Computer Science
University of Manchester

The University of Manchester

Applying Deployment Contracts to Components from Component Repositories

Kung-Kiu Lau and Vladyslav Ukis

Department of Computer Science
University of Manchester
Preprint Series
CSPP-42

Applying Deployment Contracts to Components from Component Repositories

Kung-Kiu Lau and Vladyslav Ukis

August 2007

Abstract

Currently, components in component repositories are described verbally. System developers choose components for their systems by looking at the verbal component description. In this report, we try to build an email client system using this approach. We take components from three component repositories and compose them. Following this, we run the system in desktop and web execution environments [7, 8] and discover runtime conflicts with the components. We then apply deployment contracts [5, 4, 6] to the components and perform a deployment contracts analysis at component deployment time that discovers the conflicts before runtime. This leads us to the conclusion that components with deployment contracts are better suited for component repositories since deployment contracts analysis can pre-empt runtime conflicts even before the components are composed. That is, system developers can save time and effort when composing such components.

Keywords: components, deployment contracts, component repositories

Contents

List of Figures

List of Tables

1 Introduction

Currently, components in component repositories are described verbally. System developers choose components for their systems by looking at the verbal component description. In this report, we try to build an email client system using this approach. We take components from three component repositories, Shanghai Component Repository [3], Infragistics Component Repository [2] as well as ComponentSpot Component Repository [1], and compose them. Following this, we run the system in desktop and web execution environments [7, 8] and discover runtime conflicts with the components. We then apply deployment contracts [5, 4, 6] to the components and perform a deployment contracts analysis at component deployment time that discovers the conflicts before runtime. This leads us to the conclusion that components with deployment contracts are better suited for component repositories since deployment contracts analysis can pre-empt runtime conflicts even before the components are composed. That is, system developers can save time and effort when composing such components.

2 Email Client System

The system we want to build is an email client. It must be able to retrieve emails from an arbitrary email server using POP protocol [9] given user credentials. Furthermore, it must be possible to print an email into a pdf-file for offline storage and further distribution. Moreover, we want to build 2 email clients. One for desktop use, like Microsoft Outlook, and another one for web use, like Yahoo! Email.

In order to implement the system, we take a look at three component repositories that have free components available capable of providing required functionality. In particular, we require a component that can retrieve emails from an email server, another one that can display them to the user as well as one that can print an email into a .pdf file.

We want to implement the system in .NET and therefore require our components to be .NET components.

2.1 Components

We found a component for retrieving emails from an email server in Shanghai component repository [3]. It is an open-source repository of components that contains .NET components. The description of the component that we found is as follows:

POPComponent .NET Class library in C# for communicating with POP3 Servers (retrieve/delete messages and attachments). Supports decoding files in Base64/QP encoding, including attachemnts, MS-TNEF format, and MS mht file. Compiles on Mono, DotGNU and .NET framework.

It promises to be able to retrieve messages from POP3 servers. So, we chose it.

In order to display emails, we turned to another component repository referred to as Infragistics Component Repository [2]. It contains commercial .NET components with a 30-day free trial period. We found a whole set of GUI components in the repository for desktop and web environments. The description of the components we found is as follows:

NetAdvantage for .NET is a comprehensive suite of controls, components, and tools for the .NET platform, for both Windows Forms and ASP.NET. Whether you're looking to easily

add AJAX to your web applications, add awesome power and control to your WinForms apps, or show the world you know what good UI is all about, NetAdvantage for .NET has what you need. Keep reading to find out more and explore the many ways that this suite can make your life easier and your users happier.

NetAdvantage promises to provide good UI components for Windows Forms, desktop environment, and ASP.NET, web environment. Therefore, we chose these components for our system.

The last component that we need is a component than can produce .pdf files. We found one in ComponentSpot Component Repository [1]. It is a repository of both commercial and open-source components. The component we found there is described as follows:

PDFLIB is a .NET library that allows .NET programmers to create .pdf files from within their Windows Forms and ASP.NET applications. Various methods allow programmers to decide where and how text and lines will be inserted. The PDFLib.Objects namespace contains the PDF object model, with objects such as Catalog, Page, PDFFont, and the PDFDocument.

The component promises to be able to produce .pdf files. So, we chose it for our system.

With these components chosen from three component repositories, we embark on building our email client system. First, we build a desktop version in section 2.2 and then a web version in section 2.3.

2.2 Desktop Email Client System

In order to build a desktop email client system using components we chose above, we first investigate the way the POPComponent works. The initalisation is done as follows:

(1) POPComponent popComponent = new POPComponent();
(2) popComponent.ReceiveMessage += new MessageHandler(popClientMessageReceived);
(3) popComponent.Connect(myMailServer, myPort);
(4) popComponent.Authenticate(myUserName, myPwd);

In (1), the component is instantiated. In (2), the main program subscribes to the event "ReceiveMessage" sent by the component whenever an email is received. In (3), the component connects to an email server and in (4), it authenticates the user.

The component starts receiving emails when the method "GetMessages" is called:

```
popComponent.GetMessages();
```

The emails are received in form of events processed by the routine "popClientMessageReceived":

```
private void popClientMessageReceived(MIMEParser.Message theMessage)
{
  ...
  ultraGrid1.Add(theMessage.From + " " + theMessage.Subject);
  ...
}
```

The routine mainly adds an email to the Grid View component (instance name "*ultraGrid*1") from Infragistics component repository [1].

When we start the system and press the button "Retrieve E-Mail", we get a window shown in Figure 1.

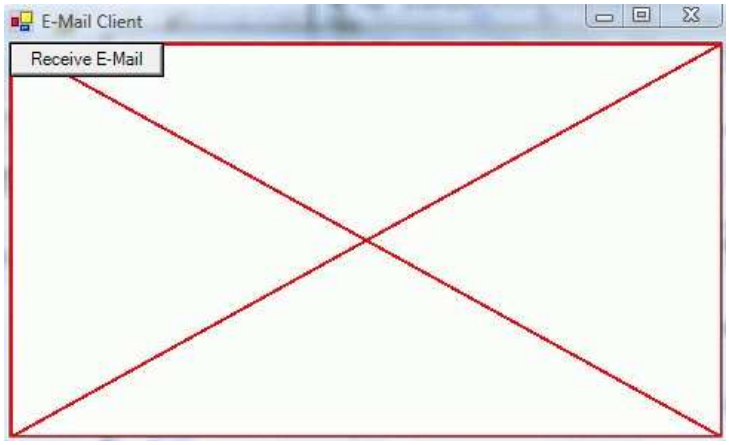

Figure 1: Desktop email client failure.

The window with the email client is jittering. After a debugging session, it turns out that the problem is as follows:

System.InvalidOperationException was unhandled by user code
Message="Cross-thread operation not valid: Control 'ultraGrid1' accessed from a thread other than the thread it was created on."

An unclear problem at first. Then, however, after another debugging session and paying attention to threading, it becomes clear that the Grid View component was instantiated on the main thread but then accessed from the thread that executed event notification from the POPComponent. That is, the POPComponent does not just send its events every time an email is received but does so on a thread different from the one that called its "GetMessages" method. This behaviour cannot be tolerated by the Grid View component.

2.2.1 Replacing POP component

In order to overcome the threading problem from above, we try to find another component that is able to receive emails from an email server. The Shanghai component repository offers such a component with the following description:

OpenPOP is a .NET component which allows you to check, retrieve and delete email messages from a standard POP3 server. This component support multiple attachments and able to decode Quote-Printable and Base64 encoded attachment.

The initialisation of the OpenPOP component is similar to POPComponent above. The main difference is that received emails are not returned in form of events but rather a message can be retrieved by a caller using the method "GetMessage" offered by the component:

[1] Details are hidden here.

```
int Count=popClient.GetMessageCount();
for(int i=Count; i>=1; i−=1)
{
  OpenPOP.MIMEParser.Message m = popClient.GetMessage(i,false);
  ultraGrid1.Add(m.From + ” ” + m.Subject);
}
```

Here, we loop through all available emails and retrieve them one by one using the method “GetMessage” [2].

When running the system, we get the output shown in Figure 2.

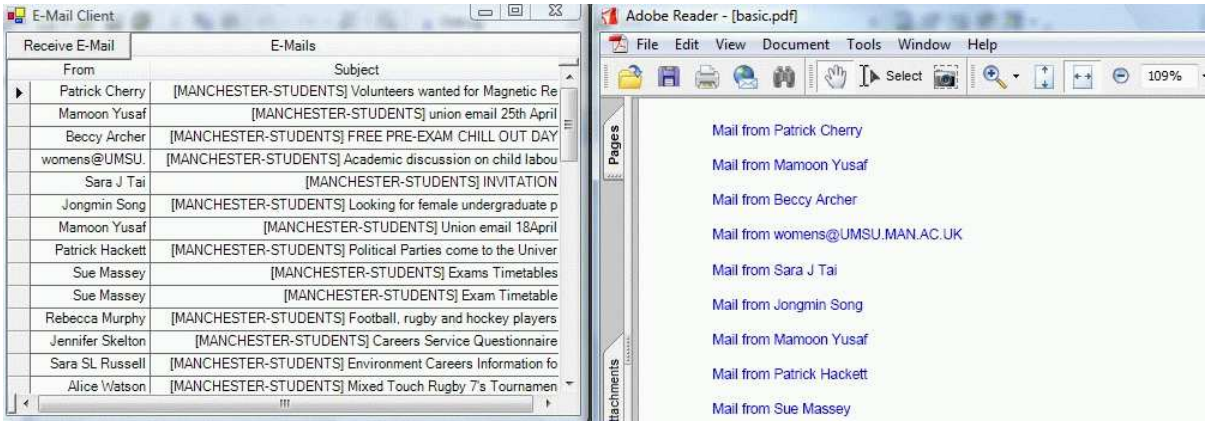

Figure 2: Desktop email client successful execution.

On the left, we can see a list of emails retrieved by the OpenPOP component from the mail server and displayed by the Grid View component. On the right, we can see the list of emails in .pdf format, created by the .pdf component and loaded into Acrobat Reader.

Now that we have build a desktop email client. In the next section we try to build a web email client.

2.3 Web Email Client System

In this section, we want to build a web-based email client system. Unfortunately, we cannot just take our desktop email client system from section 2.2 and deploy it on the web server. This is because the GUI of the system will have to be changed to render HTML so that a web browser can read and display it.

Moreover, we want our web email client system to be highly scalable and deployable even on a web server farm where a separate system instance is running on a separate machine from the farm. Therefore, we do not want to make use of ASP.NET state storages (session and application) as far as possible.

We initialise the OpenPOP component in the standard “PageLoad” method of the site:

```
protected void PageLoad(object sender, EventArgs e)
{
```

[2]Details are hidden here.

```
  if (!IsPostBack)
  {
    myOpenPopComponent = new POPClient();
    myOpenPopComponent.Connect(myMailServer, myPort);
    myOpenPopComponent.Authenticate(myUserName, myPwd);
    int Count = myOpenPopComponent.GetMessageCount();
    MessageCountLabel.Text = Count.ToString() + " E-Mails available";
  }
}
```

In the event handler of the button "Retrieve E-Mail" we retrieve the emails [3]:

```
protected void GetMessagesButtonOnClick(object sender, EventArgs e)
{
  ...
  Message aMessage = myOpenPopComponent.GetMessage(aMessageNumber, false);
  ...
}
```

When we start the system, it looks like shown in Figure 3.

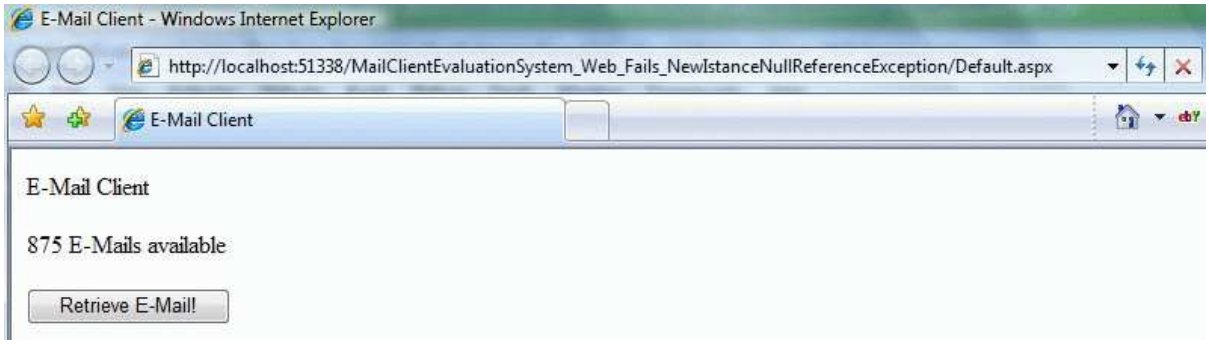

Figure 3: Web email client start view.

However, when we press the button "Retrieve E-Mails" we get the screen shown in Figure 4.

After a debugging session, we find out that the error occurs because the OpenPOP component is not initialised in the event handler of the button "Retrieve E-Mails". This happens because by default in the ASP.NET environment the system is instantiated anew on each request. So, the first request happens when the site is requested and initialised in "PageLoad" method. And the second request happens when the button "Retrieve E-Mails" is clicked. However, on second request the OpenPOP component is not initialised any longer but is a fresh instance.

To avoid the problem we try to put the initialisation code of the OpenPOP component into the event handler of the button "Retrieve E-Mails". The emails can be retrieved then. However, this works only once. When trying to retrieve emails again, we get the screen as shown in Figure 6.

[3]Details are omitted here.

Figure 4: Web email client failure due to uninitialised OpenPOP component.

Figure 5: Web email client failure due to repeated authentication on pop server.

After further debugging and reading pop server documentation, we find out that a pop server may not allow repeated authentications within a certain time period after a successful authentication to avoid spam attacks. So, putting component intialisation code into the event handler of the button "Retrieve E-Mails" does not give us a solution.

The only way to make the system work is to put the initialised OpenPOP component into the session state storage offered by ASP.NET in the "PageLoad" method and retrieve it on subsequent requests. In such a scenario, however, we may run into threading problems as different requests to the web server are processed on different threads and thus the OpenPOP component will be accessed concurrently. (For web server farm deployment, we would also need to check every time if the component is available and if not, create and put it in into the session state. However, this may entail the problem from Figure 6.)

So, in the "PageLoad" method after initialisation, we store the OpenPOP component in the session state as follows:

Session["PoPComponent"] = myPopComponent;

and retrieve the component in the event handler of the “Retrieve E-Mails” button with the code:

myPopComponent = Session[”PoPComponent”] as POPClient;

We then get a running system as shown in Figure 6. However, as mentioned above, threading issues may arise if such a system was deployed in a production environment. In our tests, with several clients accessing the system, no threading issues arose.

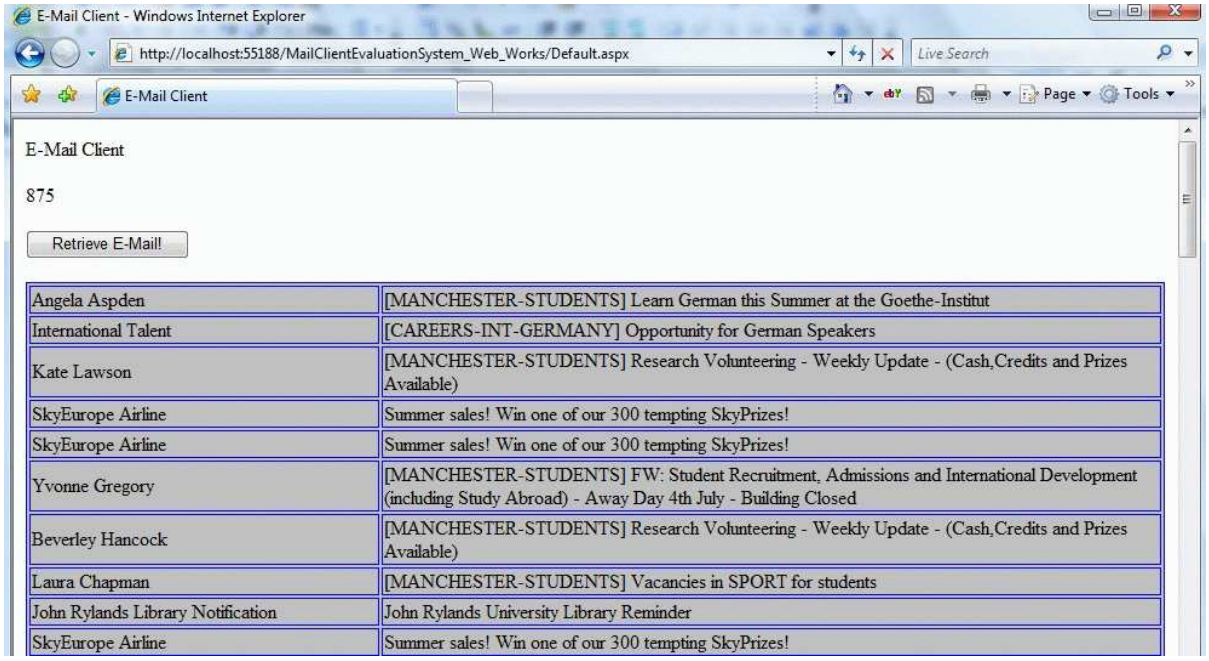

Figure 6: Web email client success.

The .pdf component could be integrated without problems as in the case of the desktop email client.

Now that we have seen several problems while building the systems. In the next section, we apply deployment contracts to the components from component repositories and see if deployment contracts analysis can help spot the conflicts between the components before they are actually composed.

3 Applying Deployment Contracts to Components

In order to perform the deployment contracts analysis of components, we first have to define deployment contracts for each of them. Components from component repositories do not contain deployment contracts. Their descriptions are limited to the verbal descriptions from section 2.1. In this section, we define deployment contracts for the components to enable deployment contracts analysis performed in the next section.

3.1 POPComponent with Deployment Contracts

Deployment Contracts Analyser contains a tool that helps the component developer define deployment contract for a component. The tool does not perform the task automatically though. For the POPComponent, we input the information into the tool shown in Figure 7.

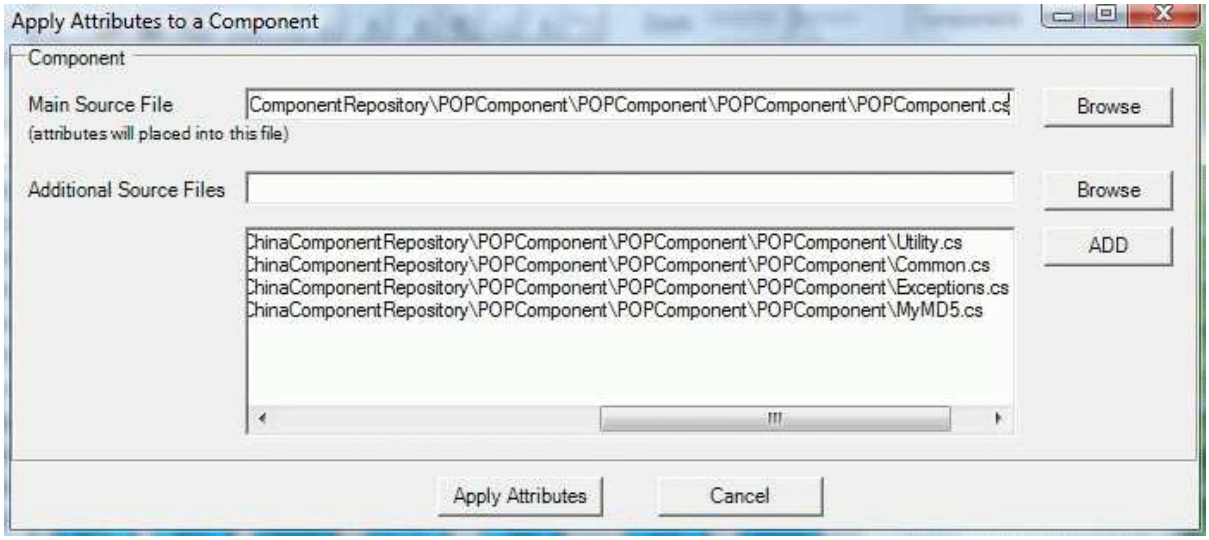

Figure 7: Defining Deployment Contract for POPComponent using Deployment Contracts Analyser.

We specify the main component file, which is the file with the main component class that gets instantiated on component instantiation, as well as other files that belong to the component. We then get a list of attributes that may be relevant to the component's deployment contract. After filtering the list and analysing the component to apply other relevant attributes, we get the create the following deployment contract for it:

```
[UsedMailServer("configurable", "", "", UsageNecessity.Mandatory)]
public class POPComponent
{
  ...
  [AsynchronousMethod]
  [AccessComponentTransientState(StateAccess.Read | StateAccess.Write)]
  [RequirePreviousMethodInvocation("Connect")]
  [IssueCallback("ReceiveMessage", ExecutingThread.InternallyCreatedThread,
   UsageNecessity.Mandatory)]
  public void GetMessages()
  {
    ...
  }

   [CallbackRegistration]
   public event MessageHandler ReceiveMessage;
```

```
  ...
}
```

At the component level, we see the attribute "UsedMailServer" indicating that all component methods make use of it. On the method "GetMessages", we see the attributes

- "AsynchronousMethod" indicating that the method is asynchronous
- "AccessComponentTransientState" indicating that the method access component's transient state, i.e. the connection to the mail server
- "RequirePreviousMethodInvocation" indicating that the method "Connect" must be executed before the method "GetMessages"
- "IssueCallback" saying that the method will issue a callback registered with "ReceiveMessage" event on an internally created thread

We had problems with this component in section 2.2. We will see later on if deployment contracts analysis can detect these problems.

3.2 OpenPOP Component with Deployment Contracts

Following the same procedure as in the previous section, we create the following deployment contract for the OpenPOP component.

```
[UsedMailServer("configurable", "", "", UsageNecessity.Mandatory)]
public class POPClient
{
  ...
  [AccessComponentTransientState(StateAccess.Read | StateAccess.Write)]
  [RequirePreviousMethodInvocation("Connect")]
  public MIMEParser.Message GetMessage(int intNumber, bool blnOnlyHeader)
  {
    ...
  }
  ...
}
```

It also makes use of a mail server in all its methods, accessed component state and requires initialisation to be executed beforehand.

3.3 Grid View Component with Deployment Contracts

The Grid View component we used in our system is commercial and therefore, we could not obtain its source code to put a deployment contract in it. However, as we already know the problem it caused from our experience in section 2.2, we designed a wrapper around it that does nothing else than forward all the calls to the original Grid View component and applied the deployment contract to the wrapper to enable deployment contracts analysis.

```
[UsedMailServer("configurable", "", "", UsageNecessity.Mandatory)]
public class POPClient
{
  ...
  [AccessComponentTransientState(StateAccess.Read | StateAccess.Write)]
  [RequirePreviousMethodInvocation("Connect")]
  public MIMEParser.Message GetMessage(int intNumber, bool blnOnlyHeader)
  {
    ...
  }
  ...
}
```

3.4 PDFLib Component with Deployment Contracts

The PDFLib component is also commercial. Therefore, we do not get access to its source code. We did not experience any problem with it. However, we know that is accesses file system to create and store .pdf files. Therefore, we created a wrapper around it with the following deployment contract to enable deployment contracts analysis.

```
[UsedMailServer("configurable", "", "", UsageNecessity.Mandatory)]
public class POPClient
{
  ...
  [AccessComponentTransientState(StateAccess.Read | StateAccess.Write)]
  [RequirePreviousMethodInvocation("Connect")]
  public MIMEParser.Message GetMessage(int intNumber, bool blnOnlyHeader)
  {
    ...
  }
  ...
}
```

Having components with deployment contracts, in the next section, we perform deployment contracts analysis and see if it can detect conflicts we encountered in section 2.

4 Deployment Contracts Analysis

In section 2, we considered 4 systems:

1. Email client system with POPComponent, Grid View component and PDFLib component deployed into the desktop environment. The system failed due to a threading problem between POPComponent and PDFLib component.

2. Email client system with OpenPOP component, Grid View component and PDFLib component deployed into the desktop environment. The system succeeded.

3. Email client system with OpenPOP component, Grid View component (for web) and PDFLib component deployed into the stateless web environment. The system failed due to incompatible state management of the environment and state handling in OpenPOP component.

4. Email client system with OpenPOP component, Grid View component (for web) and PDFLib component deployed into the web environment with use of session state. The system succeeded but we suspected possible threading issues.

Now that let us see if deployment contracts analysis of components can also give us these insights before the components are actually composed by the system developer.

4.1 System 1

System 1 consists of the POPComponent, Grid View component and PDFLib component, and is deployed into the desktop environment.

In order to enable deployment contracts analysis of this system, we first have to define requests that can go through the system. In this case, we only have one request consisting of initialising the POPComponent, retrieving emails from the mail server, displaying retrieved emails using Grid View component and creating a .pdf file with a list of emails or text of an email. The request creation is done graphically using Deployment Contracts Analyser (DCA) Tool as shown in Figure 8.

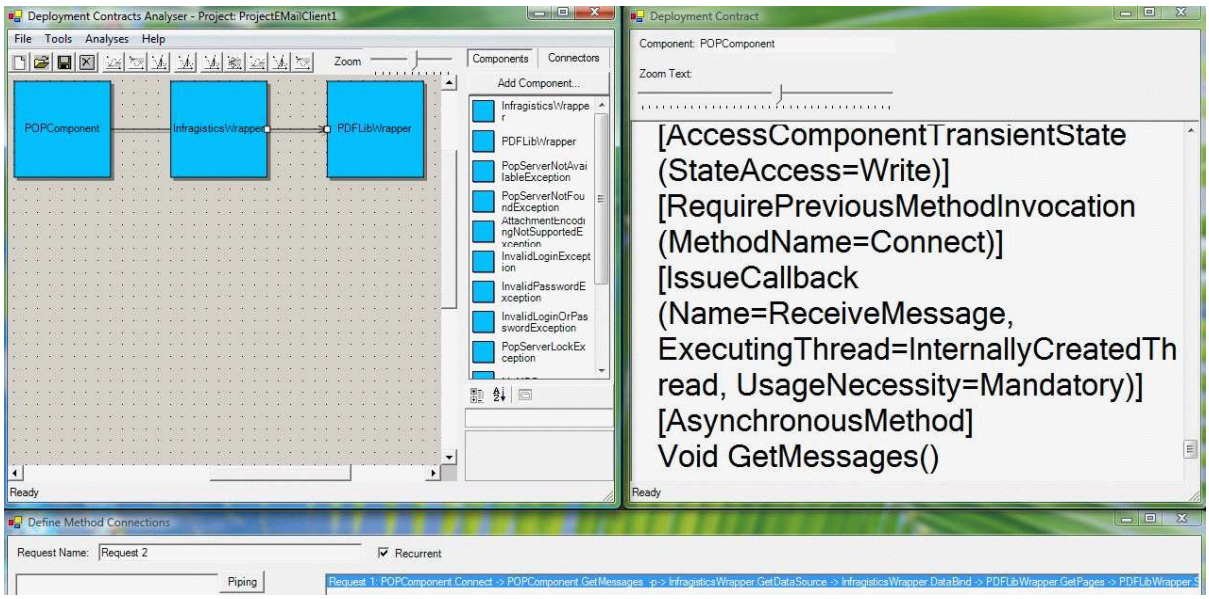

Figure 8: Defining a request using Deployment Contracts Analyser Tool for the System 1.

The request is shown at the bottom in Figure 8 and is created by selecting component methods in the right upper window in the figure.

Once the request is created, we can run the actual automatic deployment contracts analysis that is shown in Figure 9.

The result of the analysis can be seen at the bottom in Figure 9. It shows exactly the error that we encountered with the system in section 2.

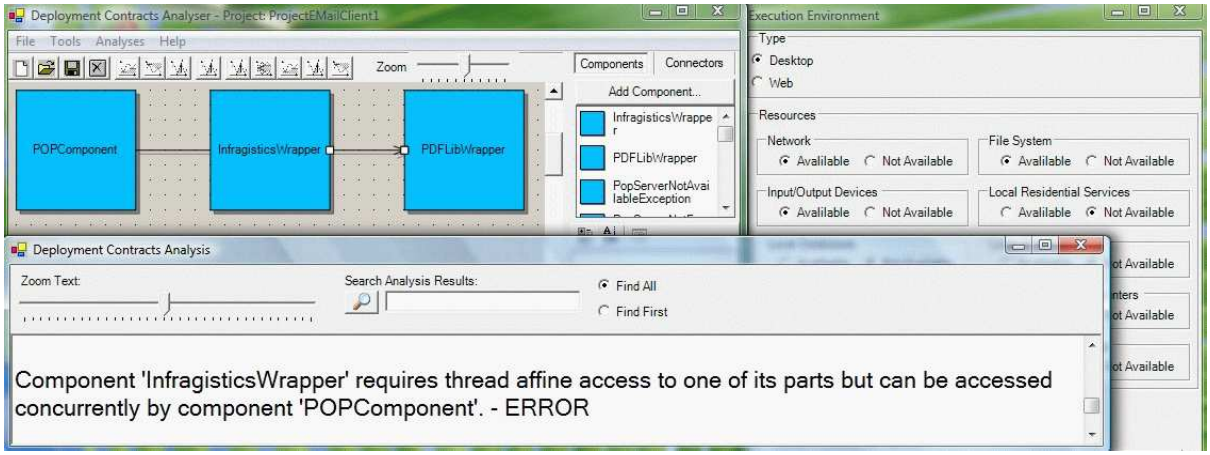

Figure 9: Deployment Contracts Analysis for the System 1.

4.2 System 2

System 2 consists of the OpenPOP component, Grid View component and PDFLib component, and is deployed into the desktop environment.

In order to enable deployment contracts analysis of this system, we first have to define requests that can go through the system. In this case, we only have one request consisting of initialising the OpenPOP, retrieving emails from the mail server, displaying retrieved emails using Grid View component and creating a .pdf file with a list of emails or text of an email. The request creation is done graphically using DCA as shown in Figure 10.

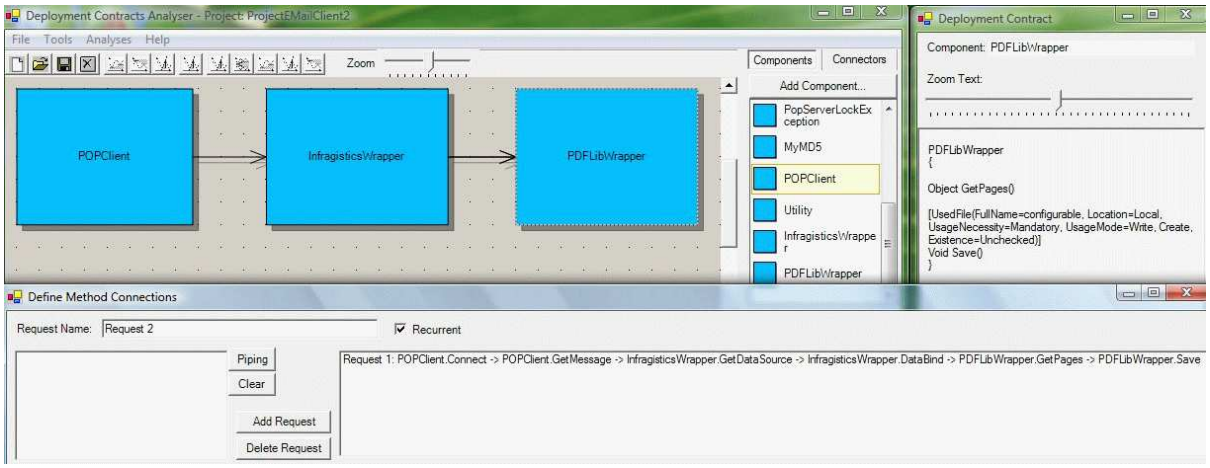

Figure 10: Defining a request using Deployment Contracts Analyser Tool for the System 2.

The request is shown at the bottom in Figure 10 and is created by selecting component methods in the right upper window in the figure.

Once the request is created, we can run the actual automatic deployment contracts analysis that is shown in Figure 11.

The result of the analysis can be seen at the bottom in Figure 11. It shows no errors, which

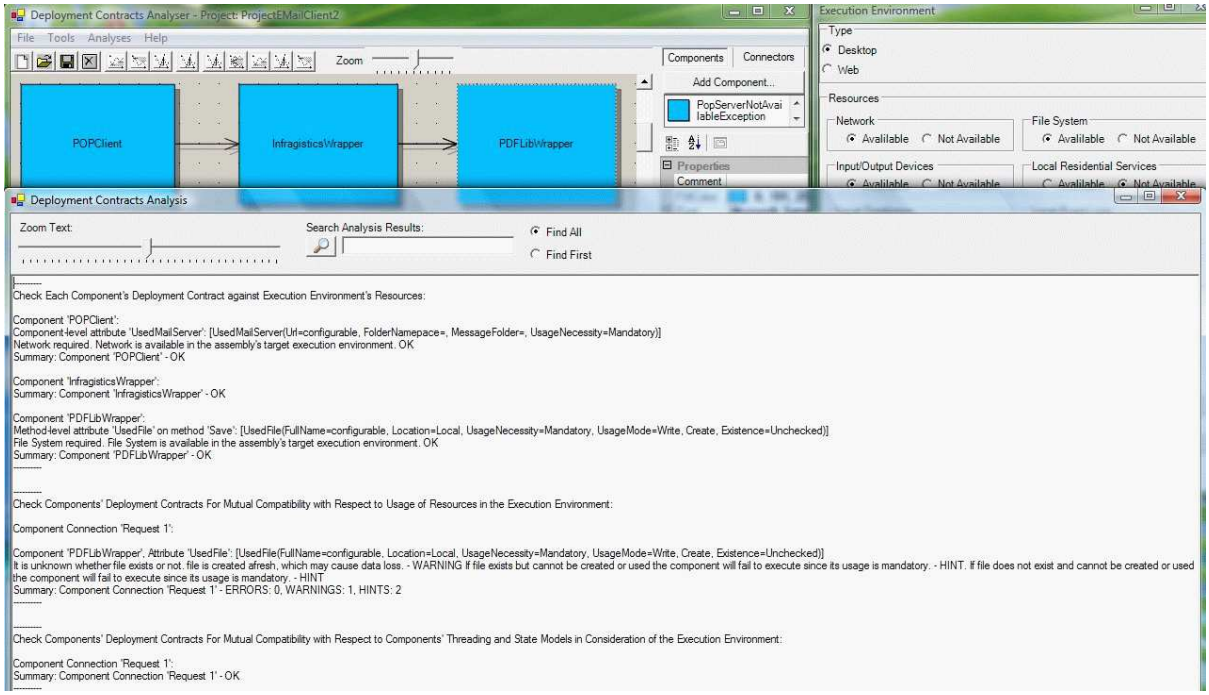

Figure 11: Deployment Contracts Analysis for the System 2.

is the behaviour we observed in section 2. The analysis only points out that the PDFLib may be problematic if it cannot re-create the file that already exists. Furthermore, data loss may occur if the PDFLib successfully recreates an existent file with important data.

4.3 System 3

System 3 consists of the the OpenPOP component, Grid View component (for web) and PDFLib component, and is deployed into the stateless web environment. The deployment contracts analysis for the system is shown in Figure 12.

The analysis detects the problem with state we witnessed in section 2.

4.4 System 4

System 4 consists of the OpenPOP component, Grid View component (for web) and PDFLib component, and is deployed into the web environment with use of session state. The deployment contracts analysis for the system is shown in Figure 13.

The analysis warns us that the state is only retained for the period of a user session inviting us to check if this is suitable for the system. However, the analysis does not detect our suspicion of a threading issue with the OpenPOP component. This is, however, impossible since the analysis does not know what will be put into the session state.

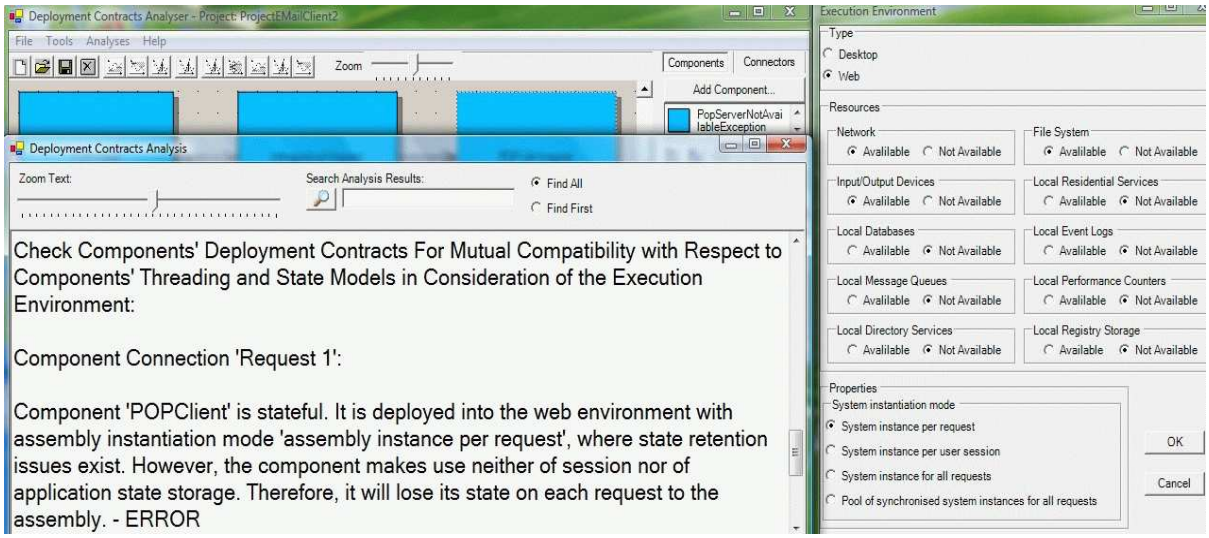

Figure 12: Deployment Contracts Analysis for the System 3.

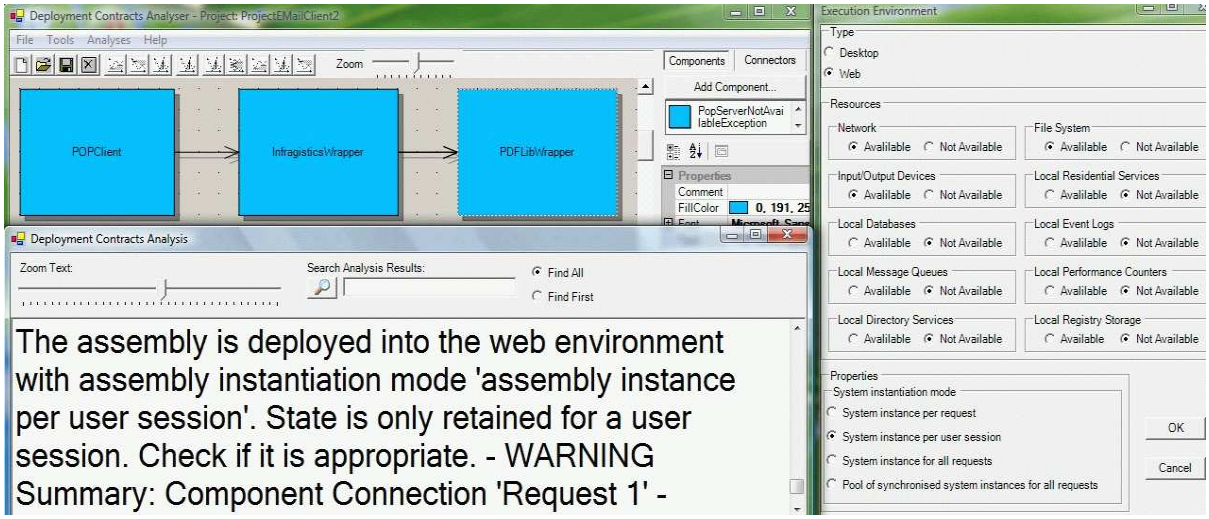

Figure 13: Deployment Contracts Analysis for the System 4.

5 Conclusion

In this report, we tried to build an email client system by drawing on existing components from three component repositories [3, 2, 1]. We discovered runtime conflicts with components. Following this, we applied deployment contracts to the components and performed deployment contracts analysis. The analysis showed the conflicts at component deployment time, before runtime. Therefore, we conclude that it is useful to have components with deployment contracts in a component repository since this saves time and effort for system developers who compose components. With deployment contracts, conflicts with components are discovered at deployment time even before components are actually composed. Without them, components

are discovered only at runtime after the whole system has been built.

References

[1] ComponentSpot Component Repository. http://www.componentspot.com.

[2] Infragistics Component Repository. http://www.infragistics.com/downloads/default.aspx.

[3] Shanghai Component Library. http://www.sstc.org.cn.

[4] K.-K. Lau and V. Ukis. A Reasoning Framework for Deployment Contracts Analysis. Preprint 37, School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK, June 2006. ISSN 1361 - 6161.

[5] K.-K. Lau and V. Ukis. Defining and Checking Deployment Contracts for Software Components. In *Proceedings of the 9th International Symposium on Component-Based Software Engineering*, volume 4063 of *LNCS*, pages 1–16, Stockholm, Sweden, June 2006. Springer.

[6] K.-K. Lau and V. Ukis. Deployment Contracts for Software Components. Preprint 36, School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK, February 2006. ISSN 1361 - 6161.

[7] K.-K. Lau and V. Ukis. A Study of Execution Environments for Software Components. In *Proceedings of the 10th International Symposium on Component-Based Software Engineering*, LNCS, Boston, USA, July 2007. Springer.

[8] K.-K. Lau and V. Ukis. On Characteristics and Differences of Component Execution Environments. Preprint 41, School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK, February 2007. ISSN 1361 - 6161.

[9] Polychronis Tzerefos, Colin Smythe, Ilias Stergiou, and Srba Cvetkovic. A comparative study of simple mail transfer protocol (smtp), post office protocol (pop) and x.400 electronic mail protocols. In *LCN*, pages 545–554, 1997.