Graphical User Interface Using a Reflective Architecture[*]

Mahesh Patel, Sara Stoecklin, Daniel G. Schwartz
Department of Computer Science
Florida State University
Tallahassee, FL 32306, U.S.A.
{mahpatel, stoeckli, schwartz}@cs.fsu.edu

Abstract: ***Reflective architectures are used to allow a software system to define its behavior at runtime based on information that can be stored in metadata. In this paper we expand the approach used to build systems using reflective architectures to include the development of graphical user interfaces. We give an example of a dynamically built GUI for a generic case-based reasoner. This example shows how a clean separation of domain-specific data from application operational behavior can be used to create GUIs dynamically using a defined metadata dictionary. It represents an appropriate solution if the system domain changes or if users need to dynamically configure and extend their applications. This reflective architecture can lead to a system that allows users to modify the behavior of a program without programming.***

Keywords: Case-Based Reasoning, Metadata Architecture, Reflection

1 Introduction

The complexity of today's problem environments and the increasing expectations placed on the development of software make it difficult to design systems that satisfy all the expectations of the users. The pressure placed on developers to produce software in a short period of time jeopardizes the development of consistent and reusable software.

[*]This work was supported by the US Army Research Office, grant number DAAD19-01-1-0502.

One time consuming activity is the development of user interfaces to support a system. In this paper, we address a specific instance of this, namely Graphical User Interface (GUI) development. GUIs are normally developed for application programs, which in turn are developed as applications for specific problem domains. Accordingly, the GUI development in this context requires that programmers build domain specific GUI components for screens and visual reports. They typically do this by using generic components such as text fields, labels, etc. They then define the domain-specific attributes, such as the needed component labels, text field lengths, ranges of valid values, error messages, component colors, etc. If it is later desired to build a GUI for a similar, but different application, however, most of the original code needs to be rewritten. The results of the earlier time-consuming development activity normally cannot be reused.

With the advent of reflective software architectures [1, 2, 3], however, it is now possible to design application frameworks that apply across a large collection of closely related, or structurally similar, application domains. Such a framework embodies abstractions of the key elements of a collection of domains in such a way that a specific application for a given domain can be obtained by instantiating the framework. The instantiation process entails specifying the necessary domain-specific elements at the level of metadata. Because of this, software frameworks employing this approach are said to embody "adaptive architectures." They can be adapted to new application domains simply by modifying the metadata. The aim of this paper is to show how this approach can be used for GUI generation.

We illustrate the technique by describing its use in conjunction with an adaptive framework for case-based reasoning.

2 An Adaptive Case-Based Reasoning Framework

An adaptive generic case-based reasoning (CBR) framework has been presented in [4]. For any CBR system there are three input files: the problems data file consisting of problems in the domain, the cases data file which has a history of previous problems together with their solutions, and the metadata file holding the domain-specific data. Cases are problem-solution pairs, where the problems are described by a collection of features. Input problems use these same features. For each input problem, a similarity metric is applied to conduct a search of the case library for those cases whose problem features are most similar to it. This similarity metric employs reflection from the metadata, where the metadata contains a listing of all the possible features, their data types, etc., and most importantly, the "comparator" used for evaluating the similarity between two different values (problem and case) for this feature. The similarity between the input problem and a case in the library is then computed as a combination of the results returned by the comparators for the individual features.

The paper [4] describes this framework, together with a specific instance of it. The instance is a case-based version of the well-known "Snort" network intrusion detection system (NIDS) [5]. Snort contains a set of if-then rules where the premises describe features of incoming packets that may be considered suspect, and the conclusions describe actions to be taken (e.g., raise an alert) if a packet having these features is detected. The current Snort rule set has more than 1500 such rules. Snort rules are reinterpreted as cases by taking the premises as case features (describing problems) and the conclusions as case actions (solutions).

3 Reflective GUI Development

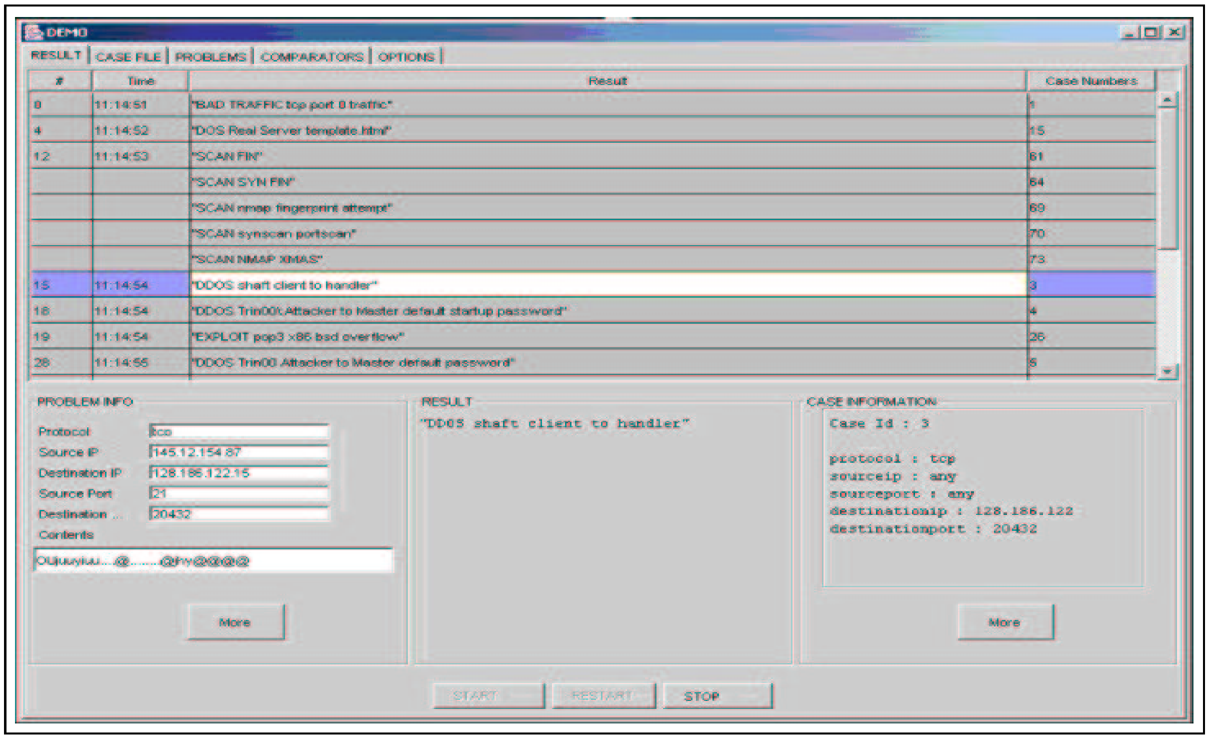
We illustrate the adaptive approach to GUI development with an application of it to GUI generation for the above CBR framework. As mentioned, both input problems and cases are characterized by a set of features. In the Snort NIDS implementation, the features include such items as protocol name, source IP address, source port, destination IP address, destination port, and packet contents (payload). The salient information about each such feature is stored in metadata. This information includes: (1) feature name, (2) data type of the feature, (3) comparator, (4) label name, (5) type of component in Java Swing used to display the contents, and (6) sequence number of the feature, i.e. the order in which it should be displayed.

Attribute	Value
Feature Name	sourceip
Data Type	String
Comparator	IPRange
Label Name	Source-IP
Component	JTextField
Sequence Number	1

Table 1: Metadata for feature Source IP

An entry in the metadata dictionary for the feature "Source IP", is shown in Table 1. The feature name is used to identify the feature internally throughout the code. The data type, as the name implies, is the kind of data the feature will describe. The comparator indicates the type of comparator to use for that particular feature. The label name is the name to be displayed as a label in the GUI. The component describes the particular Java Swing component to be used to display the feature value. Finally the sequence number is the order in which the feature is to be displayed relative to other features.

A GUI that displays this information can be hard coded using appropriate Java Swing components. The aim here, however, is to automatically generate the GUI from the metadata. This involves creating a package that can do this, i.e., which takes the metadata as input and outputs the GUI. This adaptive capability of the package is implemented



#	Time	Result	Case Numbers
8	11:14:51	"BAD TRAFFIC tcp port 0 traffic"	1
4	11:14:52	"DOS Real Server template.html"	15
12	11:14:53	"SCAN FIN"	61
		"SCAN SYN FIN"	64
		"SCAN nmap fingerprint attempt"	69
		"SCAN synscan portscan"	70
		"SCAN NMAP XMAS"	73
15	11:14:54	"DDOS shaft client to handler"	3
16	11:14:54	"DDOS Trin00\:Attacker to Master default startup password"	4
19	11:14:54	"EXPLOIT pop3 x86 bsd overflow"	26
28	11:14:55	"DDOS Trin00 Attacker to Master default password"	5

PROBLEM INFO

Protocol	tcp
Source IP	145.12.154.87
Destination IP	128.186.122.15
Source Port	21
Destination ...	20432
Contents	OUjuuyiuu....@.......@hy@@@@

RESULT

```
"DDOS shaft client to handler"
```

CASE INFORMATION

```
Case Id : 3

protocol : tcp
sourceip : any
sourceport : any
destinationip : 128.186.122
destinationport : 20432
```

Figure 1: The actual GUI.

using the process of reflection discussed in the foregoing. The resulting package thus embodies a GUI generation process that is domain independent.

3.1 GUI Creation

The GUI generated for the CBR domain is shown in Figure 1. In review of the code for the abovementioned package, there are no variables for specific GUI components. The code uses the metadata to define the type of GUI component needed for each attribute, e.g., a text field for a feature name. It also defines any domain specific information associated with that component, such as the string used in a label name, the length of the text field, the color of the component, and any string comprising an error message associated with the component.

In the upper portion of Figure 1, domain independent information tabs are used. These are as follows: (1) Result: used to display results of the CBR, (2) Case File: where cases can be viewed and added, (3) Problems: all the input problems, (4) Comparators: the comparators used for each feature, (5) Options: allows selecting alternate case and/or problem (packet) files.

These tabs are fixed for all possible domains of the CBR. The Options tab enables specification of the domain under consideration for the CBR. For example, if the CBR is detecting intrusion using the Snort like detector, the domain would be SNORT, whereas if it is performing facial recognition, the domain would be FACIALRECOGNITION. The Case File, Result, and Problems tabs are supplied with metadata information making them domain specific after a domain has been selected from the Options Tab. The comparators are selected from the metadata in the specific domain.

As an illustration of the adaptive nature of the GUI, consider the PROBLEM INFO panel in the southwest corner of the GUI. This box displays problem information (packets in the case of the SNORT domain). The label information, the component type, and the size of the component are all defined by the SNORT domain metadata. Changing domains would require changing the metadata, which in turn results in a change in the labels, components, attributes, and sizes of the items in the PROBLEM INFO panel. A snippet of the code for the generation of this GUI is depicted in Figure 2. This code creates instances of the Swing components JTextField and JLabel and then fills in the parameter information needed for these components from the domain metadata before placing them on the panel at runtime. This is where the reflection API provided by Java is applied. The GUIs in the Case File and the Result tabs are also generated from metadata in a similar manner.

The advantage of specifying the layout of the GUI in metadata is that the look of the GUI can be changed without compiling the GUI code or the CBR code. Any change made to the metadata is reflected in the appearance of the GUI at run time.

```
.
.
try{
        Class textFieldClass = Class.forName(ddRecord.getComponentType());
        If (ddRecord.getComponentType().equals("JTextField"))
        {
                JTextField jTextField=  (JTextField) textFieldClass.newInstance();
                jTextField.setBounds(xx+85,yy,(2*xsize),ysize);
                jTextField.setText(tempPF.getFeatureValue());
                problemInfoPanel.add(jTextField);
                yy+=20;
        }
}
catch(......){

}
```

Figure 2: Snippet of GUI code.

3.2 Code Specific Details for Creating the GUI

To create the screen in Figure 1, we use some of same metadata as used by other components of the application, in this case packets. To this is added further metadata indicating how to display the packet features. One such set of metadata was shown in Table 1. For the steps involved, consider again the example in Figure 2. This shows the code required to create a Java JTextField by reading the information about its placement and attributes from the metadata in Table 1.

The first line after the "try" statement is used to create an object of the type Class. The name of the class object to be created is obtained from the meta-data as indicated by the method ddRecord.getComponentType(). In the case of the metadata shown in Table 1, this would be the JtextField class. The next line creates an object of the type JTextComponent (from the Swing package). This object belongs to a class of objects from which other components are inherited. The next line sets the placement of the created component with reference to the other components. After this, the value is set to the feature value of the component. Finally, the created component is added to the panel to be displayed.

This illustrates the general manner in which components are put on the GUI. Other features may require more than a simple JTextField component, and hence can similarly make use of JTextArea. This snippet of code assumes that the component being specified to display a feature value will be a subclass of JTextComponent.

References

[1] Joseph W. Yoder and Reza Razavi, Metadata and adaptive object-models, *ECOOP'2000 Workshop Reader; Lecture Notes in Computer Science, vol. 1964*, Springer Verlag, 2000.

[2] Joseph W. Yoder and Reza Razavi, Adaptive object-models (poster session abstract), *Companion Papers of OOPSLA'00*, Minneapolis, MN, ACM Press, October 2000.

[3] Joseph W. Yoder, Federico Balaguer, and Ralph Johnson, Architecture and design of adaptive object models: Intriguing Technology Presentation at the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'01, *ACM SIGPLAN Notices*, ACM Press, December 2001.

[4] Daniel G. Schwartz, Sara Stoecklin, and Erbil A. Yilmaz, A case-based Approach to network intrusion detection, *Fifth International Conference on Information Fusion, IF'02*, Annapolis, Maryland., July 7-12, 2002, pp. 1084–1089.

[5] Snort, The Open Source Network Intrusion Detection System, available at http://www.snort.org