From: IAAI-01 Proceedings. Copyright © 2001, AAAI (www.aaai.org). All rights reserved.

Constraint-Based Modeling of InterOperability Problems Using an Object-Oriented Approach

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 is an application for Automated Diagnosis of InterOperability Problems. Interoperability testing involves checking the degree of compatibility between two networking devices that implement the same protocol. In ADIOP, each interoperability test case is first modeled as a Constraint Satisfaction Problem. Object-Oriented Programming is used to implement ADIOP. In this paper, we present the modeling language we use in ADIOP and how it allows the user to easily and efficiently create test cases and use them for diagnosis. The specific domain of application is interoperability testing of protocols in ATM (Asynchronous Transfer Mode) networks.

Introduction

We present a simple modeling language that allows the user to build models of 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 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 using a many-models architecture 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. The diagnosing part of ADIOP is not addressed in details in this paper.

Copyright © 2001, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

CSP Modeling of InterOperability Testing

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 paper is in modeling interoperability tests. 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 electro-mechanical systems."

A protocol specification is usually written by an organization such as standardization bodies (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 is written by one of these organizations. In the protocol we are using in this paper that is PNNI (Private Network-Network Interface), 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 details 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. Each test case's result provides a 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.

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 1996) (Riese 1993a) (Riese 1993b).

This CSP model can then be used to test the interoperability of two devices by checking the observations against the CSP model. 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, then the interoperability test passes.

Many Models Architecture

In this design, the CSP models are derived from the test cases in the test suite written from the protocol specification (Figure 1). In this paper, we use this form of modeling where each test is represented as a CSP. If the observations form a solution to this CSP then the corresponding test case passes. This is then repeated for each test case in the test suite.

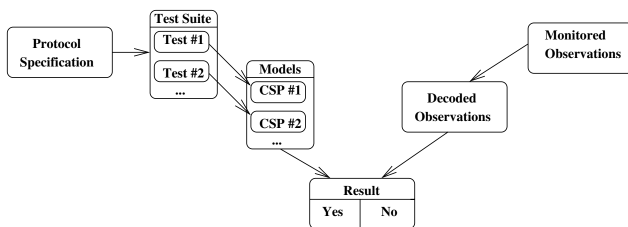

Figure 1: Many Models Architecture

The advantages of this form of modeling are that:

- It is easy to come up with 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
- 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, which is not discussed in this paper, addresses inconsistencies independently of their origin
- Some parts of testing might be included in more than one test case causing redundant testing. This is not a major concern since we can copy parts of one model into another one.

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 provides conveniently the development tools for GUI-based and web-based software. Our system ADIOP is implemented using Java.

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.

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.

There are many properties in OOP that make modeling more efficient. Two of which we are interested in here are: Encapsulation and Inheritance.

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 can not 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.

Description of the CSP Modeling Process Using OOP

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 de-

vices 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.

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

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.*

Definition 3 (Metaconstraint) : *A metaconstraint is a set of constraints relating variables belonging to one or more metavariables. Constraints are defined using variables as their arguments.*

In this paper, 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 (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 paper. 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 paper.

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 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. 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 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 2) 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.

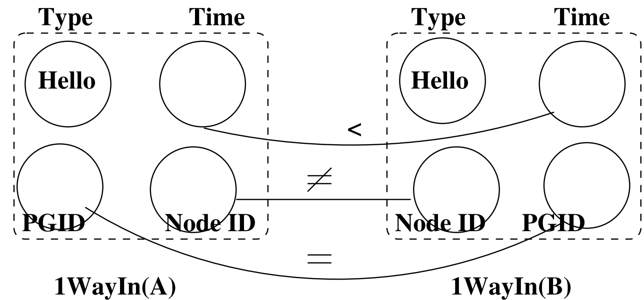

Figure 2: A Modeling Example

A $PACKET statement is used to declare metavariables (packets). More details of the modeling language are provided in a later section of this paper. 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 and showing all the variables belonging to this packet (Figure 3). This list can be used for stating constraints between these different variables.

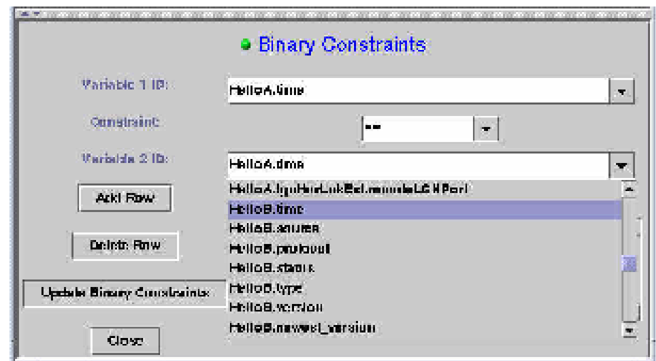

Figure 3: Packet's Parameters List

Modeling with Objects

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 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 Object-Oriented approach (OO), where a packet is represented as an object.

Since 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 which 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 4 shows a representation of the directory structure of the *packet* package.

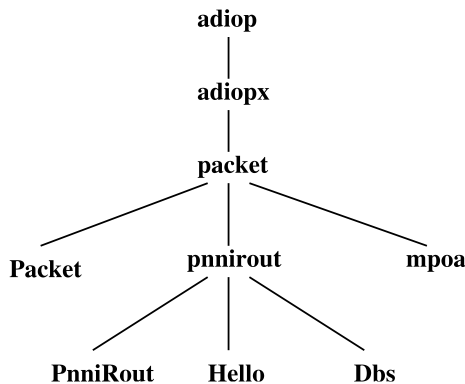

Figure 4: Directory Structure of the *packet* Package

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).

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 packets. As shown in figure (Figure 5), 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 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.

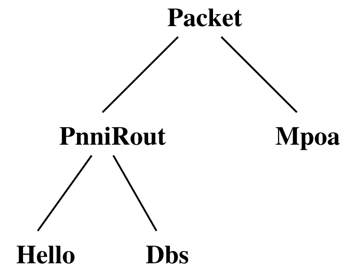

Figure 5: Class Hierarchy of the **Packet** Class

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 this 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 these packet types to have them available for use by the decoder and the CSP modeling component. 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.

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

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

Figure 6: Protocols List in the Test Suite Builder Window

Each test case is built as a file with the *.iop* extension. This file may contain a description of the test case taken usually from the IOP specification document. This file's main section is the CSP model defining the variables and constraints for this test case. The CSP model is defined between two ADIOP keywords, i.e., **$CSP** and **$ENDCSP**.

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.

The packet types shown in figure 7 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.

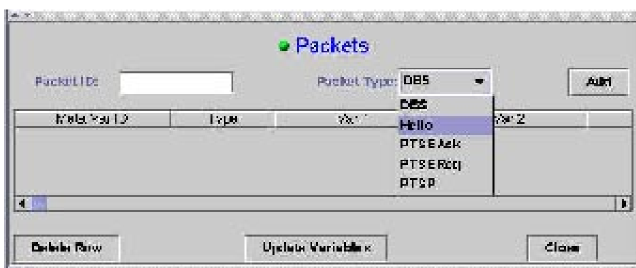

Figure 7: Packet Types List in the Test Suite Builder Window

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 3). 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** function.

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. This is not the subject of what we present in this paper.

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.

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 8).

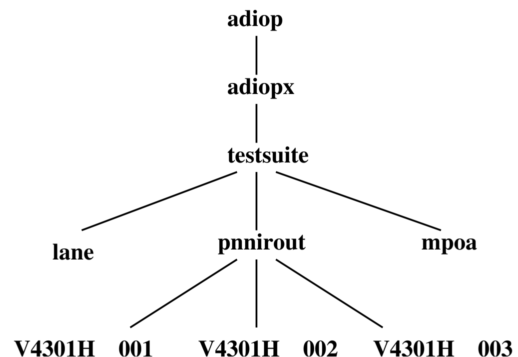

Figure 8: The *testsuite* Directory Hierarchy

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 9 shows the menu generated in the Decoder/Diagnoser window.

Modeling Language

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

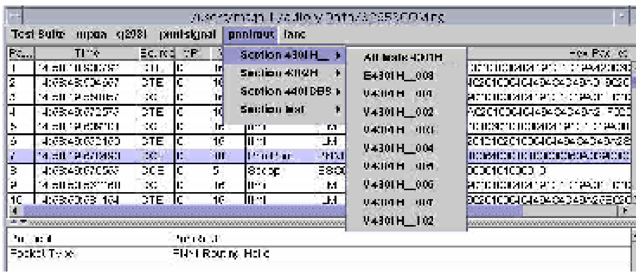

Figure 9: Test Suite Menu

- **$CSP**: This states that the CSP model declaration starts at this point. This statement is added automatically when a test case is created.
- **$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 **$PACKET packet_name packet_type** is sufficient for defining all the parameters and methods needed for this packet including its fields (CSP variables) that can be used for stating constraints.
- **$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 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 have both to be parameters in one or two of the objects declared with $PACKET. **operation** can be one of the following operations: $==, !=, <=, >=, <, or >$.
- **$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 a broader constraint statements. The function used here can be made reusable by storing it under the textit"util" directory of ADIOP. This also allows for the use of a more complex functions.
- Comments can be included using the "// comments"

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 can not
            operate.
```

The following is a CSP representation of this test 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
```

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.

Application of CSP Modeling

The CSP models are used to diagnose and solve interoperability problems (figure 1). All the test cases built using the ADIOP's modeling component are accessible through the menu in the Decoder/Diagnoser window of ADIOP (figure 9).

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.

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.

The advantage of CSP is that it is a reasoning mode that provides both modeling and problem solving within the same framework. 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 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.

We did an evaluation of ADIOP's debugging component and the summary shows that 50 test cases out of 69 built produced a meaningful explanation, which makes about 73% of test cases.

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. 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.

(Puget & Leconte 1995) propose to give access to the constraints as first class citizen of the CLP language. They implemented their approach into an OO language, where constraints are explicitly represented by objects. Their implementation, ILOG Solver, used an abstract machine which 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. Again 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.

(Roy & Pachet 1997) discuss the problem of the representation of 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 solve efficiently complex constraint satisfaction involving Smalltalk structures. The same as what we stated in the previous references can be said about the difference between this work and ours.

(Paltrinieri 1994b) has abstracted both variables and constraints as defined in the classical CSP to a new, more compact model, called 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. The OOCSP is converted into an equivalent CSP, which is then solved through a traditional constraint-programming language. The definition of CSP is enhanced through concepts deriving 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 closest 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.

Summary

In this paper we discussed CSP modeling of interoperability testing using Object-Oriented Programming. CSP modeling was introduced in Section 1. The different modeling architectures were presented in Section 2 and why we opted for a many-models architecture. The CSP modeling process using OOP was outlined in Section 4. A more detailed description of how objects are used in modeling is provided in Section 5. In Section 5.2, the class hierarchy and inheritance that we used in CSP modeling are presented. The modeling GUI is covered in Section 6. Section 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 is the subject of Section 8. A full example of

CSP modeling of an interoperability test case is shown in Section 9. We finally present related work in Section 10 on the integration of CSP and OO.

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. This work has benefited from the reviewers' comments.

References

Campione, M., and Walrath, K. 1998. *The Java Tutorial, Second Edition: Object-Oriented Programming for the Internet (Java Series)*. Addison-Wesley Pub Co.

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.

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

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.

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

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.

Sqalli, M., and Freuder, E. 1996. 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.

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

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