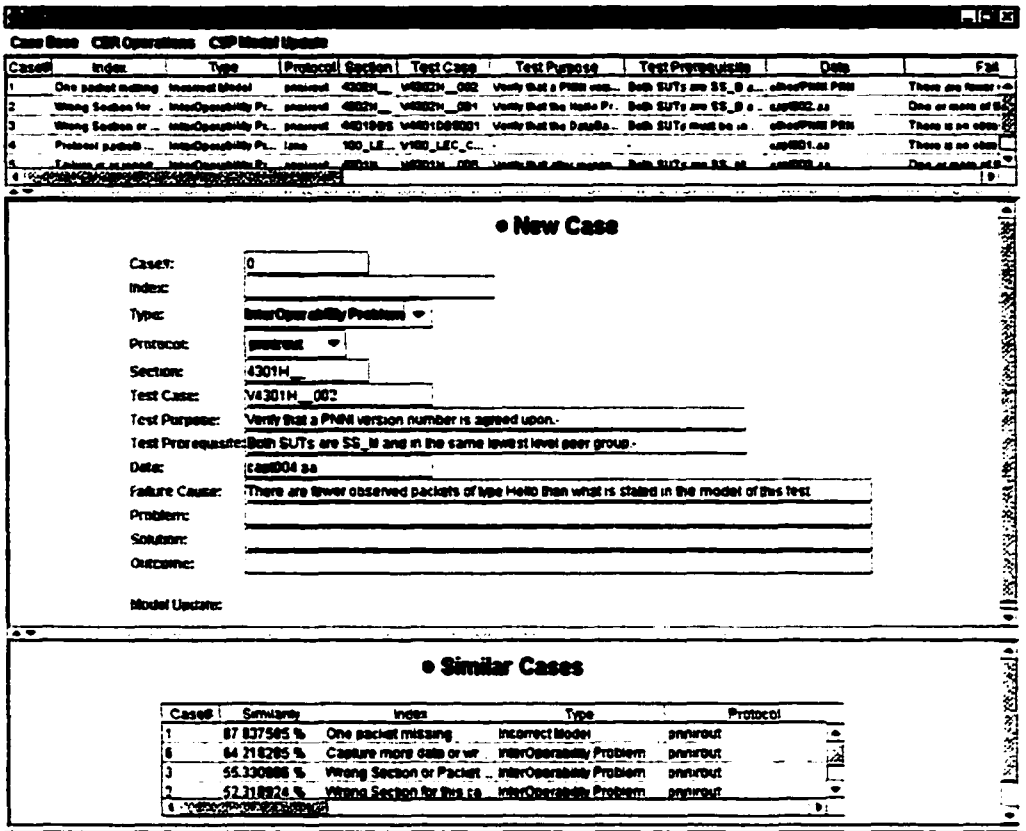

Figure 12 - Advisor/CBR Window

Another way to access the Advisor/CBR window from the main ADIOP window is to use the main menu as described in (Figure 13).

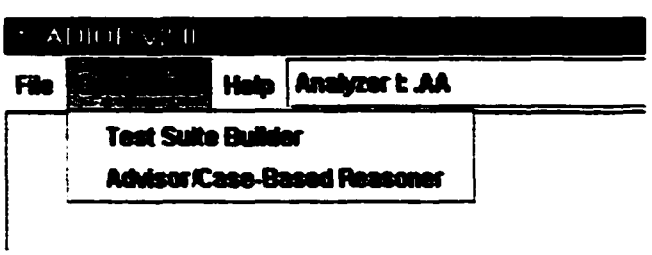

Figure 13 - Access to Advisor from Main ADIOP Window

The Advisor Window (Figure 12) shows all the cases stored in the case-base in the top panel. This case-base content can be found in the "./adiop/adiopx/debug/casebase" file. The middle panel shows the information about the new case. If the 'Advisor' was called from the result's window of a failed test case, then the 'New Case' will contain information generated from this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

test case's result. If the 'Advisor' window was called directly from the main ADIOP window, then the new case will have empty fields.

The third panel (bottom panel) of the 'Advisor' window contains the cases stored in the case base ranked by their similarity to the new case.

4.3.2. Retrieve Similar Cases

The user may want to change the 'Type' field and retrieve similar cases just so they have a better idea of what type of case they have at hand. The similarity value will be higher for the closer type of cases.
The type can be chosen from the list of provided items (Figure 14) or the user may add a new type. However, it is better not to add too many types so as to make it easy to track cases by type. The types provided by ADIOP should be sufficient for most cases to be stored. If there is an error in the model and it needs to be debugged, we will usually use 'Incorrect Model' or 'Incomplete Model'. For almost all other cases, we will use 'Interoperability Problem' as this means that the issue has to do with the data captured and not with the model of the test case.

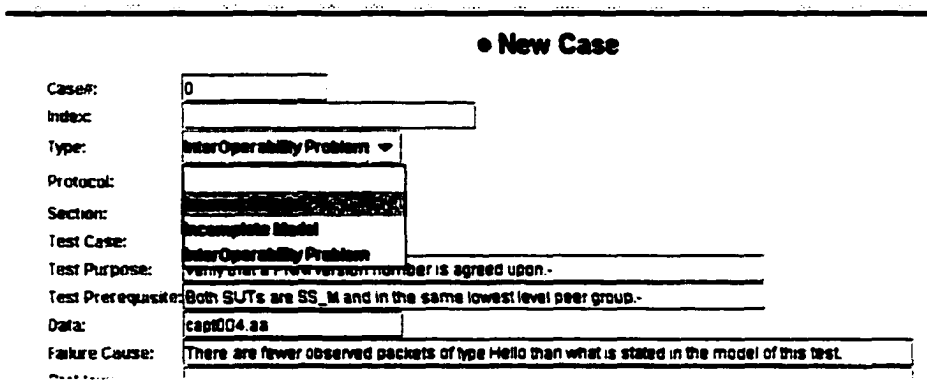

Figure 14 - Cases' Types

When the type is chosen, the user may retrieve similar cases by clicking on the right menu item under 'CBR Operations' (Figure 15).

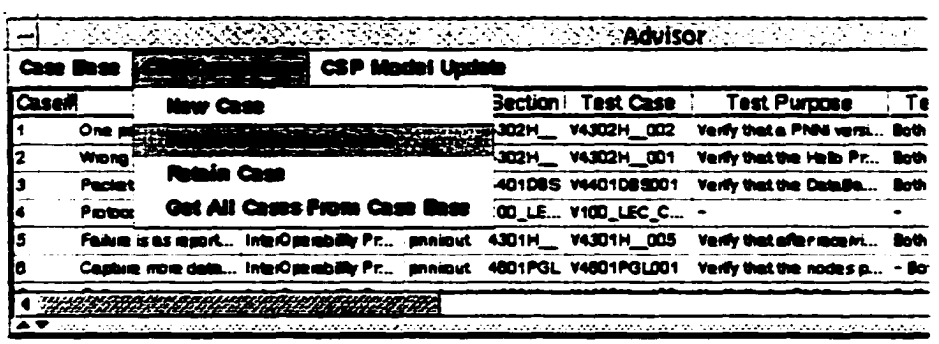

Figure 15 - Retrieve Similar Cases Menu

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Then the user can check the bottom panel for the new similarity values an ranking.

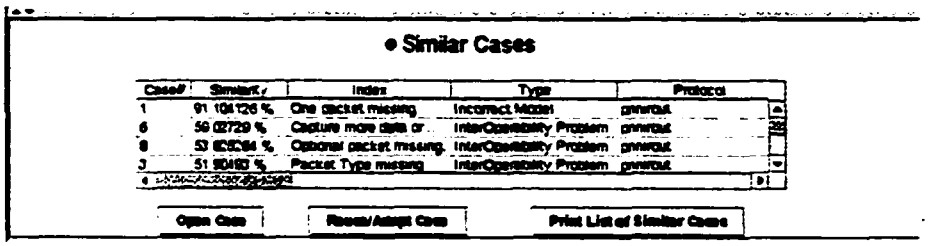

Figure 16 - Similar Cases Table

The user can then click on one of the test cases (Figure 16) that they think are the most similar to the new situation (failure). Then click on the "Reuse/Adapt Case' button to reuse information from this similar case into the new case. A new window appears containing information about case reuse and adaptation (Figure 17).

4.3.3. Reuse/Adaptation of a Case

The "Case Adaptation' window (Figure 17) shows features of the new case, the similar case chosen, and the adapted case generated in addition to the similarity value for each feature between the new and similar case and the weight used for each feature.

The weights are defined in "./adiop/adiopx/util/Constants.java". It is **recommended** that these values are only modified by the person responsible for the CBR component of the ADIOP system, so as not to cause any inconsistencies on how the similar cases are being ranked.

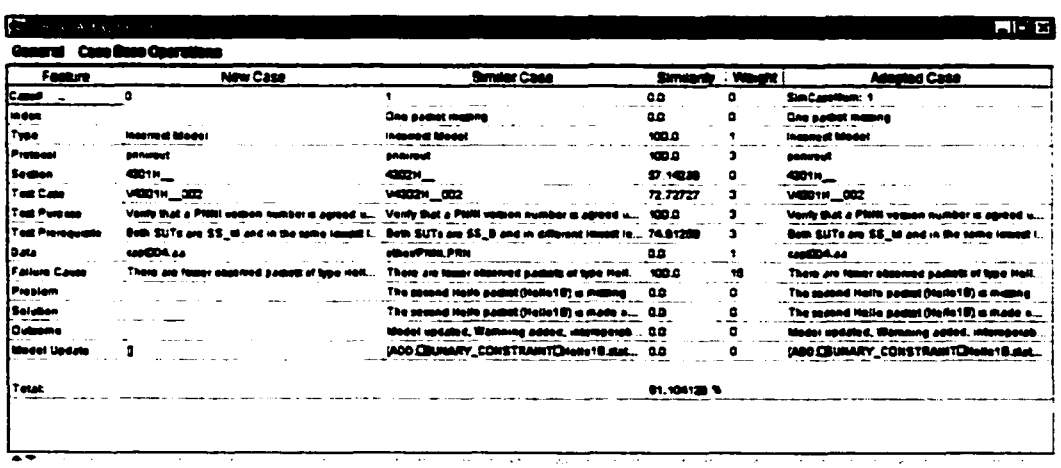

Figure 17 - Case Adaptation Window

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The user can make changes to values of two columns: the 'New Case' column and the 'Adapted Case' column.

It is possible to 'Compute Similarity' if the user makes changes to the 'New Case' values. There is a menu item for this purpose (Figure 18). This action will appropriately update the 'Similarity' column values.

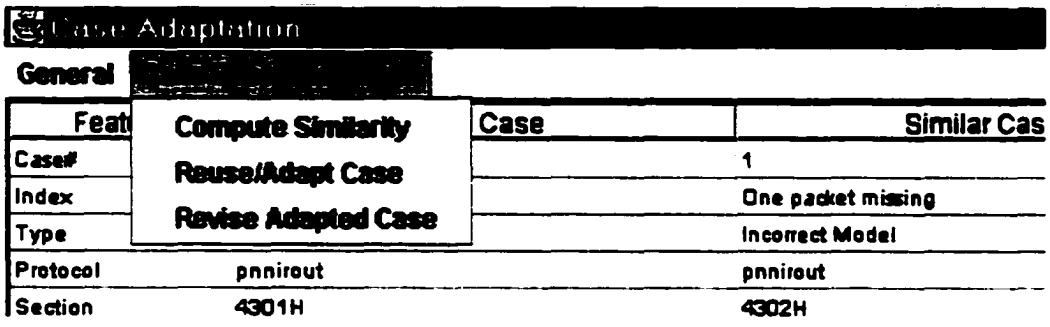

Figure 18 - Case Adaptation Menu

It is also possible to "Reuse/Adapt Case" if the user makes changes to the 'New Case' values. There is a menu item for this purpose (Figure 18). This action will appropriately update the 'Adapted Case' column values.

4.3.4. Revise Adapted Case

ADIOP-V2.0 provides only a simple adaptation method. So the user may choose to revise the adapted case to the new situation at hand.

When the user has made all the changes and adapted the new case using a similar case, they can "Revise Adapted Case" by choosing the appropriate menu item (Figure 18). This will update the features' values in the "New Case" Panel (Figure 19) of the "Advisor" window.

At this stage the user has to make sure that the case is adapted correctly and mainly check that the "Model Update" value is set to the right statements if the test case model is to be updated. Although, there is always a chance to correct these statements at later stages, it is more convenient to do it here.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

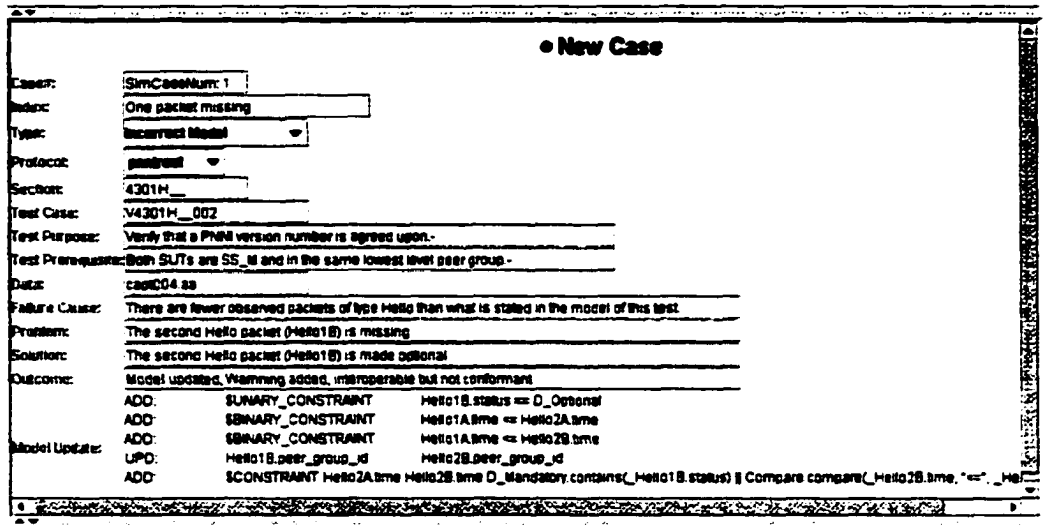

Figure 19 - Revise Adapted Case

4.3.5. Update Test Case Model

The user may want to update the model of the test case that led to the failure if they decide that the model is incorrect or incomplete. For cases with the "Type" feature set to "Interoperability Problem", the "Model Update" feature is empty and not being used.

The statements in the "Model Update" field are used to update the model (Figure 19). They can be either: 'ADD', 'DEL', or 'UPD' statements. The statements are executed in the order they are defined. An 'ADD' statement adds a new statement (usually a constraint) to the CSP model. A 'DEL' statement would delete a statement (usually a constraint) from the CSP model if it does exist (The statement is rather commented out so it is easier for the user to know which statements are unused/deleted). An 'UPD' statement replaces one variable with another.

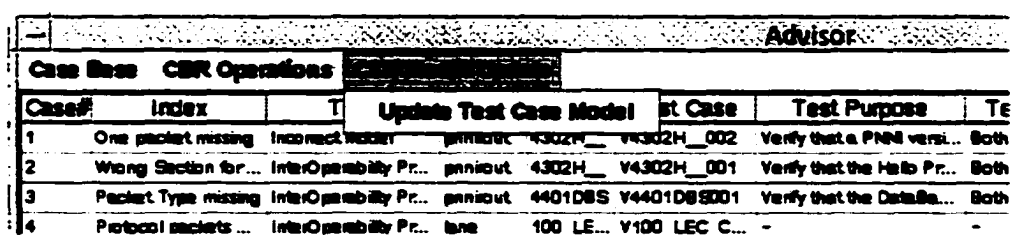

Figure 20 - Update Test Case Model Menu

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To update the model using these statements, the user can click on **CSP Model Update → Update Test Case Model** menu item (Figure 20).

The Test Suite Builder Window appears with the test case model updated using the statements from the revised case (Figure 21).

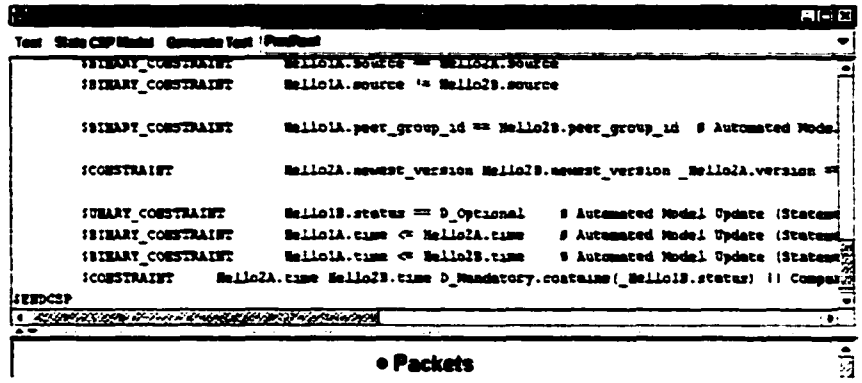

Figure 21 - Update Test Case Model

The user may update the 'Test Case ID' and the 'Update Version' of the new updated test case (Figure 22).

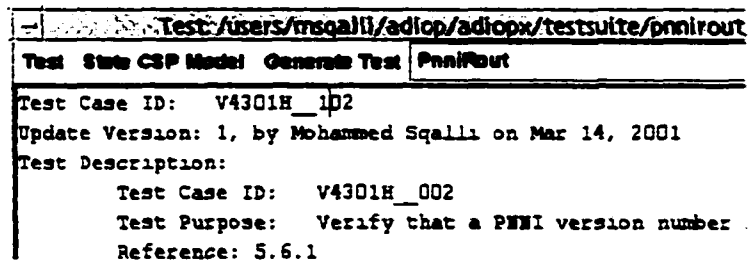

Figure 22 - Version of Updated Test Case

The user can then click on **Test → Save** to save the new updated model (Figure 23). It is **recommended** that the updated test case model be saved in a different file so that it can be tested for a period of time before becoming part of the set of test cases frequently used/run.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

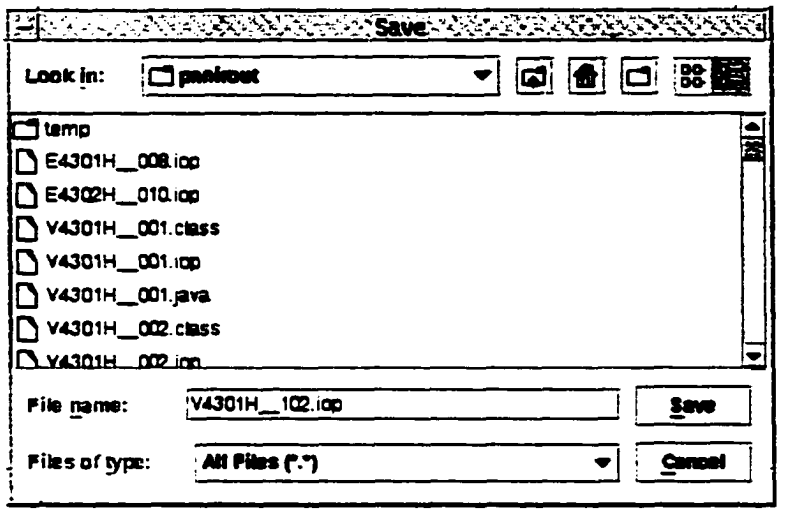

Figure 23 - Save Updated Model

The user can then click on **Generate Test → Generate Test From CSP Model** to generate the new updated test case. More information can be found in section **4.4.7** (Figure 24).

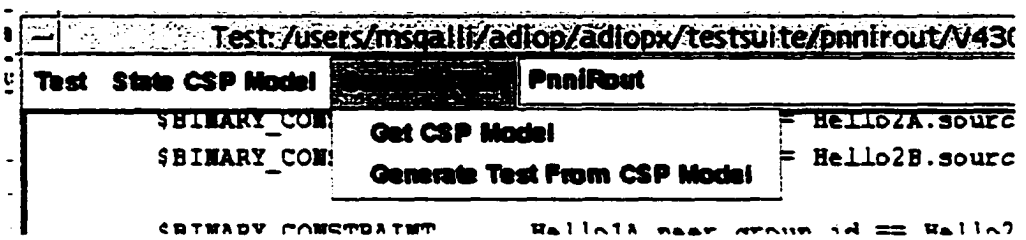

Figure 24 - Generate Updated Test Case

The user can then click on **File → Decode** in the main ADIOP window to get the "Decode" window and run the new updated test case as explained in earlier sections of this manual (Figure 25).

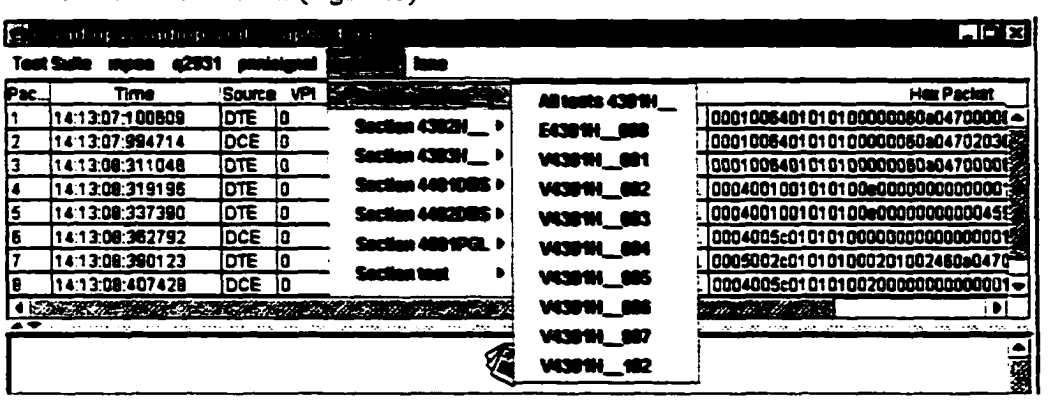

Figure 25 - Run Updated Test Case

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.3.6. Retain a New Revised Case

If the new case is different from old cases in the case base. Usually if similarity between all the old cases and the new one is **< 70%** then the user should consider adding the new case to the case base.

The user has to fill all the features with the appropriate values. The "Index:" feature should give a summary of what the case is all about so as to make it easy to understand in future uses of this case. The user must carefully decide on what value to put for each feature (specific or general) according to how this case is going to be used in the future. Then the user can click on **CBR Operations → Retain Case** in "Advisor" window. The new case is given a new number in the case-base and added to the file "adiopx/adiop/debug/casebase".

It is usually better to work with fewer cases in the case-base library. That is why it is **recommended** not to add more cases unless they really add more information to the already existing cases in the case-base.

4.3.7. Other Advisor menus

The first menu allows for general operations on the case base table and the Advisor window (Figure 26).

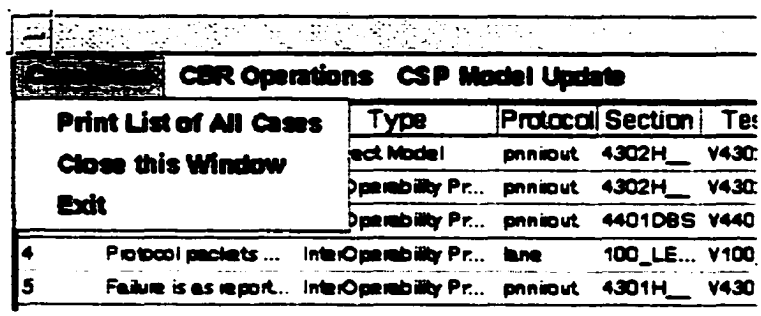

Figure 26 - Advisor menu.

The **CBR Operations → New Case** allows the user to add a new case from scratch. The new case is given a new number to be used if stored in the case-base (Figure 27).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

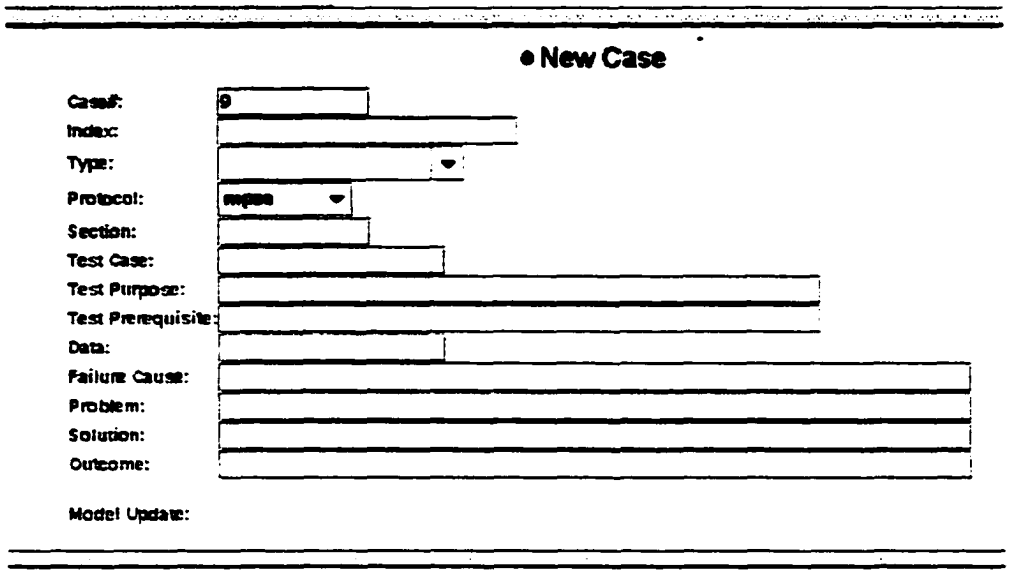

Figure 27 - New Empty Case

The **CBR Operations → Get All Cases From Case Base** menu allows the user to retrieve all cases from the case-base. This is useful if the case-base has changed manually. This is useful if the user needs to make changes to an existing case or delete a case (in this case, the sequential numbring of cases has to be set again) in the case-base "./adiop/adiopx/debug/casebase" as there is no GUI actions performing this. This is **not recommended** unless necessary.

4.4. Test Suite Builder

4.4.1. Open Test Suite Builder Window

From the main ADIOP window choose the menu **Operations → Test Suite Builder** and a new window appears that allows the creation of new test cases.

4.4.2. Choose the protocol

Optionally, you can choose a different protocol from the list box in the menu.

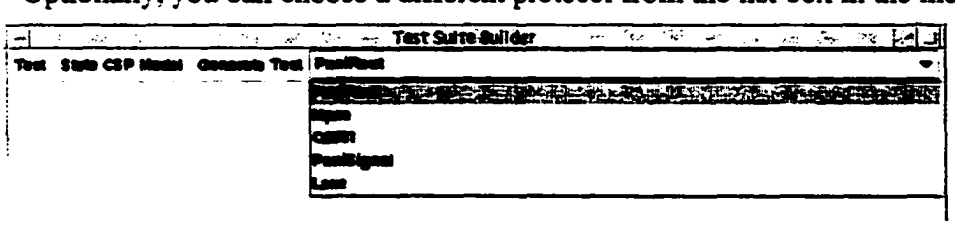

Figure 28 - Test Suite Builder Protocols

ADIOP constructs this list box and many data structures in this window that are related to the protocols and their implementation (e.g., type of packets) from the structure of the directories under "**./adiop/packet/**". So, if a new protocol is added

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

or more packet types are added/deleted, the list box and other data structures will get updated when the protocol is chosen.

4.4.3. Open a test case

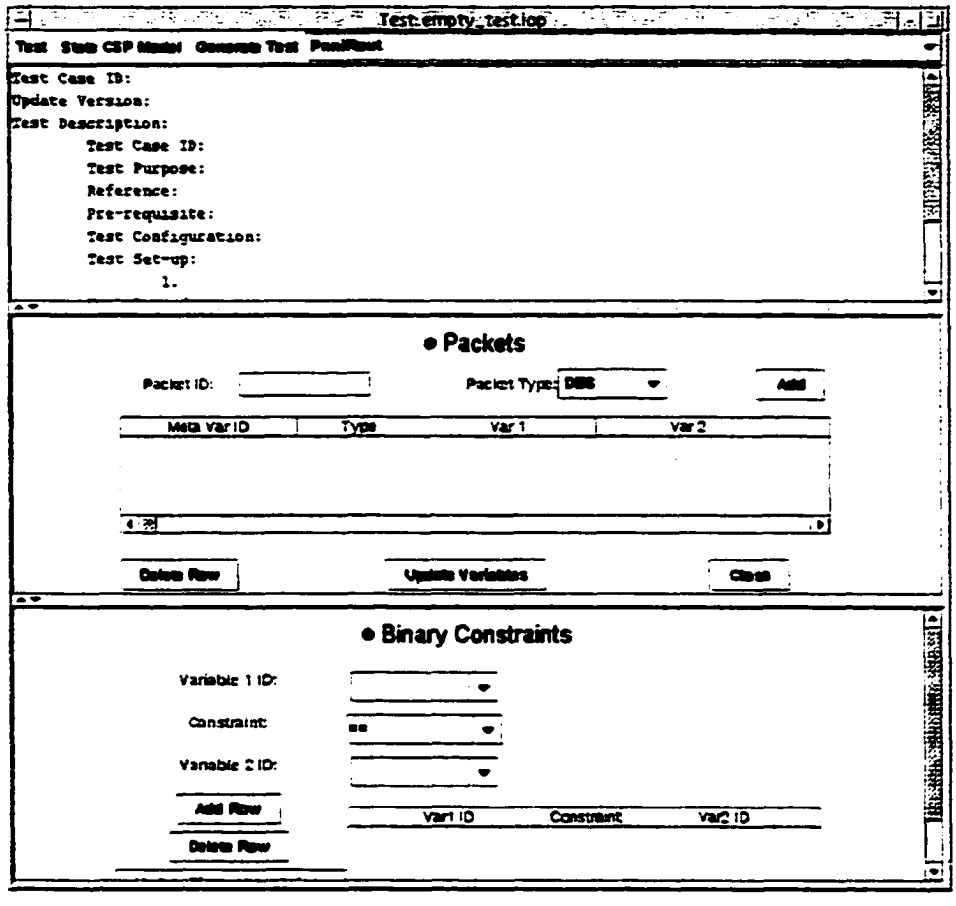

Figure 29 - An Empty Test Case

From the menu choose **Test → Open** to open an existing test case or **Test → New** to create a new one.

4.4.4. State the CSP Model

See **Annex 1** for more detailed information about the CSP model.

Note: "Delete Row" and "Close" buttons functionality is not implemented.

4.4.4.1. Start CSP Model

From the menu choose **State CSP Model → Start CSP Model** to include the statement $CSP to the test case which means this is where the test case declaration (CSP model of this test case) starts. It also adds an $ENDCSP statement at the end of the script which means this is where the test case declaration ends.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

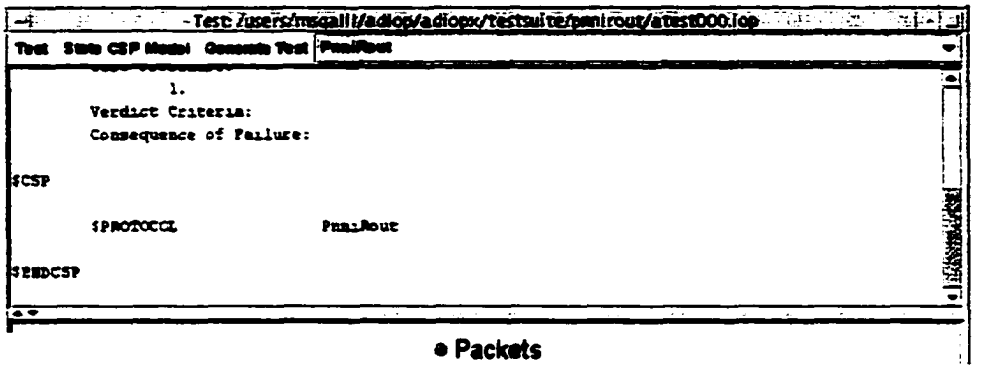

Figure 30 - Initial Test Case Declaration

4.4.4.2. Declaring Packets

From the menu choose **State CSP Model → Packets** to declare packets in the CSP model. A new panel called "Packets" appears and which allows for the addition of more packets into the CSP model.

The different fields shown in this panel are:

- ✓ "Packet ID:" field is the name to be used for a packet.
- ✓ "Packet Type:" field is a list of packet types from the protocol being used to choose from. This list is dynamically loaded from the structure of the "./adiop/packet/*protocolName*" directory.

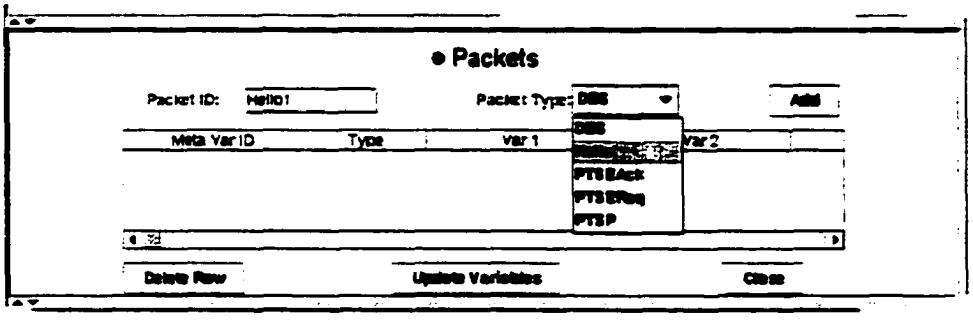

Figure 31 - Packet Types

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

✓ "Add" button is used to add the packet declared to the list of packets.

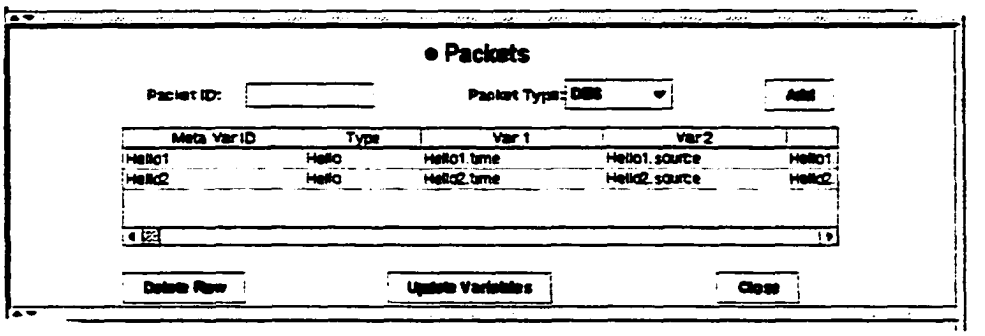

Figure 32 - List of Packets to Add

✓ "Update Variables" button is used to add the set of declarations to the CSP model.

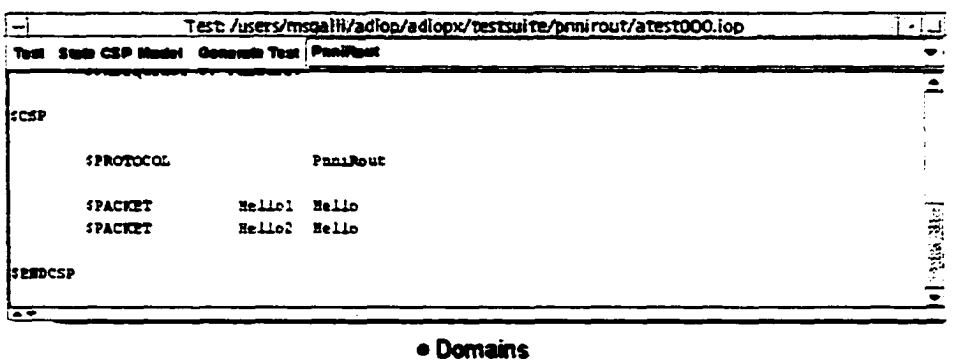

Figure 33 - Packets Added to CSP declaration

4.4.4.3. Domains

-A domain is a set of discrete values. A domain can be used to declare a unary constraint. From the menu choose **State CSP Model → Domains** to declare domains in the CSP model. A new panel called "Domains" appears and which allows for the addition of more domains into the CSP model.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

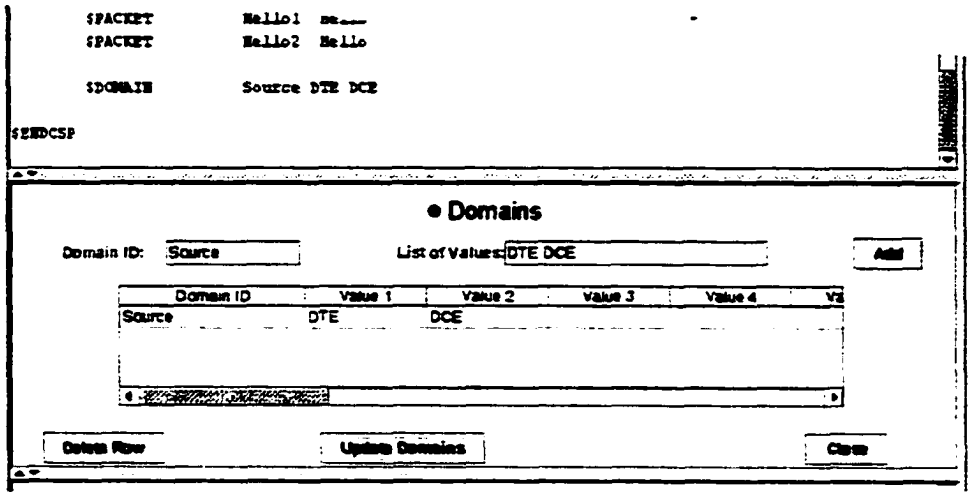

Figure 34 - Domain Declaration

The different fields shown in this panel are:

- ✓ "Domain ID:" field is the name to be used for a domain.
- ✓ "List of values:" field is a list of domain values to be the set of this domain.
- ✓ "Add" button is used to add the domain declared to the list of domains.
- ✓ "Update Domains" button is used to add the set of declarations to the CSP model.

Some domains (i.e. D_Mandatory and D_Optional) are declared by default and can be used with the status variables to state that a packet or an information element is mandatory or optional.

4.4.4.4. Unary Constraints

Unary constraints are constraints involving only one variable (packet's field). From the menu choose **State CSP Model → Unary Constraints** to declare unary constraints in the CSP model. A new panel called "Unary Constraints" appears and which allows for the addition of more unary constraints into the CSP model.

The different fields shown in this panel are:

- ✓ "Variable ID:" field is the name of the variable (packet's field) to be used.
- ✓ "Constraint:" field is a list of constraints (operations) to choose from.
- ✓ "Domain ID or Value" field can be either a domain declared using $DOMAIN (See 4.4.4.3) or a value.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

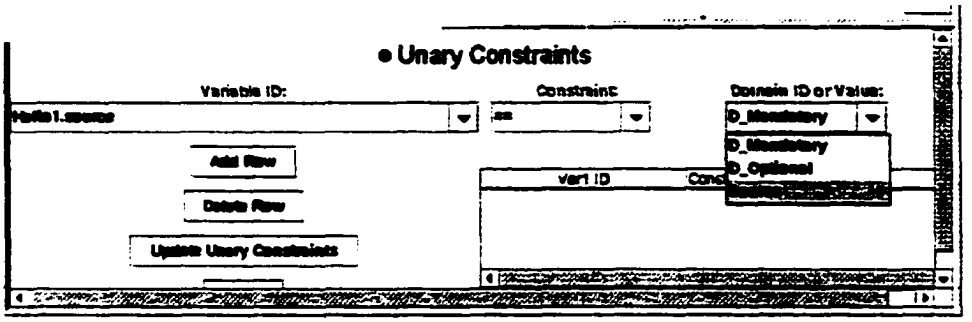

Figure 35 - Domains Available for Unary Constraints

- ✓ "Add Row" button is used to add the unary constraint declared to the list of unary constraints.
- ✓ "Update Unary Constraints" button is used to add the set of declarations to the CSP model.

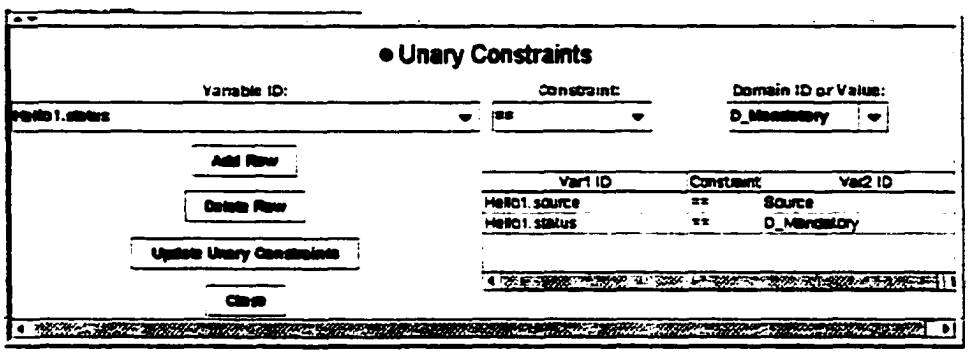

Figure 36 - Unary Constraints Declaration

4.4.4.5. Binary Constraints

Binary constraints are constraints involving two variables (packets' fields). From the menu choose **State CSP Model → Binary Constraints** to declare binary constraints in the CSP model. A new panel called "Binary Constraints" appears and which allows for the addition of more binary constraints into the CSP model.

The different fields shown in this panel are:

- ✓ "Variable 1 ID:" field is the name of the first variable (packet's field) to be used.
- ✓ "Constraint:" field is a list of constraints (operations) to choose from.
- ✓ "Variable 2 ID:" field is the name of the second variable (packet's field) to be used.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

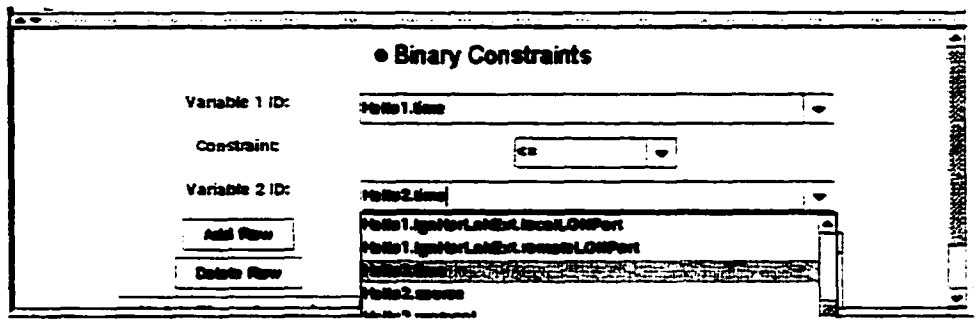

Figure 37 – Variables for Binary Constraints Declaration

✓ "Add Row" button is used to add the binary constraint declared to the list of binary constraints.

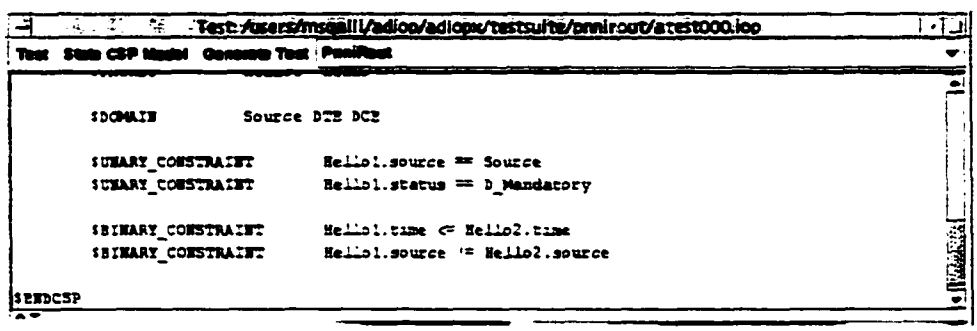

Figure 38 - Constraints Added to the CSP Model

✓ "Update Binary Constraints" button is used to add the set of declarations to the CSP model.

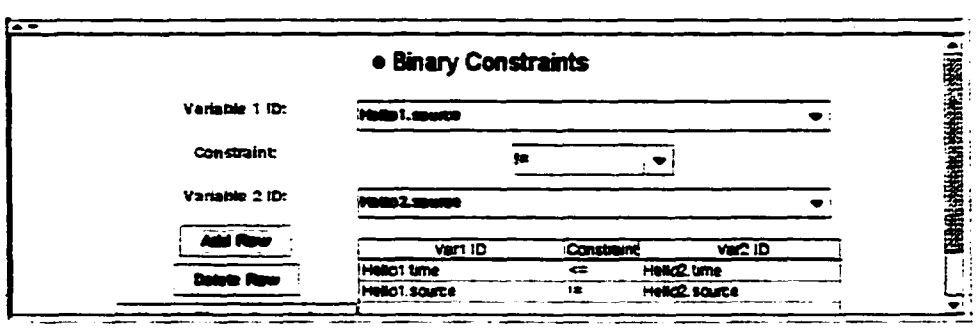

Figure 39 - Binary Constraints Declaration

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.4.6. General Constraints

General constraints are constraints involving one or two variables (packets' field). From the menu choose **State CSP Model → General Constraints** to declare general constraints in the CSP model. A new panel called "Constraints" appears and which allows for the addition of more general constraints into the CSP model.

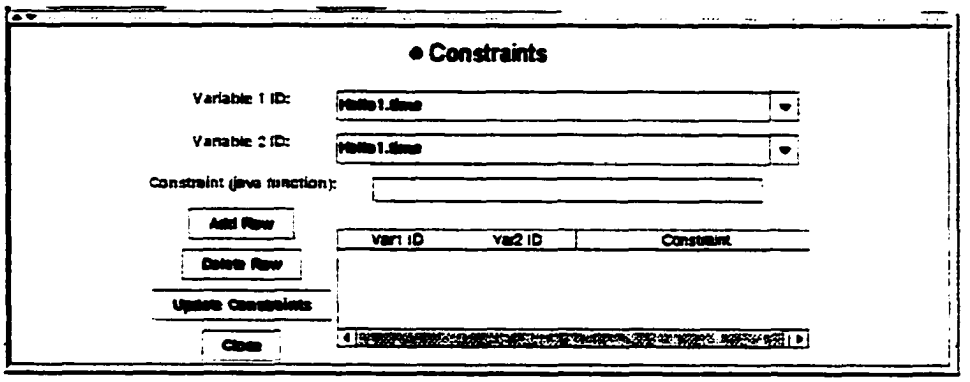

Figure 40 - General Constraints Declaration

The different fields shown in this panel are:

- ✓ "Variable 1 ID:" field is the name of the first variable (packet's field) to be used.
- ✓ "Variable 2 ID:" field is the name of the second variable (packet's field) to be used.
- ✓ "Constraint (Java function):" field is constraint that is represented using Java functions and involving one or two of the variables declared.
- ✓ "Add Row" button is used to add the general constraint declared to the list of general constraints.
- ✓ "Update Constraints" button is used to add the set of declarations to the CSP model.

Warning: There is a bug in the general constraints functionality. After you generate the test case, if you get an error message, you may have to add the "_" character at the beginning of some packet names used in the "java function" part. You will know which ones from the error message.

4.4.4.7. End CSP Model

From the menu choose **State CSP Model → End CSP Model** to include the statement $ENDCSP to the test case which means this is where the test case declaration (CSP model of this test case) ends. This step is usually not needed since "Start CSP Model" will add it as well.

4.4.5. Save a test case

From the menu choose **Test → Save** to save a test case. A test case must be saved using ".iop" extension and under "./adiop/testsuite/*protocolName*"

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.6. Get CSP Model

From the menu choose **Generate Test → Get CSP Model**. This will get the CSP model declared in the text window and put the variables (packets) declared in the "Packets" panel under the list of packets.

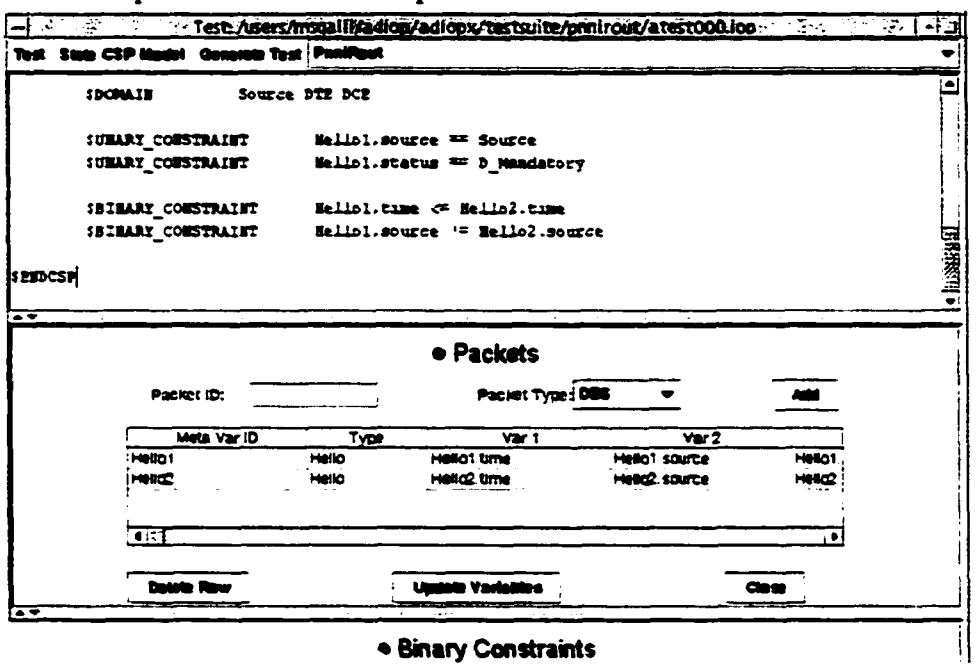

Figure 41 - Get CSP Model

4.4.7. Generate Test from the CSP Model

From the menu choose **Generate Test → Generate Test from the CSP Model**. This will create a ".java" file with the same name as the ".iop" file, then it will compile it to create the ".class" file. This is the file that will be loaded when this test case is run from the Decoder/Diagnoser Window.

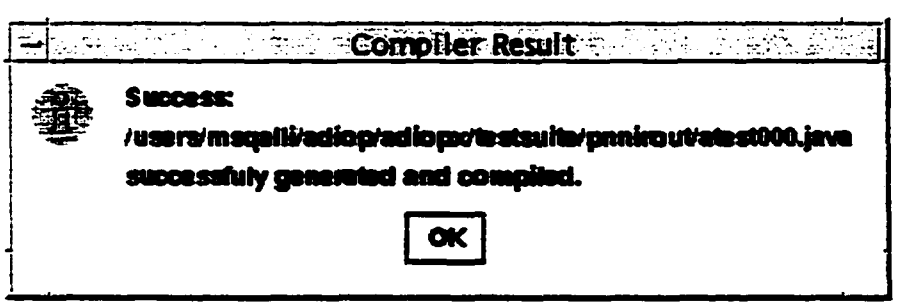

Figure 42 - Result of Generate Test From CSP Model

If there are errors in the compilation, you will get them in the above "Compiler Result" Window.

Refer to sections 4.2.1, 4.2.2, and 4.2.3 for more information on how to run this test case.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.8. Close a test case

From the menu choose **Test → Close** to close the test case that is opened.

Annex: Description of the CSP Modelling Process

In terms of modeling, we propose to model each test from the test suite as a CSP. This guarantees that the CSPs obtained are small and can be solved efficiently. This is also closer to how interoperability testing is done in the real world since the companies testing their devices prefer to get a report of specific tests and failures. The breakdown of the interoperability testing into small tests allows us to do incremental testing and detect easily problems at each level of this testing.

We also propose to use the object-oriented methodology to model these tests. In interoperability testing, an analyzer is usually used to collect the data between the two devices tested. The data collected is then decoded as packets/frames, each representing an event. Thus, it is natural to represent the CSP in term of events. Each event contains many parameters which should be checked against other events' parameters to test for interoperability. Since the constraints exist between the events' parameters, we choose to represent each parameter as a variable in the CSP. The constraints represent restrictions on these variables. However, It is a tedious work to state each one of these variables separately.

The idea is then to represent an event as a metavariable in the CSP representation and each observed event as a metavalue. A metavariable or a metavalue is an object or instantiation of an object representing an event including all the parameters (fields in the object), and methods to manipulate data in these events. A binary metaconstraint is a set of constraints relating variables belonging to two metavariables. The concept of metaconstraint is an abstract concept of representation and design purposes.

Another advantage of this is that one can state an object in the model without having to know all the parameters of that object. This allows for a very concise CSP model statement. From this statement, the system generates the CSP model which is an object with variables as fields and constraints as methods. This model is then integrated to the system and used for testing.

The CSP model is stated in a declarative way. The user needs to specify the events that are expected to be observed for the test to pass. These frames/events are represented as objects.

- $PACKET OneWayInA Hello: This states that the model contains a packet of type `Hello' named `OneWayInA'. Since the `Hello' class is already stored and contains all the information about this type of packets, this statement creates all the necessary fields for the metavariable `OneWayInA', including `time', `source', `status', and `type'.
- The domains are declared in a similar fashion: $DOMAIN D_Hello Hello
- Unary constraints state the name of the variable and the domain of values of this variable: $UNARY_CONSTRAINT OneWayInA.type D_Hello
- Binary constraints are declared as relations between two parameters: $BINARY_CONSTRAINT OneWayInA.time < OneWayInB.time

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- General constraints allow for a larger scope of constraint declaration. They can be either unary or binary: $CONSTRAINT OneWayInA.time OneWayInB.time f(OneWayInA.time,OneWayInB.time) where f(x,y) is a Java statement that returns a boolean and has x and y as its parameters.

By defining packets, there is no need to state each variable separately. And when the packets are defined, the system provides a menu with all the variables belonging to these packets. This menu can be used for stating constraints between these different variables.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix D

Approval of Protocols from the Institutional Review Board (IRB)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UNIVERSITY OF NEW HAMPSHIRE

Office of Sponsored Research
Service Building
51 College Road
Durham, New Hampshire 03824-3585
(603) 862-3564 FAX

LAST NAME	Sqalli	**FIRST NAME**	Mohammed
DEPT	Engineering (Systems Design) Computer Science	**ORIG APP'L**	11/13/2000
		IRB #	2436
OFF-CAMPUS ADDRESS (If applicable)	234-3445 Uplands Dr. Ottawa, Ontario KIV-9N6, Canada	**REVIEW LEVEL**	EXE
		DATE OF NOTICE	12/7/2001

PROJECT TITLE Diagnosing InterOperability Problems and Debugging Models by Enhancing Constraint Satisfaction with Case-Based Reasoning

Thank you for returning to the Institutional Review Board (IRB) your completed Exempt Project Final Report form indicating the above project is closed. Thank you also for enclosing a report of findings for this study

For the IRB,

Julie F. Simpson
Regulatory Compliance Manager
Office of Sponsored Research

cc. File
Prof. Eugene C. Freuder, Computer Science

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UNIVERSITY OF NEW HAMPSHIRE

Office of Sponsored Research
Service Building
51 College Road
Durham, New Hampshire 03824-3585
(603) 862-3564 FAX

LAST NAME	Sqalli	**FIRST NAME**	Mohammed
DEPT	Engineering (Systems Design) Computer Science	**APP'L DATE**	11/13/2000
		IRB #	2436
OFF-CAMPUS ADDRESS (if applicable)	21 Hogan Street, Apt. #10 Nepean, Ontario K2E-5E8, Canada	**REVIEW LEVEL**	EXE
		DATE OF NOTICE	11/13/2000

PROJECT TITLE Diagnosing InterOperability Problems and Debugging Models by Enhancing Constraint Satisfaction with Case-Based Reasoning

The Institutional Review Board for the Protection of Human Subjects in Research has reviewed the protocols for your project as Exempt as described in Federal Regulations 45 CFR 46, Subsection 46.101 (b), category 2 .

Approval is granted to conduct your project as described in your protocol. **Changes in your protocol must be submitted to the IRB for review and approval prior to their implementation. Also, if you experience any unusual or unanticipated results with regard to the participation of human subjects, please report such events to this office promptly as they occur.** Upon completion of your project or after one year, whichever is shorter, please complete the enclosed pink Exempt Project Status Report form and return it to this office.

The protection of human subjects in your study is an ongoing process for which you hold primary responsibility. In receiving IRB approval for your protocol, you agree to conduct the project in accordance with the ethical principles and guidelines for the protection of human subjects in research, as described in the following three reports: Belmont Report; Title 45, Code of Federal Regulations, Part 46; and UNH's Multiple Project Assurance of Compliance. The full text of these documents is available on the OSR information server at http://www.unh.edu/osr/compliance/Regulatory_Compliance.html and by request from the Office of Sponsored Research

If you have questions or concerns about your project or this approval, please feel free to contact our office at 862-2003. Please refer to the IRB # above in all correspondence related to this project. The IRB wishes you success with your research.

For the IRB,

Kathryn B Cataneo
Executive Director
Office of Sponsored Research

cc: File

Prof Eugene C Freuder, Computer Science

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.