A Non-Invasive Approach to Assertive and Autonomous Dynamic Component Composition in Service-Oriented Paradigm

Fei Cao
(University of Alabama at Birmingham, Birmingham, AL, USA
caof@cis.uab.edu)

Barrett R. Bryant
(University of Alabama at Birmingham, Birmingham, AL, USA
bryant@cis.uab.edu)

Rajeev R. Raje
(Indiana University Purdue University, Indianapolis, IN, USA
rraje@cs.iupui.edu)

Andrew M. Olson
(Indiana University Purdue University, Indianapolis, IN, USA
aolson@cs.iupui.edu)

Mikhail Auguston
(Naval Postgraduate School, Monterey, CA, USA
auguston@cs.nps.navy.mil)

Wei Zhao
(University of Alabama at Birmingham, Birmingham, AL, USA
zhaow@cis.uab.edu)

Carol C. Burt
(University of Alabama at Birmingham, Birmingham, AL, USA
cburt@cis.uab.edu)

Abstract: Component-based software composition offers a development paradigm with reduced time-to-market and cost while achieving enhanced productivity, quality and maintainability. Existent work on the composition paradigm are of a static composition paradigm, which is not sufficient in distributed environment, in which both constituent components and assembled distributed system are subject to dynamic adaptation. This paper presents two types of dynamic composition for distributed components: assertive and autonomous over .NET based Web Services environment. Two case studies are provided: the first one illustrates at a low level how the underlying infrastructure enables the dynamic composition; the second one illustrates at a high level how dynamic compositions are specified.

Keywords: dynamic component composition, Service Oriented Architecture, Web Services, assertive composition, autonomous composition, intermediate code manipulation, aspect-oriented programming, aspect weaving, .NET, Common Language Runtime
Categories: D.2.3, D.2.12, D.2.13, D.2.7, D.3.3, H.3.5, I.2.8

1 Introduction

With the increasing demand for scalability, reasonability and correctness of software systems, software development has evolved into a process of composing existing software components, as opposed to constructing a new software system completely from scratch [Heineman, 01]. Economically, by reducing time-to-market, this approach has improved the economic and productivity factors of software production [Devanbu, 96]; Technically, by separating overall functionality into small units, component-based software development also offers a means for better manageability [Brown, 00] and predictability [Hissam, 03] of the constructed software system.

Features of Distributed Components

With the advancement of internet technology, component-based software development has unleashed its impact into the distributed environment, while exhibiting such new features as follows:

a. The scope of component selection and reuse is extended. Consequently, component composition requires a prerequisite discovery process for identifying a matching component.
b. Distributed components are usually heterogeneous with respect to implementation languages, and host platforms. With different type systems or component models, interoperation between components will not be possible without leveraging proper bridging technology.
c. Because of the unpredictability of network transport, not only functional properties, but also non-functional properties (e.g., Quality of Service [Raje, 02], economical properties such as pricing of service) are of critical concern to guarantee the proper delivery of services offered by the assembled distributed software systems. QoS includes availability, throughput, and access control, to name a few.
d. The coupling between components is loose. A deployed component in a distributed system is subject to frequent adaptation[1] or replacement with a new version to accommodate ever-changing business requirements externally as well as the computing resource status internally. Those requirements can be either functional or non-functional.

WS as a New Paradigm for Distributed Component Composition

Those new features pose new problems for developing software systems based on distributed components. Recent years have seen the emergence of Web Services (WS) technology as a new component-based software development paradigm in a network-centric environment based on the Service Oriented Architecture (SOA) [Colan, 04], the open standard description language XML and transportation protocol HTTP. Consequently, distributed component composition can be achieved by wrapping heterogeneous components with a WS layer for interoperation. Using WS as a common communication vehicle, component interoperation is greatly simplified

[1] Here adaptation is defined as component compositon and decomposition; component composition and decomposition are the means to realize adaptation.

compared with such bridging technology as CORBA[2], where different interoperation implementations are needed for each pair of components contingent on their underlying implementation technologies. In the remaining part of this paper, the term component in a distributed environment is equivalent to a WS: we use it to correlate the canonical concept of a software component [Szyperski, 02].

The Need of a Dynamic Component Composition Paradigm in WS

In addition to offering an interoperability infrastructure for distributed components, WS also incorporates service discovery infrastructure in accordance with SOA. With problem (a) and (b) being embraced, current WS technology is yet to address the concerns as set forth in (c) and (d). Specifically,

1). in mission critical scenarios such as finance or military, there is a need for *guarantee of service availability* continuously, rather than shutting down the system for services adaptation;
2). in distributed environments, service consumption experiences are dynamic and desirable to be seamless, thus the customizability of service dynamically is of vital importance in a service-oriented environment.

As such, static component composition is not adequate, and both functional and non-functional property adaptation need to be applied in a dynamic fashion. Along this line, this paper presents a dynamic component composition paradigm in WS environment for adapting WS functionally and non-functionally while maintaining the availability of WS.

This paper presents a dynamic component composition paradigm based on the .NET Common Language Runtime (CLR) [Gough, 02]. We chose .NET because it is a thorough, fundamental re-architecting of a distributed computing platform based on WS, while other application server support for Web Services tend to be designed more as another client, or presentation tier for the back-end systems, with communication tier based on RMI or RMI/IIOP rather than a strictly XML protocol based such as .NET [Newcomer, 02].

This paper is organized as follows: Section 2 describes background information. Section 3 provides an overview of the approach, as well as salient features. Section 4 describes design and implementation. Section 5 provides two case studies. Section 6 provides the benchmarking for the approach. Section 7 describes related work. We conclude in Section 8 together with the description of future work.

2 Background

The .NET framework is a platform for software integration, with Common Language Runtime (CLR) for integrating software at the single operating system process scale, and with XML WS for integration at the internet scale. CLR is the .NET equivalent to the Java virtual machine, but offers more features such as using Common Intermediate Language (CIL) based on the Common Type System (CTS) [Gough, 02] to translate .NET languages before execution, thereby offering cross-language interoperability for .NET languages based on CIL. The code to be translated into CIL

[2] CORBA® - Common Object Request Broker Architecture: http://www.omg.org/corba/

and then to be executed by CLR is also called *managed code*. Also CIL includes rich metadata information for describing software module contracts to achieve *managed execution*, with the benefits of security and scalability.

The scope of this paper is not to at provide a full description of .NET CLR and XML WS, but rather to present our approach on capturing WS at the CLR level, then applying in-memory CIL code manipulation at runtime to realize dynamic component composition.

3 Overview of the Approach

3.1 Runtime Code Manipulation Through Assertive and Autonomous Composition Rules

Figure 1 provides an overview of the dynamic composition approach. In the left pane of the *execution unit*, the .NET XML WS, which is specified with Web Service Description Language (WSDL), is a layer built on top of .NET applications (1), which in turn runs over CLR (2). Consequently, .NET based XML WS can leverage the benefits of managed execution, where the .NET application is captured in the form of CIL (2), which is to be Just-In-Time (JIT) compiled into native code and executed (3). Therefore, by manipulating CIL derived from the XML WS implementation language, WS components can be composed at runtime.

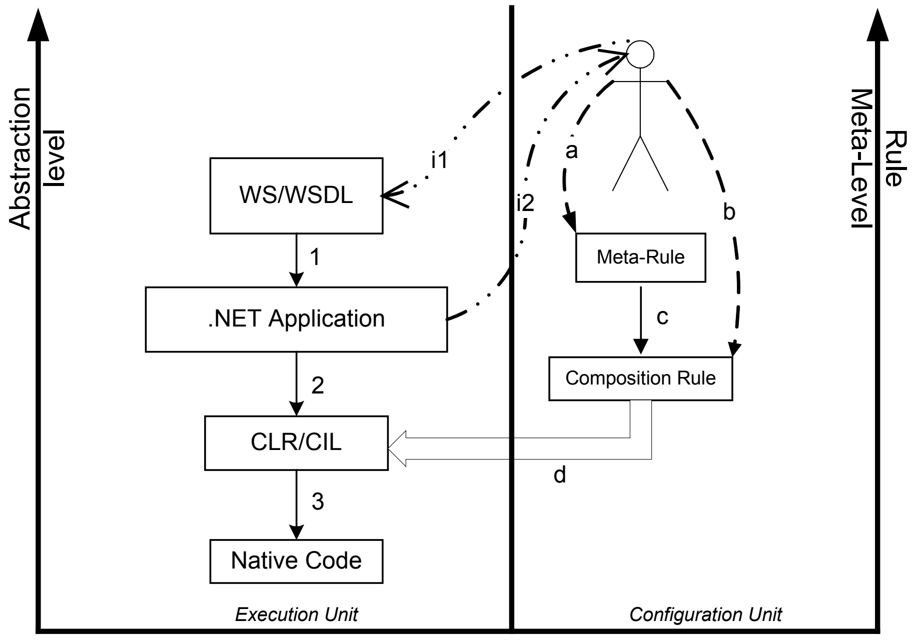


Figure 1: Overview of the dynamic composition approach

The manipulation of CIL is illustrated in the right pane of the *configuration unit*, which is comprised of a stack of composition rules with a meta-level hierarchy. Composition rules are specifications for component composition (d). Meta-rules are specifications of triggering conditions for applying the composition rules, and the firing of the composition rules is enabled through a rule execution engine automatically (c). The use of rule engine for applying composition rules is useful for implementing *autonomous compositions* based on the runtime status quo. The actor icon represents a configuration console in a manual manner for both meta-rules (a) and composition rules (b). While the composition enabled through path (a->c->d) represents autonomous composition, the composition path of (b->d) represents the *assertive composition*. The configuration decision is based on WSDL exposed by WS (i1); WS itself can in turn assume the configuration role for specifying component composition reactively (i2).

3.2 Salient Features

The dynamic component composition approach also includes the following salient features:

1). Non-invasive

▪ *Non-invasive to application code for separation of composition concerns* The WS composition is realized through in-memory IL manipulation as opposed to off-line invasive code change. The non-invasive change is often desirable as a WS vendor may deliver the software package in binary form. Also even though it is possible to derive CIL from a .NET executable using some de-compilation tools, invasively changing either original source code or derived CIL code will require unloading, recompiling and redeployment of the original WS application, which compromises the availability of WS. Moreover, the invasive change of WS code will pollute the original application such that recovering it will become difficult, which introduces the common version control problems for software systems.

▪ *Non-invasive to platform for portability*. The composition through manipulation of CIL at runtime (Figure 1-d) requires the interception of the managed execution. Instead of re-implementing the CLR such as rewriting open source CLR Rotor [Stutz, 03] to invasively add a listener for execution interception at the compromise of portability of CLR, we use a pluggable, configurable CLR profiling interface to achieve this goal, which can be enabled and disabled based on composition needs with ease to reduce unnecessary overhead.

2). Language neutral for cross-language component composition

By specifying composition rules based on WSDL, which in turn is based on alanguage neutral XML schema[3], and code manipulation at the intermediate code (CIL) level based on language neutral CTS, WS components implemented in different .NET languages can be composed across language boundaries.

3). Adaptable composition.

[3] http://www.w3c.org/2001/XMLSchema

With the configuration unit as a separate entity applied to runtime as shown in Figure 1, not only is the composition concern separated, but also it can be updated to realize adaptable composition at runtime.

The following section presents in detail the design and implementation of the dynamic component composition in Peer-to-Peer (P2P) scenarios, particularly, how the composition rules are specified to facilitate assertive and autonomous configuration.

4. The Design and Implementation of Dynamic Component Composition

4.1 Peer-to-Peer (P2P) Component Composition

Figure 2 illustrates the architecture for the dynamic component composition based on .NET WS environment. In our work, each component is hosted in an infrastructure *DynaCom*, which is essentially a profiler-enabled CLR to be detailed in Section 4.2. DynaCom is used as a proxy to for components to interoperate with components in other locations through WS. Meanwhile, DynaCom can intercept the execution of the hosting components and change the behaviour of the executing components dynamically. DynaCom is based on our prior work on using a profiling approach for dynamic service provisioning [Cao-a, 05], but here it is tailored to component composition.

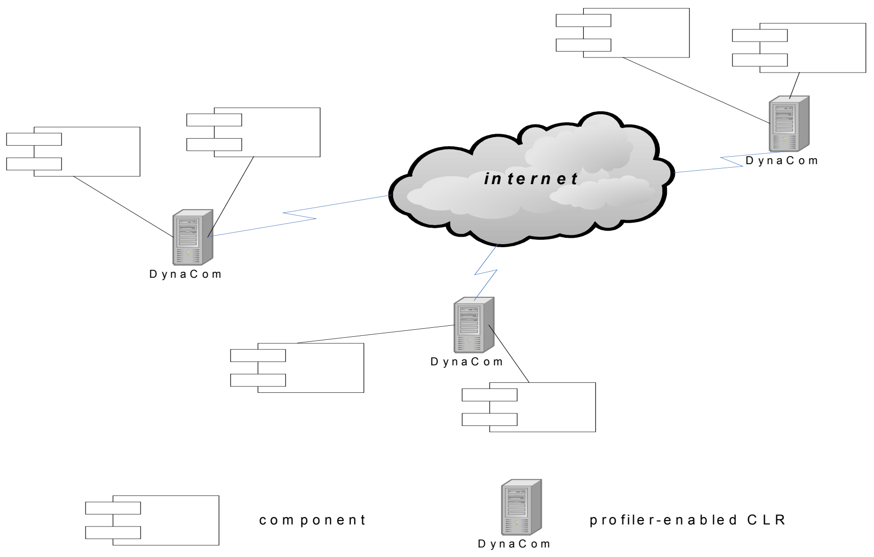


Figure 2: The P2P component compositions in .NET WS environment

The component composition model shown in Figure 2 represents a P2P paradigm, which is the primary composition model to be addressed in this paper. This choice is based on the observations that P2P and dynamic composition are tightly associated:

1). *P2P as an agile mode to accommodate dynamic features.* While WS orchestration by executing BPEL4WS[4] in the execution engine represents a centralized composition model, it has been observed that such a composition model compromises scalability, availability, and security for the server [Chen, 01]. With the highly dynamic features in distributed environment, P2P component composition paradigm will be more widely used.

2). *Dynamic composition is the necessary means for realizing P2P computation in a distributed environment.* While component composition usually requires the generation of glue/wrapper code [Cao, 02], the physical location for hosting the generated glue/wrapper code is a hard problem in P2P mode without central management and storage units. Dynamic composition, with glue/wrapper code generated in memory and JIT compiled and executed at runtime, provides a solution for P2P component composition without the physical code placement issues.

4.2 DynaCom Exposed

Figure 3 provides an anatomy of DynaCom. The part enclosed by the big square represents the enabling mechanism for dynamic composition, which is transparent to the components to be composed above the big square.

Our work is built upon the ASP.NET[5], a WS implementation package based on the .NET framework. In ASP.NET, Internet Information Service (IIS)[6] is used to accept the incoming WS SOAP (Simple Object Access Protocol) [Newcomer, 02] message transported over HTTP (1). Upon acceptance of the WS request encoded as a SOAP message, an IIS filter will launch a work process (aspnet_wp.exe), which in turn will launch CLR (2) to run the WS application in the mode of managed execution. At this point, the WS application is rendered as in CIL subject to be JITcompiled into native code and executed (6). In order to adapt WS, it is needed to intercept the WS call at the CIL level before it is compiled. While it is reasonable to implement the expected functionalities in the CLR open source of millions of lines of code such as Rotor [Stutz, 03], we feel it too expensive an effort. Instead, we use the CLR profiling API to implement a *Profiler* as event handlers, and register them as listeners for the events generated from the CLR (3). In contrast to the conventional publisher/listener model, which is often of a client-server relationship, the profiler here will be mapped into the same address space for the profiled application as an in-process server.

[4] BPEL4WS - Business Process Execution Language for Web Services - http://www-128.ibm.com/developerworks /library /specification/ws-bpel

[5] http://asp.net

[6] http://www.microsoft.com/WindowsServer2003/iis/default.mspx

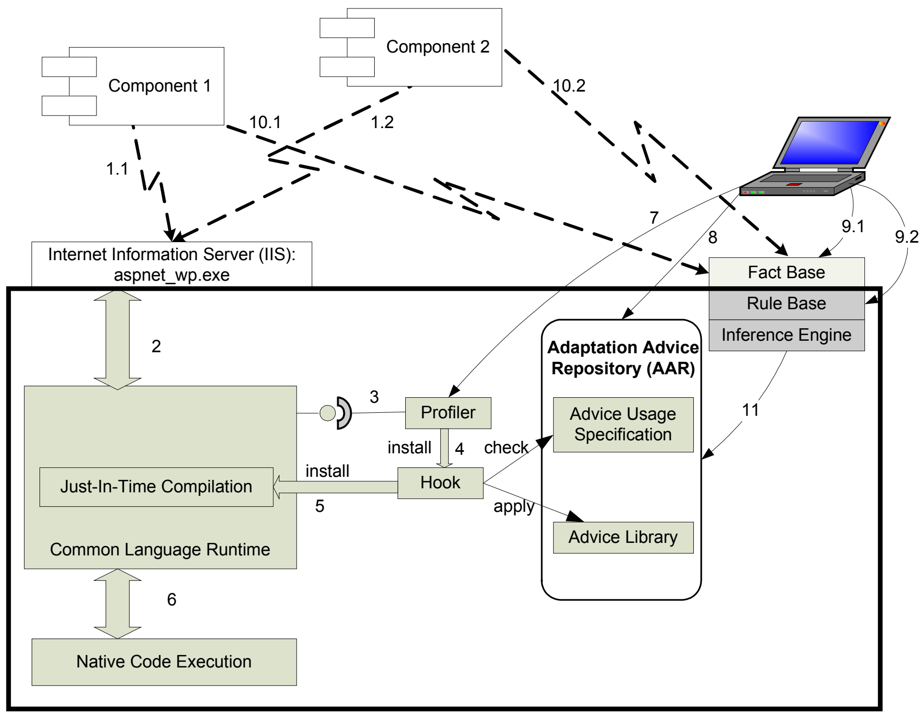


Figure 3: The Architecture of ***DynaCom: Dyna****mic* ***Com****ponent composition enabling unit, which includes the part enclosed by a bold-border rectangular and the IIS, facts. The parts of IIS and facts are accessible to the remote components, while the enclosed part of DynaCom are only accessible locally. The dashed line of 1 and 10 represents remote access, while all the remaining solid lines represent local access. The laptop icon represents the local configuration unit to DynaCom.*

The events generated from the CLR are the result of managed execution, including but not limited to garbage collection, class loading/unloading, CLR startup/shutdown and JIT compilation. The event of our interest is JIT compilation, for which we implement in-memory CIL manipulation for the event handler. The adapted CIL will then be JIT compiled and executed resulting in changed WS behavior. A one-shot change to CIL will reduce the traceability of adaptation, impede the removal of the imposed adaptation (thus incapable of dynamic decomposition), and restrict the flexibility of further adaptation. Therefore, we interpose a *Hook* code (4,5) in the WS application to be adapted, which will check the *Adaptation Advice Repository* (AAR) for applicable adaptation advice. The term “advice” is further explained in the next section. AAR is located in a shared memory for fast access during in-memory CIL manipulation. The AAR includes an Advice Library storing predefined reusable advice in the compiled managed code form, as well as an *Aspect Usage Specification* (AUS) component to indicate applicable advice for WS. The Profiler and the AAR are subject to external configuration (7-11): for 7, the configuration is used to narrow down the scope of profiling; for 8-11, the configuration is used to dynamically specify adaptation rules, among which 8 corresponds to a direct manipulation of adaptation rules, while 9-11 corresponds to indirect manipulation of adaptation rules through a rule inference engine. The inference engine can dynamically inject AUS into AAR based on the rule specification, which is to be detailed in Section 5.2. The laptop icon in the upper-right corner represents the local configuration unit. The configuration unit for DynaCom can adopt a GUI interface or an API interface. In our work, we use a simple console for the local configuration unit handling configuration 7-9, while configuration 10-11 is realized through an API interface.

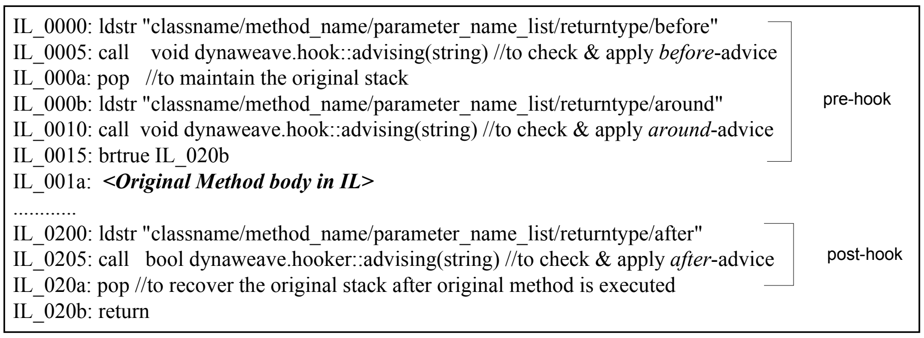
4.3 Dynamic Component Composition Through Dynamic Aspect Weaving

4.3.1 Modularized Component Composition

In Figure 2, each DynaCom only hosts 2 components, which is for simplicity purpose in illustration. In reality, a DynaCom may be hosting multiple components. Consequently, a component handling a crosscutting concern may be expected to be composed with multiple other components. Thereafter, it is not possible to specify adaptation for every individual component upon changing of requirements. Instead, there needs to be a means to abstract the adaptation in a modularized way. Aspect-Oriented Programming (AOP) [Kiczales, 97] offers a means to abstract cross-cutting concerns in a modularized way called an *aspect*, and the concerns can be weaved using weaver technology into the base program based on the *join point model*, which specifies the destination to weave concerns. In the same vein, we specify the adaptation advice in the AAR in a modularized way following AOP style[7]. To weave and unweave the specified advice, we instrument the hooks at both the entry (*pre-hook*) and exit point (*post-hook*) of the WS method to be adapted, which are used to check into the AAR to see if corresponding *before advice* and *after advice* is applicable: the former performing some pre-processing before the actual WS method execution, while the latter performs some post-processing immediately before the WS method execution returns. Such pre- and post- processing capacity can be used to instrument codes for addressing non-functional concerns, such as applying access control upon the entry into the WS method, or applying state persistency service for the executed WS application upon the end of the WS call. Also included in the pre-hook are the instructions to check if an *around advice* is specified or not, and a jump instruction to redirect the execution to the exit point of the WS application. The jump instruction is to be activated if an around advice is found valid in the AAR. With around advice, the original WS will be replaced with new behaviour specified in that around advice. Consequently, not only the original WS can be decorated, it can also be overridden completely, which is necessary when a buggy WS is identified and

[7] AOP also offers a means for separating composition specification from components to be composed, with the underlying weaver realize the composition. As such, in case the components to be composed do not involve crosscutting concerns, the component composition is still specified in the same way as an aspect weaving specification with AUS.

needs to be removed, or an old service module need to be updated. The around advice sufficiently offers a delegation and wrapping approach for component composition which is exemplified in Section 5. By using a hook for weaving, advice can be applied dynamically and proactively. Meanwhile, unweaving advice can be realized by dis-activation of the corresponding AUS in AAR. Figure 4 is the CIL manipulation template for adapting a WS method.



```
IL_0000: ldstr "classname/method_name/parameter_name_list/returntype/before"
IL_0005: call    void dynaweave.hook::advising(string) //to check & apply before-advice
IL_000a: pop   //to maintain the original stack
IL_000b: ldstr "classname/method_name/parameter_name_list/returntype/around"
IL_0010: call  void dynaweave.hook::advising(string) //to check & apply around-advice
IL_0015: brtrue IL_020b
IL_001a:  <Original Method body in IL>
............
IL_0200: ldstr "classname/method_name/parameter_name_list/returntype/after"
IL_0205: call   bool dynaweave.hooker::advising(string) //to check & apply after-advice
IL_020a: pop //to recover the original stack after original method is executed
IL_020b: return
```

Figure 4: Instrumentation of IL code of a WS method

4.3.2 Specifying Component Composition via Aspect Usage Specification

The AOP weaving specification in AspectJ [Kiczales, 01] can be adapted for component composition specification in terms of aspect weaving as illustrated in Table 1.

Component Composition		**Aspect Weaving Specification**
Sequential	a precedes b	after (a) {b; }
	a follows b	before (a) {b; }
Wrapping	a is wrapped by b at the beginning and c at the end	around (a) {b; proceed(); c; }
Overiding	a is overriden by b	around (a) {b; }

Table 1: Composition specification in the form of aspect weaving

The aspect weaving specification is represented in AUS. The type system used in the AUS in AAR can be based on the object-oriented CTS of CIL, for which each CLR hosted language is translated to before being JIT compiled. Therefore, such specification is applicable to all WS applications running in CLR, which provides a language-neutral way for AUS. However, writing adaptation AUS based on low level CTS is error-prone and not necessary for high-level AUS. As a result, AUS is written in XML rather than in CTS, which is based on the following observations:

1) **Necessity**

- Components delivered may be in binary form with source code being unavailable, thus AUS at the application code level is not feasible. On the other hand, components in the .NET WS environment are exposed through the WSDL interface, which offers a reference point for specifying WS component adaptation.
- AUS, as the specification reflecting the business requirement adjustment (by composing and decomposing related components), should have an abstraction level close to business requirements, rather than being tied to underlying implementation details.
- XML-based specification for AUS can be directly serialized and queried by hooks using XML manipulation APIs such as DOM or SAX or XQuery[8].

2) Sufficiency

- Web Service Description Language (WSDL) is based on the XML Schema, which is another language neutral type system that can be mapped to the language-neutral CTS. The XML Schema based specification is parsed and translated to CTS to be matched against the string provided by the hook such as described in IL_0000, IL_000b, IL_0200 in Figure 4. The AUS in AAR accords with XML schema as illustrated in Figure 5.

```
<wsdl:operation name="apply_advice">
  <wsdl:input message="tns:advicetype"/>
  <wsdl:input message="tns:return_type"/>
  <wsdl:input message="tns:classname"/>
  <wsdl:input message="tns:methodname"/>
  <wsdl:input message="tns:parameter_list"/>
  <wsdl:input message="tns:advicename"/>
</wsdl:operation>
```

Figure 5: The AUS schema

Associated with each *advicename* is the path information for actual advice in the form of managed code stored in the AAR. All the advice code is defined as a template with the tuple <*Classname, Methodname, Parameter_List*> as parameters, which offers reusability of advice. Such advice can be pre-built in any .NET language and compiled into managed code. If a matching advice is found, then the advice code will be loaded from the corresponding path and called. In our work, the wild-card characters are also supported for AUS.

[8] http://www.w3c.org

4.3.3 Autonomous Component Composition Using Rule Inference Engine

Functionality for the composed distributed software systems can be predicted based on the constituent components [Hissam, 03], thus a component composition based on functional requirements can be specified assertively. In contrast, non-functional properties such as pricing based on end-to-end delay (service consumption duration) for composed distributed software systems can only be reasoned about at runtime because of their dynamic characteristics. As such, a distributed software system needs to self-adapt itself by composing and decomposing components autonomously to achieve the expected QoS. The self-adaptation decisions can be collectively built into a knowledge base proactively and retroactively. Therefore, the complete dynamic component specification in terms of dynamic, autonomous aspect weaving takes the following rule:

```
  apply [aspect_name] when [logical_condition]
```

The *`when`* clause represents the condition under which the action *`apply [aspect_name]`* need to be performed. Consequently, the AUS schema in Figure 5 will be augmented with an attribute *`when`* for the *`wsdl:operation`* element. The use of rule inference provides a means for not only separation of concerns between business rules and the underlying technical implementations for component compositions, but also autonomous composition at runtime.

In our work, we use Jess [Friedman-Hill, 05] as the underlying inference engine, which is a forward and backward chaining rule engine for the Java platform. Associated with the inference engine are the fact bases and the rule base as shown in Figure 3. The rule base is only accessible to the local hosting site, and represents local autonomous composition policies; comparatively, the fact base is exposed to both the local and remote site, which can be manipulated by either the local configuration unit, local components, or remote components. The fact bases of different DynaCom are federated, and a local rule engine can query remote fact base for triggering an action. This is useful when a local composition rule is dependent on remote component status (which is reflected in the remote fact base). For example, the unavailability of a remote components during a certain period of time will trigger the local component to connect to an alternative component, which offers a means of fault tolerance.

Jess offers a hybrid programming paradigm between the Java language and declarative rule specification: the Java code can invoke the Jess rule engine while the Jess rules invoke Java code. In order for Jess fact base to interoperate with remote components, as well as to enable the Java-based inference engine interoperable with .NET environment, we wrap the Java-based Jess API with a WS layer using Java WSDP[9].

---

[9] Java WSDP – Java Web Services Developer Pack – http://java.sun.com/webservices/jwsdp/index.jsp

5 Case Study

In this section we present two case studies. The first one is an assertive dynamic composition example which is also intended to illustrate how every part shown in Figure 3 works together. The second one showcases a dynamic composition paradigm of autonomous composition.

5.1 Composition Crosscutting Credit Authorization Components

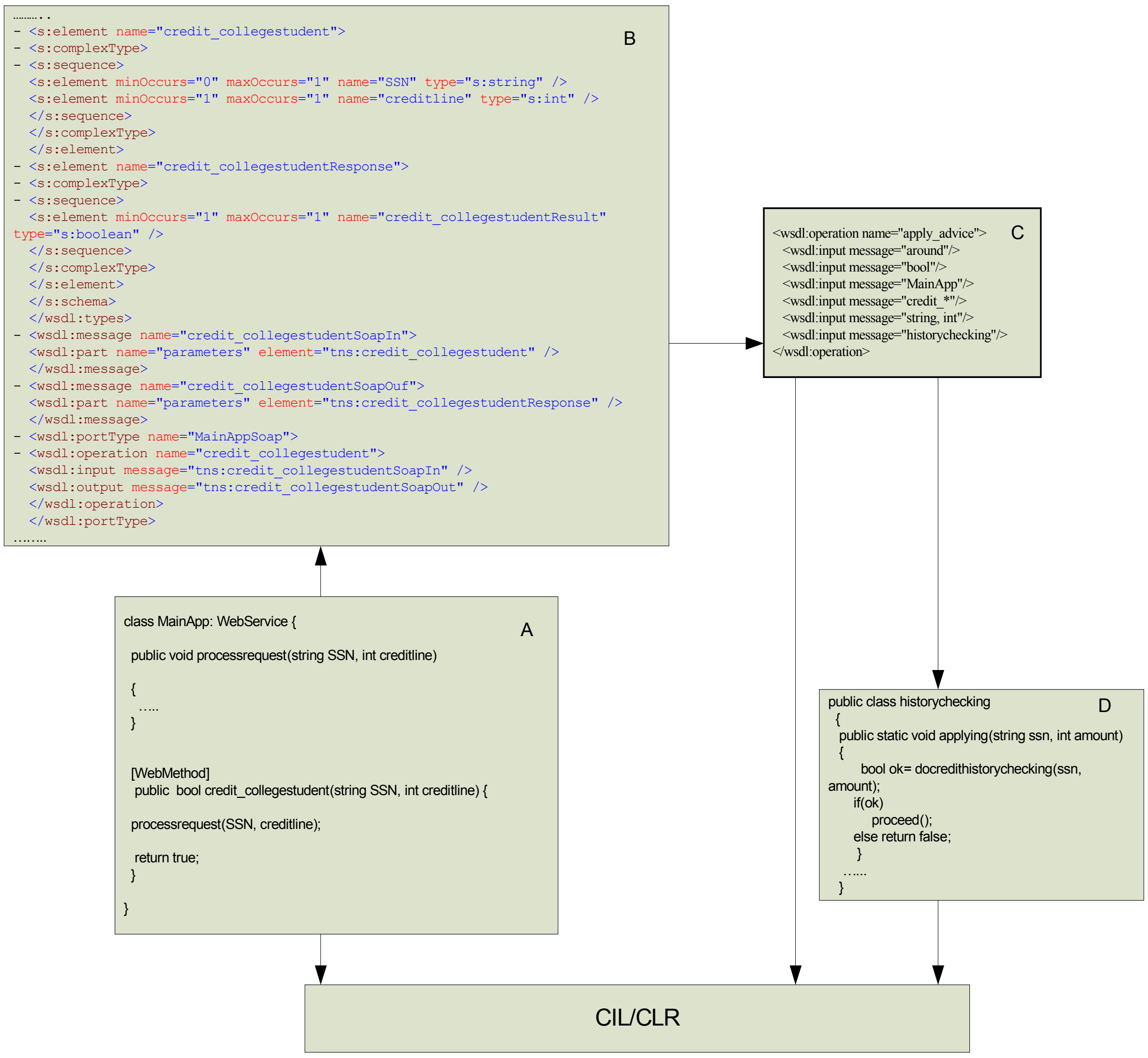
Figure 6 provides an example of a college student credit authorization WS to demonstrate the assertive dynamic component composition for a non-functional concern: access control. Figure 6-A provides a simple WS application written in C#, which provides a WS method for authorizing credit card application based on the Social Security Number (SSN[10]) and the expected credit line. The corresponding WSDL in Figure 6-B can be automatically generated from the source code in Figure 6-A based in ASP.NET, which in turn is to be exported and used as the basis for AUS as well. Figure 6-C is an AUS with an around advice to apply credit history checking before any credit card application request is processed. The AUS represents a sequential composition specification for a component encapsulating crosscutting concerns (here *HistoryChecking*). The wild card specification in credit_* represents all credit application with the request name preceded with "credit_". Figure 6-D is the source code for the pre-built credit history checking advice, which can be written in any .NET language (here C#) and is compiled and persisted in the managed code form. The type systems in Figure 6-A, Figure 6-C, Figure 6-D are translated into CIL and matched up in CLR. Once a match holds, the advice in Figure 6-D will be called by the hook instrumented at runtime. The WS application source code level detail is transparent to AUS in Figure 6-C, as well as to the HistoryChecking component in Figure 6-D. By instrumentation of intermediate code, component composition can be realized across language boundaries without invasively changing application source code.

5.2 Composing Travel Planning Components

The former section demonstrates how each part in DynaCom is integrated together for assertive dynamic component composition, particularly how the intermediate code manipulation enables the component composition across language boundaries without invasively accessing the application source code. This section will further explore the dynamic composition for multiple components for travel planning, which not only includes assertive dynamic composition, but also autonomous dynamic composition using the Jess rule inference engine. Complementing the previous case, this case focus on the user level component composition specification as opposed to dwelling on the low level intermediate code manipulation.

In Figure 7, the boxed part contains the WS components for travel planning, with those above the box representing the types used in the WS components. Each customer plans the travel through a travel agent *Travel_Agent (TA)*. The travel agent

[10] An identification number used to identify income earners in the United States.



```
………..
- <s:element name="credit_collegestudent">
- <s:complexType>
- <s:sequence>
  <s:element minOccurs="0" maxOccurs="1" name="SSN" type="s:string" />
  <s:element minOccurs="1" maxOccurs="1" name="creditline" type="s:int" />
  </s:sequence>
  </s:complexType>
  </s:element>
- <s:element name="credit_collegestudentResponse">
- <s:complexType>
- <s:sequence>
  <s:element minOccurs="1" maxOccurs="1" name="credit_collegestudentResult"
type="s:boolean" />
  </s:sequence>
  </s:complexType>
  </s:element>
  </s:schema>
  </wsdl:types>
- <wsdl:message name="credit_collegestudentSoapIn">
  <wsdl:part name="parameters" element="tns:credit_collegestudent" />
  </wsdl:message>
- <wsdl:message name="credit_collegestudentSoapOuf">
  <wsdl:part name="parameters" element="tns:credit_collegestudentResponse" />
  </wsdl:message>
- <wsdl:portType name="MainAppSoap">
- <wsdl:operation name="credit_collegestudent">
  <wsdl:input message="tns:credit_collegestudentSoapIn" />
  <wsdl:output message="tns:credit_collegestudentSoapOut" />
  </wsdl:operation>
  </wsdl:portType>
……..
```

```
<wsdl:operation name="apply_advice">
  <wsdl:input message="around"/>
  <wsdl:input message="bool"/>
  <wsdl:input message="MainApp"/>
  <wsdl:input message="credit_*"/>
  <wsdl:input message="string, int"/>
  <wsdl:input message="historychecking"/>
</wsdl:operation>
```

```
class MainApp: WebService {

 public void processrequest(string SSN, int creditline)

 {
   …..
 }


 [WebMethod]
  public  bool credit_collegestudent(string SSN, int creditline) {

 processrequest(SSN, creditline);

  return true;
 }

}
```

```
public class historychecking
 {
  public static void applying(string ssn, int amount)
  {
        bool ok= docredithistorychecking(ssn,
amount);
       if(ok)
            proceed();
       else return false;
     }
   ……
 }
```

Figure 6: Composing credit authorization component assertively(A+D)

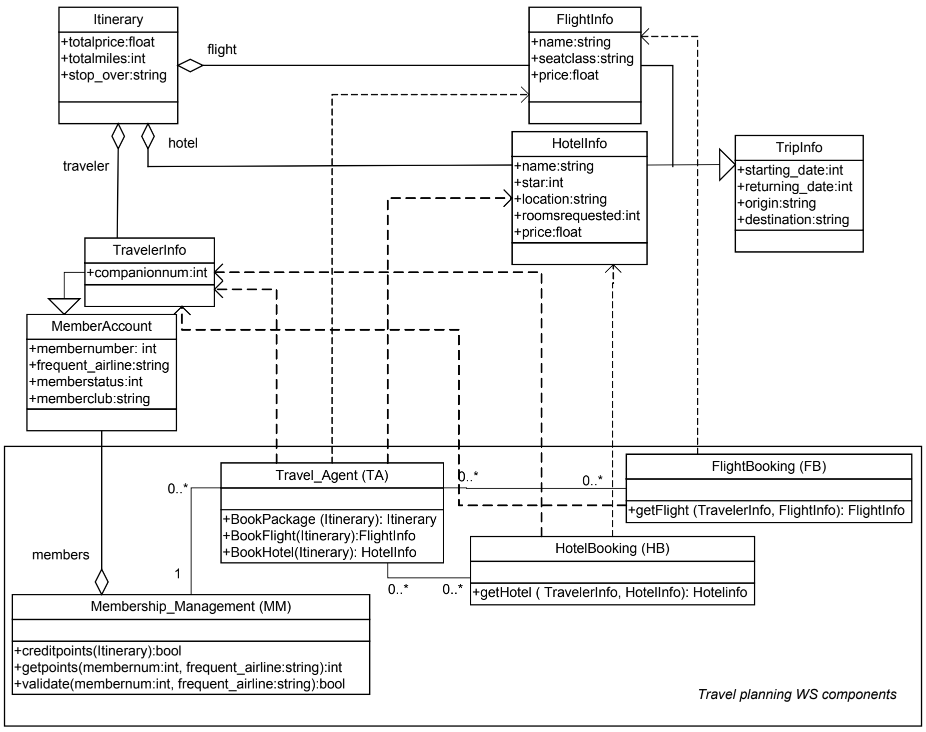


Figure 7: Class diagram for travel planning WS components

will handle both the booking of flight, *FlightBooking (FB)* and hotel, *HotelBooking (HB)*. Every traveler can credit his mileage into his own frequent flier number through *Membership_Management (MM)*. He can book the travel package including both hotel and flight, or just book one of them. He can also book for group travelers. The result of the travel booking process is the itinerary information (*Itinerary*), which includes the total cost of the trip. All those WS components in the box are loosely coupled and dynamically bound based on their partnership, service charge, and QoS. Figure 8 illustrates the travelling components composition process with sequence diagram. The italicized part represents the dynamically composed components; the TA and its associated methods represents the static front end travel agent components to the customers with back end components dynamically composed on demand.

5.2.1 Static Front End

During travel planning, the customer starts from TA WS method *BookPackage*, with the backend components dynamically composed to fulfill the travel planning purpose. The TA serves as front end components to the customers to be dynamically bound to backend WS components, and the *BookTravel* method is implemented as shown below:

```
Itinerary BookPackage (Itinerary it)
 {
     FlightInfo fi;
```

```
    HotelInfo hi;
    fi=BookFlight (it);
    hi=BookHotel(it);
    return combine(it1,it2);
  }
```

5.2.2 Dynamic Backend

While the front end code as shown above is static to the customer side, there are some dynamic component composition concerns in the backend that is transparent to the customers:

- Dynamic partnership

The front end TA component may have dynamic partnership with back end FB and HB (we assume membership management is centralized and statically bound in this case in accordance to the real world examples, where membership such as Social Security Account is centrally administrated by the appropriate government agency) based on their mutual contract, service charge (if the service charge is exceeding the budget, the partnership will be cancelled and a new partner will be identified), or QoS (if the service of the current partner is down, an alternative partner need to be identified). As such, the partnership should be established dynamically, which is also subject to dynamic change consequently. Figure 8 illustrates the dynamic partnership establishment by using two <<*create*>> messages before the call of *BookPackage*,

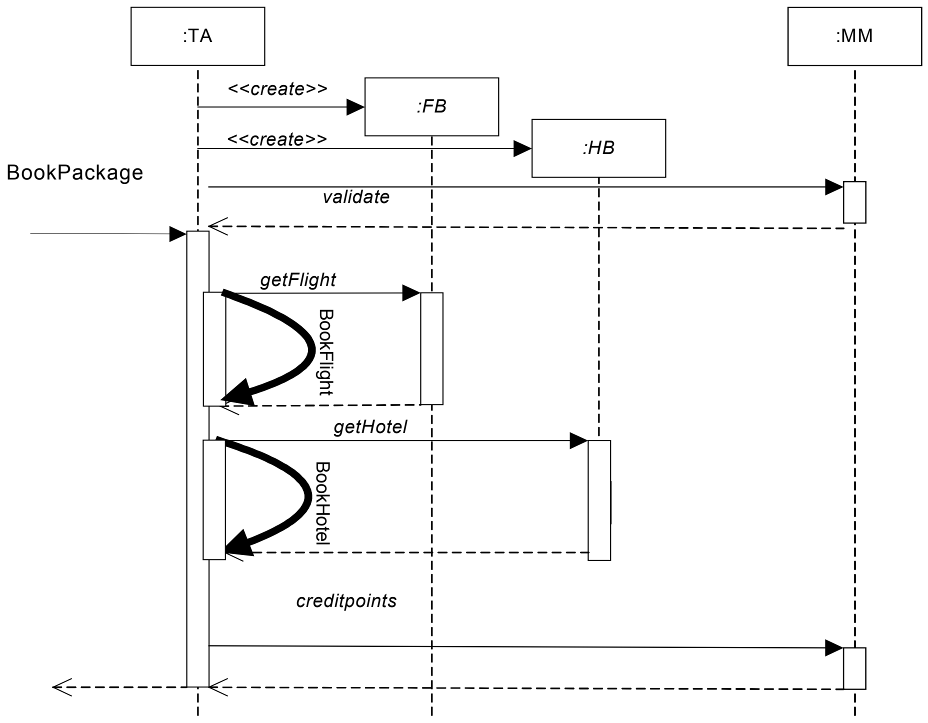


Figure 8: Dynamic composing travel planning WS components

which can be translated into the following dynamic composition specification using before advice[11].

```
   before(Itinerary *.BookPackage (Itinerary it))
   {
    this.fb= new FB(…); //the "…"part provides the
//information referencing the actual FB component that
//the instantiated object is bound to
    this.hb= new HB(…);
   }
```

Furthermore, the front end BookFlight and BookHotel code is dynamically overridden to delegate to the actual methods of FB and HB respectively. This is achieved using around advice as shown below:

```
   around (FlightInfo *.BookFlight (Itinerary it))
    {
     return fb.getFlight (it.traveler, it.flight);
     }

   around (HotelInfo *.BookHotel (Itinerary it))
    {
      return fb.getHotel (it.traveler, it.hotel);
        }
```

▪ Dynamic membership management

With the tightening security measures, the customer's background is subject to be checked by the central member management (MM) unit upon designated period of time. As such, a rule is added in Jess that for a given duration, the membership will be validated (e.g., background checking, passport verification) for each *BookPackage* call. Assume during the period July 1, 2005, to September 20, 2005, all traveller's membership will be validated by MM. To enable the Jess rule engine to trigger the dynamic composition of validation behavior, we need to:

1) capture the execution of *BookPackage* and relay the values into Jess fact bases;
2) have a bridge from Jess to .NET for rules to directly manipulate AAR in Figure 3.

As is mentioned in Section 4.3.3, we use WS to wrap a Java class, which in turn can interoperate with Jess. Thus, a .NET based WS component can interoperate with Jess rules. Specifically, to achieve 1), we add into the "before advice" for BookPackage the following code:

[11] For illustrative purpose, we use the syntax resembling AspectJ to specify the component composition, which in turn will be translated into XML representation as described in Section 4.3.2.

```
before(Itinerary *.BookPackage (Itinerary it))
{
  ……
  …… //above are other advice code which are ignored
    //here for clarity

WS_Jess.assert("membernumber",
                it.traveler.membernumber  );
WS_Jess.assert("airline",
               it.traveler.frequent_airline);
Date date=getdate();
WS_Jess.assert("date",date);
//the above three lines add three
//facts to the Jess fact base through WS-Jess bridge
}
```

To achieve 2), we define a Java class which is used as a relay between Jess and .NET platform, so that whenever the rule fires, AAR in .NET can be manipulated from Jess. The Java class is defined as follows:

```
class Jess_WS{
 public static void
     apply(string advicetype, String  returntype,
           String classname, String methodname,
            String parameterlist, String advicename)
     {
      … //code to interoperate with .NET to update AAR;
     }
}
```

The parameter list is consistent with the XML elements as shown in Figure 5. The Jess rule is specified as follows, which calls into the Java class Jess_WS:

```
(bind  ?aus  (new  Jess_WS))  ;;aus_wrapper  is  the  Java
         ;;wrapper for writing AUS
         ;;into  the  AAR  through  Java-WS  bridge  using
         ;;Java WSDP as ;;described in Section 4.3.3
(bind ?para (str-cat ?membernumber "/" ?airline))
         ;;the  values  of  ?membernumber  and  ?airline
         ;;are  fed  into  the  fact  base  by  the  before
         ;;advice for BookPackage
(defrule security_control
 (date ?d &:(>= ?d 20050701)&:(< ?d 20050920))
  =>(?aus   apply  "before",  "",  "TA",  "BookFlight",
   ?para, "MM.validate"))
```

The last line defines a Jess rule specifying once the booking date is between July 1, 2005 and September 20, 2005, the membership validation advice will be applied through Jess-Java-WS interoperation before the call of *.BookFlight in .NET environment. Once the condition is satisfied during runtime, the corresponding rule will be applied autonomously for dynamic composition. Furthermore, as the Jess rule exists as a separate entity for configuration from the execution logic, the composition rule can be adapted as needed at runtime as well.

Likewise, dynamic composition can be applied to credit travel points after the travel reservation, using after advice:

```
after(Itinerary *.BookPackage (Itinerary it))
{ MM.creditpoints(it);
}
```

Furthermore, dynamic composition can be applied either assertively or autonomously as shown above for other non-functional property guarantees including but not limited to budgeting (if the cost of the requested service exceeds the budget, either to choose a cheaper service or to remove subcomponents for reducing cost), and load balancing (if current load is over capacity, the service requests are to be delegated to alternative components). As those composition specifications overlap the aforementioned dynamic composition specifications in principle, details are omitted here.

6 Performance Evaluation

Using the profiler to handle all the events generated from all managed execution in CLR is expensive and will degrade system performance significantly. Therefore, we apply optimization at three levels through configuring the profiler as indicated in (7) in Figure 3:

1). As the CLR can be launched from a shell, Internet Explorer, ASP.NET, and other customizable CLR hosts for managed execution, we configure the profiler to skip profiling for all non-ASP.NET modules hosted in CLR, which can be filtered easily based on the name of the module that launches the CLR.
2). We could further trim unnecessary profilings based on class name, or CIL method. This is possible because all managed code is translated to CIL, and the CIL level information can be derived from the corresponding WSDL for the WS; this is also necessary to avoid profiling system classes and methods.
3). We mask all unnecessary events except JIT compilation events, which is needed for handling CIL manipulation.

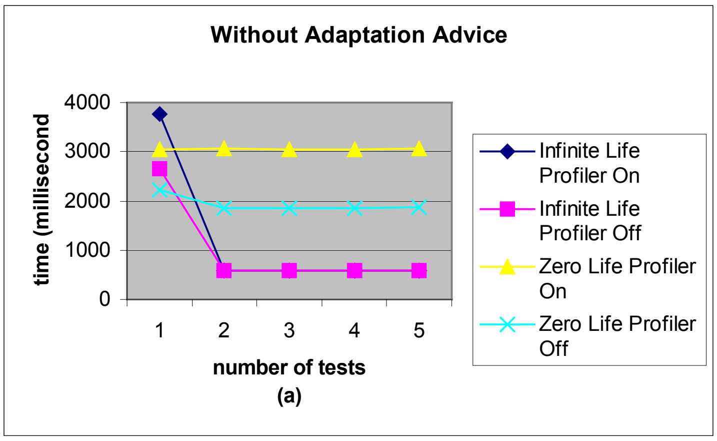
To evaluate the influence of CLR profiling-based WS adaptation on performance, we implemented a simple WS server application with 100 loops for calling a method, which contains only a single plus calculation in its body. We host this WS application on a Dell Workstation with Intel XEON CPU 2.2GHx, 1.00GB RAM, which is installed with Win XP professional version 2002 with IIS 5.1, .NET framework version 1.1.4322. We configured the profiler so that the method is to be profiled and adapted with log advice to write to a file a line of strings. A WS stub is generated by compiling the corresponding WSDL for this simple WS application. The WS stub is instrumented together with a simple client application for the client application to call the server-side WS. The client side is hosted on a Dell PC with Intel Pentium 4 CPU 1.80 GHz, 512 MB RAM which resides on the same LAN environment as the server so as to minimize the network influence during the server side performance benchmarking.

Note that the CLR profiling-based approach only applies to managed code to be loaded and JIT compiled. Therefore, we run ASP.NET in the managed mode for profiling WS to realize dynamic adaptation. ASP.NET can load one worker process to handle a pool of WS requests. Once the worker process is launched to serve the first WS requests in the pool, it continues to serve other WS requests in the same pool until the end of its lifecyle without itself being reloaded into CLR, thus it fails to profile the other WS applications in the same pool. Therefore, we adjust the setting for ASP.NET so that a new worker process will be created for each WS request so that each WS call can be captured by the Profiler and thus is adaptable. The goal of our tests is to evaluate how the adjustment of worker process lifetimes (Figure 9-a), and the enactment of profiling-based dynamic adaptation (Figure 9-b) affect the performance of WS provisioning in the peer-to-peer composition model.

For the case in Figure 9-a, we did not provide any adaptation advice when adjusting the worker process life between *zero life* (a new worker process is created for each WS request) and *infinite life* (the same worker class is used for multiple WS requests). The absence of advice execution will help clarify the influence of the changing life of a worker process on the system performance.

There are significant differences between the first call and the remaining calls for an infinite life case as the first call involves the creation of a new worker class, thus incurring more overhead than the remaining WS calls which reuse the original worker process. Also the presence of profiling does not affect performance much in the case of infinite life, as the worker process is no longer to be reloaded for new WS requests, thus the new WS will not be adapted, and the event handler in the profiling API is ignored. In comparison, the worker process with zero life will incur a performance degradation of 1.7 times slower with profiling on than with profiling off. With the absence of the profiler, the overhead incurred by adjusting from infinite life to zero life will be 3.0 times. With the absence of advice, the overall performance degradation (with profiling on, zero life for worker class) against the conventional WS provisioning scenario (with profiling off, infinite life for worker class) for this WS provisioning is 3.0*1.7=5.1. Figure 10 illustrates the performance degradation.

In Figure 9-b, we focus on evaluating the influence of active advice on the overall performance. Therefore, the worker process is set with zero life. We found the number of active advice will not affect the performance linearly, as the AUS are stored in the paging file to be shared by hooks, which constitutes a minor overhead in comparison to that incurred by hook instrumentation and calling of advice. The weaving of a matching advice in the case of zero life in Figure 9-b incurs a performance degrade of 2.2 times. Therefore, the overall performance degradation (with profiling on, zero life for worker class) against the conventional WS provisioning scenario (with profiling off, infinite life for worker class), by synthesizing the result descibed in the preceding paragraph, will be 2.2*5.1=11.2.



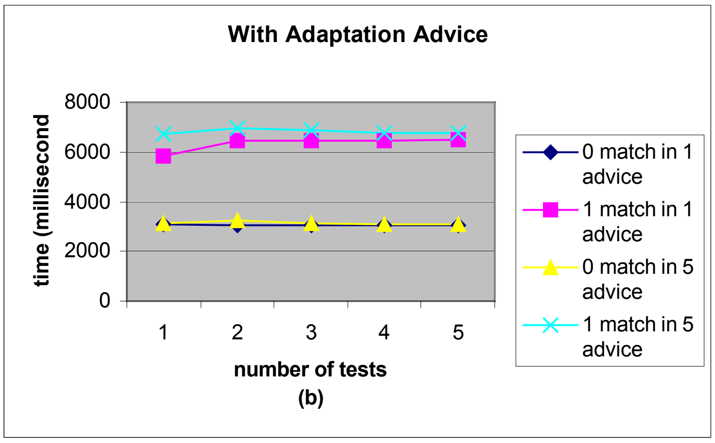


Figure 9: Benchmarking dynamic WS adaptation

In the real world deployment, we can reduce the overhead by setting the worker class to zero life at the adaptation time, then resetting it to infinite time after adaptation is done. Of course this assumes a predicable adaptation process.

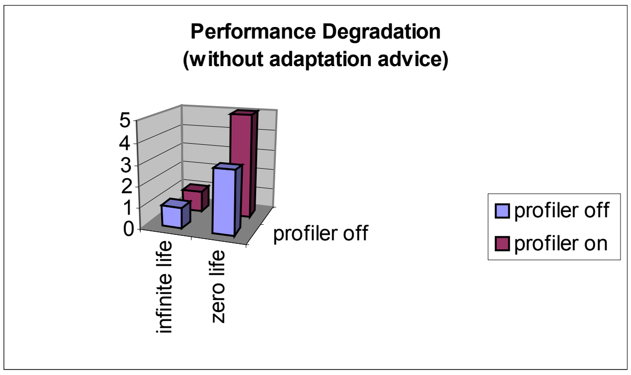


Figure 10: Performance degradation with 0 adaptation advice

7 Related Work

Component composition can be enacted at design level (e.g., [Clarke, 02], [Keller, 98]), and application code level (e.g., [Hölzle, 93], [Mezini, 98], [Seiter, 99]). In contrast, our work on component composition is enacted at intermediate code level without introducing new language constructs. With a lower-level of abstraction, our work enables cross-language component composition, while the above work restrict the component composition to a specific language. Also, none of the aforementioned work on component composition is applied at runtime, which is however necessary in distributed computing environment.

The Composition pattern has been proposed in [Clarke, 01], which uses a UML template for specifying composition of crosscutting concerns at a high level and maps sequence diagrams into AspectJ code. Our composition pattern is represented with a comprehensive framework rather than just a design-level pattern. Also a sequence diagram is used here for illustrating the dynamic partnership, with each object in the sequence diagram corresponding to a partner when mapped to dynamic composition specification. In contrast, each object in a sequence diagram ia synthesized to an aspect construct in AspectJ in [Clarke, 01]. While AOP has been applied to distributed systems for resolving crosscutting concerns ([Pulvermuller, 99], [Zhang, 03]), here we dedicate AOP to the composition purpose: for composing components handling cross-cutting concerns in a modularized way, and for separating composition from components. Moreover, we use the Jess inference engine to autonomously apply aspect weaving for component composition. While the work described in [Yang, 02] also aims at applying an aspect-oriented approach to dynamic adaptation, they only offer a means for making the AOP-based adaptation ready, without presenting any solution on how to use rule engines to trigger the adaptation. Additionally, [Duzan, 04] presents a prototype implementation in the QuO toolkit for an aspect-based approach to programming QoS-adaptive applications. In contrast, our work is targeted on loosely coupled service oriented computing as opposed to tightly coupled distributed object computing in QuO, where adaptation rules are triggered by exceptions thrown from runtime.

Our work also incorporates non-functional concerns into WS component composition. Prior work such as IBM's Web Services Level Agreement (WSLA) [Dan, 02] and HP's Web Service Management Language (WSML) [Sahai, 02] incorporate the notion at higher-level presentation, rather than address it at a lower-level platform layer. We believe a treatment at a platform layer is necessary toward thoroughly addressing non-functional concerns for WS.

Our work is rooted in the UniFrame project ([Raje, 02], [Olson, 05]), which aims at creating a framework for seamless integration of distributed heterogeneous components. In UniFrame, component composition is also following the peer-to-peer paradigm, which is enabled through a discovery services in search of a matching component. Once a searched component does not match the requirement functionally or non-functionally, the search process will be launched again, which exhibits the autonomous features similar to that described in the work presented here. While the work presented here is scoped at the service-oriented computing paradigm for component composition, the principles can be integrated into UniFrame as well.

8 Conclusion and Future Work

This paper presents a dynamic component composition approach under service-oriented paradigm in the .NET environment. By using intermediate code manipulation, component composition is 1) possible to cross language boundaries so long as they are CLR-compliant; 2) achieved in a non-invasive manner; 3) implemented not only in an assertive manner, but also in autonomous manner using a rule inference engine; 4) specified using the AOP paradigm for separating composition specification from components to be composed, and for modularized composition of components handling cross-cutting concerns, with hooks used to weave and unweave advice at runtime proactively and retroactively. Moreover, as the WS components can be exposed with XML-based WSDL, the component composition can be specified with language neutral XML, which is further mapped to language-neutral type system CTS, with low-level CTS transparent to upper level composition decision makers. The experimental results show the profiling-based dynamic composition approach is encouraging with the appropriate control over the profiling scope in the WS scenario. Even though the approach presented in this paper is .NET based, the principle also applies to other platforms with adequate software vendor support.

With the different abstraction levels involved as shown in Figure 1, one future direction is to investigate the model-driven approach ([Cao-b, 05], [Frankel , 03], [Lédeczi, 01]) for modeling component composition concerns, so that component composition can be represented in high-level models which reduces the gaps between business requirements and underlying implementation, with AAR and rules as shown in Figure 3 automatically synthesized from models. We would also like to explore the use of mobile agents in the peer-to-peer component composition scenario where composition decisions can be federated and communicated seamlessly, for which security is also of vital concern in the future research.

Acknowledgements

This research is supported in part by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

References

[Brown, 00] A. W. Brown, Large-Scale Component-Based Development, Prentice Hall, 2000.

[Cao, 02] F. Cao, B. Bryant, R. Raje, M. Auguston, A. Olson, C. Burt, Component Specification and Wrapper/Glue Code Generation with Two-Level Grammar Using Domain Specific Knowledge, In Proc. Int. Conf. on Formal Engineering Methods, October 2002, 103-107.

[Cao-a, 05] F. Cao, B. R. Bryant, S.-H. Liu, W. Zhao, A Non-Invasive Approach to Dynamic Web Service Provisioning, In Proc. IEEE Int. Conf. on Web Services, July 2005, (*to appear*).

[Cao-b, 05] F. Cao, B. R. Bryant, W. Zhao, C. C. Burt, R. R. Raje, A. M. Olson, M. Auguston. Model-Driven Reengineering Legacy Software Systems to Web Services, 2005 (*submitted*) .

[Chen, 01] Q. Chen, M. Hsu, Inter-Enterprise Collaborative Business Process Management, In Proc. Int. Conf. on Data Engineering, April 2001, 253-260.

[Clarke, 01] S. Clarke, R. J. Walker, Composition Patterns: An Approach to Designing Reusable Aspects, In Proc. Int. Conf. on Software Engineering, IEEE Computer Society, May 2001, 5-14.

[Clarke, 02] S. Clarke, Extending Standard UML with Model Composition Semantics, Sci. Comput. Program, 44(1), 2002, 71-100.

[Colan, 04] M. Colan, Service-oriented architecture expands the vision of Web Services, 2004, http://www-106.ibm.com/developerworks/webservices/library/ws-soaintro.html.

[Dan, 02] A. Dan, A. R. Franck, A. Keller, R. King, H. Ludwig, Web Service Level Agreement (WSLA) Language Specification, 2002, http://dwdemos.alphaworks.ibm.com/wstk/common /wstkdoc/services/utilities/wslaauthoring/WebServiceLevelAgreementLanguage.html.

[Devanbu, 96] P. Devanbu, S. Karstu, W. Melo, W. Thomas, Analytical and Empirical Evaluation of Software Reuse Metrics, In Proc. Int. Conf. on Software Engineering, IEEE Computer Society, March 1996, 189-199.

[Duzan, 04] G. Duzan, J. P. Loyall, R. E. Schantz, R. Shapiro, J. A. Zinky, Building Adaptive Distributed Applications with Middleware and Aspects, In Proc. Int. Conf. on Aspect-Oriented Software Development, March 2004, 66-73.

[Frankel , 03] D. S. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley, 2003.

[Friedman-Hill, 05] E. J. Friedman-Hill, Jess 7.0, The Rule Engine for the Java Platform, Sandia National Laboratories, 2005.

[Gough, 02] J. Gough, Compiling for the .NET Common Language Runtime (CLR), Prentice Hall PTR, 2002.

[Heineman, 01] G. T. Heineman, W. T. Councill, Component Based Software Engineering: Putting the Pieces Together, Addison-Wesley, 2001.

[Hissam, 03] S. A. Hissam, G. A. Moreno, J. A. Stafford, K. C. Wallnau, Enabling predictable assembly, Journal of Systems and Software, 65(3), 2003, 185-198.

[Hölzle, 93] U. Hölzle, Integrating Independently-Developed Components in Object-Oriented Languages, In Proc. European Conference on Object-Oriented Programming, July 1993, 36-56

[Keller, 98] R. K. Keller, R. Schauer, Design Components: Towards Software Composition at the Design Level, In Proc. Int. Conf. on Software Engineering, April 1998, 302-311.

[Kiczales, 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, In Proc. European Conference on Object-Oriented Programming, June 1997, 220-242.

[Kiczales, 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An Overview of AspectJ, In Proc. European Conference on Object-Oriented Programming, June 2001, 327-353.

[Lédeczi, 2001] Á. Lédeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai, Composing Domain-Specific Design Environments, IEEE Computer, 34(11), 2001, 44-51.

[Mezini, 98] M. Mezini, K. J. Lieberherr, Adaptive Plug-and-Play Components for Evolutionary Software Development. In Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications, October 1998, 97-116.

[Newcomer, 02] E. Newcomer, Understanding Web Services, Addison Wesley, 2002.

[Olson, 05] A. M. Olson, R. R. Raje, B. R. Bryant, C. C. Burt, M. Auguston, UniFrame-a Unified Framework for Developing Service-Oriented, Component-Based, Distributed Software Systems, Service-Oriented Software System Engineering: Challenges and Practices, Idea Group, 2005, 68-87.

[Pulvermuller, 99] E. Pulvermuller, H. Klaeren, A. Speck, Aspects in Distributed Environments, In Proc. Generative Component-based Software Engineering, September 1999, 37-48.

[Raje, 02] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, C. C. Burt, A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components, Concurrency and Computation: Practice and Experience, 14(12), 2002, 1009-1034.

[Sahai, 02] A. Sahai, V. Machiraju, M. Sayal, L. J. Jin, F. Casati, Automated SLA Monitoring for Web Services, 2002, http://www.hpl.hp.com/techreports/2002/HPL-2002-191.pdf

[Seiter, 99] L. M. Seiter, M. Mezini, K. J. Lieberherr, Dynamic Component Gluing, In Proc. Int. Symposium on Generative Programming and Component-Based Software Engineering, September 1999, 134-164

[Stutz, 03] D. Stutz, T. Neward, G. Shilling, Shared Source CLI – Essentials, O'Reilly Press, 2003.

[Szyperski, 02] C. Szyperski, D. Gruntz, S. Murer, Component Software: Beyond Object-Oriented Programming, 2nd ed., Addison-Wesley/ACM, 2002.

[Yang, 02] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, P. K. McKinley, An Aspect-Oriented Approach to Dynamic Adaptation, In Proc. The First Workshop on Self-healing Systems, November, 2002, 85-92.

[Zhang, 03] C. Zhang, H.-A. Jacobsen, Refactoring Middleware with Aspects, IEEE Trans. Parallel Distrib. Syst. 14(11), 2003, 1058-1073.