International Journal of Software Engineering
and Knowledge Engineering
Vol. 19, No. 4 (2009) 507–528

World Scientific
www.worldscientific.com

TRAP.NET: A REALIZATION OF TRANSPARENT SHAPING IN .NET

S. MASOUD SADJADI*

School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA
sadjadi@cs.fiu.edu
http://www.cs.fiu.edu/∼sadjadi/

FERNANDO TRIGOSO

1317 St. Tropez Cir. #1306
Weston, FL 33326, USA
fernando_trigoso@ultimatesoftware.com

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.

Keywords: Transparent shaping; dynamic adaptation; static adaptation; TRAP.NET; .NET Attributes; reflection; metadata; reference redirection; adapt-ready; .NET Framework; CIL; ILDASM; ILASM.

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

*Corresponding author.

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 effectiveness 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 classifies TRAP.NET based on how, when and where software composition takes place. Section 5 describes the research challenges we faced. Section 6 demonstrates how to use TRAP.NET using an example application and shows its performance overhead is negligible. Section 7 compares TRAP.NET to some related work. Finally, Sec. 8 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 precoded solutions to common program requirements in the form of class libraries. It also manages the execution of programs written specifically for this framework [2]. 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 [3]. 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 *AdaptReady* attribute to automatically make them adapt-ready. The code snippet in Fig. 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

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

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

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.

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 from 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 Fig. 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 [4].

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.

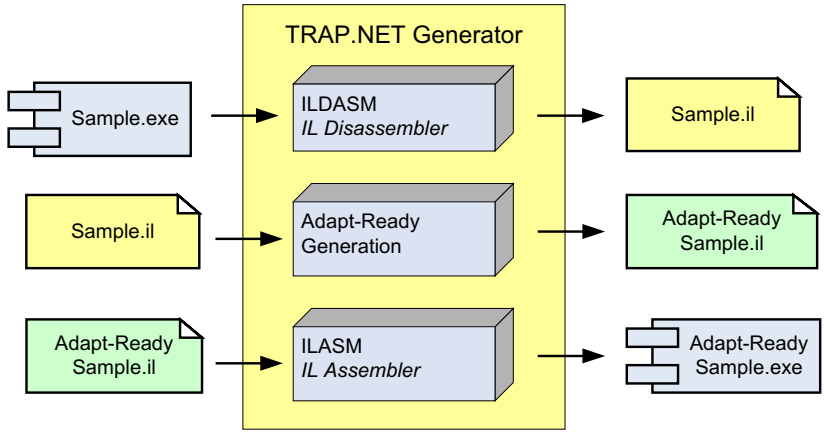

Fig. 2. TRAP.NET Round-tripping.

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.

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 Fig. 3 we show what an adapt-ready method would look like 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

```
[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.

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

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 4 shows the dependency of these 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

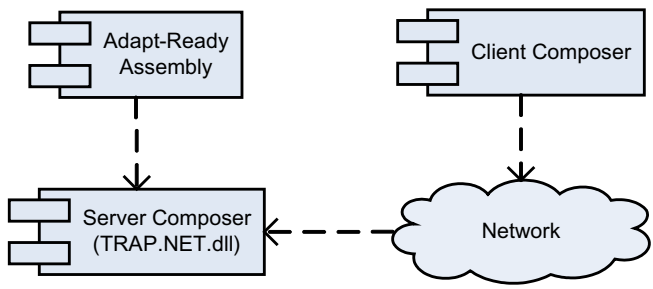

Fig. 4. Dependency of components at runtime.

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. The reference redirection process is explained in the next section which discusses member access in more detail.

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.

3.4. *Member access*

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. Figure 5 illustrates a dynamic method with external references before redirection.

3.4.1. *Reference redirection*

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 reference 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. Figure 6 shows the reference redirection steps.

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 by parsing the byte code of delegate methods.

Every member has a unique metadata identifier. These metadata identifiers — also referred to as metadata tokens — represent member references. These tokens

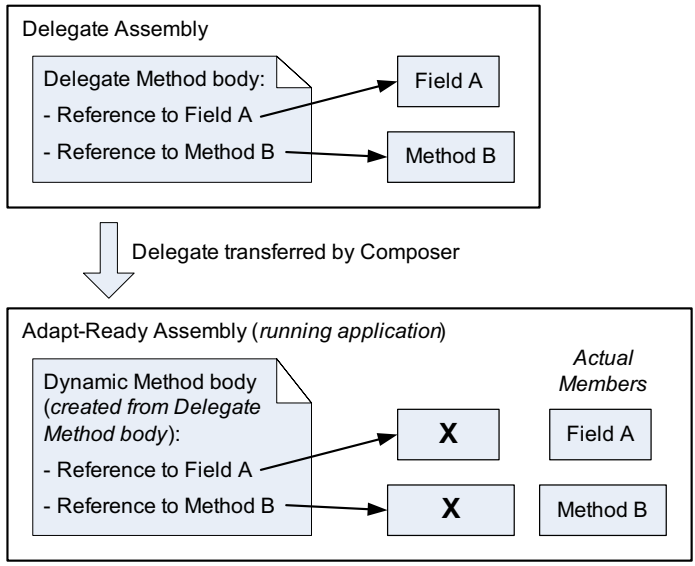

Fig. 5. Dynamic method before reference redirection. Notice that references in the dynamic method point to 'X' which represents nothing.

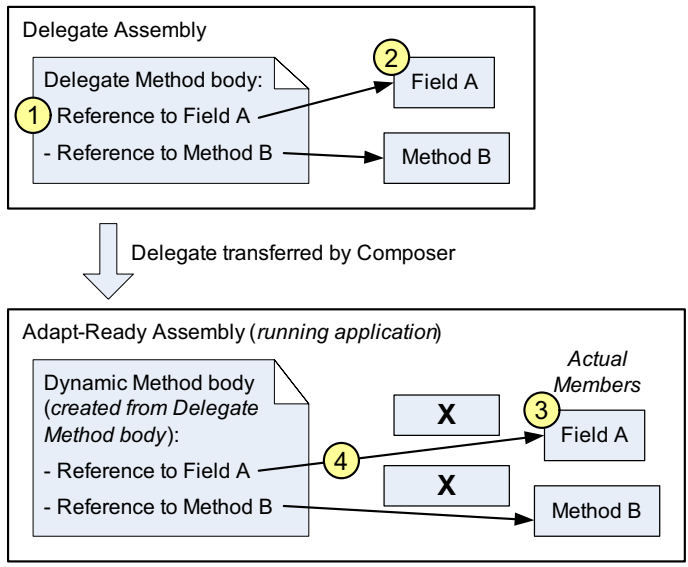

Fig. 6. Reference redirection steps. Step 1, find references; step 2, resolve members; step 3, locate members in adapt-ready assembly and; step 4, replace references.

can be extracted from the byte code of delegate methods. The byte code of methods can be obtained using ILDASM which places the corresponding byte code of each IL instruction as comments inside the method code.

The token parser is the component that finds metadata tokens by parsing the body of delegate methods. When the user selects a delegate method for adaptation, the composer disassembles the delegate assembly using ILDASM. The token parser then parses the contents of the delegate method looking for these tokens.

After the tokens — which represent references — are found the composer needs to resolve their corresponding members. It uses the method *ResolveMember* in the *Module* class to resolve the members in the delegate assembly. This method takes a token identifier as a parameter and returns an object that encapsulates the member. The return object could be either of these types: *MethodInfo*, *ConstructorInfo*, *FieldInfo*, *PropertyInfo* or *EventInfo*. These reflection types provide all the information we need to locate the member with the same signature in the running application.

To locate a member the composer uses the methods *GetMember*, *GetMethod*, *GetField*, etc. in the *Type* class. The composer calls these methods with respect to the adapt-ready assembly so they return members in this assembly. These methods take a member signature and return the member with that signature. Therefore, the signature the composer passes to these methods is the signature of a member in the delegate assembly and the return value is the corresponding member in the adapt-ready assembly.

As mentioned earlier, tokens represent member references. Therefore, to replace a reference the composer needs to replace its token. Moreover, tokens are part of the byte code of a method. Hence the composer can replace tokens referring to members in the delegate assembly with tokens referring to members in the adapt-ready assembly at the byte code level. To achieve this replacement the composer needs two pieces of information.

First, it needs to know the token of the member in the adapt-ready assembly. This token can be easily retrieved since we already know the member metadata information which includes the token identifier. Second, it needs to know the offset in the byte code at which to execute the replacement. The token parser calculates this offset, which at the time of finding references also calculates the offset in the byte code at which delegate references appear.

With these two pieces of information available, the composer proceeds to replace the tokens in the body of the delegate method. 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.

3.4.2. *Current member access scope*

The member access scope is limited by the composer's ability to locate members. At the time of this writing, the functionality to locate members in the adapt-ready

application was almost complete. Different types of members require different calls with different parameters to the reflective methods used to find members. Future work is expected to complete this functionality. The following list shows the type of members that can be located with the current functionality of the composer.

- *Local fields* are fields in the same class the adapt-ready method belongs to; the composer can locate public, protected and private instance and static local fields.
- *Local methods* are methods in the same class the adapt-ready method belongs to; the composer can locate public, protected, private, internal and protected internal instance and static methods. It can also locate overloaded methods.
- *External fields* are fields in classes different from the class of the adapt-ready method; the composer can locate public instance and static external fields.
- *External methods* are methods in classes different from the class of the adapt-ready method; the composer can locate public instance and static external methods. It can also locate overloaded methods.
- *External constructors* are constructors of classes different from the class of the adapt-ready method; the composer can locate public external constructors. Static constructors can only be called by the runtime. It can also locate overloaded constructors.
- *Parent fields* are fields declared in the base class of the class the adapt-ready method belongs to; the composer can locate public and protected instance and static parent fields.
- *Parent methods* are methods declared in the base class of the class the adapt-ready method belongs to; the composer can locate public, protected, internal and protected internal instance and static parent methods. It can also locate abstract, virtual and new public parent methods. It can also locate overloaded parent methods.

The functionality to locate members has passed testing with at least one sample of each member type listed above. A few features are missing that will allow the composer to access all the types of members in the adapt-ready application. Some of these features are listed below.

- Property and Event access.
- Access to member in other modules that are part of the adapt-ready application.

4. TRAP.NET Classification

This section provides a classification of TRAP.NET using the taxonomy introduced by McKinley *et al.* [5] based on *how*, *when* and *where* software composition takes place.

4.1. *How to compose and TRAP.NET*

The first dimension of this taxonomy is *how* composition is implemented, that is, what specific mechanisms are used to enable compositional adaptation. There are a

variety of techniques available. Among those we have: function pointers, wrappers, proxies, metaobject protocol, aspect weaving, middleware interception, integrated middleware, etc. TRAP.NET uses two of these techniques — metaobject protocol and aspect weaving.

A *metaobject protocol* is a mechanism supporting intercession and introspection that enable modification of program behavior. TRAP.NET uses two main metaobject protocols: the .NET reflection components and dynamic methods. The reflection components in .NET are used to discover information about the program; and the dynamic methods are used to modify the program's behavior.

Aspect weaving consists in weaving code fragments that implement crosscutting concerns into an application dynamically. The TRAP.NET Generator performs the aspect weaving of code that supports dynamic adaptation. It generates and inserts code fragments into methods according to their signature. Moreover, the generator adds the adaptive crosscutting concern at compilation time to the intermediate language code of the application. Therefore, the original source code is never modified.

4.2. *When to compose and TRAP.NET*

McKinley *et al.* [5] also differentiate approaches according to when the adaptive behavior is composed with the business logic. They state the following: "Generally speaking, later composition time supports more powerful adaptation methods, but it also complicates the problem of ensuring consistency in the adapted program. For example, when composition occurs at development, compile, or load time, dynamism is limited but it is easier to ensure that the adaptation will not produce anomalous behavior. On the other hand, while runtime composition is very powerful, it is difficult to use traditional testing and formal verification techniques to check safety and other correctness properties." From this argument, they further divided composition into two main approaches: static composition and dynamic composition.

In *static composition*, also called static adaptation, software is composed at development, compile, or load time. If an adaptive program is composed at development time, then any behavior is *hardwired* into the program and cannot be changed without recoding. Adaptive behavior can also be added at compile time with aspect-oriented languages such as AspectJ. Different aspects can be weaved for different environments at compile or link time. These applications are called *customizable* and they only require recompilation to fit to a new environment. *Configurable* applications delay the final adaptation decision until the application loads a particular component. Although composition at load time is still a form of static composition, it is very useful for applications that run on different environments. These type of applications only need to adapt to its environment when they area loaded. For example, if a user starts an application in a handheld device, the display components loaded are going to be different than the ones loaded if the same application was executed on a regular desktop computer.

In *dynamic composition*, also called dynamic adaptation, software is composed by a composer at runtime. In this case, a composer could replace or modify algorithmic and structural units during execution without halting or restarting the program. Additionally, they differentiate two types of dynamic composition approaches according to whether the composer is allowed to change the business logic of the application. *Tunable software* prohibits the modification of code for the business logic. It only supports fine-tuning of crosscutting concerns. In contrast, *mutable software*, allows the composer to change any functionality of the program's components, including the business logic functionality.

TRAP.NET follows the static and dynamic composition approaches. To achieve static composition, the composer only allows adaptation when the application loads. To achieve dynamic composition, the composer allows changing the implementation of the business logic of the application at run time. Therefore, TRAP.NET creates both configurable and mutable software.

4.3. *Where to compose and TRAP.NET*

This is the final dimension in the taxonomy presented by McKinley *et al.* It focuses on where in the system the composer inserts the adaptive code. There are three main areas where this insertion may occur: the middleware layers, the application code or the operating system. Our work focuses in the middleware layers and the application code since these are the areas that affect TRAP.NET.

Middleware layers can provide facilities that aid recomposition at runtime. For example, the Java Virtual Machine (JVM) and common language runtime (CLR) assist dynamic recomposition through reflection facilities provided by the Java language and the .NET framework. In the case of TRAP.NET, these reflection facilities are extensively used by the generator and composer. However, the adaptive code is not inserted in these layers.

The adaptive code can also be inserted in the application code in two ways. The developer can explicitly add the code to support dynamic adaptation or the adaptive code can be woven into the functional code. TRAP.NET follows the latter approach. The generator weaves the adaptive aspect at compile time and the composer inserts the adaptive code at runtime. The TRAP.NET approach offers a way to add adaptive behavior to applications transparently with respect to its original code.

5. Research Challenges

We faced several challenges in the research of how to implement transparent shaping for the .NET framework. Previous implementations of transparent shaping worked with the original source code and were language dependent. Even though .NET caters many languages, it also centralizes everything in a lower layer of abstraction at the Common Intermediate Language level. This centralization gave us the

opportunity to make this particular implementation of transparent shaping language independent. The fact that we chose to work with the CIL level was another challenge.

Intermediate language resembles assembly code thus it is more complicated to use and understand than a fourth-generation language such as Java or C#. In order to achieve language independence, we had to become familiar with the intermediate language syntax and usage. After we learned enough about intermediate language, we researched a way to add the adaptive aspect at this level. To add the adaptive aspect at this level, we followed the interception and redirection principles and investigated a way to implement them in IL code. We believe this approach was the harder path to take but it was the right one. The easier path would have been providing different implementations of transparent shaping for different .NET languages. We may have ended up with TRAP/C#, TRAP/VB, TRAP/J#, etc. These different implementations would not have provided any new contributions. Implementing transparent shaping at a higher-level language has already been performed with TRAP/J and TRAP/C++. Instead we chose to provide a more difficult implementation that applications written in different languages can use.

Another challenge was making this implementation more transparent to the user. Other implementations of transparent shaping present the user with two different user interfaces — one for the generator and another one for the composer. The user interface for the generator actually displays all the methods and classes in an assembly. From this list the user can select the methods or classes to make adapt-ready. This approach is not very common in the .NET world. To overcome this obstacle we researched the .NET development practices and decided to use attributes so the user can annotate methods to make adapt-ready. Attributes are very common in .NET and they are used in a variety of ways.

The next challenge we had to overcome was the fact that delegate methods may reference members in the delegate assembly. The problem arises when a delegate method is ported into the executing adapt-ready application. The member references of the delegate method are not in the execution context of the running application. Therefore we investigated and implemented a technique to redirect the references from the delegate assembly to the executing assembly.

6. TRAP.NET Case Study

This section will show how to provide dynamic adaptation to a sample application. The application used in this example sorts and searches a set of numbers which are received from an external process. For the sake of the example let's assume that once deployed this application shall not be stopped. However, to allow for dynamic change, the developers of this sample application made it adapt-ready using TRAP.NET. When first deployed the external process was sending an unsorted list of 5 million numbers. The sample application would sort them and then search for a subset of those numbers. After some time, the external process

starts sending a sorted list of 5 million numbers. Since the sample application does not know that the numbers are already sorted, it continues to sort them and search for a subset of those numbers. Using TRAP.NET we can improve the performance of this application at run time by providing a new implementation of the method that sorts and searches. The new implementation of this method would just search for a subset of these numbers. Obviously, this is very simplistic scenario; nevertheless, it proves the usability of TRAP.NET.

The following steps show how to use TRAP.NET on this sample application to make it adapt-ready. First, as shown in Fig. 7, we decorate the method that searches numbers with the *AdaptReady* attribute. The implementation of method *SearchNumbers* is rather complex. It calls the methods *Sort* and *Search*, it also uses local and class variables and it returns a *long* data type. We intentionally added this complexity to demonstrate some of the member access that TRAP.NET supports.

After the development of the sample application is completed. It can be built and then made adapt-ready with the TRAP.NET Generator which can be used through a Visual Studio add-in or a standalone windows application. Figure 8 shows the user interface of the Visual Studio add-in.

```
[AdaptReady(true)]
public long SearchNumbers()
{
    long timeToSort = Sort();

    _allNumbersFound = Search();

    return timeToSort;
}
```

Fig. 7. Sample application method with *AdaptReady* attribute.

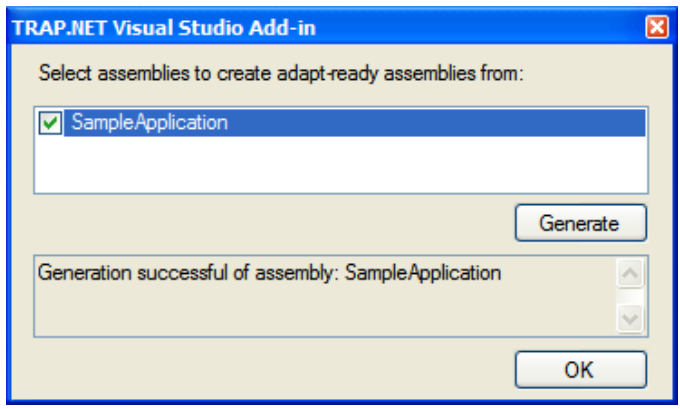

Fig. 8. TRAP.NET Visual Studio add-in.

```
[AdaptReady(true)]
public long SearchNumbers()
{
    long timeToSort = 0;

    _allNumbersFound = Search();

    return timeToSort;
}
```

Fig. 9. New implementation of *SearchNumbers* to avoid sorting.

The Sample Application is now adapt-ready, it can be deployed and start receiving numbers from the external process. When we decide to change its behavior we can develop the delegate with the new implementation of its *SearchNumbers* method. Figure 9 shows the new implementation.

Using the client composer we can access the server composer which is part of the running sample application. The endpoint of the server composer was set by another attribute that was placed in the main method of the Sample Application before deployment. The client composer can be exposed through a web or windows application. Figure 10 shows the windows application user interface of the client composer. This interface shows the server composer endpoint, the delegate assembly to be loaded and adapt-ready methods mapping.

After the user clicks adapt, the client composer would send the delegate method to the server composer through the network which would then flag the

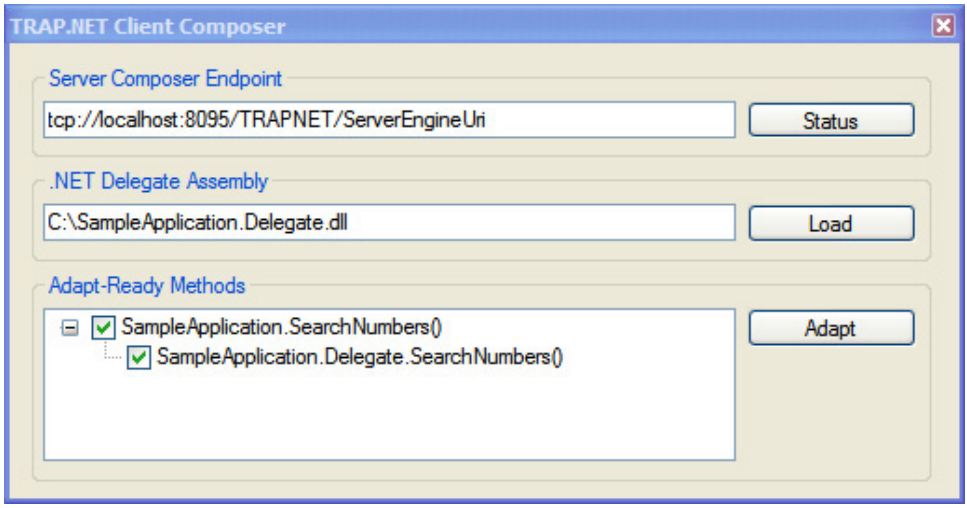

Fig. 10. TRAP.NET Client composer.

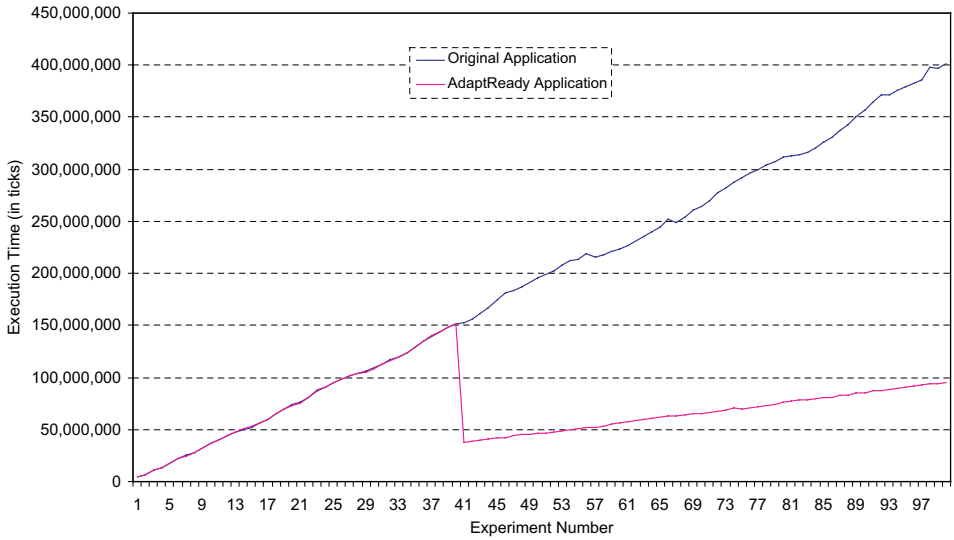

Fig. 11. Performance analysis; the adapt-ready application is adapted at the 40th experiment.

SearchNumbers method as adapted. The next time the runtime calls the *SearchNumbers* method, the new implementation, provided by the delegate, will be used.

We analyzed this application with and without TRAP.NET. We executed the original application several times with different number arrays recording these results. Then we executed the adapt-ready version of the original application. We noticed that the executions times were extremely close, thus there was not a significant performance penalty. The reason being that TRAP.NET adds an $O(1)$ constant complexity. Then we adapted the adapt-ready application so it would benefit the new delegate logic or searching numbers. Obviously, we saw a dramatic decrease in the execution time as compared to that of the original application. For this case study, TRAP.NET proved to be very useful as it did not decrease the performance of the adapt-ready application and it drastically increased the performance of the adapted application. Figure 11 shows the results of our performance analysis.

7. 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. For example, Iguana/J [6] extends JVM; mKernel [7] extends EJB; DynamicTAO [8] extends TAO [9], Open ORB [10], QuO [11], Orbix [12], ORBacus [13], IRL [14], Eternal [15], and ACT [16] extend CORBA. 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 [17], Aspect.NET [18], Hyper/J [19], DemeterJ (DJ) [20], JAC [21], Composition Filters [22], ARCAD [23, 24], Reflex [25], Kava [26], Dalang [27], Javassist [28], and TRAP/J [29]. 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 support to all .NET programming languages, 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 .NET language that it was developed in 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.

Finally, we would like to acknowledge other useful frameworks, such as RAppia [30], that allow the developer of adaptive and autonomic systems to use predefined generic components for example to instantiate the most appropriate strategy to different configurations, without having to re-implement a customized solution for each configuration. Such approaches although are very helpful for developing new adaptive and autonomic systems, they do not address incorporation of adaptation logic in existing system with no source code available. Of course, the generic and pluggable components of such frameworks can be incorporated to existing applications using TRAP/J and TRAP.NET facilities without the need of manual modification to the source code; hence, we consider TRAP approach to be complementary to these frameworks. Note that the rationale behind using .NET Attribute at development time is just for developer's convenience and TRAP.NET really does not need to have access to the source code.

8. Conclusion

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 [31]. 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 OISE-0730065, OCI-0636031, IIS-0552555, and HRD-0317692). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the NSF and IBM. 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. .NET Framework, *Wikipedia*, 2 March 2007, available at URL: http://en.wikipedia.org/wiki/ .NET_framework.
3. Attributes Overview, *MSDN Library for Visual Studio 2005*, 2005, available at URL: http://msdn.microsoft.com/library/en-us/cpguide/html/cpconattributesoverview.asp.
4. Serge Lidin, Expert .NET 2.0 IL Assembler (Apress. 2006) pp. 389–408.
5. Philip K. McKinley, S. Masoud Sadjadi, Eric P. Kasten and Betty H. C. Chen, Composing Adaptive Software, *Computer* (2004), pp. 56–64.
6. B. Redmond and V. Cahill, Supporting unanticipated dynamic adaptation of application behavior, in *Proceedings of ECOOP*, 2002.
7. J. Bruhn and G. Wirtz, mKernel: A manageable kernel for EJB-based enterprise applications, in *Proceedings of the First International Conference on Autonomic Computing and Communication Systems (Autonomics 2007)*, 2007.
8. F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. C. Magalhães and R. H. Campbell, Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB, in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms* (Middleware 2000), New York (2000).
9. D. C. Schmidt, D. L. Levine and S. Mungee, The design of the TAO real-time object request broker, *Computer Communications* **21** (1998) 294–324.
10. G. S. Blair, G. Coulson, P. Robin and M. Papathomas, An architecture for next generation middleware, in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England (1998).
11. J. A. Zinky, D. E. Bakken and R. E. Schantz, Architectural support for quality of service for CORBA objects, *Theory and Practice of Object Systems* **3** (1997).
12. IONA Technology, (Orbix) available at URL: http://www.iona.com/products/orbix.htm.
13. IONA Technologies Inc., ORBacus for C++ and Java version 4.1.0. (2001).

14. R. Baldoni, C. Marchetti and A. Termini, Active software replication through a three-tier approach, in *Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems (SRDS02)*, Osaka, Japan, 2002, pp. 109–118.
15. L. Moser, P. Melliar-Smith, P. Narasimhan, L. Tewksbury and V. Kalogeraki, The eternal system: An architecture for enterprise applications, in *Proceedings of the Third International Enterprise Distributed Object Computing Conference (EDOC'99)*, 1999.
16. S. M. Sadjadi and P. K. McKinley, ACT: An adaptive CORBA template to support unanticipated adaptation, in *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan, 2004.
17. 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)*, LNCS 1241 (Springer-Verlag, 1997).
18. Vladimir O. Safonov, Aspect.NET — an aspect-oriented programming tool for Microsoft.NET, in *Microsoft Research SSCLI RFP II Capstone Workshop 2005*, September 2005.
19. H. Ossher and P. Tarr, Using multidimensional separation of concerns to (re)shape evolving software, *Communications of the ACM* **44** (2001) 43–50.
20. K. Lieberherr, D. Orleans and J. Ovlinger, Aspect-oriented programming with adaptive methods, *Communications of the ACM* **44** (2001) 39–41.
21. R. Pawlak, L. Seinturier, L. Duchien and G. Florin, JAC: A flexible and efficient solution for aspect-oriented programming in Java, in *Proceedings of Reflection 2001*, LNCS 2192, 2001, pp. 1–24.
22. L. Bergmans and M. Aksit, Composing crosscutting concerns using composition filters, *Communications of ACM* (2001) 51–57.
23. P. C. David, T. Ledoux and N. M. N. Bouraqadi-Saadani, Two-step weaving with reflection using AspectJ, in *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, 2001.
24. Z. Jarir, P.-C. David and T. Ledoux, Dynamic Adaptability of Services in Enterprise JavaBeans Architecture, in *Seventh International Workshop on Component-Oriented Programming (WCOP'02)* 2002.
25. E. Tanter, J. Noy'e, D. Caromel and P. Cointe, Partial behavioral reflection: Spatial and temporal selection of reification, in R. Crocker and G. L. Steele, Jr., (eds.): *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*, Anaheim, California, 2003, 27–46.
26. I. Welch and R. J. Stroud, Kava — A reflective Java based on bytecode rewriting, in W. Cazzola, R. J. Stroud and F. Tisato, (eds.), *Reflection and Software Engineering*, LNCS 1826 (Springer-Verlag, Heidelberg, 2000), pp. 157–169.
27. I. Welch and R. Stroud, Dalang — a reflective extension for Java, technical report CS-TR-672, University of Newcastle upon Tyne, East Lansing, Michigan (1999).
28. S. Chiba, *Load-time Structural Reflection in Java*, LNCS 1850, 2000.
29. 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.
30. L. Rosa, L. Rodrigues and A. Lopes, A framework to support multiple reconfiguration strategies, in *Proceedings of the First International Conference on Autonomic Computing and Communication Systems (Autonomics 2007)*, 2007.
31. J. Zhang, Z. Yang, Betty H. C. Cheng and P. K. McKinley, Adding safeness to dynamic adaptation techniques, in *Proceedings of the ICSE 2004 Workshop on Architecting Dependable Systems*, Edinburgh, Scotland, May 2004.