TRAP.NET: A Realization of Transparent Shaping in .NET

S. Masoud Sadjadi and Fernando Trigoso
School of Computing and Information Sciences
Florida International University, Miami, FL, U.S.A
{sadjadi, ftrig001}@cs.fiu.edu

Abstract

We define adaptability as the capacity of software in adjusting its behavior in response to changing conditions. To list just a few examples, adaptability is important in pervasive computing, where software in mobile devices need to adapt to dynamic changes in wireless networks; autonomic computing, where software in critical systems are required to be self-manageable; and grid computing, where software for long running scientific applications need to be resilient to hardware crashes and network outages. In this paper, we provide a realization of the transparent shaping programming model, called TRAP.NET, which enables transparent adaptation in existing .NET applications as a response to the changes in the application requirements and/or to the changes in their execution environment. Using TRAP.NET, we can adapt an application dynamically, at run time, or statically, at load time, without the need to manually modify the application original functionality-hence transparent.

1. INTRODUCTION

The goal of our ongoing research is to improve software adaptability. Imagine a world where software systems do not have to stop every time there was a need for adapting the software to the new changes in its requirements or in its execution environment. For example, as a wireless user moves from one wireless cell to another, the applications available to the user must adapt to different environments and resources with minimal interruption in their service. Critical systems such as financial networks or power systems cannot afford to shut down due to the need to adapt to new conditions or security attacks. A hurricane prediction application running for hours cannot be restarted just because a few resources are out of service.

Adaptable software presents itself as a possible solution to the above problems. An application is said to be *adaptable* if it can change its behavior dynamically at runtime, which may be due to changes in its environment or due to new functional or non-functional requirements [1]. Unfortunately, developing adaptable software is non-trivial. An adaptable application involves both *functional code* and *adaptive code*. Functional code implements the business logic of the application while adaptive code implements the adaptation logic which enables the application to be adaptive. Usually, these two types of code are blended into one source code making its maintenance and adaptation difficult.

The *transparent shaping* programming model poses a solution to solve the difficulty of developing adaptable applications [1]. This model allows the design and development of adaptable applications without the need to modify their source code.

Transparent shaping produces adaptable programs in two steps. In the first step, usually executed at compile time, an existing program is transformed so its behavior can be adapted at runtime. And in the second step, executed at runtime, these transformations receive the new behavior so the program can be adapted.

The first step of transparent shaping generates an adapt-ready program. An *adapt-ready* program is a program whose behavior is initially equivalent to the original program except for the fact that it can be adapted at startup and/or run time.

Applying transparent shaping to object-oriented programs yields a new programming model called *Transparent Reflective Aspect Programming* (TRAP) [1]. In this paper, we provide a realization of transparent shaping following the TRAP model, called TRAP.NET, which is targeted for .NET applications.

TRAP.NET provides a language-independent mechanism for transparently producing adapt-ready programs from existing programs in .NET. TRAP.NET also provides a mechanism to adapt these adapt-ready programs. This adaptation can be static, at load time, or dynamic, at runtime.

Static adaptation is more restrictive than dynamic adaptation. Static adaptation can only occur once when the application is loading and it is useful for applications that will be deployed on different platforms. These applications only need to adapt to their corresponding platforms at startup time. Dynamic adaptation is useful for applications that need to adapt to changes in their environment or to new requirements without halting.

The major contributions of this paper are summarized as follows. First, we assess the expressiveness and effec-

tiveness of .NET Attributes in providing a means to label what portions of an existing application should become adaptable. *Attributes* are pieces of metadata information that can be placed in the source code of .NET applications. TRAP.NET uses this metadata information to identify which pieces of functionality should be made adapt-ready. Second, we researched and developed a language-independent software tool that realizes the first step of transparent shaping appropriate for .NET applications. We call this software tool the *generator*. The generator automatically generates an adapt-ready application independent of the programming language used in the development of the original application. Finally, we researched and developed a language- and platform-independent software tool that realizes the second step of transparent shaping. We call this software tool the *composer*. The composer allows new adaptive behavior—to be added at startup or runtime—to replace the existing adapt-ready behavior.

The remainder of this paper is organized as follows. Section 2 provides background on the .NET technologies that TRAP.NET uses. Section 3 explains the design, implementation, and operation of TRAP.NET. Section 4 compares TRAP.NET to some related work. Finally, Section 5 offers some concluding remarks.

2. BACKGROUND

This section provides a brief overview of the .NET technologies that TRAP.NET uses to implement the transparent shaping model.

The Microsoft .NET Framework is a software component that provides vast pre-coded solutions to common program requirements in the form of class libraries. It also manages the execution of programs written specifically for this framework [6]. The .NET Framework provides a run-time environment called the co*mmon language runtime* (CLR, or just runtime) that runs .NET applications and provides services that make the development process easier. When compiling a .NET program, the compiler translates the source code into *common intermediate language* (CIL, or just IL), which is a CPU-independent set of instructions that can be efficiently converted to native code. CIL, formerly called *Microsoft intermediate language* or MSIL, resembles an object-oriented assembly language. TRAP.NET is language-independent as it provides adaptation at the CIL level.

When a .NET compiler produces code in CIL, it also produces the corresponding metadata. *Metadata* is "data about data;" in the programming context it is data about the program. Metadata describes the types in the code, including the definition of each type, the signatures of each type's members, the members that the code references, and other data that the runtime uses at execution time. The CIL and metadata are contained in a portable executable (PE) file referred to as *module*. The presence of metadata in the module along with CIL enables the code to describe itself. The runtime locates and extracts the metadata from the module as needed during execution.

.NET Reflection is the component that provides class libraries to access the metadata of .NET applications. Therefore it contains classes to describe every programming element such as assemblies, modules, namespaces, types, fields, methods, attributes, events, etc. Using reflection, TRAP.NET can access the metadata and the IL code of existing applications. TRAP.NET takes advantage of this feature and puts all the adaptive-related metadata into the same file and this way, there is no need to load a separate file with the metadata information about the adaptive behavior.

Reflection can also be used to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object. Then it can invoke the type's methods or access its fields and properties. TRAP.NET can discover information about all the elements of an application. Also, it can modify an application's behavior mainly through the usage of dynamic methods. *Dynamic methods* are methods that can be generated and executed at runtime.

.NET Attributes are keyword-like descriptive declarations. They resemble programming languages reserved keywords such as *public* or *private.* These two keywords further define the behavior of class members by describing their accessibility to other classes. Because compilers are designed to recognize these predefined keywords, a developer does not traditionally have the opportunity to create their own. The CLR, however, allows the addition of attributes to annotate programming elements such as types, fields, methods and properties [7]. These attributes can be extracted using runtime reflection services. TRAP.NET uses attributes to label methods that should become adapt-ready and to gather metadata information about these methods.

3. TRAP.NET OVERVIEW

This section provides an overview of TRAP.NET from its usage perspective. It describes the steps required to achieve dynamic adaptation with TRAP.NET at development time, compile time and run time. Static adaptation is presented at the end of this section since it reuses the techniques used to achieve dynamic adaptation.

3.1. At Development Time

To tailor the TRAP approach to the .NET development practices we allow the user to manually place .NET Attributes to annotate which methods should be adapt-ready. In our particular case we implemented a custom attribute which we call the *AdaptReady* attribute. By placing this attribute, the user is staging the application so it is suitable for the generator at compilation time. The generator can then look for the methods with the *Adap-*

tReady attribute to automatically make them adapt-ready. The code snippet in Figure 1 shows this attribute with a method called *Some-Method*.

Another important step that occurs at development time is the fact that the user has to add a reference to the TRAP.NET class library. The *AdaptReady* attribute is defined in the TRAP.NET class library, thus to successfully compile the application with this attribute, the application has to reference this class library. This particular reference coalesces the original application and TRAP.NET into one application. This union makes the generation process much easier since most of the functionality to support adaptation can be placed in the TRAP.NET library. Therefore, most of the adaptive code that needs to be weaved can be simple calls to functions in the TRAP.NET library.

```
[AdaptReady(true)]
public void Some-Method()
{
    /* some implementation */
}
```

Figure 1. A method with the *AdaptReady* attribute.

3.2. At Compile Time

After the annotated application is compiled as it normally would, the user can send its application to the generator. The generator can be executed from a plug-in for Visual Studio or form the command-line.

The generator is in essence an aspect weaver which adds the adaptive aspect to the methods annotated with the *AdaptReady* attribute. The adaptive aspect in this case consists of hooks that will intercept and redirect the control flow as appropriate. As mentioned earlier, since all .NET assemblies are compiled into CIL, the generator can weave the adaptive aspect in CIL code. The addition of the adaptive aspect at this level makes the TRAP.NET generator language independent and transparent with respect to the original source code.

When the generator is executed, it receives as input the annotated assembly. This assembly is immediately loaded into an *Assembly* object provided by the .NET reflection facilities. This object allows the discovery of the types and methods of the staged assembly. The generator iterates through the list of types and methods and finds all the methods with the *AdaptReady* attribute.

At this point, as illustrated in Figure 2, first the annotated assembly is disassembled using ILDASM, which is a tool distributed with the .NET Framework, to create a text file with the intermediate language (IL) code of the assembly. Next, the source code of the method, in IL, is used as a base for the generation of the adapt-ready IL code. Finally, the adapt-ready IL code is reassembled using ILASM, which generates the adapt-ready assembly. The process described in this paragraph is called round-tripping, which involves three steps: disassembly, tinkering with the CIL source code, and reassembly [8].

During the tinkering phase, the generator only adds the hooks to the methods that have the *AdaptReady* attribute. The actual hooks consist of a simple *if-then-else* statement to intercept and redirect the control flow. The *if*-condition intercepts the control flow and checks whether adaptation is enabled for that particular method. If it is, the automatically generated code inside the *if*-condition redirects the control flow. It loads and invokes the new adaptive functionality for this method using a dynamic method. The *else*-condition wraps the original functionality of the method which is executed if adaptation is not enabled.

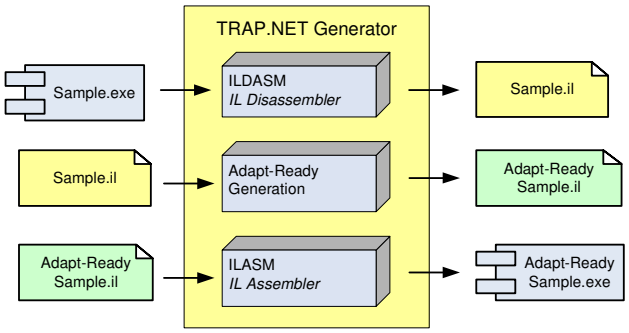

Figure 2. TRAP.NET Round-tripping.

Presenting the code of an adapt-ready method in CIL can be cumbersome due to the fact that intermediate language notation is very similar to assembly language which tends to be lengthy even for simplistic logic. Moreover, understanding CIL requires knowledge of CIL instructions and syntax. Therefore, for clarity, in Figure 3 we show what an adapt-ready method would look like in pseudo-code before and after generation.

```
[AdaptReady(true)]
SOME-METHOD()
1   call original implementation
```

(a) *Some-Method* before generation.

```
[AdaptReady(true)]
SOME-METHOD()
1   if this method is adapted
2       then call INVOKE-DYNAMIC-METHOD()
3       else call original implementation
```

(b) *Some-Method* after generation.

Figure 3. Adapt-ready method in pseudo code before and after generation.

Inside the *if*-condition, the generator has to perform code generation to invoke the dynamic method. The dynamic method has to be invoked with all the parameters of the original method which may be one or more and may be of different types. Also, the return type of the invocation needs to be casted to the return type of the original method. Using reflection the generator can determine the number and the types of parameters as well as the return

type of the method. Based on this information, the code to invoke the dynamic method is automatically generated.

Finally, besides adding the hooks, the generator also inserts a method call in the startup point of the application. When the adapt-ready application starts running, this method call is the first thing that gets executed. This method is part of the TRAP.NET class library and initializes all the components and data structures needed to support static and dynamic adaptation at runtime. As part of this initialization a communication channel is opened so the application can receive new functionality at runtime. This communication is implemented using *.NET Remoting*, which supports interprocess communication in distributed applications.

3.3. At Runtime

The last few steps to achieve dynamic adaptation with TRAP.NET occur at runtime when the adapt-ready application is executing. They are triggered when the user decides that the functionality of a particular adapt-ready method needs to adapt to some changing condition. Future work may involve automated decision making according to some policy. After this decision is made, the user develops the new functionality using the original source code. Then, the user can utilize the composer to upload the new adaptive functionality.

The composer is essentially a distributed application composed by two modules: the *client composer* and the *server composer*. The client composer is the user interface used to upload new functionality to the running application. The server composer—hosted by the TRAP.NET class library—is part of the running adapt-ready application and it serves requests from the client composer. The composer is the core of TRAP.NET because it achieves dynamic adaptation. Figure 5 shows the dependency of these components at runtime.

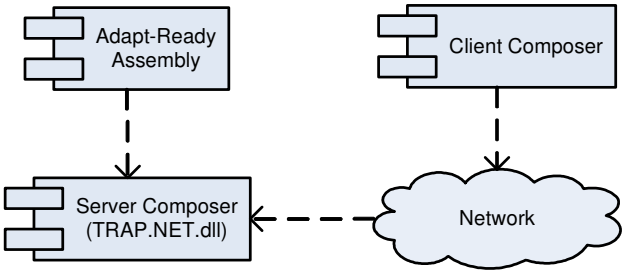

Figure 5. Dependency of components at runtime.

We developed two client composers, a Windows application composer and a Web application composer. Among all of their functionality, these composers perform three basic operations. The first one is to get the status of the adapt-ready application so the user can see which methods can be adapted. The second operation lets the user upload a new *delegate assembly* to the adapt-ready application. A delegate assembly is an assembly that contains *delegate methods*. Delegate methods contain the functionality that replaces the functionality of adapt-ready methods. Once the delegate assembly is loaded, the third operation lets the user adapt an adapt-ready method with a delegate method.

After the second operation, when user loads the delegate assembly, the server composer matches adapt-ready methods to their potential delegate methods candidates. Potential delegate methods are methods that have the same return type and parameter types as the adapt-ready method. Once this matching is complete, the results are returned in XML format to the client. The user interface then displays the adapt-ready methods with their potential delegate methods. At this time, the user may select one of the delegate methods for adaptation of its adapt-ready method. This adaptation request is sent to the server which prepares the contents of this delegate method so they are suitable for adaptation.

The preparation of the delegate method consists of the generation of a dynamic method with the contents and information of the delegate method. This generation is a complex process that relies heavily on reflection. The shell of the dynamic method has to have the same metadata as the delegate method. This metadata includes the parameter types, the declaring type and the local variables of the delegate method. After the shell of the dynamic method is completed, its contents are populated with the body of the delegate method. This body is the IL of the delegate method in byte code. Assuming the delegate method has no external references—that is, it only works with local variables and it does not call other methods—the newly created dynamic method is ready for usage. This dynamic method is stored in a data structure inside the server composer so it can be referenced when needed. The next time the adapt-ready method is called, it finds out that is has been adapted and its execution enters the *if*-condition which gets the dynamic method from the server composer. After the dynamic method is retrieved, the adapt-ready method invokes it with its current parameter values.

The process just described assumes that the delegate method had no external references to members outside the method itself. Non-trivial applications usually require access to external references. *Members* are any language element that can be referenced, i.e. methods, constructors, fields, properties and events. Each of these members may also have return types, parameters, access modifiers (such as *public* or *private*) and implementation details (such as *abstract* or *virtual*) among others. When a delegate method is developed and compiled it may reference members in the delegate assembly itself. After the delegate method is ported into a dynamic method in the running application, these references are still pointing to the delegate assembly. However, the delegate assembly is not being executed and it is out of the context of the running application thus, these references are really pointing to

nothing. Therefore invocation of a dynamic method with references to the delegate assembly would fail. The server composer solves this issue by redirecting these references to the running application.

To redirect references into the running application, the composer takes the following steps. First, it *finds* the references in the body of the delegate methods. After they are found, it *resolves* them, that is, it gets the actual delegate member being referenced so it can discover its signature. Using the signature of the delegate member, it *locates* the member with the same signature in the running application. Then, it *replaces* the reference in the delegate body so it points to the member in the running application.

The functionality that finds references in the body of delegate methods had to be developed from scratch. Reflection only provides access to the metadata of methods, not their actual contents. As a consequence, we developed a component, called the *token parser*, which finds external references in methods. Every member has a unique metadata identifier. These metadata identifiers—also referred to as metadata tokens—are used to uniquely reference members. The token parser extracts these references by parsing the byte code of delegate methods.

Using reflection and the tokens discovered by the token parser, the composer can resolve the members being referenced. After these members are resolved, it locates them in the running application. Once the composer locates these members, it can obtain their metadata identifiers or tokens. At this point the composer has the tokens that represent references in both assemblies. The composer then proceeds to replace tokens in the delegate body.

After replacement is completed, the new body is assigned to the dynamic method. The references in the dynamic method body have now been redirected to members in the running application. Now, the dynamic method is ready for invocation.

So far we have presented how dynamic adaptation is achieved at runtime. TRAP.NET can be used in a mode that enables static adaptation. This type of adaptation occurs at load time by reusing the functionality that achieves dynamic adaptation. To achieve static adaptation the generator adds code to the startup of the application to pause it as soon as it starts executing. While the application is paused, the composer is the only available component which is waiting to receive delegate methods. The user then sends delegate methods and adapts the desired adapt-ready methods. The composer then flags these methods as adapted and will not allow new adaptive functionality during the execution of the program. When the user wishes, the application will be resumed.

4. RELATED WORK

Similar to TRAP.NET, other approaches to build adaptable programs involve *intercepting* calls to functional code, and *redirecting* them to adaptive code [1]. There are two main categories of related work. The first category involves approaches that *extend middleware* to support adaptive behavior (*e.g.,* Iguana/J [3] that extends JVM). Since the role of traditional middleware is to hide resource distribution and platform heterogeneity from the business logic of applications; it is a natural place to put adaptive behavior. However, these approaches generally become outdated after a newer version of the middleware (*e.g.,* JVM) is released.

The second category includes approaches to transparently *augment the application code* with facilities for interception and redirection. Examples related to our work include AspectJ [4], Aspect.NET [5] and TRAP/J [2]. AspectJ and Aspect.NET enable aspect weaving at compile time which is similar to the task the TRAP.NET generator performs. However, they do not provide a means for dynamically weaving new code into the application at runtime.

TRAP/J is an instance of TRAP in Java. To augment an existing Java program with the required hooks, TRAP/J uses compile-time aspect weaving provided by AspectJ. Following the TRAP approach, TRAP/J operates in two phases. In the first one, at compile time, TRAP/J converts an existing program into an adapt-ready program. This conversion is accomplished using an Aspect Generator and a Reflective Class Generator. These generators produce aspects and the reflective classes. Next, these two products, with the original source code, are passed to the AspectJ compiler which weaves the generated and the original source code together to generate and adapt-ready assembly. Note that the generation process in TRAP/J generates significantly more code than the generation process in TRAP.NET. The second phase occurs at runtime when using the reflective classes, new behavior can be introduced to the application.

A limitation of TRAP/J is the fact that it can only make classes adapt-ready. Even if the user only wishes to make one method in a class adapt-ready, TRAP/J will make the entire class adapt-ready. Performance and flexibility seem affected by this limitation. In contrast, TRAP.NET overcomes this limitation since it is able to make methods adapt-ready. Moreover, methods are the units of functionality and behavior in the object oriented paradigm. Since transparent shaping focuses on changing the business logic which is hosted by methods, TRAP.NET offers a more natural implementation. The state of the object is not changed by adaptation itself. Adapting a method only changes its functionality. The new adaptive functionality may change the state of the object as it was programmed by the user.

Other contributions of TRAP.NET include its language independence, tailoring of the transparent shaping model to .NET development practices and extensive member access. TRAP.NET can make any application

adapt-ready, independently of the language that it was developed since it works at the intermediate language level. Also, by using attributes it customizes the transparent shaping model so it fits .NET development practices. Moreover, TRAP.NET is the implementation that offers the most types of member access to the adaptive functionality. The new adaptive functionality may need to access resources or members in the running application. TRAP.NET enables this accessibility.

5. CONCLUSIONS

The next major step for TRAP.NET should be the development of a case study with an application geared towards pervasive, autonomic or grid computing. This case study will produce important results to improve and evaluate TRAP.NET in many aspects including performance, reliability and usability.

Safe adaptation and security are two main concerns that remain pending in this research. Safe adaptation is the ability of a program to maintain its integrity during adaptation [9]. And, security deals with protecting an adaptable program from malicious entries [1]. These two issues are ongoing research areas for dynamic adaptation. Future work in TRAP.NET should support safe adaptation and security as these techniques become available.

The major achievement of this paper is the research on the design and development of TRAP.NET. This tool is a successful realization of transparent shaping using the TRAP model that provides dynamic adaptation of applications without the need to modify their original functionality. TRAP.NET provides the means to achieve separation of concerns when developing adaptive applications. It adheres to the aspect-oriented paradigm by adding adaptive functionality across the business logic of an application. It uses reflection to discover and modify the functionality of an application. Moreover, unlike similar tools, it intercepts only the methods selected by the user. The rest of the application remains intact maintaining its original performance. Adaptive code in TRAP.NET is achieved by developing delegates which can provide its own new functionality and reuse the already existing functionality of the adapt-ready application. This important achievement contributes to the developing area of software adaptation in order to support critical and long-time running applications such as the ones found in pervasive, autonomic and grid computing.

ACKNOWLEDGMENTS

This work was supported in part by IBM, the National Science Foundation (grants OCI-0636031, REU-0552555, and HRD-0317692). The authors are very thankful to the following students who played a significant role in the implementation of TRAP.NET: Allen Lee, Tuan Cameron, Ana Rodriguez, Juan H. Cifuentes, Javier Ocasio, Amit Patel, Mitul Patel, Enrique E. Villa, Frank Suero, Etnan Gonzalez, Edwin Garcia, Alain Rodriguez, and Lazaro Millo.

TRAP.NET is a follow-up work on the Transparent Shaping research originated in Michigan State University and the authors are thankful to Philip McKinley, Betty Chen, and Kurt Stirewalt who contributed to the original ideas in Transparent Shaping, its TRAP extension, and the realization of TRAP in Java, called TRAP/J. Our gratitude is also extended to our colleagues at Florida International University, Peter Clarke and Masoud Milani, who provided us with feedback on this work.

REFERENCES

[1] S. M. Sadjadi. Transparent Shaping of Existing Software to Support Pervasive and Autonomic Computing. *A Dissertation submitted to Michigan State University*, 2004.

[2] S. Masoud Sadjadi, Philip K. McKinley, Betty H.C. Cheng, and R.E. Kurt Stirewalt. TRAP/J: Transparent generation of adaptable Java programs. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, Agia Napa, Cyprus, October 2004.

[3] Redmond, B., Cahill, V.: Supporting unanticipated dynamic adaptation of application behavior. *In Proceedings of ECOOP*, 2002.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag LNCS 1241, June 1997.

[5] Vladimir O. Safonov. Aspect.NET – an aspect-oriented programming tool for Microsoft.NET. In *Microsoft Research SSCLI RFP II Capstone Workshop 2005*, September 2005.

[6] .NET Framework. *Wikipedia*. 2 March 2007. Available at URL: http://en.wikipedia.org/wiki/ .NET_ framework.

[7] Attributes Overview. *MSDN Library for Visual Studio 2005*. 2005. Available at URL: http:// msdn.microsoft.com/library/en-us/cpguide/html/ cpconattributesovervi-ew.asp.

[8] Serge Lidin. Expert .NET 2.0 IL Assembler. Apress. 2006, pages 389-408.

[9] Ji Zhang, Zhenxiao Yang, Betty H.C. Cheng, and Philip K. McKinley. Adding safeness to dynamic adaptation techniques. In *Proceedings of the ICSE 2004 Workshop on Architecting Dependable Systems*, Edinburgh, Scotland, May 2004.