Solving InterOperability Problems Using Object-Oriented CSP Modeling

Mohammed H. Sqalli and Eugene C. Freuder
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
msqalli,ecf@cs.unh.edu

Abstract

ADIOP (Automated Diagnosis of InterOperability Problems) models, diagnoses and debugs interoperability problems using an integration of Constraint-Based and Case-Based Reasoning. The specific problem domain is interoperability testing of protocols in ATM (Asynchronous Transfer Mode) networks. Each test case is first modeled as a Constraint Satisfaction Problem (CSP) using an Object-Oriented approach. The CSP models are then used to diagnose and solve interoperability problems. CSP algorithms are adapted to the OO approach and to take advantage of the specialized problem structure. These provide a better diagnosis of the interoperability problems including an accurate and concise explanation of the testing performed. These algorithms are evaluated to show their usefulness. A CSP model may be incomplete or incorrect and Case-Based Reasoning (CBR) is used to compensate for that. In this paper, we provide an overview of the first two parts of the ADIOP system, namely modeling and diagnosis; the third part, model debugging, is not addressed in this paper.

1 Introduction

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 (what is stated in the specifications of the protocol tested). The specific domain we are using is interoperability testing of protocols in ATM (Asynchronous Transfer Mode) networks.

The application presented in this paper is motivated by the work done at the UNH-RCC InterOperability Lab (IOL) at the University of New Hampshire. One of the main challenges in interoperability testing at IOL is how to diagnose and debug interoperability problems in a timely manner. At the present, these tasks are done manually by the experts who work at IOL. This can be an exhausting task since there can be a large amount of data to check and since no records are kept on how previous problems have been solved.

In summary, there are two major concerns to address:

1. Checking a large amount of data to find out where there is a mismatch between what is expected and what is observed.
2. Wasting time in diagnosing problems that have been diagnosed before at IOL or in solving problems that are very similar to old problems solved.

There is a need to make the process of diagnosing interoperability problems easier, quicker and more efficient. The work we present in this paper aims at solving some of these problems by automating the process of running interoperability tests and diagnosing interoperability problems.

We developed the ADIOP (Automated Diagnosis of InterOperability Problems) system to model, diagnose and debug interoperability problems using an integration of Constraint-Based and Case-Based Reasoning.

In this paper, we present a framework for modeling interoperability testing using an integration of CSP and OOP (Object-Oriented Programming). Each test case from the interoperability test suite is represented as a CSP model and implemented as an object with metavariables and constraints respectively representing its parameters and methods.

We discuss how these CSP models are used to diagnose interoperability problems. Some CSP algorithms are adapted to the OO approach and to take advantage of the specialized problem structure. These provide a better diagnosis of the interoperability problems including an accurate and concise explanation of the testing performed. We evaluate these algorithms and show their usefulness. In particular we show how specialized inference methods can provide better explanations than the traditional search and inference methods.

A CSP model may be incomplete or incorrect and CBR is used to compensate for what is missing in a CSP model. In this paper, we provide an overview of the first two parts of the ADIOP system, namely modeling and diagnosis. The third part, model debugging, is not addressed in this paper.

2 InterOperability Testing

One mission of IOL is to provide testing services for vendors of computer communications devices. The IOL is mainly used by a community of over 70 vendors to verify the interoperability and/or conformance of their computer communications products.

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 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. Two devices are interoperable if the observations match the protocol specifications.

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 with alternative solutions. The application domain used in this paper is interoperability testing of ATM protocols.

This paper is focused on testing protocols that run over ATM networks, and most of the examples used are taken from the PNNI Routing protocol. These are taken from the ATM Forum document "AF-TEST-CSRA-0111.000" which provides the test suite for performing PNNI interoperability testing [PNNI-IOP, 1999].

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 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 certain quality of service (QoS) requested by the user.

Private Network-Network Interface

PNNI (Private Network-Network Interface) protocol provides dynamic routing, supports QoS, hierarchical routing, and scales to very large networks [PNNI-1.0, 1996]. Two switches running PNNI are able to send data to each other either via direct link or by using a route. 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.

2.1 Current Problem Solving Techniques

Interoperability testing is done by analyzing the data collected using monitors. These are usually put next to a device being tested. Figure 1 shows the physical setup and the steps for interoperability testing.

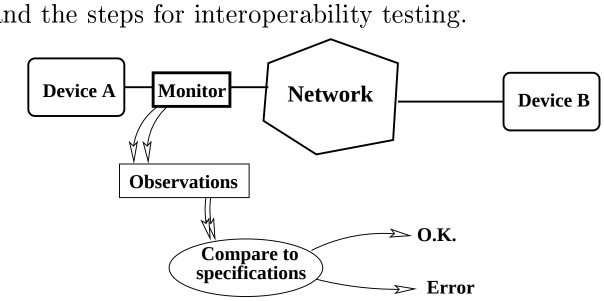

Figure 1: InterOperability Testing.

At the present, these tasks are done by the experts who work at IOL. Doing these steps manually has many disadvantages:

1. It is time consuming because the user may spend hours of analysis to find what the problem is and diagnose it. The large amount of data to be analyzed makes it an even harder and slower process.
2. It is not guaranteed that a problem that has been solved once can be solved later, or that a similar problem can be solved. The user may not remember how the problem was dealt with before, or the user

might be a different person from the one who solved the problem before. And because there is no trace kept of how the problem was solved, more time is spent on solving the same problem.

2.2 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 test case to be performed using constraint representation.

First, each test case from the test suite is modeled as a Constraint Satisfaction Problem (CSP). CSP provides more flexibility 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 (metavariables) and the constraints relating them. Then, the diagnosis is done by checking whether all the constraints are satisfied. If a diagnosis of the problem is found, then it is reported. When the system is unable to correctly diagnose the problem, Case-Based Reasoning (CBR) is applied to determine what is missing in the model of the test specification, because the model may be incomplete or incorrect.

The model debugging component of the ADIOP system including CBR is not addressed in this paper. The architecture of this component is presented in [Sqalli and Freuder, 1998].

3 CSP Modeling of InterOperability Testing using OOP

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. 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 modelling. 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 electromechanical systems." There are two folds 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 paper, we discuss the first part which involves CSP modeling using Object-Oriented Programming (OOP).

In this section we present a simple modeling language that allows the user to build models of the interoperability test cases. CSP provides a uniform framework 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 much better flexibility of representation 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 which 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 with the tester.

3.1 Many-Models Architecture

Modeling of interoperability testing is performed using a many-models architecture where each test case is modeled as a CSP. In [Sqalli and Freuder, 2001], we state the advantages and disadvantages of this design, why we used it over the One-Model architecture. In this architecture, CSP models are derived from test cases in the test suite written from the protocol specification (Figure 2).

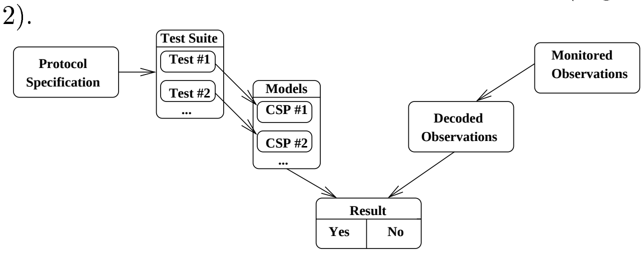

Figure 2: Many Models Architecture

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. Then the constraints defined in the CSP model are checked for consistency. If all the constraints are satisfied for an assignment, the CSP model has a solution and the interoperability test case passes. This is then repeated for each test case in the test suite.

3.2 Description of the CSP Modeling Process using OOP

In this paper, 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.

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 interoperability testing into small test cases 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 which 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 a 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.

Definition 1 (Metavariable) : *A metavariable in the CSP model refers to the representation of a packet that encompasses many variables. Some of these variables are the 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 (Metavalue) : *A metavalue in the CSP model refers to the data captured of a packet. This data is used to instantiate a metavariable.*

In the ADIOP project, we only use 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 in [Roy and Pachet, 1997] [Paltrinieri, 1994a] [Paltrinieri, 1994b] [Stone, 1995] and [Puget and Leconte, 1995].

The closest work to ours is what has been done in [Paltrinieri, 1994a] [Paltrinieri, 1994b]. 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.

There has been some related work on modeling protocol testing as well in [Marrero *et al.*, 1997], [Sqalli and Freuder, 1996a], [Riese, 1993a], [Riese, 1993b].

More details on related work to the papers mentioned above can be found in [Sqalli and Freuder, 2001].

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 to pass. These packets are represented as objects. An example of a CSP model is stated in (Figure 3) 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.

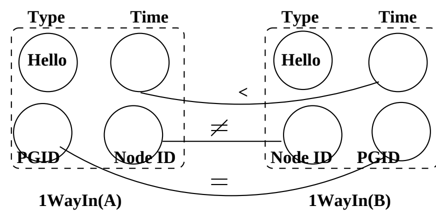

Figure 3: A Modeling Example

[Sqalli and Freuder, 2001] presents in more details the modeling language, the implementation of objects and the class hierarchy and inheritance used in the ADIOP system.

3.3 Modeling Interface

The modeling interface for ADIOP 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. The Test Suite Builder (TSB) component of ADIOP provides the functionality for modeling a test case as CSP.

The Graphical User Interface (GUI) used for modeling allows the user to declare metavariables, domains, and constraints in a very efficient manner. The user does not have to know the details of the object being manipulated.

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 4 is constructed from the structure of ADIOP directories. If a new protocol is added to these directories, it will be dynamically loaded and shown in this menu.

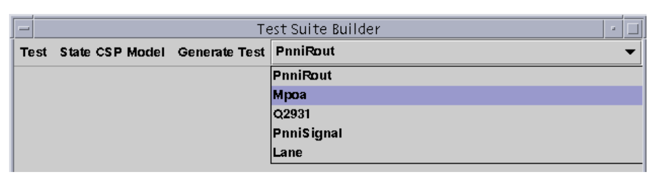

Figure 4: Protocols List in the Test Suite Builder Window

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.

The packet types 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.

The domains can be declared as a set of discrete values. These are used to declare unary 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 5).

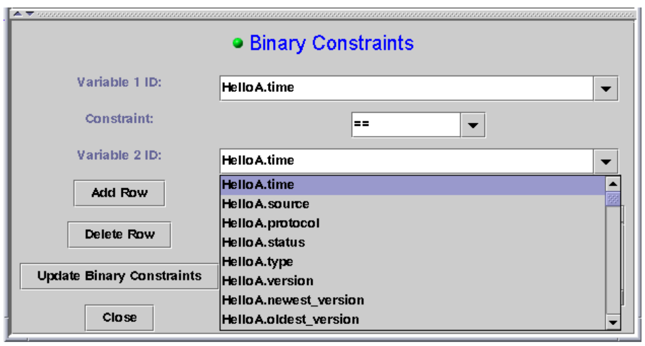

Figure 5: Packet's Parameters List

These variables are dynamically loaded using the structure of the metavariable (packet) they are part of. ADIOP provides also 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 added to the CSP model definition using the **update** button.

In addition, this GUI is used to decode packets from binary format to readable text format. It also provides the tools for running interoperability tests that have been built and generating reports of testing results.

3.4 Test Cases as Objects

When the model definition is completed, the user can generate an object for this test case. 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.

ADIOP constructs a menu in the Decoder/Diagnoser window from the structure of the directories under the *testsuite* directory where all objects representing test cases are stored. If a new protocol is added or more test cases are generated, the menu will get automatically updated. Figure 6 shows part of the menu generated in the Decoder/Diagnoser window.

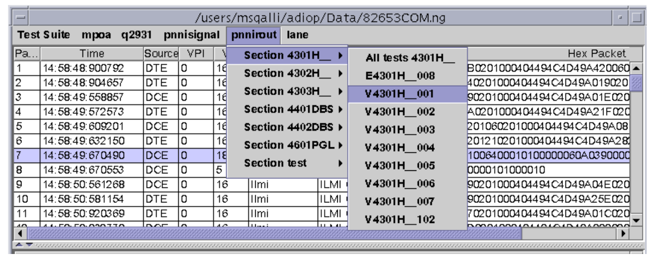

Figure 6: Test Suite Menu

3.5 Modeling Language

The model is stated in a very simple language. Details of this modeling language including the syntax, the keywords and their meaning can be found in [Sqalli and Freuder, 2001].

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 can not operate.
```

The following is a CSP representation of this test using ADIOP's modeling language:

```
$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
```

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.

4 Constraint-based Diagnosis of InterOperability Problems

As shown in the last section, the ADIOP (Automated Diagnosis of InterOperability Problems) system provides a modeling language based on CSP that allows the user to build test cases. We use OOP to implement ADIOP, and we explained why we have choosen this approach and how it is being used. Each test case is represented as an object which is the corresponding CSP model. We have also shown how this is used in a many-models architecture. [Sqalli and Freuder, 2001] presents a more detailed version of the modeling component of ADIOP.

In this section, 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 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.

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

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

Definition 3 (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 the conformance testing which states that each device by itself is conformant in its behavior to the protocol specification.*

There has been some related work on diagnosis. A device can be modeled based on its components and their expected behavior [Hamscher and Struss, 1990]. [Abu-Hakima, 1994] presents the DR (Diagnostic Remodeler) algorithm for automating model acquisition for diagnosis. The DR algorithm was implemented to combine model-based diagnosis (MBD) and fault-based diagnosis (FBD). Model-based approach has been used in [Riese, 1993a] for interpreting observations and diagnosis. In [Riese, 1993b], a protocol is represented as a set of constraints derived from an Extended FSM (EFSM). In Model-Based Diagnosis, an error is defined as: a conflict between observations and expectations based on the model.

One approach to model-based diagnosis has taken diagnosis to be a constraint satisfaction problem (CSP) [Fattah and 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 and Wallace, 1992].

4.1 Modeling, Decoding and Diagnosis

Figure 7 shows three components of ADIOP, namely modeling, decoding and diagnosis. Modeling and decoding are two steps that are needed before the diagnosis takes part. The figure shows the interrelation between these three components.

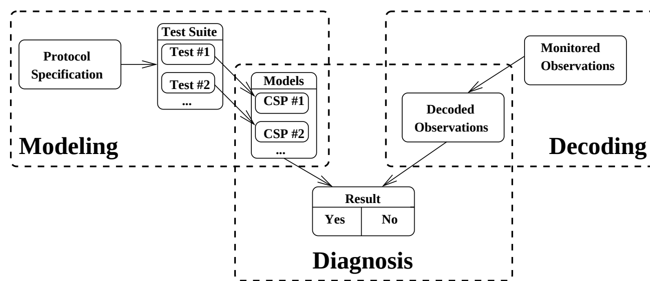

Figure 7: Modeling, Decoding and Diagnosis Components

All the test cases built using the modeling component are accessible through the menu in the Decoder/Diagnoser window of ADIOP. One input for the diagnosis component is the objects (CSP models) representing test cases that we presented in the previous section.

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 another input for the diagnosis component.

The Decoder/Diagnoser window of ADIOP is divided into two panels. The upper one shows the summary of decoded observations as its data and the test cases that were built by the modeling component as its menu list (Figure 6). The lower panel in the window shows the details of the packet highlighted in the upper panel which contains the summary of the packets observed.

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 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 later in this section. ADIOP also generates an explanation of the diagnosis. A report can be generated for one test case or a set of test cases from the same category (section).

4.2 Diagnosis of InterOperability Problems

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

Since we use CSP for modeling, this allows us to take advantage of methods and algorithms that already exist for solving CSPs. 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 an accurate and concise explanation of the problems detected.

ADIOP automates the process of interoperability testing by providing a useful tool for the user to come up with reports of interoperability testing of different devices. These reports are usually done manually by looking at the monitored data using an analyzer. Our goal is to simplify this task and provide an easy-to-use user interface that allows the decoding and the analysis/diagnosis of the observed data including reports generation.

4.3 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. 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.

Our focus in this paper 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 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 very simple and fast preprocessing of packets.

Constraint Satisfaction Methods

When we have the CSP representation of a problem, we can use different methods to solve it 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). Node consistency is the lowest type of consistency. It checks whether unary constraints (constraints involving one variable, e.g., $V < 3$) are satisfied.

The drawback of search is that the time of exploring all the possibilities grows exponentially with the number of variables. The drawback of inference is that lower consistency checking such as NC and AC are not usually enough to solve the entire problem in most cases. For solving the entire problem using only inference, one needs to perform higher consistency checking that includes many variables, but that leads again to the same problem of combinatorial explosion 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 backtrack 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 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 *et al.*, 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 and Leconte, 1995] [Wallace, 1996]. These applications have improved the CSP paradigm and made it more widely used.

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, or how many solutions the problem has.

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 instead of variables and values. The algorithm for backtracking stops at the first solution if it exists. 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.

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 8.

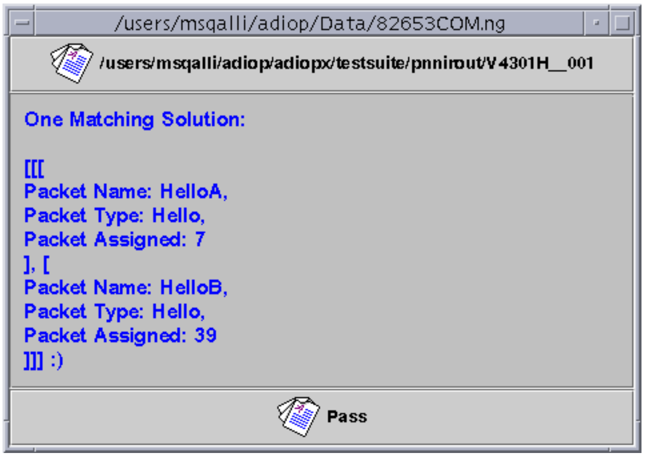

Figure 8: ADIOP's Result Window of a Successful Test Case

The user is not usually interested in learning how the solution was found. However the user may want to know the metavalues (observed packets) that were assigned to metavariables to obtain the presented solution. Multiple solutions are not important for the user as the outcome of diagnosis is not based on how many solutions were found but rather on whether there is a solution, i.e., whether the test case passes or fails. And that is why the issue we have with diagnosis and explanation has more to do with failures than with successes.

When there is no solution to the test case being executed, it fails 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 mainly 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 problem.

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 and Freuder, 1996b]

We propose to use search supplied 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.

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 fed up 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 taken advantages of 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 the last section, 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. We are rather interested in the NC where a node is a metavariable.

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 we 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 the user 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.

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. An algorithm makes use of domain reduction using the **protocol** variable. In addition, the value of the **pack-**

etType 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. 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 and 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.

4.4 Explanation

Inference is used mainly to reduce the domains of metavariables. The different results that the inference leads to are used as the input to some templates so that the user gets a useful explanation for the outcome of a test case execution. Some of these templates are used with the search algorithm but these are not very useful explanations for the user.

The results obtained using inference are used with some defined templates to provide the user with an explanation of the results of running a test case. Here we present an example of these templates:

There are less 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 packets 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 9.

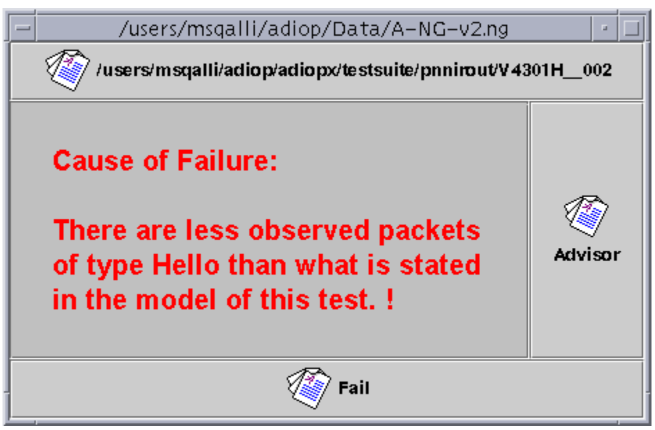

Figure 9: ADIOP's Result when packets of a packet type are fewer than required

4.5 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. An example of this is shown in figure 8 and figure 9.
- Section test cases 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 10).

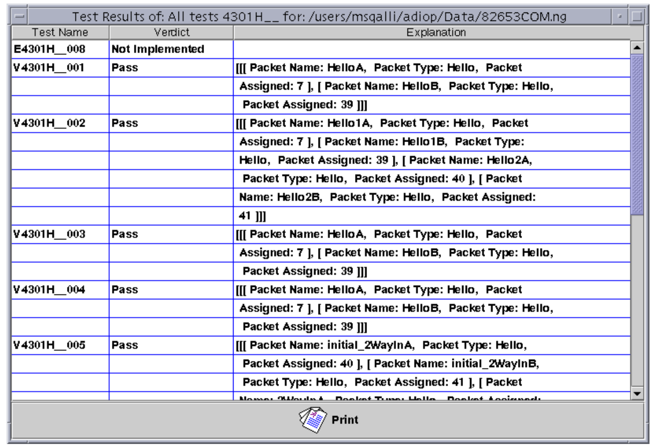

Figure 10: ADIOP's Test Cases Report of One Section

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

4.6 Evaluation

Solvability

In this section, we present an evaluation of the algorithms used in ADIOP. We used four captured data (observations). Three observations are for the PNNI Routing protocol and one is for the LANE (Local Area Network Emulation) protocol. We have used captures from real-world data obtained at 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 only.

Table 1 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 capture "82653com", excluding one test case makes the overall result jump from 232% down to 67%. We are not able to find out why we got such results for these test cases. These test cases were considered as outliers.

The "Pre" column shows the preprocessing time. Preprocessing uses the inferences we introduced earlier in this section. The "BtPre" column shows the time it takes to find a solution or none using backtracking after preprocessing.

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)
82651010	72	PnniRout	17	139.2	722.6	861.8	1129.6	**76%**
82653com	91	PnniRout	17	49	1212.6	1261.6	543.2	232%
82653com 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%**

Table 1: Summary of Results of Running Test Cases on Different Captures

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 effort that a preprocessing+backtracking uses compared to backtracking alone. A ratio of less than 100% shows that time was saved using preprocessing. The overall results show a reduction of effort to between 42% and 76% for each captured data.

The test cases we run were all related to the protocol of the data captured. If the user run 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 since we knew for which protocol each capture is, and since the users would usually make sure that the test cases being run are for a capture of the same protocol. This preprocessing is also useful when the user is not sure of what data they are testing, and it will save them a lot of time not having to search all the tree space for a solution. For instance, we tried one test case from the "Pnni Signaling" protocol on a capture from "Pnni Routing" protocol and the savings were about 100%. This is explained by the fact that preprocessing time is linear and there is no backtracking performed, while when backtracking alone is used it is a search of the whole tree of possibilities.

Explanation

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

In the PNNI capture shown in table 1 we had many test cases where the preprocessing was enough to solve the problems as 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 the preprocessing was enough to solve the problem, the explanation that was produced was a more meaningful one to the users.

Out of the 69 test cases we run, 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 test cases.

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 $O(s)$. That of "PacketTypePreprocess" is $O(n^2 s)$. Thus, the complexity for performing the preprocessing is $O(n^2 s)$. The complexity of search after preprocessing is $O(r^n)$ with $r \leq s$. When preprocessing solves the problem, then the complexity is of $O(n^2 s)$. And 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.

4.7 Summary

We discussed CSP modeling of interoperability testing using Object-Oriented Programming. CSP modeling was presented using a many-models architecture. The CSP modeling process using OOP was outlined with a description of how objects are used in modeling. We 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. A full example of CSP modeling of an interoperability test case was provided.

In the diagnosis section, 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 an overview of explanation and what templates are being employed. An evaluation of the performance of the different algorithms used was presented and showed that some of the specialized algorithms can lead to better results in term of time and explanation.

5 Acknowledgments

This material is based in part on work supported by the National Science Foundation under Grant No. IRI-9504316. Special thanks to the staff and students from the ATM consortium of the InterOperability Lab (IOL) at the University of New Hampshire for their support and feedback.

References

[Abu-Hakima, 1994] S. Abu-Hakima. Automating Model Acquisition by Fault Knowledge Re-use, DR, the Diagnostic Remodeler Algorithm. In *International Workshop on the Principles of Diagnosis*, pages 1–6, New Paltz, NY, October 1994.

[Fattah and Dechter, 1992] Y. El Fattah and R. Dechter. Empirical Evaluation of Diagnosis as Optimization in Constraint Networks. In *Working Papers of the Third International Workshop on Principles of Diagnosis (DX-92)*, 1992.

[Freuder and Wallace, 1992] E.C. Freuder and R.J. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[Hamscher and Struss, 1990] W. Hamscher and P. Struss. Model-Based Diagnosis. In *AAAI-90 Tutorial Notes, Eighth National Conference of Artificial Intelligence*, pages 1–179, Boston, Massachusetts, USA, July 1990.

[Marrero *et al.*, 1997] Will Marrero, Edmund Clarke, and Somesh Jha. Model Checking for Security Protocols. Technical Report CMU-CS-97-139, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213, May 1997.

[Paltrinieri, 1994a] Massimo Paltrinieri. 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.)*, pages 299–311, Rosario, Orcas Island, Washington, USA, May 1994. Springer.

[Paltrinieri, 1994b] Massimo Paltrinieri. Visual Environment for Constraint Programming. In *11th International Symposium on Visual Languages*, pages 118–119, 1994.

[PNNI-1.0, 1996] PNNI-1.0. *Private Network-Network Interface Specification Version 1.0 (PNNI 1.0)*. The ATM Forum, Technical Committee, March 1996. af-pnni-0055.000.

[PNNI-IOP, 1999] PNNI-IOP. *Interoperability Test for PNNI Version 1.0.* The ATM Forum, Technical Committee, February 1999. AF-TEST-CSRA-0111.000.

[Puget and Leconte, 1995] Jean-Francois Puget and Michel Leconte. Beyond the Glass Box: Constraints as Objects. In *Logic Programming, Proceedings of the 1995 International Symposium (ILPS): John W. Lloyd (Ed.)*, pages 513–527, Portland, Oregon, December 1995. MIT Press.

[Riese, 1993a] M. Riese. Diagnosis of Communicating Systems: Dealing with Incompleteness and Uncertainty. In *Proceedings IJCAI-93*, pages 1480–1485, 1993.

[Riese, 1993b] M. Riese. Diagnosis of Extended Finite Automata as a Constraint Satisfaction Problem. In *Proceedings of the Fourth International Workshop on Principles of Diagnosis (DX-93)*, pages 60–73, 1993.

[Roy and Pachet, 1997] Pierre Roy and Francois Pachet. Reifying Constraint Satisfaction in Smalltalk. *Journal of Object-Oriented Programming*, 10(4):51–63, 1997.

[Sabin *et al.*, 1994] D. Sabin, M. Sabin, R. Russell, and E. Freuder. A constraint-based approach to diagnosing distributed software systems. In *Proceedings of the Fifth International Workshop on Principles of Diagnosis (DX-94)*, 1994.

[Sqalli and Freuder, 1996a] M. Sqalli and E. Freuder. A Constraint Satisfaction Model for Testing Emulated LANs in ATM Networks. In *Proceedings of the 7^{th} International Workshop on Principles of Diagnosis (DX-96)*, pages 206–213, Val Morin, Quebec, 1996.

[Sqalli and Freuder, 1996b] M. Sqalli and E. Freuder. Inference-Based Constraint Satisfaction Supports Explanation. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 318–325, Portland, Oregon, August 4-8, 1996.

[Sqalli and Freuder, 1998] M. Sqalli and E. Freuder. 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)*, pages 266–273, Cape Cod, Massachusetts, May 24-27, 1998.

[Sqalli and Freuder, 2001] M. Sqalli and E. Freuder. 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), (to appear)*, Seattle, Washington, USA, August 7-9, 2001.

[Stone, 1995] Nicholas D. Stone. Object-Oriented Constraint Satisfaction Planning for Whole Farm Management. *AI Applications*, 9(1), 1995.

[Van Hentenryck *et al.*, 1995] P. Van Hentenryck, D. McAllester, and D. Kapur. Solving Polynomial Systems using a Branch and Prune Approach. *SIAM Journal on Numerical Analysis*, 1995.

[Wallace, 1996] M. Wallace. Practical Applications of Constraint Programming. *Constraints - An International Journal*, 1(1-2):139–168, 1996.