Journal of Object Technology
Published by ETH Zurich, Chair of Software Engineering, © JOT 2010
Online at http://www.jot.fm.

Dynamic adaptability with .NET service components

Arun Mishra[a] A.K. Misra[a]

a. Department of Computer Science and Engineering, Motilal Nehru National Institute of Technology, Allahabad, India.

Abstract In self-adaptive systems components are dynamically modified according to the execution environment requirement, where each component is a probable point of failure. Existing approaches to make such systems more vigorous and safe are both brittle and time intensive. A framework for dynamic adaptation has been designed to automate the component integration process at runtime by accessing the equivalent component from a repository of components. The .NET technology allows developers to adapt run-time components by specifying component behavior using pre- and post-assertions on the component's services. Components can be compared for behavioral equivalence by comparing the assertions. These assertions will help us to compute the utility value for each component in the repository and the component with the highest value is picked for replacement. In this paper, we describe the mechanism for component adaptation using .NET services, by considering a system in a dynamic context with proxy switcher and network switcher components.

Keywords Assertions, Component Assessment, Dynamic Adaptation, Abstract Syntax Tree

1 Introduction

In self-adaptive systems where changes can occur in their execution environment, component performance can degrade dramatically and may reduce system performance. This kind of system achieves its goal by unlimited availability of duplicate components and so imposes strong synchronization requirements on the system to distinguish corrupt and correct components, and these requirements may cause system performance to degrade to unacceptable levels [DM03]. In our work, we introduce a framework that automates the component selection and integration process. We assume that the software can be reconfigured at runtime, whenever it is not meeting its objective, by safe reassembly of components [FAPV04, PJMW07]. We present a technique to automate the component assessment procedure. Our model compares behavioral aspects of the components of the executing system to the behavioral aspects of the

Arun Mishra, A.K. Misra. *Dynamic adaptability with .NET service components.* In Journal of Object Technology, vol. 9, no. 5, 2010, pages 75–94. Available at http://www.jot.fm/contents/issue_2010_09/article4.html

components in the repository. The result of the comparison is captured by a utility value, computed for each component present in the repository, which describes the functional resemblance with the component to be replaced. On the basis of the utility value, our approach finds out which is the most suitable component for integration. This assessment is achieved by deriving Abstract Syntax Trees (AST) of the assertions and comparing them [BYMSB98].

We have done our experimental study with the self-adaptive application using .NET services, with proxy switcher and network switcher components in a dynamic context. We have implemented construction of ASTs from the attached assertions on the services of the components.

The Microsoft .NET framework improves on existing component systems, such as J2EE, COM & COMT and CORBA, in a number of ways [FJTJ03]. Many features of .NET provide much improved support for dynamic re-configuration [SGM02]. We have used built-in services of the .NET framework which allow us to recover component interfaces to find out component behavior at runtime. In this paper, we focus on how the .NET services are used to compute the metadata to support the selection of components at run-time.

The rest of the paper is organized as follows: section 2 presents the related work in the field of self-adaptive architecture, component assessment and integration. Section 3 presents the need to automate the component integration and our approach. Section 4 describes the methodology for component assessment. Section 5 provides metadata-driven solutions for component assessment using .NET services. Section 6 presents a framework for dynamically component assessment and integration. Section 7 gives an example of dynamic adaptation. Section 8 reports on experimental work and finally the conclusions and possible future work.

2 Related work

In component-based software engineering, self-adaptive software has been successfully applied to a diversity of tasks ranging from strong image interpretation to automated controller synthesis [LRS01]. Most of these focus on self-adaptive architecture [Garlan04], [Oreizy99], [FHS06] to make possible realization of their models, but have not paid much attention to the fundamental issue of the reason for software failure. Paul Robertson *et al.*'s work on automatic recovery from software failure gave many reasons why software fails [RW06]. Our approach is based on utility function whose values are calculated with the help of assertions on the services of the components. The comparison of the assertions at run-time and the process of component assessment is completely transparent to the user. Our approach builds on a successful account of software diagnosis at runtime as proposed in [WN97]. In 1999, Peyman Oreizy *et al.*, proposed an architecture of self-adaptive software. This architecture describes an infrastructure based on two processes: one is on system evolution and other is on system adaptation. Central to the author's view is the dominant role of software architecture in planning, coordinating, monitoring, evaluating, and implementing seamless adaptation [Oreizy99]. However, assessment of components at run-time for modification was lacking. Ira D. Baxter *et al.* worked on clone (duplicate code) detection. They presented simple and practical methods for detecting exact and near miss clones over arbitrary program fragments in program source code by using abstract syntax trees [BYMSB98]. As per their first step in the clone detection procedure, the source code is parsed and an AST is formed. After that, three main algorithms are

applied to find clones. Our concept of assertions matching also uses a similar technique using ASTs, but we used the meta information to build the AST.

3 Why automate the component integration and our approach

Techniques that handle component level failure are costly in terms of time taken to thoroughly scrutinize, recognize and specify a reply to all possible malfunctions of a system [HFS04]. Automating the component assessment and integration process is a necessary task to achieve optimal performance in a dynamic environment. Our objective is to introduce a framework using .NET services that automates the component integration process. We devised a lightweight assessment technique for component integration which improves overall performance when the system is under stress during the reassembly of components.

4 Component assessment

Our approach uses the concept of sufficient accuracy rather than absolute correctness [Shaw02]. Such a system should make every effort to maintain its normal operating behavior by using the best available alternative component to replace the failing one. The alternative component detection in terms of component behavior is based on component structure; that structure must include signature of required services, pre-post assertions to abstract out the hidden behavior of the services and the order in which the services are invoked in a component [FGP05]. With the help of the reflection mechanism of the .NET framework we can access the signature of services and easily detect equivalence in the number of parameters in corresponding operations, types of their parameters and their corresponding return type. To match the behavior of corresponding components, we need to prove equivalence of pre- and post-assertions of respective component services. For that we have designed the utility function that generates a scalar value for each component in the repository. The utility function computes the sum of matched assertions between services of two components. Next, we define the threshold of the component, denoting the minimum number of match operations required to replace the component. Thus, by comparing the threshold of the utility value, system should be able to decide which component is the best alternative component for a given respective component. An alternative component is a component which has equal or higher value in its utility, compared to the threshold. Now, we describe the utility function that assigns a scalar value to each component in the repository.

Valuating the utility of components

Alternate component selection is based upon a component's utility. Utility is valuated with a technique which is based on properties of an Abstract Syntax Tree (AST) [BYMSB98]. Since the valuation of component behavior depends on the pre- and post-assertions, it is necessary to add pre- and post-assertions to the services of components to abstract from the hidden behavior of components. To establish the equivalence of assertions, we build the AST from the assertions. To find exact tree matches, a number of transformations are required, which take the form of commutative operators or non-commutative operators [BYMSB98]. These changes include nodes with commutative operators like add (+) where their sub-trees are interchangeable

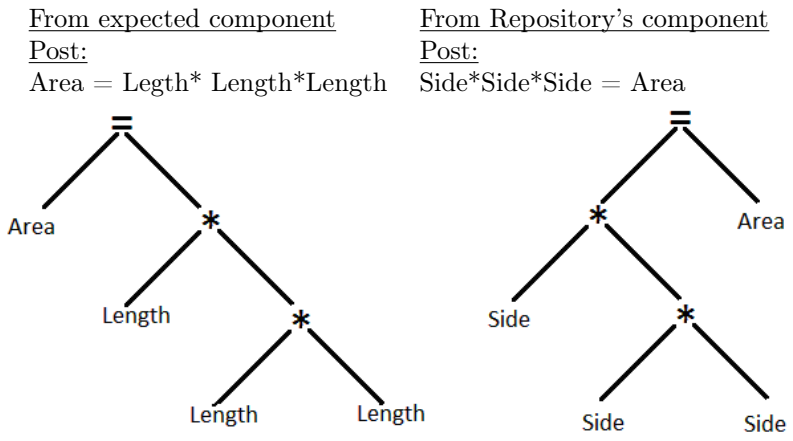

Figure 1 – AST of Post-assertion of expected component (left) and AST of Post-assertion of component from repository (right)

i.e., (x+y) same as (y+x) where x, y are sub-tree of commutative operator addition, nodes with non-commutative operators like subtract (-) where their sub-trees are not interchangeable, and nodes with numbers or variable names that are considered to be of type text. Statements including Boolean operators AND and OR are transformed into a normalized form. The opposite operator of $<$ is $>$; transformation can be applied into their sub-trees to establish similarity at the root level.

As an initial step in the utility valuation process, the assertions of the corresponding services of the respective components are parsed and ASTs produced for them. According to the commutative and non- commutative operators, there are two possibilities when comparing trees. In the first possibility, comparisons are performed when root node of both trees are equal and have a commutative operator. In second possibility, comparisons are performed when root node of both trees are equal and have a non-commutative operator.

The example in Figure 1 shows the post assertions for the computation of cube volume services. Both ASTs present commutative root node (*i.e.*, =), so according to the first possibility, we compare the left sub-tree of first AST with the right sub-tree of the other, and vice versa. After that, left and right subtrees are recursively compared. However, for the values on leaf nodes, we consider that the two sub-trees with text nodes and having no children are alike. Hence, we can easily detect the equivalence regarding both post assertions.

In previous work [MM09], we designed the basic tree equivalence algorithm to compare the ASTs. The algorithm considers above said two possibilities, and establishes equivalence of the ASTs. We use this information to assigne value to the utility function. If pre- and post-assertions are equivalent, the utility of respective component is increased by one (initially it is zero).

This process is applied recursively until all sub-trees of all pre- and post-assertions are compared. Finally, the utility value denotes the number of equivalent services in a given component in the repository with respect to a given one (the component to be replaced in the system). The same process is followed to calculate the utility of all the components in the repository.

A service has dependencies with one or more services of a component *i.e.*, if a

service is executed before the other; then such a service will react in a certain way. We call it the likely interactions of a component. To find the equivalent component, the order in which services are invoked should be equivalent between respective components. We have represented likely interactions by means of regular expressions. A regular expression represents different possible combinations of likely interactions of a component. In a previous work [MM09] we have presented an example having two components shapeArea and GeomtericalShapeCoverage. The component's interfaces are given below (specified in OCL). In order to be sure about similarity at "services execution order" level our assessment procedure needs to be run.

```
Component ShapeArea{
  double CircleArea(double radius);
  double RectangleArea(double length, double width);
  double SqureArea(double length);
}

Component GeometricalShapeCoverage{
  double GetCircleCoverage(double radius);
  double GetRectangleCoverage(double width, double length);
  double GetSqureCoverage(double length);
}
```

For the sake of simplicity, we consider first execution of CircleArea (other services can be invoked in any order and any number of times) service of component ShapeArea and GetCircleCoverage(other services can be invoked in any order and any number of times) service of component GeometricalShapeCoverage. The regular expressions for these two services are:

1. ShapeArea.CircleArea. (RectangleArea+ SquareArea)*
2. GeometricalShapeCoverage.GetCircleCoverage. (GetRectangleCoverage+ GetSquareCoverage)*

To verify the equivalence of the regular expressions, we have used again the basic tree equivalence algorithm [MM09]. The ASTs of both regular expressions are shown in Figure 2.

In the face of commutative or non-commutative operator, the type of a tree's nodes depends on regular expressions. Alternative (+) is a commutative operator type. Concatenation (.) is a non commutative operator. Iteration (*) is a unary operator type. Text node corresponds to services in the leaves of the tree.

Similarly, we can prove the equivalence between regular expressions for other services in the components.

Now, we can conclude that, detection of alternative components is possible with the help of utility function and regular expressions of component's interactions.

5 Metadata-driven solution for component assessment using .NET services

In .NET we can add assertion as metadata on the member of the component by annotating them with so-called attribute specifications. An attribute specification

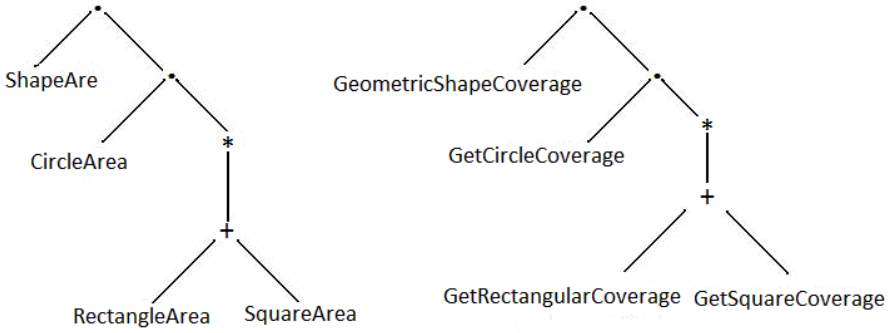

Figure 2 – ASTs for regular expressions of CircleArea service from ShapeArea Component (left) and regular expression of GetCircleCoverage service from GeometricalShapeCoverage (right)

consists of a type name which names an attribute class, plus an argument list consisting of literal expressions. The type name and the literal value are stored in the assembly by the programming language compiler (i.e. C #, VB).

At run-time, this information is used by the CLR (middleware service of .NET framework) to create an instance of the named attribute class. The CLR and the application itself can retrieve the instances associated with an element, and act upon them [CPG09]. Implementation level details for attribute specification and their processing are discussed in Section 8.

6 A framework for component assessment and integration

Figure 3 shows a framework for component adaptation at run-time. We specify the framework as a collection of integrated components, allowing run-time adaptability. It involves two units; one is an execution manager and other is a component extractor.

6.1 Execution Manager

When an application is launched Execution Manager unit continuously monitor the behavior of the application. This unit has three major components. The working of all three components has been divided into different sections:

1. Run-Time Instrumentation: While the application is running, the Execution Manager unit determines the properties of interest for evaluating application behavior in the current context. The Context Instrumentation component of execution manager is responsible for this function [Oreizy99].

2. Reasoning about collected observations: There is needed to scrutinize the run-time instrumented behavior of the application with respect to the expected behavior of the application. If any deviation is found (because of context change), the Adaptation Strategies must reason about the impact of deviation and select an appropriate component, with the help of the Component Extractor unit of framework, that best suits the current context.

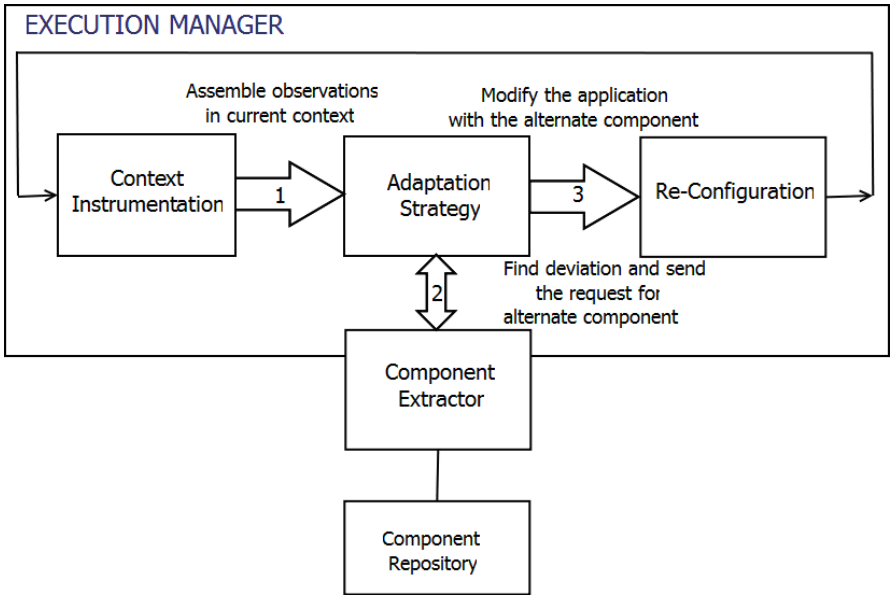

Figure 3 – Framework for Component Adaptation at Run-time

3. Change configuration: The Re-configuration component of execution manager is responsible to connect new component, block old (faulty) components and restart service to ensure the functionality of the application.

6.2 Component Extractor

The Component Extractor will provide services to the Execution Manager unit of the framework. The Adaptation Strategies component of the Execution Manager unit sends requests for alternative components. Component Extractor fetches the best alternative component (its reference) from the repository, according to the previously specified approach of Component Assessment described in section 4 of this paper. And finally, sends the reference of the extracted component to the Adaptation Strategies component of the framework.

7 Dynamic Adaptation: An example

As an example, we consider a web browser running in a system. Several changes can occur in the work environment. For example, the proxy server, through which the internet connection is available, may fail or work for a period without wired connection *i.e.*, for this period, network connection is available through wireless access point. To adapt to these changes, the application might switch to another available proxy server or switch to wireless access point. So, when such changes occur the application should adapt accordingly.

We are putting forward a dynamic adaptive system which involves two components designed to detect two types of failure in the system: Internet proxy failure and Network connection failure. Detection of point of failure in the web browser is based on hierarchical dependency of components (from top to bottom in hierarchical order;

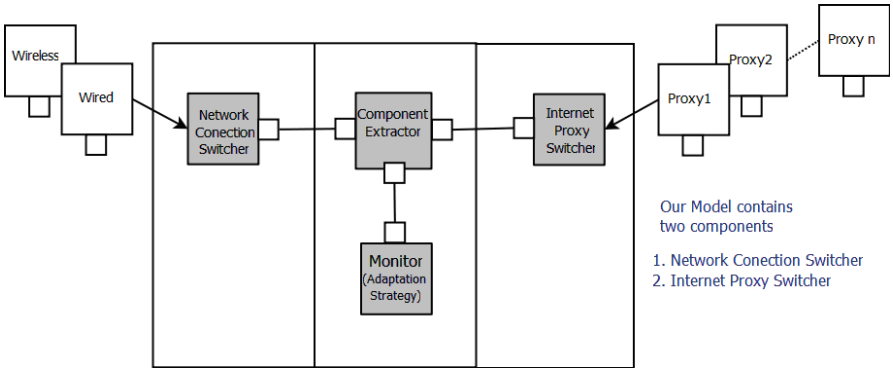

Figure 4 – Framework for the Dynamic Assimilation of Internet Proxy Switcher and Network Connection Switcher

each component depends on its predecessor). In our system, Network connection failure precedes Internet proxy failure. So, network connection is checked first for point of failure and then Internet Proxy is checked.

After detection of a fault, the system searches for the best suitable alternative component to replace the failed component and integrates it in order to revitalize the system. Selection of an alternative component at run-time is done with the help of .NET services such as Custom attribute [CPG09] and Reflection [MBI09].

Figure 4 shows the system's component framework for the dynamic integration of Internet proxy switcher and Network connection switcher.

The system framework is composed of three segments

1. Component switcher (after detection of a fault, the system may adapt either the Network connection switcher or the Internet proxy switcher)
2. Browser Helper Object (BHO) work as a Monitor
3. Component extractor

Component Switcher

Our system has two main components: one is Network connection switcher whose framework is shown in figure 6 and the other is Internet proxy switcher whose framework is shown in figure 5. When changes occur in the application's running context, the system adapts the corresponding components to minimize the impact of changes on the system. For instance, Network connection switcher replaces the failed network connection with an alternative network connection *i.e.*, when wired network connection fails, wireless network is connected. Internet proxy switcher sets the working internet proxy server IP (proxy that gives reply on pinging it) in Internet Explorer by replacing the failed internet proxy IP.

To abstract the hidden behavior of components, we add metadata in the form of pre-assertion and post-assertion on each component. Custom Attribute Service provided by Microsoft's .NET technology used to attach pre-assertion and post-assertion to each component in the component assembly [CPG09].

For Proxy switcher component, following methods contains pre-assertion and post-assertion.

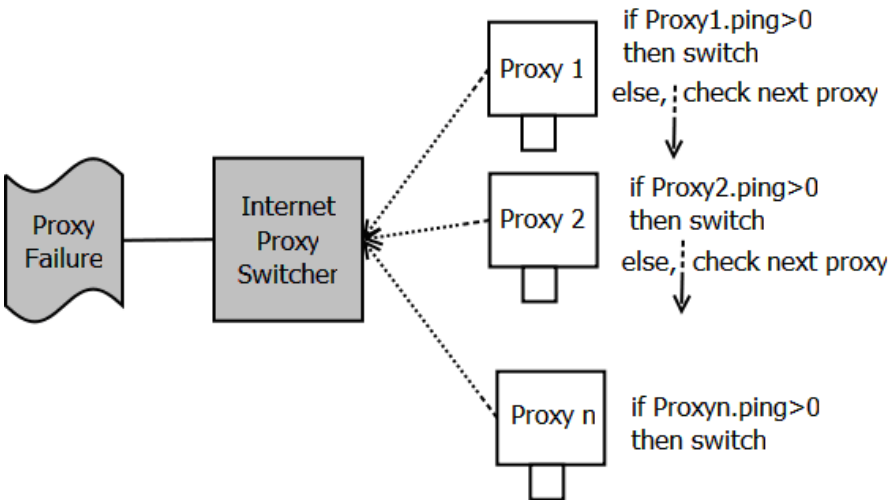

Figure 5 – Proxy switcher framework architecture

To define the assertion, we follow the prefix representation that makes easier the task of conversion of assertion into Abstract Syntax Tree.

```
Public class proxy_switcher {
 [Pre-assertion ("=(( IPAddress.TryParse(IP,outip)),true)")]
 //A function which set the poxy IP, takes IP address as input.
 //So, Pre-assertion checks the validity of IPAdress}

 [Post-assertion("=((Ping.Send(ip,1000,buffer,null)).status),true")]
 //For the same function Post-assertion checks if the IP address on
 ping gives success, with the help of send function of Ping class.

 public static void Set_Proxy(IPAddress ip);
}
```

The `proxy_switcher` component is loaded from component assembly when the current proxy set in Internet Explorer fails. The `Set_Proxy`(void Set_Proxy(IPAdress ip)) function is called, it pings the available internet proxies and sets the first found working proxy(Internet Proxy server IP that gives reply on pinging it) in Internet Explorer [ASIEP09].

For network switcher, the `pre_assertion` and `post_assertion` are as follows:

```
public class Network_switcher {
 [Pre-assertion(">(WirelesszeroConfigNetworkInterface.PreferredAccess
 Points.Count),0)")]
   //Checks if the Wireless networks (Access points) are available by
   counting the available access points
 [Post-assertion("=(WirelesszeroConfigNetworkInterface.connectTo
 PreferedNetwork(apName),true)")]
   //Connection is established with the access point, apName
```

`Network_switcher` component is loaded from component assembly when the current wired Network connection fails. Wireless function (`void wireles ()`) is called

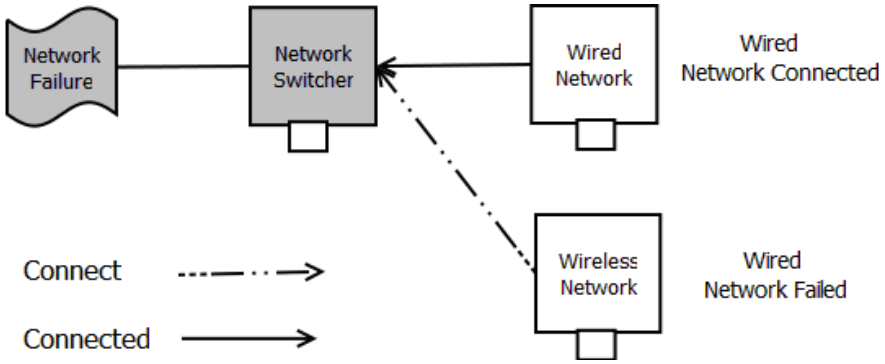

Figure 6 – Network switcher framework architecture

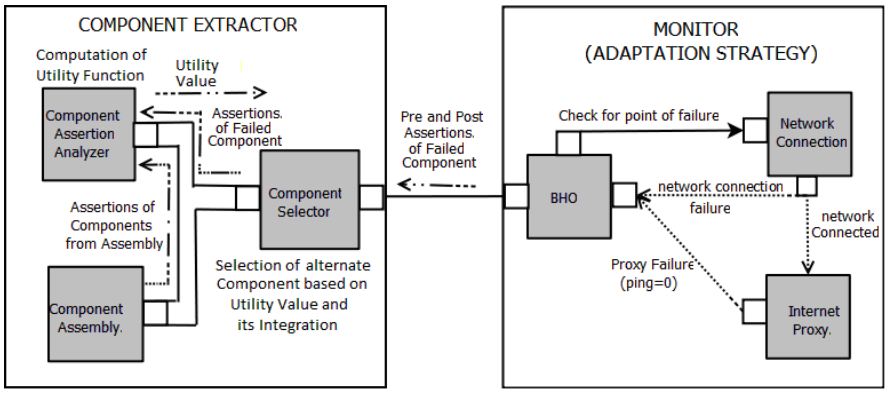

Figure 7 – Architecture framework of Component extractor and Monitor segment

and it connects to available wireless network (Access point). We used WirelessZero-ConfigNetworkInterface| present in OpenNet Class Framework (OpenNETCF.Net 2.3) provided by Microsoft, to connect with nearby points [WZCNI09].

Monitor

In our system monitoring is done by BHO (Browser Helper Object) attached to Internet explorer. BHO is a DLL module designed as a plug-in for Microsoft's Internet Explorer web browser to provide added functionality [BHO09]. Some of the additional functionalities provided by BHO are detection of browser's typical events, such as Go-Back, Go-Forward, and Document Complete and access to browsers menu and toolbar [BHOVS09]. We are using Document Complete event of BHO in our application to detect the failure of Internet browser.

BHO checks the components in the order of their hierarchical dependency (Network connection failure component first and Internet proxy failure next) *i.e.*, BHO played a major role in deciding adaptation strategy.

Whenever it encounters a failure, it passes the pre-assertion and post assertion of the failed component to Component Extractor unit of our framework. Figure 7 shows the framework architecture of Monitor segment (BHO) and Component extractor segment.

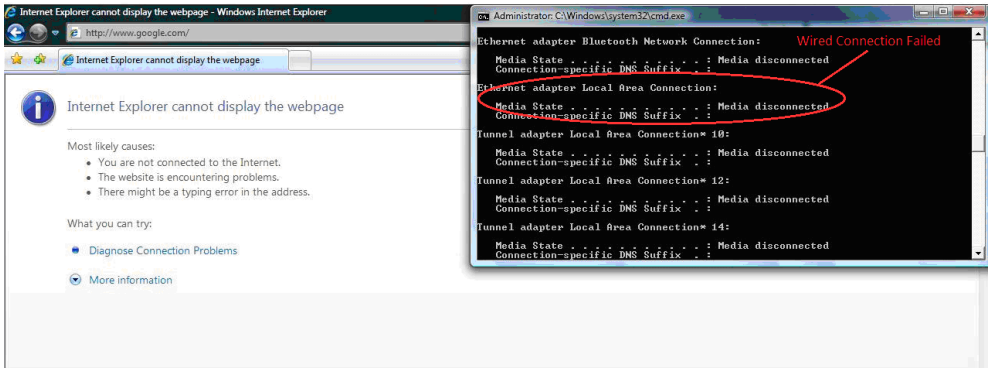

Figure 8 – This snapshot shows disconnection of wired media

Component Extractor

This segment gets the pre-assertion and post-assertion of the failed component from Monitor. It extracts the suitable alternative component (with the help of attribute services and reflection mechanism in .NET) by comparing its pre-assertion and post-assertion with the respective failed one and integrates it. Component Selector obtains the pre-assertion and post-assertion of failed component from the Monitor. It extracts all the components from component assembly and provides each component's pre-assertion and post-assertion to the Assertion Analyzer module. It compares the pre-assertion and post-assertion of the respective services and finds matches. This work is done according to the Component Assessment technique in section 4 of this paper. In our example, Component Selector also plays the role of re-configuration. It integrates the selected alternative component and re-configure the system. The coordination between Component Extractor and Monitor segment is shown in fig. 7.

8 Experimentation

We validated our approach by implementing two simple case examples: In case I the network connection failure and in case II the Internet proxy failure, is considered as a malfunction point in system execution environment. These are fairly simple adjustments in the environment. We executed our system's components in a simulated context environment. On running Internet Explorer in our system, BHO detected that there is a failure in Internet Explorer (by reading the HTML content of Internet Explorer every time the page gets loaded) and it starts finding the point of failure. In the first case, it found that the failure is at the wired network connection, and in the second case, found that the wired network is connected but the IP address of the proxy server is not replying.

To elaborate our experiment setup, we developed the prototypes for each case. These prototypes have the component's code and the snapshots which show the approach's feasibility. Prototypes are based on .NET framework's services. The first subsection presents a prototype for case I, explaining the working of BHO, Component Extractor, and Network Connection Switcher components. In the second subsection, we present a prototype, depicting the Component Extractor behavior for Internet proxy failure case.

Network Connection Failure

On running Internet Explorer in our system while it was not connected to wire network, BHO component senses this change in the execution environment. Figure 8 shows, the snapshot after the adjustments made in case of wired media.

To explain the details of our experiment, we developed the prototype explaining the working of BHO, Component Extractor, and Network Connection Switcher components.

The following code prototype (in Visual C++) depicts the working of BHO:

```
void void STDMETHODCALLTYPE BHO::OnDocumentComplete(IDispatch *pDisp,
VARIANT *pvarURL) { //dynamic html is used to catch the events of IE

  CComPtr<IDispatch> spDispDoc;

  HRESULT hr = m_spWebBrowser->get_Document (&spDispDoc);

    if(SUCCEEDED(hr)) //When page loads
      {
         CComQIPtr<IHTMLDocument2, &IID_IHTMLDocument2> spHTML;
         spHTML = spDispDoc;
    if(spHTML)
      {
         CComPtr<IHTMLElement> m_pBody;
         hr = spHTML->get_body(&m_pBody);
         BSTR bstrHTMLText;
         //Reads the content of the Html page of Internet explorer
         hr = m_pBody->get_outer
         HTML(&bstrHTMLText);
         CW2A tmpstr1(bstrHTMLText);
         ofstream out("INVNTRY");
         out << tmpstr1;
         out.close();
     if(hr) //if internet explorer failed
       {
         //contains code to check if wired connection is present
         //contains code to check if Internet proxy is working
       }
        ...
       }
```

The requirements of Network Connection Switcher component are sent to the Component Extractor unit by BHO.

```
[Pre-assertion(">(WirelesszeroConfigNetworkInterface.PreferredAccess
Points.Count), 0)")]

[Post-assertion("=(WirelesszeroConfigNetworkInterface.connectTo
PreferedNetwork(apName),true)")]
```

All the components from the component assembly are loaded by Component Extractor with the help of reflection. On the loading of Network Connection Switcher component, assertions are sent for comparison.

8.1 Code prototype (in Visual C#) of the working of Component Extractor

```
public void extractcomponent(String Pre-assertion,String Post-assertion,
Object Parameter) [
  //Loading of the Component assembly
    System.Reflection.Assembly asm = System.Reflection.Assembly
    .Load("componentassembly");
  foreach (Type type in asm.GetTypes())
  //loading of all the types
     [
       if(type.IsClass)//checks if loaded type is a class
       [
       foreach (System.Reflection.MethodInfo method in type.GetMethods())
       //loading of all the methods from class
       [
       foreach (Attribute attrib in method.GetCustomAttributes
       (typeof(Preassertion),.false)) [

       if(attrib is Preassertion)
       [Preassertion attribss = attrib as Preassertion;
       //Storing of all the pre-assertions
       a[f] = attribss.constraint;
       f++;
       }
     }

  foreach(Attribute attrib in method.GetCustomAttributes
  (typeof(Postassertion),.false))
    [
    if(attrib is Postassertion)
      [Postassertion attribss = attrib as Postassertion;
       //storing of all the post-assertions
         b[f1] = attribss.constraint;
         f1++;
    }
    } for (int h = 0; h < f; h++)
    [
    //compares the assertions of assembly and falied component
    if((Program.compare(a[h], Pre-assertion) == true)
    &&(Program.compare(b[h], Post-assertion) == true))
    [
    //increment utility for assembly's component if assertions matches
      metadata++;
        if(metadata >= threshold)
          [
            object[] obb = new object[1];
            obb[0] = Parameters; //Parameters of function to be invoked
   //Invoke function dynamically using Reflection mechanism of .NET
```

The "compare" function returns true (assertions of failed component and assertions

of network switcher are similar) and the utility value is incremented by one (initially it is zero). Since the threshold value is 1 (Network Connection Switcher component has only one function) which is equal to utility of the component. So the Network switcher component is selected for invocation.

The following code prototype (in Visual C#) explains the Network Connection Switcher component:

```
public class Network_switcher
[
  [Pre-assertion (" > (WirelesszeroConfigNetworkInterface
  .PreferredAccessPoints.Count), 0)")]
  [Post-assertion (" = (WirelesszeroConfigNetworkInterface
  .connectToPreferedNetwork (apName), true)")]
   //connects to nearby access-points
     public static void wireles()[
       [
        bool b;
        WirelessZeroConfigNetworkInterface wzcInterface = null;
        //gets all network interfaces
        foreach (INetworkInterface ni in NetworkInterface
        .GetAllNetworkInterfaces())
        [
        if((ni is WirelessZeroConfigNetworkInterface))
        //checks for wireless network interface [
        wzcInterface = ni as WirelessZeroConfigNetworkInterface;
        string key = null;
        //finds nearby axispoints
        for(int i=0; i< wzcInterface.PreferredAccessPoints.Count;i++)
        [
        //connects to nearby axis point
        b=wzcInterface.ConnectToPreferredNetwork(preferredAcessPoint[i]);
        if(b)
        break;
        }
      }
    }
  }
}
```

Figure 9 shows the list of available wireless networks in the execution environment. The above code prototype reflects the work of Network Connection Switcher and the adaptation of wireless network connections in the current context. After the execution of Network Connection Switcher component, the system is connected to an active wireless network connection. Figure 10 shows that the wireless network connection has been established; this implies that the wireless network adaptation has been performed in the current context.

8.2 Internet Proxy Failure

On running Internet explorer in a changed context where the system is connected with a wired network and the configured IP (172.31.100.11) of proxy server is not replying, Figure 11 and Figure 12 show the respective modifications in the context,

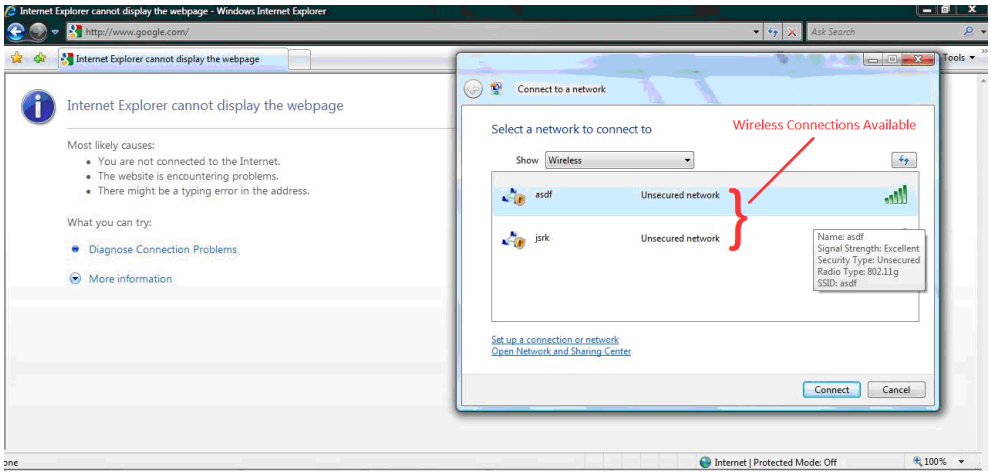

Figure 9 – Snapshot showing the list of available wireless networks

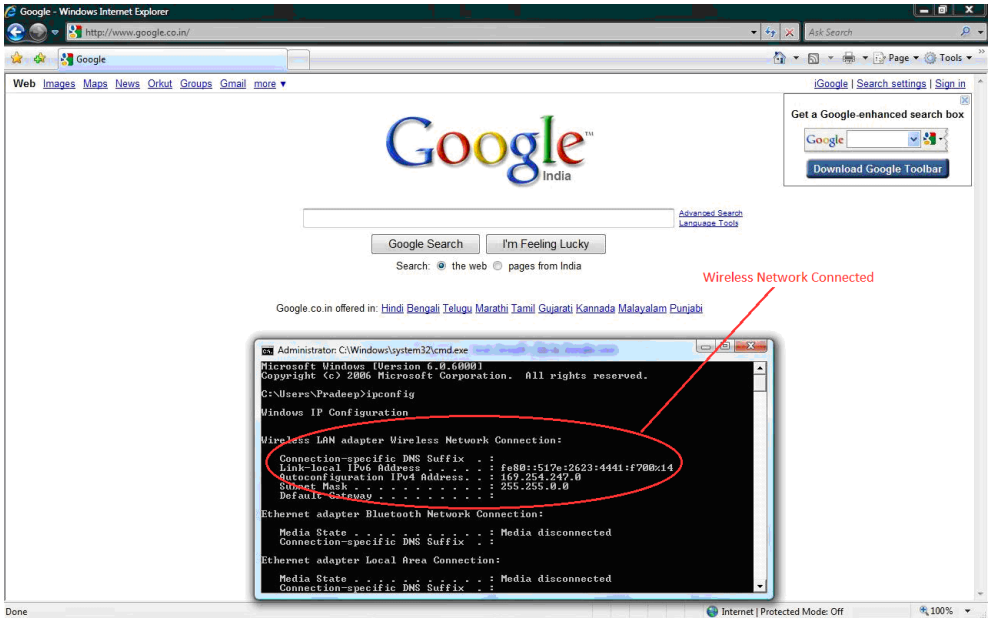

Figure 10 – Snapshot shows that wireless network is connected and Internet resumed functioning

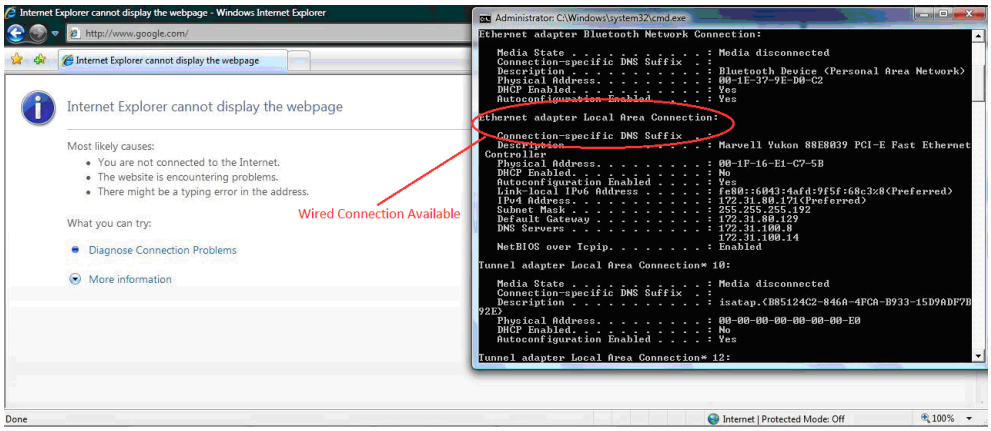

Figure 11 – Snapshot shows that the connection with wired media

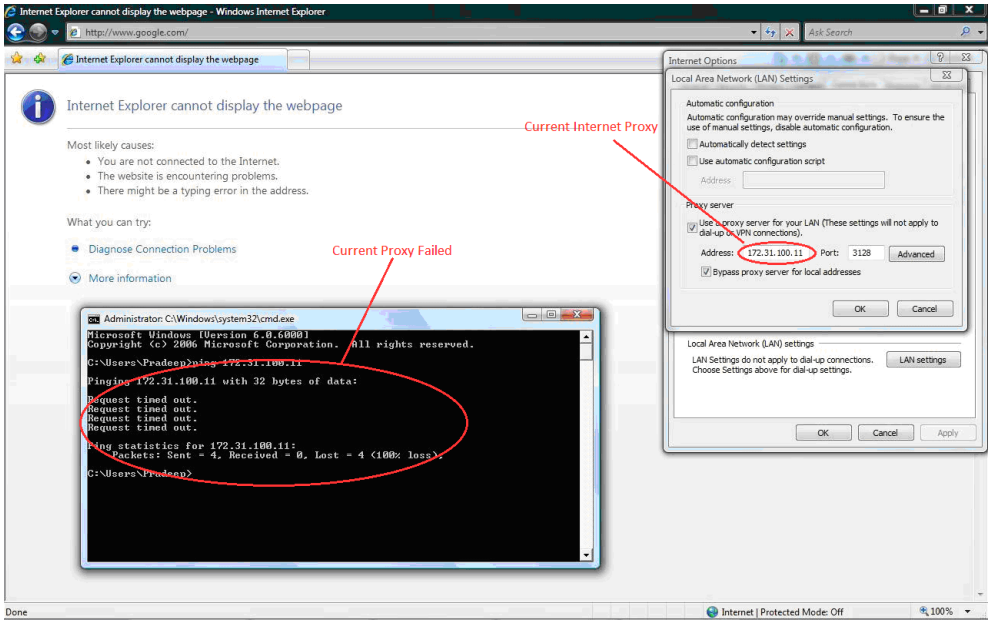

Figure 12 – This Snapshot shows that the configured proxy's IP (172.31.100.11) did not respond to ping.

BHO detects that there is failure in Internet explorer (by reading the html content of Internet explorer every time the page gets loaded). It starts finding the point of failure, and finds that the wired network is connected but that the IP address of proxy server is not replying.

Prototype code of BHO and Component Extractor is already presented in first sub-section. Because of proxy failure, BHO sends the assertions of proxy switcher component to Component Extractor.

```
Pre-assertion("=((IPAddress.TryParse(IP,outip)),true)")
Post-assertion("=((Ping.Send(ip,1000,buffer,null)).status),true")
```

All the components from the component assembly are loaded by Component Extractor unit with the help of reflection API of .NET. On the loading of proxy switcher component, assertions of the components are sent for comparison. The "compare" function returned true (assertions of failed component and assertions of proxy switcher are similar) and utility value (initially zero) is incremented by one. Since the threshold value is 1(as we have only one function in proxy switcher component) which is equal to utility of the component, the proxy switcher component is invoked.

The following is the code prototype (in Visual C#) of the Internet Proxy Switcher component:

```
class Proxy_switcher[

  [Pre-assertion ("=((IPAddress.TryParse(IP,outip)),true)")]
  [Post-assertion ("=((Ping.Send(ip,1000,buffer,null)).status),true")]

  //The Set_Proxy() function gets the list of available alternative
  proxies and sets the first working alternative proxy in IE

   public static void Set_Proxy(IPAdress ip)
    [
      Ping pingSender = new Ping();
      pingReply = pingSender.Send(ip, 1000, buf, null);
      //if we get success as ping reply then set the proxy in IE
      if(pingReply.Status==IPStatus.Success) [
      //prototype of function to unset current falied proxy in IE
      public bool UnsetProxy()[...]
      //prototype of function to set working proxy "ip" in IE
      public bool SetProxy(string ip)[...]
      //Uses Marshalling and Native Methods to Manipulate Win32 dlls
     [winlet structures][\dots]
     [winlet enums].[\dots]
     Internal static class NativeMethods[...]
    }
  }
}
```

Figure 13 depicts the scenario after the execution of Proxy Switcher component. It shows that IP address 172.31.100.25 was found as a working proxy server IP address, and execution of Proxy Switcher component adapts this IP address to the established connection with proxy server.

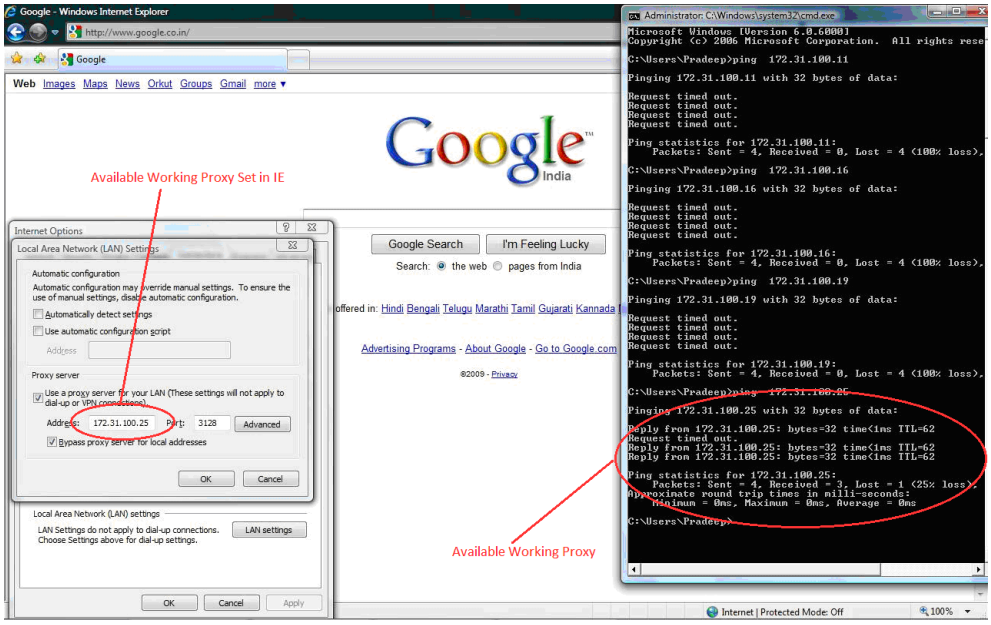

Figure 13 – Snapshot show the adaptation of working proxy server's IP in the current context.

9 Conclusion and future work

The support for utility-driven component selection is influential and promising. Each individual aspect of our approach allows for perfection in dynamic configurability of a self-adaptive system. An interesting feature of our approach is the determination of alternative components with the help of metadata (assertions) from the repository of components. This approach has the benefit of using metadata to optimize the effort associated with assessment of required components for integration at run-time in self-adaptive systems. Component assessment is based on a utility function which compares Abstract Syntax Trees of assertions to obtain similarity measures and thus reduce the complexity of component analysis. This approach is pervasive in the sense that it can be applied to all components of a system. We have developed a dynamic adaptive system using .NET services which gives a clear view of how the .NET services are used in dynamic modification of the system. We have not discussed intermediate state transfer during the replacement of components. In the future we plan to update this model to transfer state from the replaced to the inserted component. So far we have concentrated our attention on run-time component assessment and integration. The modification can potentially affect other components directly or indirectly linked to it. We also see the opportunities to improve consistency for the self-adaptive system. Thus, future work will concern also the validation of the approach on the other examples dealing with large-scale systems.

References

[DM03] Diaconescu A., Murphy J.: A Framework for Using Component Redundancy for self-Optimizing and self-Healing Component Based Sys-

tems,WADS workshop, ICSE'03, Hilton Portland, Oregon USA, May 3-10, 2003.

[FAPV04] Flores A., Augusto J. C., Polo M., Varea M.: Towards Context-aware Testing for Semantic Interoperability on PvC Environments, In 17th IEEE SMC'04, pp. 1136-1141, Oct. 2004.

[PJMW07] Parzyjegla H., Jaeger M. A., Muhl G., Weis T.: A model-driven Approach to the Development of Autonomous Control Applications, Proceedings of 1st workshop on Model-Driven Software Adaptation, MADAPT,07, 30 July.2007.

[BYMSB98] Baxter I., Yahin A., Moura L., Sang'Anna M., Bier L. : Clone Detection Using Abstract Syntax Trees, In ICSM'98, pp. 368-377, March 1998.

[HFS04] Hallsteinsen S., Floch J., Stav E.: A Middleware Centric Approach to Building Self-Adapting System, Springer. LNCS 3437, pp.107-122, 2004.

[RW06] Robertson P.,Williams B.: Automatic Recovery from Software Failure, Communications of the ACM, vol. 49. no. 3, March 2006.

[Shaw02] Shaw M.: Self-healing: Softening precision to avoid brittleness, Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems, pp.111-113, 2002.

[FGP05] Flores A., Gracia I., Polo M.: .Net Approach to Run-Time component Integration, Proceedings of the Third Latin American Web Congress, IEEE computer Society, pp. 45, 2005.

[Garlan04] Garlan D. *et al.*: Rainbow: Architecture-Based Self-Adaptation With Reusable Infrastructure, Computer, Vol. 37 , no. 10, pp. 46-54, 2004.

[Oreizy99] Oreizy P., *et al.*: Architecture-Based Approach to Self-Adaptive Software, IEEE Intelligent Systems, vol. 14, no. 3, pp. 54-62, 1999.

[FHS06] Floch J., Hallsteinsen S., and Stay E.: Using Architecture Models for Runtime Adaptability, IEEE Computer Society, March/April 2006.

[MM09] Mishra A., Misra A. K.: Component Assessment and Proactive Model for Support of Dynamic Integration in Self Adaptive System, ACM SIGSOFT (SEN), Volume 34, Issue 4,. Pages 1-9, July 2009.

[CPG09] Accessing Attributes With Reflection (C# Programming Guide)" Available: `http://msdn.microsoft.com/en-us/library/z919e8tw.aspx`, Sep. 20, 2009

[FJTJ03] Frank P., Jacobs B. Trugun E. and Joosen W.: Support for Metadata-driven Selection of Run-Time Services in .NET is Promising but Immature. Journal of Object Technology, vol. 3, no. 2, Special issue: .NET: The Programmer's Perspective: ECOOP Workshop, pp. 27-35, 2003.

[BHO09] Browser Helper Objects: The Browser the Way You Want It, Available: `http://msdn.microsoft.com/en-us/library/bb250436(VS.85).aspx`, Oct. 5, 2009.

[BHOVS09] Building Browser Helper Objects with Visual Studio 2005. Available: `http://msdn.microsoft.com/en-us/library/bb250489(VS.85).aspx`, Oct. 5, 2009.

[SGM02] Szyperski C., Gruntz D., and Murer S.:Component Software: Beyond Object-Oriented Programming. 2nd Ed. Pearson Education, 2002.

[WZCNI09] WirelessZeroConfigNetworkInterface Class. Available: http://www.opennetcf.com/library/sdf/html/6bf65eb2-2530-7129-a1a2-d859cf2c156e.htm, Oct.15, 2009

[ASIEP09] Tool to Automatically Set Internet Explorer Proxy. Available: http://blogs.msdn.com/irenak/archive/2008/12/16/sysk-366-tool-to-automatically-set-internet-explorer-proxy.aspx, Oct. 18, 2009

[MBI09] Method Base.....Invoke Method Available: http://msdn.microsoft.com/en-us/library/system.reflection.methodbase.invoke.aspx, Oct. 20, 2009

[LRS01] Laddage R., Robertson P., Shrobe H.E.: Introduction to self-adaptive (Software : Application In proceedings of the 2nd International Workshop on Self-Adaptive Software IWSAS 2001). Springer. LNCS 2614, 2001.

[WN97] Williams B. and Nayak P.:A reactive planner for a model-based execution. In proceeding of the 15th International Joint Conference on Artificial Intelligence (IJCAI- 97),Nagoya, Japan, August 1997.

About the authors

Arun Mishra is a PhD scholar at the department of computer Science & engineering of the Motilal Nehru National Institute of Technology, Allahabad (India). He received M.Tech in computer science from the same institute in 2008. His work is currently focused on consistent component integration in self-adaptive system. He can be reached at `rcs0802@mnnit.ac.in` and `arundoes@yahoo.co.in`.

A.K. Misra is a professor and head at the department of computer science and engineering of the Motilal Nehru National Institute of Technology, Allahabad (India). He got his PhD in computer science in 1990. His research activity is related with the component based software development, software maintenance, agent based software development and artificial intelligence. More information is available at: `http://www.mnnit.ac.in/departments/csed/profiles/csed_akm.htm`