Aspect-Oriented Programming with C# and .NET

Wolfgang Schult and Andreas Polze
Hasso-Plattner-Institute at University Potsdam
{wolfgang.schult|andreas.polze}@hpi.uni-potsdam.de

Abstract

Almost a year ago, Microsoft has introduced the .NET architecture as a new component-based programming environment, which allows for easy integration of classical distributed programming techniques with Web computing. .NET defines a type system and introduces notions such as component, object, and interface, which are building blocks for distributed multi-language component-based applications.

As many other component frameworks, .NET mainly focuses on functional interfaces of components. Non-functional aspects of components, such as resource usage (CPU, memory), timing behavior, fault-tolerance, or security requirements are currently not expressed in .NET's component interfaces. These properties are essential for building reliable distributed applications with predictable behavior even in cases of faults.

Within this paper, we discuss the usage of aspect-oriented programming techniques in context of the .NET framework. We focus on the fault-tolerance aspect and discuss the expression of non-functional component properties (aspects) as C# custom attributes. Our approach uses reflection to generate replicated objects based on settings of a special "fault-tolerance" attribute for C# components.

We have implemented an aspect-weaver for integration of aspect-code and componen- code, which uses the mechanisms of the language-neutral .NET type system. Therefore, our approach is not restricted to the C# language but works for any of the .NET programming languages.Introduction and Motivation

1. Introduction

Reliable computer systems used in the telecommunication industry, in cars and automated factories (process control) are often implemented as special purpose systems which are vendor-specific, expensive, hard to maintain and difficult to upgrade. Often, those systems apply proprietary techniques to achieve security and predictable timing behavior, even in case of faults. With the need of integrating multiple of those control systems into a bigger whole, requirements arise to open up proprietary systems for standard (non real-time) distributed computing technology.

Component-oriented programming provides a promising way to system composition out of units with contractually specified interfaces and explicit context dependencies. Software component can be deployed independently, they are subject to composition by third parties. There exist a number of distributed component frameworks, notably the Common Object Request Broker Architecture (CORBA) [14] , Microsoft's Distributed Component Object Model (DCOM/COM+) [4] , SUN's JavaBean Model [7] , and the relatively new .NET framework [19] .

Although all of these frameworks simplify the implementation of complex, distributed systems significantly, the support of techniques for reliable, fault-tolerant, and secure software, such as group communication protocols or replication is very limited.

Any fault tolerance extension for components needs to trade off data abstraction and encapsulation against implementation specific knowledge about a component's internal timing behavior, resource usage, interaction and access patterns. These non-functional aspects of a component are crucial for the predictable behavior of real-time and fault-tolerance mechanisms. However, in contrast to the various mechanisms describing a component's functional interface (Interface Definition Languages, Class/Method specifications), there is no general means to describe a component's non-functional properties, such as security settings, fault-tolerance measures and timing behavior.

Within this paper we present our approach towards component replication for fault-tolerance in the .NET framework. Following the idea of aspect-oriented programming [9] we have developed tools and a description technique for fault-tolerance requirements. The description technique uses the extensible "custom attributes"-mechanism of the C# programming language as an underlying representation and allows specification of fault-tolerance requirements independently from an

object's implementation. Using the .NET reflection and introspection mechanisms, C# attributes can be evaluated at runtime. We have implemented tools, which allow for automatic generation of proxy objects, which in turn manage C# object replication and implement certain fault-detection mechanisms. Although this work concentrates on the C# programming language, our approach is more general and works for all programming languages which are built upon the .NET type system.

The remainder of the paper is organized as follows: Section 2 presents related work. Section 3 gives an overview over the .NET framework. Section 0 discusses meta-programming and reflection in .NET. Using a simple calculator a case study, we demonstrate in Section 5 how fault-tolerance requirements can be expressed using C# attributes. Section 6 discusses implementation issues concerning our tools. Section 7 gives direction to future work and Section 8 finally summarizes our conclusions.

2. Related Work

The idea of providing fault tolerance as additional feature to distributed, middleware-based component systems has been lately the focus of several research activities. There exist a variety of research projects, which focus especially on the CORBA platform. Significantly less work exists in context of the Microsoft Component Object Model (COM) and the new .NET framework.

In order to describe a component's fault-tolerance (FT) requirements and fault assumptions, two general approaches exist: FT requirements and assumptions can be expressed in some sort of extended interface definition language (IDL). This solution has been used by the CORBA systems mentioned below. The other option, which is also quite common in CORBA systems, is to hard-code component configuration and FT settings in form of a set of function calls (FT-API), which is inserted into component code.

With the "Draft Adopted Submission for Fault Tolerant CORBA" [15] adopted in March 2000, OMG has been seeking to incorporate existing approaches for software fault tolerance into CORBA. Among those approaches are Electra [12] and Orbix+Isis [6] , both are CORBA ORB-implementations for reliable, distributed services. Electra extends the CORBA specification and provides group communication mechanisms, reliable multicasts, and object replication. The Electra-ORB uses services from the underlying ISIS [3] and HORUS [17] systems. Orbix+Isis [6] works also on top of ISIS [3] . A different approach has been chosen by the designers of Eternal [13] , which implements an OMG-compliant fault-tolerance infrastructure without requiring modifications to the ORB. Eternal uses CORBA interceptors to attach group communication protocol and replica management functionality to the CORBA ORB.

The concept of aspect-oriented programming (AOP) offers an interesting alternative for specification of non-functional component properties (such as fault-tolerance properties or timing behavior). There are a variety of language extensions with AspectJ [2] [11] [8] (which is a Java extension) as most prominent example.

The reflection-API as present in Java can be used to obtain runtime type information about objects and classes. Using "marker interfaces" and the "instanceof"-operator, one could implement similar mechanisms as those introduced with AOP (see [10]). However, since Java interfaces are rather an implementation mechanism than an aspect-description mechanism, this approach violates the separation of component description and implementation.

Our work is novel as it uses the new C# language construct of an attribute to express non-functional component properties without any programming language extensions and without introducing a new interface definition language. We have developed a set of tools, which allow for automatic generation of proxy classes and replica management in order to deal with crash faults of object. Our current work has focused in a static mechanism for inter-weaving functional code and aspect code however; we plan to create a more dynamic version based on the new features of the .NET framework and the Common Language Runtime (namely the ability to generate, compile, and load code dynamically into the virtual machine). Additional research will focus on more sophisticated fault assumptions (timing/omission/incorrect computation faults).

3. Overview over the .NET Architecture

Almost a year ago, Microsoft has introduced the .NET architecture as a new component-based programming environment, which allows for easy integration of classical distributed programming techniques with Web computing.

At the center of the .NET framework is an object model, called the Virtual Object System (VOS), and at center of the object model is a type system. The object model relies on basic concepts found in many object-oriented languages, such as class, inheritance, dynamic binding, class-based typing and so on. The object model is not, however, identical to the model of any of these languages. Rather, it's an attempt to define a suitable base that bridges all these languages and others.

The type system of .NET gives objects of predefined basic types, such as integers and characters, a clear place in the type system–and it provides a clean way to convert between reference and value types through "boxing" and "unboxing" operations. The result is a more coherent and regular type system than we have seen in the dominant languages so far.

Most importantly, this model is designed to be language-independent. The C# programming language directly reflects the .NET object model. NET's focus is rather on the programming model than on any specific language. The .NET framework itself is language-independent and attempts to provide a reasonable target to which all current languages can map. The framework enables compilers for multiple languages (namely C#, C++, VB) to share a common back end.

Multilanguage component mechanisms have existed before, notably CORBA and COM. But they contain a major hurdle – one has to write an interface description in the appropriate interface definition language (IDL) for every component that you make available to the world. There is no IDL with .NET: You just use classes from other languages as if they were from your own.

What this means for both component developers and component users is a dramatic simplification of the requirements put on any single development environment. You don't need libraries addressing every application area. You provide components in your domains of expertise, where you can really bring added value. Where good libraries already exist, you benefit from them at no extra cost.

4. Metadata and Reflection in .NET

Reflection is a language mechanism, which allows access to type information during runtime. Reflection has been implemented for various object-oriented programming languages, among them Java, C#, and C++. C++ is somewhat special as it implements reflection rather as an add-on (RTTI - runtime type information) than as an inherent language feature. With .NET, reflection is not only restricted to a single language, but basically anything declared as code (any .NET assembly) can be inspected using reflection techniques. There are two variants of accessing runtime type information in .NET: the reflection classes in the common language runtime library and the unmanaged metadata interfaces.

4.1. Reflection via Runtime Library

The runtime library´s reflection classes are defined in the namespace of `System.Reflection`. They build on the fact that every type (class) is derived from `Object`. There is a public method named `GetType`, which has as return value an object of the type `Type`. This type is defined in the namespace `System`. Every type-instance represents one of three possible definitions:

- a class definition
- an interface definition
- a value-class (usually a structure)

Via reflection, one may ask about almost every type attribute, including the type's access-modifier, whether it is a nested type and about the type's properties.

Metadata information is structured in a hierarchical fashion. At the highest level stands the class `System.Reflection.Assembly`. An assembly object corresponds to one or more dynamic libraries (DLLs) from which the .NET unit in question is composed. As depicted in Figure 1, class `System.Reflection.Module` stands on the next lower level of the metadata hierarchy. A module represents a single DLL. This module class accepts inquiries about the types the module contains. Proceeding further down the metadata hierarchy reveals type information for any of the building blocks making up a member of the .NET virtual object system.

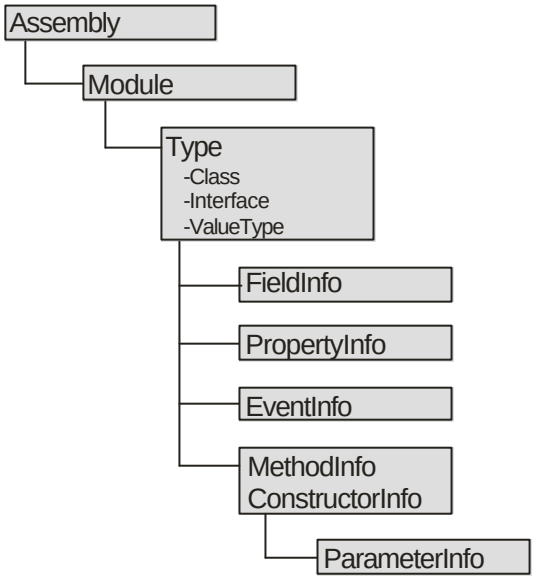

Figure 1: The Metadata Hierarchy of .NET

In each case, an instance of the class `System.Reflection.MemberInfo` represents a single data element. Such a data element may describe one of the following basic units making up an object:

- method (`System.Reflection.MethodInfo`)
- constructor (`System.Reflection.ConstructorInfo`)
- property (`System.Reflection.PropertyInfo`)
- field (`System.Reflection.FieldInfo`)
- event (`System.Reflection.EventInfo`)

Figure 2 presents an excerpt from a C++ program, which uses reflection to display all methods of a given type (`MyCalculator` in our case).

```
Type pType=typeof(MyCalculator);
MemberInfo[] arMemberInfo =
pType.GetMembers(BindingFlags.LookupAll);
int cMembers = arMemberInfo.Length;
for ( int i=0; i < cMembers; i++ ) {
  MemberTypes mt = arMemberInfo[i].MemberType;
  if(mt == MemberTypes.Method ) {
    // Downcast the MemberInfo to a MethodInfo
    MethodInfo pMethodInfo =
      (MethodInfo)arMemberInfo[i];
    Console.WriteLine(pMethodInfo.Name);
  }
}
```

Figure 2: Access to Runtime Type Information using Reflection in C#

4.2. The Unmanaged Metadata Interfaces

The unmanaged metadata interfaces are a collection of COM interfaces that are accessible from "outside" of the .NET environment. You can access them from any Windows program. The interface definition can be found in the COR.H, which is contained in the platform software development kit (platform SDK).

The interface `IMetaDataImport.IMetaDataAssemblyImport` is used for accessing metadata on the .NET assembly level. Access to this interface is obtained via a second interface, called `IMetadataDispenser`. As the name indicates, this interface "dispenses" all kinds of additional metadata interfaces, which allow read and write access to .NET metadata.

Access to the metadata dispenser is obtained via calls to the COM system as depicted in Figure 3 (here as C++ Code):

```
hr = CoCreateInstance(
  CLSID_CorMetaDataDispenser, 0,
  CLSCTX_INPROC_SERVER,
  IID_IMetaDataDispenser,
  (LPVOID*)&m_pIMetaDataDispenser );

hr = m_pIMetaDataDispenser->OpenScope(
  wszFileName,
  ofRead,
  IID_IMetaDataImport,
  (LPUNKNOWN *)&m_pIMetaDataImport );
```

Figure 3: Access to the IMetaDataImport Interface via COM

The `IMetaDataImport` interface obtained from the `OpenScope()` call provides access to the .NET assembly specified in the `wszFileName` Argument. Information about the structure of classes contained in that particular .NET assembly and their building blocks is now accessible via functions `EnumXXX` and `GetXXXProps`. The first function returns an enumeration of tokens describing the metadata available, the latter one returns information about the metadata's properties, which correspond to a particular token.

In addition to the token there exists a special way of type encoding. The function `GetMethodProps` for example gives an array of the type `PCOR_SIGNATURE` as return value. This array contains the signature of the queried element. The same information can be obtained by multiple calls to `EnumXXX` and `GetXXXProps`, however, using the signatures is the more direct approach. Signatures contain only pure type information, whereas `GetXXXProps` methods reveal also formal parameter names.

5. A C# Attribute to express Fault-tolerance Requirements

Within this Section we are presenting a simple calculator in C# as a case study. We use the calculator as basis for a discussion on how functional (C#) and non-functional (aspect) code can be combined.

5.1. The Calculator example

As depicted in Figure 4, our C# calculator has been implemented within a class `Calculator` which resides in the namespace `Calc`. Our calculator stores its operands as data-members `Op1` and `Op2`. The class implements a public member function `Add`.

```
namespace Calc {
  public class Calculator {
    public Calculator() { Op1=0; Op2=0; }
    public double Op1;
    public double Op2;
    public double Add() { return Op1+Op2; }
  }
}
```

Figure 4: The Calculator Class

5.2. Extending the Calculator to tolerate Crash-Faults

Once the calculator class has been compiled, it is available as a .NET assembly. Clients may import the assembly and instantiate calculator objects. We are now going to introduce a C# attribute which transparently adds fault-tolerance to our calculator class. With the modified class, whenever a client creates an object (via new), multiple instances of the object are created and managed consistently (replication in space).

Since the main purpose of our work so far was to investigate whether the C# language and runtime mechanisms are flexible enough to express non-functional component properties, we are assuming a very simple fault model for now. The only faults we assume to occur are crash faults at the object level. We introduce a proxy

object for replica management, which constitutes a single-point-of-failure. However, this proxy object can be seen as part of the client rather than part of the replicated service (and faults at client side are not considered at all).

Furthermore, we assume that replica consistency can be maintained without communication among the replicas. This means that replicas have to be deterministic, they are not allowed to make (concurrent) use of system services which require serialization of requests (such as `gettimeofday()`).

As discussed in Section 7, we feel that most of those simplifying assumptions can be lifted in the future. We plan to use .NET remoting in order to distribute replicated objects across machine boundaries. This would allow us to tolerate not only crashes of objects but also process or node crashes in a distributed environment. The usage of "aspect-specific templates", which is mentioned in Section 7.1, allows to generate more flexible code for replica management than demonstrated here. We plan to implement a number of consensus protocols to maintain replica consistency, among them a voting scheme (which would allow us to detect and tolerate incorrect computation faults). A master-slave replication scheme could be considered in order to deal with system calls that are not idempotent.

For our simple example, we define a C# attribute to describe fault-tolerance requirements:

```
[TolerateCrashFault(n)]
```

The parameter n indicates how many crash faults of objects implemented inside the C# component (assembly) may occur before the service provided by the component (which is adding numbers) is discontinued.

In order to tolerate n crash-faults of objects, one needs n+1 replicas of an object. So, behind the scenes, our component will create n+1 replicas whenever a client asks for a new calculator objects.

Syntactically, C# attributes may appear at every type definition. In our case, we have extended the definition of the Calculator class with an attribute, as depicted below.

```
[TolerateCrashFault(4)]
public class Calculator {
  /* ... */
}
```

There are more sophisticated fault-assumptions for replicated services than just crash faults, however, in order to demonstrate the automatic generation of code for replica management based on C# attributes, we restrict ourselves to the most simple crash-fault assumption for objects. In our case, five objects would be created and the calculator service remains accessible as long as at least one object survives.

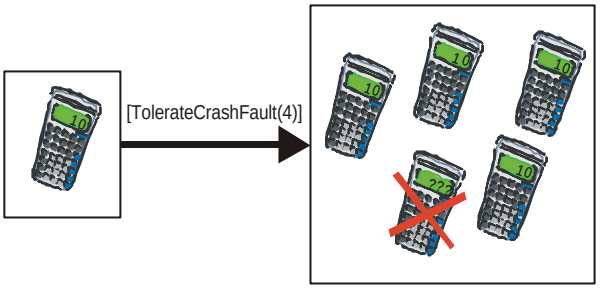

Figure 5: Replication in Space controlled by an Attribute

In order to make replication (almost) transparent to clients, the new (redundant) component has to meet the following requirements:

1. The interfaces must not change. Especially the method signatures have to remain unchanged.
2. Polymorphism and inheritance relations should remain intact. If a client is deriving from a class implemented in a component, then it should still be able to derive from that class after adding the attribute to component.
3. Changes in client-side code should be kept minimal.

Additional requirements concern the implementation of objects, which are to be replicated. Due to our current simple scheme for managing replica consistency, we assume deterministic behavior of the objects which includes the requirement to not interact with further system components (such other objects, files, non-deterministic system services).

5.3. The Aspect Weaver (Wrapper Assistant)

In aspect-oriented programming terminology, a tool, which mixes functional and aspect-code is called an aspect weaver. We have designed and implemented a tool called *WrapperAssistant*, which acts as an aspect weaver and generates code for replica management. Our tool uses introspection and reflection techniques based on metadata in the .NET Common Language Runtime (CLR) to detect function signatures exported by a component and to generate proxy classes for those classes exported by the component. The behavior of the replica management mechanism is controlled by the `TolerateCrashFault(n)` attribute.

Figure 6 shows a screen dump of a *WrapperAssistant* dialog where the user is presented a list of classes implemented in a particular .NET assembly (the calculator assembly in our case). Depending on the user's selection, the *WrapperAssistant* will generate code for the appropriate proxy classes, whose signatures will be identical with the original classes.

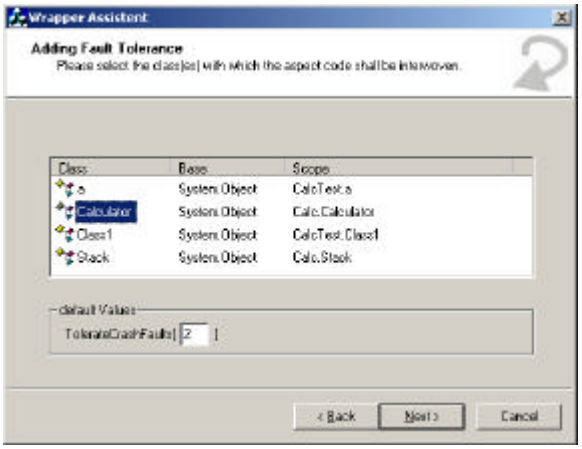

Figure 6: Choosing the Classes

On client side, only minimal changes to the original code are required. In fact, all the client programmer has to do in order to use the added fault-tolerance features of a component is changing a single line of code.

```
using proxy;    // clients have to import the
                // proxy namespace
// using calc; /// in order to activate the
// replica management and fault-tolerance
// features

void Calculate() {
  Calculator c = new Calculator();
  // this comes from the proxy namespace
  c.Op1=3;
  c.Op2=7;
  Console.WriteLine(c.Add());
}
```

Figure 7: A Client using Calculator-Proxy

The only change required in our client-code is commented out in line 2 – instead of `using the calc` namespace, the client now uses the `proxy` namespace. The actual implementation remains untouched.

6. Implementation of the WrapperAssistant

Interception of calls into a component – either at runtime or by source-code substitution at compile-time – is a standard way to transparently add code, which modifies the behavior of a component and implements certain non-functional aspects (fault-tolerance in our case). Figure 8 illustrates the control flow necessary to invoke a function on multiple replicas of an object. The return values of those function invocations have to be combined into a single value, which is sent back to the client. Under the crash-fault assumption, it is sufficient to simply forward the first value obtained from any replica. There are various design alternatives:

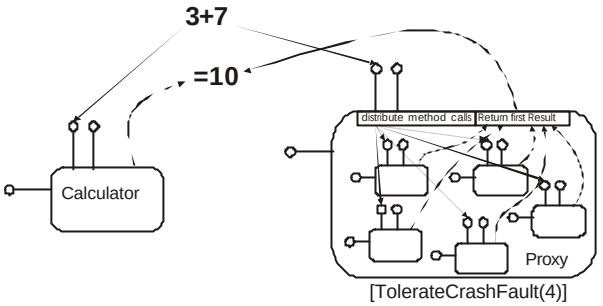

Figure 8: Invoking a Function at Replicated Objects

6.1. Tool Generated Code vs. Runtime Delegation

There exist two main options to implement function call redirection for a component. The first option would be using object-oriented function pointers (delegates in C# jargon) to redirect calls to a specially augmented object into a series of calls to object replicas. This method is very flexible and allows for dynamic redirection (reconfiguration) of function calls. However, access to replicated member variables and component properties cannot be handled in this fashion.

Generation of code for replica management at compile time was the second option. Our aspect weaver follows this approach and generates proxy classes for replica management. A proxy class is derived from a given class and implements the original class' signature. However, instead of actually implementing all methods, the proxy maintains data structures for replica management and forwards function calls. Additionally, it implements setter/getter methods for each member variable present in the original C# class. This way, access to member variable of replicated objects is handled.

6.2. Source Code vs. Intermediate Language Type Info

The next question that arises is how to obtain an interface's signature. The classical compiler-approach would be to parse the declaration of the component in its original languages. This would require access to components' source. Also, it would restrict the aspect weaver tool to components written in a single programming language.

We have decided to follow the .NET approach and use runtime type information to derive the signatures of classes and their members from a binary .NET assembly's metadata. There are two different way to programmatically deal with metadata. The first option is based on the reflection API present in the C# language. Using the unmanaged metadata COM interfaces as accessible from C++ is the second option. Because of early beta status of some of the C# tools, we opted to use

the unmanaged metadata interfaces for the implementation of our *WrapperAssistant* tool.

6.3. Generation of a Proxy Class

The *WrapperAssistant* generates classes for replica management within a separate namespace (`proxy` in our case). These classes directly extend the public classes implemented in a given component. For the calculator example the following code is generated:

```
namespace proxy {
  public sealed class Calculator:Calc.Calculator
  {
    /* ... */
  }
}
```

Figure 9: Definition of a Proxy Class

The next step is to overwrite each member function of the original class with a version, which has an identical signature but – instead of actually implementing the function – forwards function calls to multiple replica objects. To gain access to the assignment of public variables of the original class, they are defined as properties in the tool-generated proxy class.

For the calculator this would look as follows:

```
new public double Op1 {
  get { /* ... */ }
  set { /* ... */ }
}
```

Figure 10: Getter/Setter Methods for Data Members

Within the constructor of the proxy class, the appropriate number of base class instances has to be created. The `TolerateCrashFaults` attribute as defined in Figure 11 supplies that number.

```
public sealed class
TolerateCrashFaults:System.Attribute {
  private int m_i;
  public TolerateCrashFaults(int i) {m_i=i; }
  public int Count
  { get {     return m_i+1;  } }
}
```

Figure 11: Definition of TolerateCrashFaults-Attribute

The constructor internally stores the number of tolerable errors. Variable `_Count` contains the number of replicas that have to be created. Figure 12 shows an excerpt from the proxy class' constructor.

```
public Calculator(): base() {
  int _Count=0;
  System.Attribute[] _arAtt =
    System.Attribute.GetCustomAttributes(
      GetType());
  foreach(System.Attribute _attr in _arAtt) {
    if(_attr is TolerateCrashFaults)
      _Count=((TolerateCrashFaults)_attr).Count;
  }
  _bc=new Calc.Calculator[_Count];
  int i;
  for(i=0;i<_bc.Length;i++) {
    try { _bc[i]=new Calc.Calculator(); }
    catch(System.Exception) { _bc[i]=null; }
  }
}
```

Figure 12: Creation of Replicas

At first the constructor checks for the `TolerateCrashFaults` attribute. The attribute then is read and the constructor creates `_Count` memory slots (as Array `_bc`). Those are then filled with references to the object replicas.

Each overwritten member function in the proxy class passes its function-call to every instance referenced in the array. For the `Add` function this looks as follows:

```
public new double Add()
{ int i;
  double _RetVal=new double();
  for(i=0;i<_bc.Length;i++) {
    if(_bc[i]==null) continue;
      try { _RetVal=_bc[i].Add(); }
      catch(System.Exception) { _bc[i]=null; }
    }
    return _RetVal;
}
```

Figure 13: Function Call Forwarding

7. Future Work

7.1. Aspectspecific Templates

Definition of so-called join points for interweaving aspect-specific code and functional component code is a standard problem when using AOP. Interception of method calls at runtime is an approach chosen by many aspect systems. However, this allows only for invocation of aspect specific code before and after each function call.

We have developed a set of so-called aspect specific templates, which define rules for source-code substitution. Those templates define special points of interweaving (join points). The code is written in the target language (C# in our case). The substitution is then carried out at these points. Figure 14 presents an example for a template, which deals with the fault-tolerance aspect and specifies how to extend a method call. The parts written bold within the pointed parentheses are interweaving points.

```
public <MODIFIER> <RESULTTYPE>
<METHODNAME>(<PARAMDECLARATION>) {
  int i;
  <RETVALINIT>
  for(i=0;i<_bc.Length;i++) {
    if(_bc[i]==null) continue;
    try {
      <RETVALASSIGN>_bc[i].<METHODNAME>
        (<PARAMLIST>);
    } catch(System.Exception) { _bc[i]=null; }
  }
  <RETVALRETURN>
}
```

Figure 14: Template for Function Call Redirection

Our set of templates deals with various aspects of generation of proxy code. Specific templates focus on:

- Namespaces
- Classes
- Methods
- Arrays
- Constructors

The mechanism is very flexible. Templates allow for the specification of aspect-code without even knowing the components with which it will be used.

7.2. Managing Aspect-Information at Runtime

The use of attributes to declare aspects has further advantages. Since attributes are implemented as classes (which derive from `System.Attribute`), they may carry constructor code as well as additional methods. The aspect weaver then cannot only determine weather an aspect is defined; it also can call these functions to obtain additional information for the weaving process. Using additional methods declared for an attribute, it is also possible to change the attribute's semantics during runtime of a program. This would allow adapting to changes in the environment. For example, one could define an attribute, which uses either replication in space (if there are enough computing nodes available) or replication in time in order to tolerate crash faults of objects or processes. The switch over between those different implementation strategies would be managed by the class implementing the attribute – and thus be transparent to any clients using a replicated service. This feature clearly exceeds the flexibility of more static approaches, which use an IDL-like language to express aspect information.

Another part and the next step of our current work is dynamic weaving. This means that the weaver is integrated in the runtime environment. Instead of using the `new` statement to create an object, the weaver is called to generate an instance of the object implemented inside a .NET component. The .NET system supports the possibility to creating and executing code at run time. We are currently studying restrictions imposed on component interfaces by this approach.

8. Conclusions

The concept of aspect-oriented programming (AOP) offers an interesting method for specification of non-functional component properties (such as fault-tolerance properties or timing behavior). The new component-based programming environment, introduced by Microsoft almost a year ago, allows for easy integration of classical distributed programming techniques with Web computing. As many other component frameworks, .NET mainly focuses on functional interfaces of components

However, the Common Language Runtime, which is the foundation of the .NET framework, supports introspection and reflection for .NET components (assemblies). Using these mechanisms, our research focuses on the application of aspect-programming techniques to the .NET framework.

Within this paper, we have discussed how the new C# language construct of an attribute can be used to express non-functional component properties without any programming language extensions. We have developed a set of tools, which allow for automatic generation of proxy classes and replica management in order to deal with crash faults of object. We have outlined how the static mechanism for inter-weaving functional code and aspect code can be replaced by a more dynamic version based on the new features of the .NET framework and the Common Language Runtime. Additional research will focus on more sophisticated fault assumptions (timing/omission/incorrect computation faults).

References

[1] T. Archer, "Inside Microsoft C#", ISBN 0-7356-1288-9, Microsoft Press.

[2] AspectJ Hompage, http://www.aspectj.org/, 2001

[3] K.P.Birman, "The Process Group Approach for Reliable Distributed Computing", Communications of the ACM, Vol. 36, No.12, December 1993, pp.37-53.

[4] D. Box, "Essential COM", ISBN 0-201-63446-5, Addison-Wesley, February 1998.

[5] S. Hanenberg, R. Unland, "Concerning AOP and Inheritance", Dept. of Mathematics and Computer Science University of Essen.

[6] Isis Distributed Systems Inc. and Iona Technologies Limited, Orbix+Isis Programmer's Guide, 1995.

[7] SUN Microsystems, “JavaBeans: The Only Component Architecture for Java Technology”, http://java.sun.com/products/javabeans/.

[8] G.Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, “Getting Started with AspectJ“, Communications of the ACM, Vol. 44, Issue 10, October 2001, pp. 59-65

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin, “Aspect Oriented Programming”, In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finnland, Springer Verlag LNCS 1241, June 1997.

[10] K.Lieberherr, D. Orleans, J. Ovlinger; “Aspect-Oriented Programming with Adaptive Methods”, Communications of the ACM, Vol. 44, Issue 10, Oktober 2001, pp. 39-41

[11] C. V. Lopes, G. Kiczales, “Recent Developments in AspectJ”, Xerox Palo Alto Research Center.

[12] S.Maffeis, “A Flexible System Design to Support Object Groups and Object-Oriented Distributed Programming”, in Proceedings of ECOOP’93, Lecture Notes in Computer Science 791, 1994.

[13] P.Narasimhan, L.E.Moser, P.M.Melliar-Smith, “Strong Replica Consistency for Fault-Tolerant CORBA Applications”, in Proceedings of Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), 2001.

[14] OMG, "The Common Object Request Broker: Architecture and Specification", Object Management Group, Inc., Framingham, MA, USA, 1995.

[15] OMG, “Draft Adopted Submission for Fault Tolerant CORBA (Part 1 and 2)”, doc.omg.org/ptc/00-03-04, doc.omg.org/ptc/00-03-05, March 2000.

[16] M. Pietrek, http://msdn.microsoft.com/msdnmag/issues/1000/metadata/metadata.asp.

[17] A. Polze, J. Schwarz, M. Malek, “Automatic Generation of Fault-Tolerant Corba-Services”, in Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS) USA'00, Santa Barbara, August 2000, IEEE Computer Society Press, 2000.

[18] R.van Renesse, K.P.Birman; “Fault-Tolerant Programming using Process Groups”, in F.Brazier, D.Jones (Eds.) “Distributed Open Systems”, Computer Society Press, 1994.

[19] J. Richter, D. Box, several articles about .NET; SYSTEM-Journal 02/2001 to 05/2001, redtec publishing, Unterschleißheim, Germany.

[20] Workshop “Microsoft .net Crash Course for Faculty and PhDs”, Microsoft Research, Cambridge, England, September 3-6, 2001.