Vol. 44 No. 7 IPSJ Journal July 2003

Regular Paper

Effective Automated Testing for Graphical Objects

Juichi Takahashi†,†† and Yoshiaki Kakuda††

Recently, software testers have become increasingly reliant on automated testing. The automated testing methods consist of three phases: test case design, execution, and verification. However, to accomplish these three phases, presents a dilemma because of the lack of a verification function. Hardly any commercial automated testing tools can efficiently compare graphical objects, though Graphical user interface (GUI) software is now more crucial than text-based user interface. This paper describes a technique that aids automatic behavior verification for a particularly difficult problem: determining the correctness of screen and paper output. A methodology for capturing and comparing the output is presented, and a case study using Microsoft(R) PowerPoint(R) is described.

1. Introduction

Automated testing involves executing test cases and verifying the results programmatically instead of relying on human ability. Research shows automated testing can save up to 80% [1),2)] of testing costs, because test cases can be executed much faster automatically than manually. Moreover, it is impossible to complete some tests manually [3)].

One problem with automated testing is that there are so many graphics in desktop applications and Web applications. Automated tools have difficulty fetching and comparing graphical objects. For example, although Web applications should be able to handle many types of objects, controls, and images, most existing automation tools have trouble fetching the objects' information. In addition, no research has been done on testing for printing. Hardly any automated testing tools can handle printing objects. The same issues face testers using third-party interface controls[4)] and developers using custom controls. In those cases, testers must physically watch the running test case on their computer screen and printed-paper, because existing tools cannot have the ability to automatically verify the behavior of such objects. This approach is not optional, because the main benefit of automated testing is that it increases the number of test cases that can be applied. Sitting in front of a computer to manually verify test results negates the time saved through automated execution. Although there are objects which are hard to verify by using automated tools, some common objects can be handled and verified by in this way, such as strings, files, memory information, menu objects, windows size and attribute, contents of communication data, and screen images. Image objects, such as disk or screen images, are stored in the file system, and testers can perform standard binary file comparison to compare such images. Screen images are compared bit by bit. However, if an application uses graphical images, bit-by-bit comparison is the only way to compare them. It is thus difficult to compare actual images with expected images, because of storage constraints (the images tend to be large) and time constraints (bit-by-bit comparison is computationally intensive). Although almost all automated verification tools are capable of comparing drawing objects, several researchers recommend against capture/replay automated testing for drawing objects because the verification is overly sensitive to any change [1),5),6)].

Thus, in this research, a reliable and cost-effective graphical automated testing method, called the API and PostScript comparison method, is offered for testing drawing objects.

2. Verification Method of Graphical Objects

An important issue for verification is how to compare graphical objects. With testing of screen images, the main problem is that only a bit-by-bit verification method is currently available [7),8)]. With testing of printing image, there is no method of verifying test results. At first, we will consider which verification method may be used to test graphics-related software.

† SAP Labs Tokyo
†† Faculty of Information Sciences, Hiroshima City University

2.1 Verification of Graphical Screen Images

A number of applications can be used to draw graphical images. Examples include Web, Computer-Aided design (CAD), and desktop publishing applications. Formally, when testing graphical screen images, testers used the bit-by-bit comparison method. This section describes two ways of verifying graphical images by the bit-by-bit comparison method, and explains the API comparison method that we developed [9].

2.1.1 Bit-by-Bit Comparison

This method, which is used by most commercial tools [7),8)], compares graphical images bit by bit. Its main drawback is that it is too difficult to compare pre-saved bitmap images with actual bitmap images. For example, some testers save an image with a Windows title bar and some testers do not. Consequently, when a tester compares a pre-saved bitmap with an actual bitmap image, the comparison program may judge that the verification failed because the two objects are not equivalent. It is true that there are other techniques for comparing objects intelligently by means of bitmap images, for example various biometric verifications [11)], such as fingerprint, face [12)], signature, and iris recognition, are used in the real world. However, those methods are still being researched and their algorithms have not yet been applied to software testing.

2.1.2 API Comparison

In modern commercial operating systems, such as Microsoft Windows(R) and UNIX, applications do not access graphic devices directly, because the operating systems are designed for multi-tasking to protect against conflict of demands for use of shared graphics hardware resources. Therefore, applications use the same system APIs to access graphic devices. Drawing applications use graphical APIs to render screen images. Calls to such APIs are interrupted and the information is stored as drawing information (**Fig. 1**). To take as an example, when drawing a line, PowerPoint uses the Win32 graphic API [13)] (LineTo API), which passes the data to the operating system. In addition, most applications running on Windows operating systems use the common Win32 graphic API. In this case, a tool (see Section 6, "Tool Development") interrupts the "LineTo" API and stores the "LineTo" API information.

One of the benefits of this approach is that application testing does not require its source code to interrupt APIs' information. On the other hand, when a method is required that uses source code and interrupts the APIs, testers should prepare all related source code, including that developed by external companies, open source code, and so on. For various reasons, it is sometimes difficult to obtain such source code. In addition, testers are required to keep the same version during the entire development process, from the source code to the testing of the binary files. Activity of configuration management entails further work for testers.

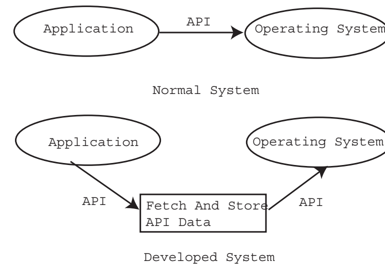

Fig. 1 API comparison model.

2.2 Verification of Printed Graphical Images (PostScript(R) Comparison)

Graphical images are not only drawn on the screen, but may also be printed on paper by means of a printing device. By using PostScript commands to test printing objects, we can test them efficiently. Usually, it is difficult to fetch printed-image information. The objects are on paper, and there is no method of obtaining printed information using automated testing tools. An image scanner may be useful, but scanned information is too ambiguous. Therefore, in this research, we use PostScript commands to fetch printed-image information. The PostScript language was developed by Adobe(R) in 1985 [14)] and has been used widely for desktop publishing applications. PostScript commands are not constructed from a subset of dot information but are logical. A graphical object can be regenerated [15)] and used to analyze objects [16)] by means of PostScript commands. On the other hand, it is very difficult to use the dot information generated by a non-PostScript printer. For the same reason, testers do not use the bit-by-bit comparison technique (the testing results may be sensitive). When an application commands an operating system to print images, the operating system receives API commands from

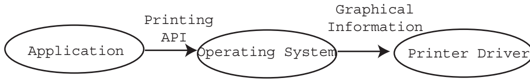

Fig. 2 PostScript comparison model.

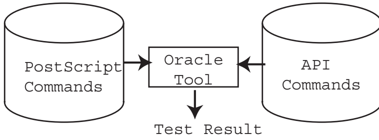

Fig. 3 Mixed comparison model.

the application and passes graphical information to the printer driver. The printer driver then prints the objects on paper (**Fig. 2**). In the PostScript printing system, the application can generate PostScript commands and store them as a file. The file can be used to compare pre-saved objects with test result objects. In this method, called PostScript comparison technique, testers can use printing images as logical objects, such as lines, circles, and triangles [14]. For instance, to draw a line on paper from (0, 0) to (100, 100), an application generates the following PostScript commands:

```
0 0 moveto
100 100 lineto
```

The "moveto" command sets the current point to (x, y) and the "lineto" command defines the point to which the line will be drawn.

2.3 Verification of Printed Graphical Images by Means of Screen Graphical Images

The previous section dealt with verification of printed graphical images. Yet it is rare for an application to have only printing functions without the ability to render images on screen, and it is equally unusual for printing functions to be tested individually. Typically, an application has both on-screen drawing and printing functions. Thus, in a typical testing scenario an application tests graphical screen images first, and then goes on to ensure those screen images match printed graphical images. In addition, when testers choose to use this testing method instead of only "PostScript Comparison", they usually do not have to design printing test cases. While they are running screen image test cases, a tool (see the following section, "Tool Development") can execute and verify the printing test results simultaneously (**Fig. 3**). This means that the screen image test cases are automatically duplicated as printing test cases, and executed and verified.

3. Benefits of Using the API and PostScript Comparison Approach

The API and PostScript comparison method offers a large number of benefits. Some of the most important of these are explained.

3.1 API Comparison System

3.1.1 System Configuration

Bit-by-bit verification is extremely sensitive to changes [1]. Various uncontrollable factors can often affect results. For example, it is very difficult to ensure that all members of a testing team have the same type of machines and graphic systems. One tester may have an advanced graphic system with resolution of over $1,600 \times 1,200$ pixels, while another may have a normal graphic system with a resolution of $1,024 \times 768$. Any differences between computing environments can cause failure, even when screen images match. On the other hand, when the API comparison method is used, the difference can be adjusted by the vector comparison method (see Section 5.2 "Vector Comparison"). Thus, testers can test graphical objects on any type of computing system.

3.1.2 Disk Size

When testers store a drawing image with screen resolution of $1,600 \times 1,200$ pixel screen resolution at three bytes per pixel, the image requires 5.76 M bytes ($1,600 \times 1,200 \times 3$). Storing files of this size is acceptable only when a test case is considered strategically crucial. In this case, they can either use an image-compression method to reduce the disk sizes, or they can use the API comparison method.

3.1.3 Processing Speed

Similar issues exist for disk size. Bit-by-bit comparison requires time.

3.1.4 Expansion and Reduction Testing

A number of drawing and CAD applications support object expansion and reduction functions. When testing such applications, testers are required to exercise and verify expansion and reduction functions. In testing expansion and reduction functions using the bit-by-bit method, it is necessary to store images for every possible zoom rate. When testing zoom rates from 1% through 100% in increments of 1%, there may be 100 test cases, and testers must store 100 bitmaps to verify the test results. These storage tasks are time-consuming and require large amounts of hard drive space. In addition, verification is not straightforward, be-

cause even difference of a single bit causes failure in bit-by-bit comparison techniques. On the other hand, testers can test any type of expansion and reduction by the API and PostScript comparison method. They do not have to store 100 images to compare the results. When using a vector comparison method, it is sufficient to store a single image to compare an original and a post-tested result for any degree of zoom rate (see the section titled "Vector Comparison").

3.1.5 Model-Based Testing

In some cases, it is better to use model-based testing. However, if model-based [4),17)] and monkey tests [10)] are performed by using a bit-by-bit comparison method, manual verification of the test results is a major problem for those testing for drawing objects. On the other hand, testers can use the API comparison method, and can execute model-based and monkey tests using the automated verification method we developed.

3.2 PostScript Comparison System

It is difficult to describe the benefits of the PostScript method, because little research on automated printing testing has been reported to date. For instance, in Microsoft and SAP, most testers manually verify test results for printed-paper images. It is inevitable that such work consumes both large amounts of time and money. Moreover, in the case of complicated images, it is very difficult to compare printed images and screen images. Myers advises, "Thoroughly inspect the result of each test" [7)]. The PostScript comparison method can meet his recommendation for testing graphical objects.

4. Analysis of Graphical Objects and Defects: Toward the Use of API Comparison

In this section, we will examine the types of graphical objects used by PowerPoint. We will also consider whether the API and PostScript comparison methods can detect all kinds of graphical object defects. As a result, the advantages and limitations of our method can be seen.

4.1 Type of Graphical Objects

Lines, triangles, and rectangles: Most drawing objects are composed of simple line objects. A triangle is made up of three connected lines, a rectangle of four lines, and a polygon of four or more lines. It is possible to fetch and compare such objects by the API comparison method, because the APIs contain the information describing the lines that compose the objects.

Bitmap fonts: Text is displayed by using font or text APIs. When an application uses a text-handling API, any commercial automation testing tool can fetch text information. Yet, for a variety of reasons, developers sometimes choose not to use text-handling APIs to display text. In such cases, bitmap fonts are used, so text information is rendered by a using bitmap instead of text-handling APIs. When text information is stored as a bitmap, it cannot be fetched by existing commercial automation testing tools. The only way to fetch such information is to load and store it as a bitmap object. It is possible to fetch and compare such objects by the API comparison method. However, there is no advantage in using the API and PostScript comparison method as opposed to the bit-by-bit comparison method. Therefore, we will not discuss this topic in this paper.

Image objects: Because a large number of multimedia applications and tools have been developed over the last decade, testers are required to test images of a large number of types, such as bitmap, GIF (Graphics Interchange Format), and JPEG (Joint Photographic Expert Group) images. It is possible to fetch and compare such objects by the API comparison method. However, as in the case of bitmap fonts, there are no advantages in using the API and PostScript comparison method rather than the bit-by-bit comparison method.

4.2 Types of Graphical Defects

Software defects can be categorized as improperly constrained input, improperly constrained stored data, improperly constrained computation, and improperly constrained output [19)]. In accordance with this idea, we can categorize the types of graphical defects as improperly constrained computing coordinates, improperly computed color information, and improperly constrained output.

4.2.1 Improperly Constrained Computing Coordinates

One common defect is that drawing objects are often rendered by using the wrong coordinates. This defect can be found by the API comparison method, since the API and PostScript comparison methods use coordinate information. Consider the following report of real defect in Microsoft PowerPoint:

Application: Microsoft PowerPoint 2000

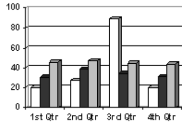

Fig. 4 Defect 1.

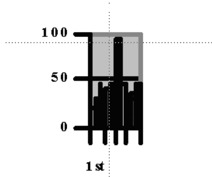

Fig. 5 Defect 2.

Steps:
1. Launch Microsoft PowerPoint.
2. Insert a graph (**Fig. 4**).
3. Minimize the graph.
4. Click at another location (not on the graph).
5. Restore the original size.

Expected Result: The graph is shown as in the original image (Fig. 4).

Actual Result: The graph is shown with corruption (**Fig. 5**).

4.2.2 Improperly Computed Color Information

Applications use graphical APIs to show objects with color information. First, the operating system receives API calls from the application, and then it calculates the color information on the basis of the specific capabilities of the installed graphic card. Next, the operating system passes the information to the graphic card driver. The problem is that there is a huge variety of different graphic devices and they all vary in their ability to display colors; moreover, it is difficult to develop an application to suit all of these graphic systems. Another problem is that drawing a graphic uses multiple software modules and hardware, including the application, operating system, graphic card drivers, and graphic card hardware (**Fig. 6**). Consequently, defects may be in the application, the operating system, the graphic driver, or any combination thereof. However, since computing color information is usually much simpler than computing coordinates, in our experience, the number of defects is relatively small. Moreover, most defects are caused by the graphic driver and graphic hardware because there are so many types of graphic card in the world [20]. In addition, though the API comparison method can fetch color information, it is quite simple to calculate the color information code. Thus, in this research, we shall not include verification of color information, since a very small number of defects is anticipated.

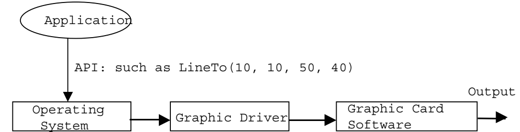

Fig. 6 Graphical object output.

4.2.3 Improperly Constrained Output

As explained above, to output graphic images, the data created by the application goes through the application, API, operating system, graphic driver, and graphic card software (Fig. 6). Obviously, some graphic driver software and card-controlling software contains defects. We call such defects improperly constrained output defects. Virtually no method (including the API and bit-by-bit comparison methods) can find defect of this type, because they usually occur within the graphic driver code. Therefore, in this paper, we do not focus on testing improperly constrained output type defects.

5. Comparison Method

In this research, two kinds of comparison methods, point-based and vector-based comparison are offered to reduce the need for bit-by-bit comparison. In these comparison techniques, testers can use images as logical objects, such as lines, circles, and triangles. As explained in the previous section, the API and PostScript comparison system will fetch the API and PostScript commands. For example, to draw a line from screen coordinates (0, 0) to (100, 100), an application calls an API as follows (on Microsoft Windows platforms).

```
POINT pPoint[2];
pPoint[0].x = 0;     pPoint[0].y = 0;
pPoint[1].x = 100;   pPoint[1].y = 100
LineTo(pPoint);
```

The tool then fetches the information and stores it for comparison.

5.1 Point-Based Comparison

Once an API command from an application has been fetched, the API command information is saved as point information to be analyzed and compared. For instance, in the case of two rectangles (**Figs. 7** and **8**), the coordi-

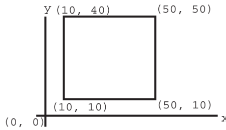

Fig. 7 Point-based comparison 1.

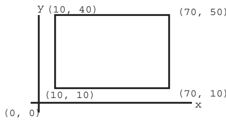

Fig. 8 Point-based comparison 2.

Table 1 Comparison results.

Fig. 7	**Fig. 8**	**Test Result**
(10, 10)	(10, 10)	Pass
(10, 50)	(10, 50)	Pass
(50, 10)	(70, 10)	Fail
(50, 50)	(70, 50)	Fail

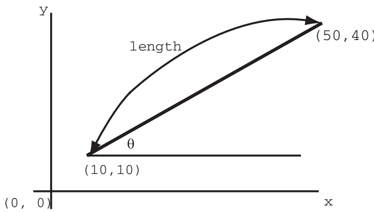

Fig. 9 Vector comparison.

nates of the rectangles will be compared one by one. The test results are shown in **Table 1**.

5.2 Vector Comparison

In a vector comparison system, point information generates vector information, which is structured by length and angle (**Fig. 9**).

In Fig. 9, the value of from vector (10, 10) through (50, 40) is calculated as follows:

$$length = \sqrt{(10-50)^2 + (10-40)^2}$$

$$\Theta = \sin^{-1}(\tfrac{40-10}{length})$$

The vector comparison technique is useful because testers can easily accomplish expansion and reduction testing, as explained in the previous section titled "Expansion and Reduction Testing". For example, **Fig. 10** shows the line zoomed out 50% from Fig. 9. coordinates are (25, 20) = (50, 25) ×0.5 and (5, 5) = (10, 10) ×0.5.

The length and angle are as follows:

$$length = \sqrt{(5-20)^2 + (5-20)^2}$$

$$\Theta = \sin^{-1}\left(\tfrac{25-5}{length}\right)$$

Even when the application zooms out 50%,

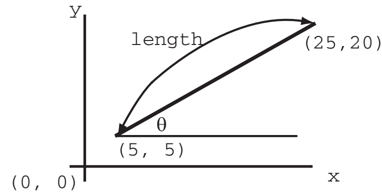

Fig. 10 50% Zooming.

the angle remains the same. The length can also be calculated as follows:

Original length = (zoom rate) × (changed length)

In similar objects, the angles are the same, and the lengths are similar. The bit-by-bit comparison method cannot directly compare similar objects.

6. Tool Development

To realize the idea of API comparison, we developed a tool that can fetch graphical API information on PowerPoint. Because verification of graphic screen images is much more complicated to realize than verification of printed graphical images, we explain our tool in a separate section. PowerPoint uses common components of Microsoft(R) Office(R). When drawing graph objects, PowerPoint uses an Office component named GRAPH9.EXE. In order to store API information, we changed the GRAPH9.EXE binary file to capture the API information. The GRAPH9.EXE file is edited by using a binary editor. One string in GRAPH9.EXE's was changed from "GDI32.dll" to "GDI23.dll" (indicated in bold font in the following **Fig. 11**) to access gdi23.dll instead of gdi32.dll (**Fig. 12**).

After the string has been changed, GRAPH9. EXE calls gdi23.dll when PowerPoint uses graphical APIs such as LineTo( ) and Polygon( ). Gdi23.dll has two functions: to store API information and to pass the API information to the operating system. As Fig. 12 shows, the API information goes into gdi23.dll and is passed into gdi32.dll. Consequently, the API information is stored on the hard disk drive. The stored information is used to compare objects (this comparison will be discussed in a later section). The following source code is a part of gdi23.dll that fetches and stores the LineTo commands.

```
KERNEL23_API BOOL WINAPI
    myLineTo(HDC hdc, int iX, int iY)
{
```

```
Address:  Hex code
------------------------------------------------------
D50: 50493332 2E648C8C 00004744 4932332E PI32.dll  GDI23.
D60: 646C6C00 4B45524E 454C3332 2E646C6C dll KERNEL32.dll
D70: 00004D53 4F392E44 4C4C0000 6F6C6533  MS09.DLL  ole3
D80: 322E646C 6C005553 45523332 2E646C6C 2.dll USER32.dll
------------------------------------------------------
```

Fig. 11 Changed Binary Image for GRAPH9.EXE.

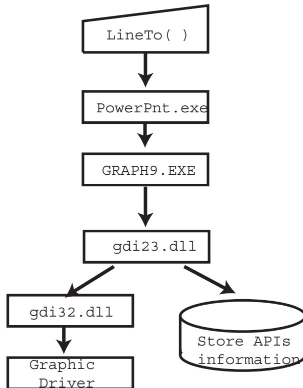

Fig. 12 Gdi32.dll behavior.

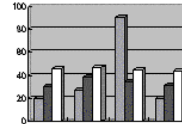

Fig. 13 Fetching information.

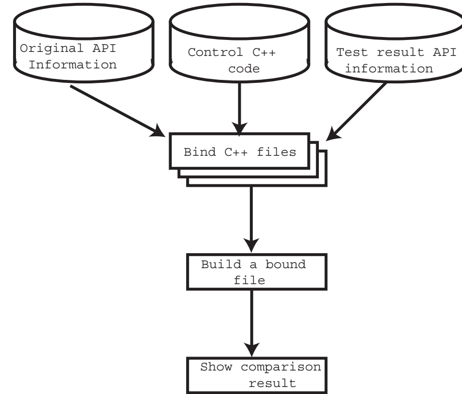

Fig. 14 Flowchart for Gdi23.dll.

```
typedef BOOL
(CALLBACK *LPFN)(HDC, int, int);
HINSTANCE bltH_Dll;
LPFN bltPtrFn_Function;
BOOL ReturnValue;

//Loading Dll
char ptrChr_DllPath[MAX_STR];
GetSystemDirectory(ptrChr_DllPath,
 MAX_STR);
strcat(ptrChr_DllPath, "\\gdi32.Dll");
//Load gdi32.dll
bltH_Dll=LoadLibrary(ptrChr_DllPath);
//In case failing the library
_ASSERT(bltH_Dll);
if (bltH_Dll==NULL)
{
ErrorLoading("LineTo");
return 0;
}
bltPtrFn_Function = (LPFN)
GetProcAddress(bltH_Dll, ''LineTo");
//In case failing the function
_ASSERT(bltPtrFn_Function);
if (bltPtrFn_Function == NULL)
{
FreeLibrary (bltH_Dll);
return 0;
}
FreeLibrary (bltH_Dll);
ReturnValue =
bltPtrFn_Function(hdc, iX, iY);
//To store LineTo information into disk
LogFile2(
"LineTo(*hdc,", iX, iY, LINETO_LOG);
return ReturnValue;
}
```

6.1 Fetching Drawing Information

The tool described in this paper specifically targets Microsoft PowerPoint. It fetches drawing information from APIs calls (Fig. 12). For the object in **Fig. 13**, the tool could fetch the points of the polygons, e.g.

```
Polygon 1:(34,172),(53,158),(248,158),
          (229,172), (34, 172)
Polygon 2:(34,172), (34,22), (53,8),
          (53,158), (34,172)
Polygon 3:(53, 158), (53,8), (248,8),
          (248,158), (53,158)
```

and so on.

6.2 Comparing Graphical Objects

In order to compare an original object and a test result, the tool has a comparison function. After the tool has stored the original information and the test result, it binds the original program and the test result and prepares a C++ program that includes the APIs' comparison routine. The bound file is then compiled and linked in order to compare the original rendered object with the test result (**Fig. 14**).

7. An Example: Microsoft PowerPoint

Microsoft PowerPoint is used as a sample application to confirm the efficiencies of the API and PostScript comparison technique. Power-

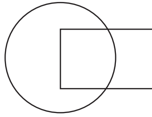

Fig. 15 Simple object (original).

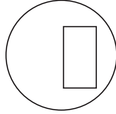

Fig. 16 Simple object (modified).

Point is one of the most popular graphical presentation applications, and is sufficiently complicated to demonstrate the API and PostScript comparison technique working on a real application.

7.1 Verification of Graphical Screen Images

In the previous section, we described what the API comparison method can efficiently test for logical graphical objects (lines, circles, polygons, and so on). Next, we will test simple logical objects by using the API comparison method, and will then extend the API comparison case study to a real application object.

7.1.1 Simple Line and Circle Objects

Lines and circles are basic graphical objects that are often built into more complicated graphical objects. Before confirming whether the API comparison works with real-world graphical objects, we first try a simple demonstration. Test Case 1: Compare two simple objects.

1. Launch PowerPoint.
2. Draw a rectangle and a circle (**Fig. 15**).
3. Save the object as the original object.
4. Change the rectangle width (**Fig. 16**).
5. Save the object as the destination object.

Expected result: The comparison program shows the differences between the objects.

The API comparison method stores the original object (Fig. 15) formation as:

```
Ellipse(hdc,87,51,204,168);
Rectangle(hdc,147,80,238,146);
```

On the other hand, the API comparison method stores the result object (Fig. 16) as:

```
Ellipse(hdc,87,51,204,168);
Rectangle(hdc,147,80,185,146);<-diff.
```

We can thus clearly determine that the two objects are different.

7.1.2 Complicated Graph Objects

We have already demonstrated a simple graphical object. In order to prove that the API comparison method can achieve more advanced testing efficiency, we now show that it can test real-world, and complicated objects. The target object is one of the functions of PowerPoint's Graph (Fig. 13). Graph's objects are mainly structured in terms of polygons. To simplify the analysis and results, the tool tests only the polygon object information. When two objects (Figs. 13 and **17**) are compared, the former differs from the original object only in the value of the first bar value (Fig. 13). The comparison program should show that only one bar is different. The following is a test case: Test Case 2: Compare two different objects.

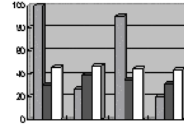

Fig. 17 Modified object.

1. Launch PowerPoint.
2. Insert a graph object (Fig. 13).
3. Save the object as the original object.
4. Change the value of the 1st bar (Fig. 17).
5. Save the object as the destination object.

Expected result: The comparison program shows the differences between the objects. Test Analysis:

- 42 polygons are detected.
- Two polygons have different point values.
- Nine lines have different values.

Test Result: PASS

In this case, there are 42 polygons. Comparing Figs. 13 and 17, the program detects two polygons that are different: the differences include 9 lines. Thus, the tool can detect the difference as expected. It is important to note that the bit-by-bit comparison method hardly ever finds the differences or the causes of problems. The bit-by-bit comparison tool only shows differences caused by group of bitmap information. On the other hand, the API comparison method can show APIs that cause defects. It is clear that when developers debug the defect, the API comparison gives more logical defect information and permits faster debugging than the bit-by-bit comparison method.

7.2 Verification of Printed Graphical Images

As explained in the previous section, in order to verify printed images with pre-saved images, we use the PostScript comparison method.

7.2.1 Tool Development

A tool was developed to use the PostScript comparison method and verify printed images. The tool is capable of:

- Fetching PostScript commands.
- Comparing rendered objects via PostScript commands.

Fetching Printing Information: Fetching PostScript information is much simpler than fetching API information. PowerPoint is able to save information on PostScript commands as a file, and we use the files to compare each PostScript command.

Comparing Rendered Objects via PostScript Commands: Fetched PostScript information can be saved as a file. However, we cannot simply compare pre-saved and test result files, because saved PostScript files include various types of information such as PostScript command definitions, dates, and so on. Therefore, the tool filters out non-rendering-related PostScript commands and compares the filtered data. If the two files are different, the tool shows a test failure and indicates the differences between the graphical objects.

7.2.2 Testing

As we demonstrated in the previous section, we used both simple and complicated cases to confirm whether the PostScript comparison method can test printing images with real-world applications.

7.2.3 Simple Line and Circle Objects

Test case 1 was executed and PowerPoint generated the following PostScript commands for the original object (Fig. 15):

```
5296 1762 moveto
3446 1762 lineto
3446 3112 lineto
5296 3112 lineto


4244 3210 moveto
4469 2985 4596 2680 4596 2362 curveto
....
```

PowerPoint also generated the altered object (Fig. 16):

```
4196 1762 moveto <--difference
3446 1762 lineto
3446 3112 lineto
4196 3112 lineto <--difference

4244 3210 moveto
4469 2985 4596 2680 4596 2362 curveto
....
```

When we compare the test results for two objects, we can see that two lines of PostScript commands (indicated in bold font) have different values. This successful test was conducted by using the PostScript comparison method.

Complicated Graph Objects: Test case 2 (described in the previous section) was also performed by using the PostScript comparison method. Test analysis:

- 265 lines were detected.
- Eight lines had different point values.

The results shows that the pre-saved object and the tested object are not the same. Some of the line objects have different line lengths. Test Result: PASS.

7.3 Verification of Printed Graphical Images by Screen Graphical Images

To verify printed graphical images with verified graphical screen images, we use both API and PostScript comparison methods.

7.3.1 Tool Development

We developed a tool with three functions:

- Deconstructing polygon objects into line objects
- Adjusting
- Comparing rendered objects in the form of screen and printing images

Deconstructing Polygon Objects into Line Objects: Unfortunately, depending on the application, PowerPoint has different architectures for printing and drawing on-screen functions. For drawing on the screen, PowerPoint uses polygon commands, whereas for paper printing, it uses the lineto command. Thus, a tool is required for deconstructing a polygon command into lineto commands. For example, let us consider a square represented by the polygon command for drawing on-screen, such as:

```
rPoint[0].x = 0;   rPoint[0].y = 0;
rPoint[1].x = 100; rPoint[1].y = 0;
rPoint[2].x = 100; rPoint[2].y = 100;
rPoint[3].x = 100; rPoint[3].y = 0;
Graph->AddPolygon(rPoint, 4);
```

The polygon command above can be deconstructed into lineto commands by the tool as follows:

```
0    0    moveto 0    100 lineto
100  100  lineto 100  0   lineto
0    0    lineto
```

After the tool has converted polygon information into lineto information, the on-screen and printed graphical information is compared by using lineto commands.

Adjusting: Naturally, applications do not

Table 2 Test result.

Command	GDI		Postscript		GDI				Postscript				ASIN		Length Rate		Delta for Lines				Test Result
	x	y	x	y	diff x	diff y	length	diff y/ lentgh	diff x	diff y	length	diff y/ legth	GDI	Post	Zoom Rate	Error Rate	GDI		Postscript		
																	x	y	x	y	
moveto	49	218	1502	3552																	
lineto	55	214	1571	3502	6	-4	7.2	-0.6	69	-50	85.2	-0.59	-35.9	-33.7	11.8	-0.3%					PASS
lineto	357	214	5202	3502	302	0	302.0	0.0	3631	0	3631.0	0.00	0.0	0.0	12.0	-2.1%					PASS
lineto	351	218	5133	3552	-6	4	7.2	0.6	-69	50	85.2	0.59	35.9	33.7	11.8	-0.3%					PASS
lineto	49	218	1502	3552	-302	0	302.0	0.0	-3631	0	3631.0	0.00	0.0	0.0	12.0	-2.1%					PASS
moveto	55	214	1571	3502													6	-4	69	-50	PASS
lineto	55	18	1571	1140	0	-196	196.0	-1.0	0	-2362	2362.0	-1.00	-90.0	-90.0	12.1	-2.3%					FAIL
lineto	357	18	5202	1140	302	0	302.0	0.0	3631	0	3631.0	0.00	0.0	0.0	12.0	-2.1%					PASS
lineto	357	214	5202	3502	0	196	196.0	1.0	0	2362	2362.0	1.00	90.0	90.0	12.1	-2.3%					**FAIL**
lineto	55	214	1571	3502	-302	0	302.0	0.0	-3631	0	3631.0	0.00	0.0	0.0	12.0	-2.1%					PASS
moveto	78	218	1852	3552													23	4	281	50	PASS
lineto	78	22	1852	3021	0	-196	196.0	-1.0	0	-531	531.0	-1.00	-90.0	-90.0	2.7	77.0%					FAIL
lineto	84	17	1921	2964	6	-5	7.8	-0.6	69	-57	89.5	-0.64	-39.6	-39.8	11.5	2.7%					PASS
lineto	84	214	1921	3502	0	197	197.0	1.0	0	538	538.0	1.00	90.0	90.0	2.7	76.8%					**FAIL**
lineto	78	218	1852	3552	-6	4	7.2	0.6	-69	50	85.2	0.59	35.9	33.7	11.8	-0.3%					PASS

generate the same coordinate values for screen and printing images. For example, the applications may generate the following PostScript commands for printing:

```
0 0  moveto 100 100 lineto
```

and the following API commands for rendering on screen:

```
rPoint[0].x = 0;   rPoint[0].y = 0;
rPoint[1].x = 50;  rPoint[1].y = 50;
Graph->AddPolygon(rPoint, 2);
```

When the API commands and PostScript commands are compared, the tool indicates that the two objects are not equivalent. Therefore, the tool is required to calculate the zoom rate and use this to compare the objects. The zoom rate is calculated as follows:

$$\text{Zoom rate} = \frac{\sum_1^n \frac{length(GDI)}{length(PostScript)}}{n} \quad .... \quad \text{(n: number of lines)}$$

7.3.2 Testing

Test case 2 (described in the previous section) is also performed in the API and PostScript comparison methods. Test Analysis:

- Forty-two polygons are detected in the screen object.
- The forty-two polygons are deconstructed into 168 lines.
- Four lines have different values.

The result is that pre-saved object and the tested objects are not the same. The test result indicates that four line objects have different line lengths, as expected. Test result: PASS.

To explain the details of the calculation process, we will choose a part of the information from the entire set of test results. As shown in **Table 2**, three polygons are separated into 12 lineto commands and three "moveto" commands, and pointing information is indicated. First, when we look at each of the points in the first two columns (under the heading "GDI" in the table and shown in white letters against a colored background) we see that the coordinates 49, 218 and 55,214 are fetched by the API comparison system. The coordinates are generated by the following commands:

```
rPoint[0].x = 49; rPoint[0].y = 218;
rPoint[1].x = 55; rPoint[1].y = 214;
```

In the next two columns (under the heading "PostScript" in the table), the coordinates 1502, 3552 and 1571, 3502 are fetched by the PostScript comparison system. The coordinates are generated by the following commands:

```
1502 3552 moveto
1571 3502 lineto
```

Each line length is

7.2 ($\sqrt{(49-55)^2+(218-214)^2}$) and

85.2 ($\sqrt{(1502-1571)^2+(3552-3502)^2}$).

Therefore, at this time we cannot say that the GDI and PostScript objects are the same. But we can find a fixed zoom rate of around 12. This shows that most of the objects are similar, but some of them are not. According to Table 2, two lines have 77.0% and 76.8% error rates, and this indicates that two graphical objects are not equivalent. Next, we compare angles. In Table 2, we calculate the angle of every line by using the pointing information, and this shows that there are a small number of differences of sin^{-1} (under the heading ASIN in the table) between the GDI object and the PostScript object. These differences can be ignored, because they may be caused by accidental errors in calculation (see the section titled "Error Rate"); furthermore, we can say that the two objects have the same angles for all lines. Finally, even when the lengths are similar and the angles are the same, we cannot say two objects are equiva-

lent. We need to check the positions of lines, because the objects should be shown in the correct positions on screen and paper. For example, the objects in **Fig. 18** have similar lengths and the same angle lines. However, the two objects are not equivalent, because the relevant coordinates of their two lines are different. Thus, in Table 2, we calculate delta x and delta y. When analyzing the three factors above, we can say that the comparison tools indicate that the screen object and the printed object are not the same, and that the testing meets the expected result.

7.3.3 Error Rate

Unfortunately, we see some errors during the calculation process. In **Table 3** there is a 13% error rate. However, we consider that this error rate might be acceptable, for the following two reasons: First, PowerPoint is not required to produce accurate images and printed output, since it is a business presentation tool. Users may not require accurate printed images. Second, GDI gives a value of 8.1 for the length, whereas PostScript gives a value of 85.2 when there is a 13% error rate. Since Windows graphical API only allows integer to be used as value [13], the error rate tends to be high when the value is small enough. Of course, when we test larger graphical objects, the error rate can be reduced. Let's us turn now to other types of objects, such as circles, ovals, and triangles. A circle has two attributes: location of the center, and the diameter. It is entirely fair to say that the location of the center has an error rate quite similar to that of the location of a line, while the diameter has the same error rate as the length of a line. Consequently, we may not see a much larger error rate for circle objects than for line objects. For triangles and other objects, we can be fairly certain that there is not a large error difference between lines and triangles or other objects, since objects usually consist of lines and arcs.

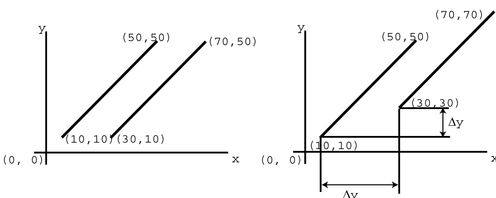

Fig. 18 Relative positions of objects.

7.3.4 External Factors

In real-world testing, when we formalize a model, we may encounter obstacles from external factors, such as sequential API and PostScript commands.

Drawing on the screen by APIs: In drawing on the screen, it may be possible to have a different sequence of API calls for presaved and test result objects. However, we have never encountered any unexpected sequence of APIs in our research. Even so, there is a slight chance that an application and operating system may generate an unexpected sequence of APIs. In order to formalize this method, we added a sorting function to our developed tool to adjust this type of altered sequence.

Printing by Using PostScript commands: In our research, we used a Hewlett-Packard(R) Laser Jet4000 PostScript printer with a Microsoft(R) Windows 2000(R) printer driver. The issue here is the PostScript driver design. For example, when there is a line from 0,0 to 100, 100, we expect the PostScript printer driver to generate the following code:

```
0 0 moveto
100 100 lineto
```

On the other hand, since the PostScript language grammar [14] does not define anything sequentially, there is a possibility that the PostScript driver may instead generate another set of code:

```
100 100 moveto
0 0 lineto
```

In addition to the Hewlett-Packard(R) Laser Jet4000 PostScript printer, we used a Xerox DocuPrint 4512 and an Apple LaserWriter II NTX. Fortunately the three printers generated the same sequential PostScript commands, and our tool operated without any trouble. Yet, there is a slight chance that a printer driver may generate unexpected sequential code, as explained for the API call above. We are sure that the design of PostScript drivers will dif-

Table 3 Error Rate.

Command	GDI		Postscript		GDI				Postscript				ASIN		Length Rate		
	x	y	x	y	diff x	diff y	r	(diff y)/r	diff x	diff y	r	(diff y)/r	GDI	Post	diff	Zoom Rate	Error Rate
moveto	188	116	3164	2321													
lineto	195	112	3233	2271	7	-4	8.1	-0.5	69	-50	85.2	-0.59	-35.9	-29.7	-6.2	10.6	10.3%
lineto	176	112	3033	2271	-19	0	19.0	0.0	-200	0	200.0	0.00	0.0	0.0	0.0	10.5	10.6%
lineto	171	116	2964	2321	-5	4	6.4	0.6	-69	50	85.2	0.59	35.9	38.7	-2.7	13.3	-13.0%
lineto	188	116	3164	2321	17	0	17.0	0.0	200	0	200.0	0.00	0.0	0.0	0.0	11.8	0.1%

fer only slightly among printer drivers, and we emphasize that the effort of changing the tool is much smaller than that required for testing graphic applications.

8. Expanding the API and PostScript Comparison Method (Fetching Other Types of Graphical Object)

In the preceding sections, we have demonstrated how tests can be efficiently executed for PowerPoint graphical objects (mostly constructed from lines) by API and PostScript comparison. In general, the API and PostScript comparison method is able to fetch figures of any of the types, such as lines, triangles, squares, and circles, which we tested on PowerPoint 2000. On the other hand, there may be other attributes that determine the form of graphics, such as thickness, thinness, and overlapping. In this section, we attempt to explain whether the API and PostScript comparison method can fetch and compare shapes of figures (lines and triangles) as well as attributes of figure (thickness and thinness). Consequently, we demonstrate that the API and PostScript comparison can test any type of application, including Microsoft PowerPoint.

8.1 Thick and Thin Lines

First, we confirm that the API comparison method can detect the widths of figures (such as lines and rectangles). In the Windows system, developers usually use a CreatePen( ) API in order to determine the width of lines, such as:

```
//Draw a line as 2 points width.
//"2" indicates the width
CreatePen(PS_SOLID, 2, NULL);
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);
//Draw a line as 5 points width.
//"5" indicates the width
CreatePen(PS_SOLID, 5, NULL);
MoveToEx(hdc, 100, 100, NULL);
LineTo(hdc, 200, 200);
```

In the above code, the width of data includes the CreatePen API. In addition, we can easily fetch and compare the APIs by using the API comparison methods and compare the line widths (**Fig. 19**). Next, we confirm that PostScript commands can be used for testing thin and thick lines. The “setlinewidth” command is used for determining to decide line width, such as 6.25 for a thin line and 25 for a thick line (Fig. 19).

In the code below, thick and thin lines

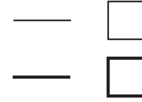

Fig. 19 Thin and thick line.

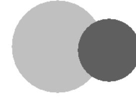

Fig. 20 Overlapped objects 1.

Fig. 21 Overlapped objects 2.

are clearly indicated by the “setlinewidth” PostScript command, and the PostScript comparison method that we developed can test the attributes.

```
%%% thin line
6.25 setlinewidth
1039 1041 moveto
2054 1041 lineto
%%% thick line
25 setlinewidth
1039 1041 moveto
2054 1041 lineto
```

8.2 Overlapped Objects

If two graphical objects are overlapped in different ways (**Figs. 20** and **21**), the graphical images must be different. For the present, we examine whether API and PostScript comparison method can precisely fetch and distinguish graphics of these types.

API comparison: When the two objects' APIs are fetched, the sequence can affect the graphical image. The first-called API is shown in the background, and the second-called API is shown in foreground. The following Win32 APIs show how the overlapped objects are configured in a Windows system. The only difference between Figs. 20 and 21 is the overlapping sequence.

```
//For Fig.\,20
Ellipse(hdc, 0, 0, 500, 500);
Ellipse(hdc, 0, 400, 1000, 500);

//For Fig.\,21
Ellipse(hdc, 0, 400, 1000, 500);
Ellipse(hdc, 0, 0, 500, 500);
```

These APIs also shows that the API comparison method can handle multi-window sys-

tems. When we test graphical objects on multiple windows, API comparison is required to distinguish between the sequences of API commands. For example, a tool distinguishes APIs for the first-opened and second-opened windows. Thus, testing multiple windows by API comparison is the same as testing overlapped objects.

PostScript comparison: PostScript commands have the same drawing system as API command for printing foreground and background objects. The codes below are PostScript commands for showing overlapped objects (Figs. 20 and 21).

```
%First circle, drawn on the left side
1535 1230 moveto
1209 1230 945 1494 945 1820 curveto
945 2146 1209 2411 1535 2411 curveto
1862 2411 2126 2146 2126 1820 curveto
2126 1494 1862 1230 1535 1230 curveto

%Second circle, drawn on the right side
2197 1466 moveto
1975 1466 1796 1646 1796 2868 curveto
1796 2089 1975 2269 2197 2269 curveto
2419 2269 2599 2089 2599 1868 curveto
2599 1646 2419 1466 2197 1466 curveto
```

As the above code shows, the sequential difference may only affect the foreground and background locations of graphical objects. In short, it is possible that our API and PostScript comparison method can employ overlapped objects. There may also be color, shade, and transparency attributes in graphic applications. But we do not foresee any difficulties in fetching these attribute by the API and PostScript comparison method on PowerPoint or other graphic applications. Thus, it seems reasonable to suppose that the API and PostScript comparison method can be used with most graphical applications.

9. Related Work

It is difficult to find work related to this graphical object verification. One possible exception is jRapture [21], a Java-based operational testing method which captures interactions between a Java program and operating system, including GUI, file, and console inputs. Its concept is similar to that of our API and PostScript comparison method, but it depends on the Java programming language. On the other hand, our API and PostScript comparison approach is independent of the programming language and focuses more on graphical objects, since they are the most difficult to handle using current testing techniques. In addition, jRapture tends to reveal software defects that occur on the users' site (often beta users). However, our API and PostScript comparison method tends to find complicated graphic defects and reduce automated testing costs.

10. Conclusion

Our research has shown that the API and PostScript comparison method can be useful for comparing graphical objects automatically. This is a great advance for automated verification work of what has traditionally been a manual, labor-intensive verification process. Though testers have struggled to compare drawing and printed images automatically, we can now offer an alternative method to verify rendered images using API and PostScript comparison. This research has demonstrated the basic steps for realizing API and PostScript comparison techniques:

API comparison was demonstrated on Microsoft PowerPoint by comparing points in each of two rendered graphical objects on screen. In this research, we used the Microsoft Windows operating system. In order to generalize the use of our method, we may need to realize our techniques on the Linux and UNIX operating systems. Supporting our techniques on Windows systems requires complicated tool development, because we were not able to use any Windows source code, but using UNIX and Linux would be much easier since we could use their source code. Thus, we can assume that this API comparison could easily be implemented on those operating systems.

PostScript comparison was also demonstrated on Microsoft PowerPoint to allow automated testing of printed images. Such testing can be expensive and time-consuming, and is difficult to automate. However, this PostScript comparison method makes it possible to automate such testing. In addition, although it is impossible to test graphical objects by using advanced testing approaches, such as model-based testing [17], test case generation [23], dumb monkey [10], random testing, and so on (these methods cannot be implemented due to lack of verification functions), we now have the option of using them with graphical images on both screen and paper.

References

1) Dustin, E., Rashka, J. and Paul, J.: *Automated Software Testing*, Addison-Wesley (1999).
2) Fewster, M. and Graham, D.: *Software Test Automation*, Addison Wesley, New York, USA (1999).
3) Beizer, B.: *Black-Box Testing*, John Wiley & Sons, Inc., New York (1995).
4) Dustin, E.: Lessons in Test Automation, *Software Testing & Quality Magazine* (Sep./Oct. 1999).
5) Kaner, C.: Improving the Maintainability of Automated Test Suites, *International Software Quality Week* (1997).
6) Marick, B.: When Should a Test Be Automated?, *International Software Quality Week*, (May 1998).
7) Mercury Interactive, *WinRunner Users Guide* (2000).
8) Rational Software Corporation, *Using Rational Robot Release 7.5*, Rational Software Corporation, MA (1999).
9) Takahashi, J.: An Automated Oracle for Verifying GUI Objects, *ACM Software Engineering Note* (Jul. 2001).
10) Nyman, N.: Application Testing with Dumb Monkeys, *International Conference on Testing Computer Software* (1999).
11) Pentland, A. and Choubury, T.: Face Recognition for Smart Environments, *IEEE Computer*, Vol.33, Issue 2, pp.50–55 (Feb. 2000).
12) Pankanti, S. and Bolle, R.: Biometrics: The Future of Identifications, *IEEE Computer*, Vol.33, Issue 2, pp.46–49 (Feb. 2000).
13) Microsoft Co.: *Microsoft Win32(tm) Programmer's Reference*, Microsoft Press, WA, USA (1993).
14) Adobe Systems, Inc.: *PostScript(R) language Reference*, Third Edition, Addison-Wesley Publishing Company (1999).
15) Ginsburg, A., Marks, J. and Shieber, S.: A Viewer for PostScript Documents, *Proc. ACM Symposium on User Interface Software and Technology* (1996).
16) Giuffrida, G., Shek, E. and Yang J.: Knowledge-Based Metadata Extraction from PostScript Files, *Proc. 5th ACM Conference on ACM 2000* (2000).
17) Takahahsi, J. and Kakuda, Y.: Extended Model-Based Testing toward High Code Coverage Rate, *Quality Connection* (2002).
18) Myers, G.J.: *The Art of Software Testing*, New York, John Wiley & Sons (1979).
19) Whittaker, J. and Jorgensen, A.: Why Software Fails, *ACM Software Engineering Note*, Vol.24, Issue 4, pp.81–83 (Jul. 1999).
20) Takahashi, J.: Is Special Software Testing Necessary Before Releasing Products to an International Markets?, *International Quality Week* (Jun. 2000).
21) Steven, J., Chandra, P., Fleck, B. and Podgurski, A.: jRapture: A Capture/Replay Tool for Observation-Based Testing, *International Symposium on Software Testing and Analysis* (Aug. 2000).
22) Whittaker, J. and Thomason, M.: A Markov Chain Model for Statistical Software Testing, *IEEE Trans. on Software Eng.*, Vol.20, No.10, pp.812–824 (Oct. 1994).
23) Memon, A.M., Pollack, M.E. and Soffa, M.L: Automated Test Oracle, *International Symposium on Foundations of Software Engineering* (2000).

(Received April 30, 2002)
(Accepted April 3, 2003)

Juichi Takahashi received M.S. degree in Software Engineer at the Florida Institute of Technology and is Dr. Eng. candidate at Hiroshima City University. He is currently quality manager at SAP Labs Tokyo. He has been worked at Microsoft in both U.S.A. and in Japan for 8 years as Software Test Lead. It follows that his interests include software testing and quality assurance. He is a member of IEEE, ACM, and IPSJ.

Yoshiaki Kakuda received the B.E., M.Sc., and Ph.D. degrees from Hiroshima University, Japan, in 1978, 1980 and 1983, respectively. From 1983 to 1991, he was with Research and Development Laboratories, Kokusai Denshin Denwa Co., Ltd. (KDD). He joined Osaka University from 1991 to 1998 as an Associate Professor. He is currently a Professor in the Department of Computer Engineering, Faculty of Information Sciences, Hiroshima City University, since 1998. His current research interests include network software engineering and assurance networks. He is a member of IEEE (U.S.A) and IPSJ (Japan). He received the Telecom. System Technology Award from Telecommunications Advanced Foundation in 1992.