TRAP/J v2.1: An improvement for Transparent Adaptation

Technical Report FIU-SCIS-2007-09-01
May 2007

S. Masoud Sadjadi, Luis Atencio, and Tatiana Soldo
Autonomic and Grid Computing Research Laboratory
School of Computing and Information Sciences
Florida International University, 11200 SW 8th Street, Miami, FL 33199
{sadjadi, laten001,tsold001}@fiu.edu

Abstract

With the advent of mobile, pervasive, and grid computing, software systems must be designed to dynamically adapt to changes that might occur in their runtime environments. Certainly, careful system design and modeling are key factors for systems to be complete. However, as technology changes and new forms of technology continue to emerge, predetermining all possible scenarios in which a system may be running is nothing short of impossible.

These issues can be addressed with a tool called TRAP/J (Transparent Reflective Aspect Oriented Programming in Java. However, the first implementation of this tool performs poorly on demanding applications, severely lacked usability, and provided very limited support for adaptation. In this paper, we will be addressing various issues in the first implementation of TRAP/J and we have developed a new version, TRAP/J v2.1, which is aimed at providing better performance and usability over the original TRAP/J.

TRAP/J 2.1 is focused on improving the performance of the generation and adaptation phases of Transparent Adaptation and keeping in mind ease of usability. This will allow a decision support system –in our case, a user— to benefit from a user friendly, interactive, web based Composer Interface through which new behavior can be inserted into an application remotely at runtime or startup time. In addition, it has a Generator Interface that allows users to choose which classes they wish to make adaptable.

1. Introduction

Transparent Adaptation has definitely been a big step in the design of autonomic systems, which promise to resolve the ever increasing complexity and size of software systems developed today. This is, in big part, given by the need of systems to adapt or alter their behavior autonomically in response to high level human policies and free users of many potentially conflicting concerns like quality-of-service (QoS), security, availability, battery life, network failures, etc. Thus, not all decisions must be made at the time of design of a complex system; needs for improvements and enhancements can later be addressed via adaptation tools. In grid computing, for example, the physical runtime environment can change according to the discovery and failure of resources –number of nodes, memory; or even programmatically such as the insertion of a new optimal algorithm that takes advantage of grid resources more efficiently. All of these factors can potentially change depending on the execution environments in which systems are in; if need be, more efficient algorithms can be developed by experts separately and then inserted when application is executing. For more information on this topic, please refer to Transparent Grid Enablement (TGE) where TRAP/J 2.1 is used to accomplish this.

Furthermore, tools that enable transparent adaptation such as TRAP/J should be efficient. If adaptation would incur so much overhead that the overall performance of the system suffers dramatically, in these cases adaptation will not be feasible. In other words, the performance of an application that is being adapted should be very close to the original application. TRAP/J 2.1 enhances the overall performance of its predecessor by shortening both its compile-time and runtime architectures. Basically, the critical execution path of the system has been redesigned much shorter and efficient.

In addition, TRAP/J 2.1 is now very user friendly. Using a client-server architecture it benefits users with a web based GUI Composer Interface that clearly shows the runtime status of the adapt-ready program including its classes, instances created, adaptive behavior (delegates) being used, and others. This gives users an opportunity to make clear decisions on where and when to insert adaptable behavior. To allow for adaptable behavior to be inserted at startup time or runtime, the adapt-ready application can be started in two modes: run mode, paused mode. During run mode, users can interact with the adapt-ready application through the GUI interface and insert adaptive code at runtime. In paused mode, user can start the application with default adaptable behavior. The Generator Interface allows users to choose from a project a subset of classes that they want to make adaptable, including any standard java library classes. These classes will essentially form the adapt-ready application.

In future work, we will be discussing new and more automated approaches to Transparent Adaptation as well as adding a decision-support system that could monitor performance of the adapt-ready application and make decisions as to where and which type of adaptation should take place in compliance with high level human guidance.

2. Background Information

Transparent Adaptation or Transparent Shaping allows applications to change their behavior statically (at startup time) or dynamically (at runtime) without any modifications to the original source code. TRAP/J combines a software engineering paradigm, called Aspect-Oriented Programming (AOP) and Structural Reflection, to provide the capabilities for Transparent Shaping, enhance modularization of code, and promote code reuse.

TRAP/J makes extensive use of a component called AspectJ which extends Java with features that realize the Aspect-Oriented Programming (AOP) and Aspect-Oriented Software Development (AOSD) paradigms in Java. AOP is the means by which TRAP/J enhances the level of modularization and reduces code entanglement or spaghetti code provided by the implementation of crosscutting concerns. For example, non-functional concerns such as profiling, tracing, security, or Quality of Service (QoS) often crosscut horizontally several different modules, as in figure 1.

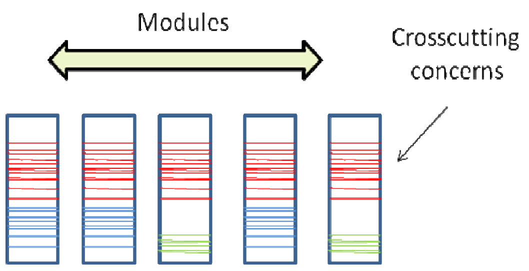


Figure 1. Non-functional concerns crosscutting among different modules

Furthermore, through the AspectJ compiler (ajc) generic hooks are weaved into the application at compile time. These hooks (called join points) are the points where the desired crosscutting functionality would take place. In AOP, the `aspect` is the basic unit of abstraction (much like classes in Java) and is composed of two constructs (`pointcut` and `advice`). Pointcuts sense when a hook has been reached in program execution, and it is at that point where it redirects the call to an advice that contains the implementation of the new functionality.

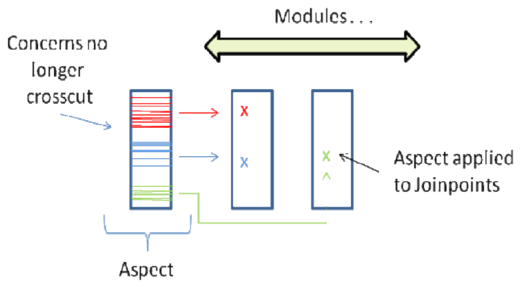


Figure 2. Aspects place hooks on source code and modularize crosscutting functionality

In addition, reflective programming, typically common in languages that run in a virtual machine environment, allow programs to introspect (observe) and possibly modify its structure and behavior at runtime. In other words, through reflection, a developer can obtain information about the structure of a class such as its methods and fields. With this information (metadata), reflection allows developers to create classes and instances at runtime.

Furthermore, we will briefly explain the common terminology used in the development and use of both versions of TRAP/J. TRAP/J adaptation is composed of two phases: 1) The Generation phase generates an adapt-ready version of an existing application; it includes the Aspect and Reflective classes. The Aspect class performs the interception and redirection of code through its generic hooks. The Reflective class wraps the adaptable class and provides the mechanism to manage and insert adaptive code. 2) The Composition phase allows insertion of new code at startup time or runtime. Adaptive behavior is provided through the implementation of adaptive classes, which are called delegate classes.

3. Previous System: TRAP/J

We will describe in high level the characteristics of the first implementation of TRAP/J relating to its performance and usability. In the following section we will see how TRAP/J v2.1 addresses these issues.

3.1 Compile Time Model

TRAP/J's compile time model consists of three classes: Aspect, BaseLevel, and MetaLevel. The Aspect class is generated by the Aspect Generator and the rest by the Reflective Class Generator as shown in figure 1. The Aspect is in charge of intercepting calls from methods of the original application and redirecting them to the new behavior. The BaseLevel class provides implementation for all local and inherited public, static, and final methods of the adaptable class. The BaseLevel wraps its equivalent adaptable class and extends it with mechanisms to search and invoke the adaptive behavior in the Delegate classes through the MetaLevel class.

Figure 1 illustrates how the adapt-ready application is generated. The adapt-ready application is composed of the Aspect, Reflective Classes, and the original application; all of which are input to the Aspect compiler who is in charge of weaving the generic hooks into the original application.

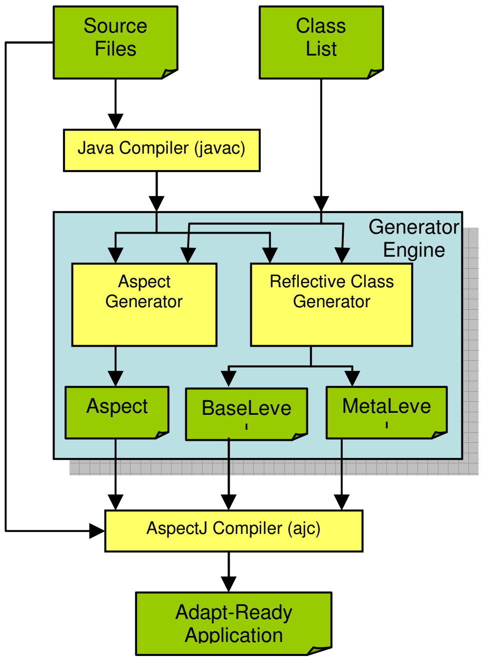


Figure 1. TRAP/J Compile time model

3.2 Runtime Model

One of the main goals of transparent adaptation is to keep the performance of the adapt-ready application very close to the original application (as if no adaptation was provided). In other words, the level of overhead must be reduced to a minimum. TRAP/J has basically four layers in its runtime model: original application, Base Level, Meta Level, and Delegate.

One issue is that the critical execution path of TRAP/J has too many levels of indirection when adaptable behavior is being invoked. To keep things simple following figure 2, suppose a user decides to make an application, OrigApp, adapt-ready. Within the application, the user decides to make `OrigClass` adaptable. When an instance `OrigClass` gets created (`ocInst`), the Aspect intercepts this call and redirects it to construct a Base Level instance. This is possible due to the fact that the Base level instance extends the `origClass` instance. Therefore, any method calls meant for `OrigClass` will actually result in invoking the equivalent (overridden) method calls on the Base Level instance. The overridden method obtains the reflective information about the invoked method and uses its reference to a Meta Level instance to determine if any of the loaded delegates implement a matching method. If a matching delegate method exists, the adaptive behavior gets invoked; otherwise, the original method will be invoked. The key issue is that this entire process happens for any call to a method inside any of the adaptable classes whether a Delegate with a matching method has been inserted or not.

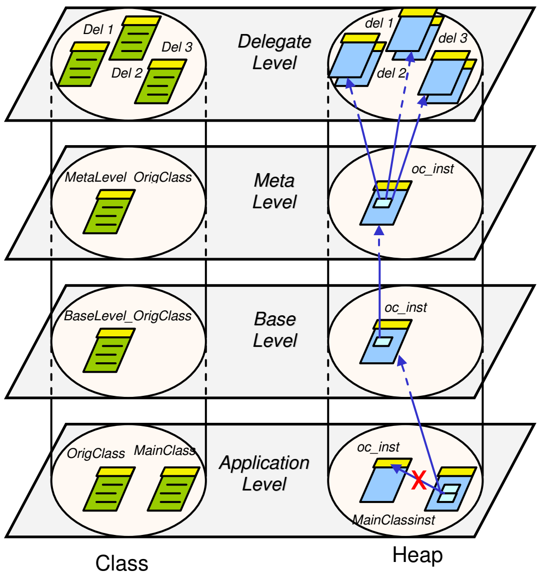


Figure 2. TRAP/J runtime model

As we can see in figure 3, the main issue is where the decision is made to determine whether adaptive behavior is present. This introduces a lot of overhead to the original application because for every method call of an adaptable class, expensive reflective calls and intense loop iterations occur unconditionally prior to even knowing whether adaptive behavior is available; hence, the application's performance suffers greatly. This degradation in performance is not constant, it varies depending on the application and the amount of adaptive behavior provided. In fact, the greater the number of adaptable classes and the greater the number of inserted delegate classes, the more overhead the application will suffer. But even in the event that only one adaptable class has been declared and little adaptation has been provided, if the application is making calls to methods of the adaptable class inside the body of a loop, the performance will still suffer.

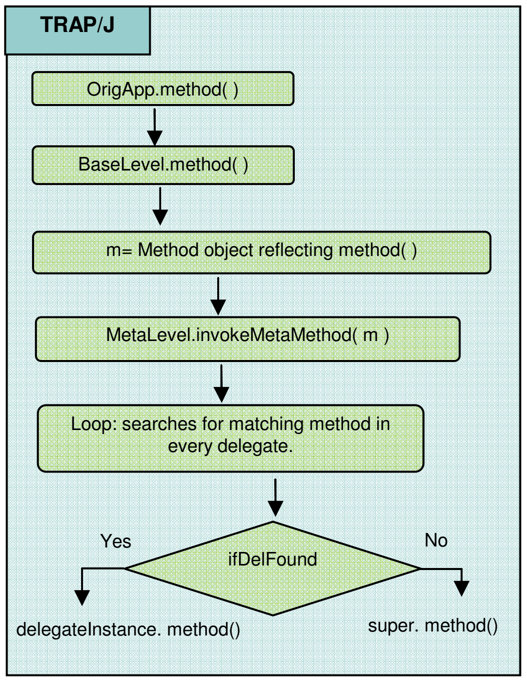

TRAP/J
OrigApp.method( )
BaseLevel.method( )
m= Method object reflecting method( )
MetaLevel.invokeMetaMethod(m)
Loop: searches for matching method in every delegate.
Yes
ifDelFound
No
delegateInstance. method()
super. method()

Figure 3. TRAP/J execution path activity diagram

3.3 Usability

Both generation and adaptation phases lack of a clear GUI interface. The generator engine is invoked through the command line by specifying configuration settings and the file name of the list of classes the user wants to make adaptable. The adaptation phase is done through a delegate management console, where a user types the name of the delegate he wants to use. Both interfaces do not interact with the user. The delegate management console provides no information as to the current status of adaptation; namely, which classes or instances are being adapted and by which delegates. The lack of information about the state of adaptation is the main issue of this tool.

4. New System: TRAP/J v2.1

TRAP/J v2.1 is a redesign of TRAP/J and addresses the main concerns previously discussed

4.1 Compile time Model

The generator engine is similarly composed of an Aspect Generator as well as a Reflective class generator. The latter generates only one class, WrapperLevel, which essentially merges the previous MetaLevel and BaseLevel into one, more simplified and manageable, class. The Aspect class is essentially the same as in the original version of TRAP/J –redirect the method calls made to an adaptable class to their respective Wrapper Level implementations. Aside from merging both layers, the Wrapper Level stores the delegates that have been inserted using a smart delegate insertion mechanism efficiently. This will become an important factor in when measuring the runtime performance of the new version.

In addition, the Generator also includes an Adapt-ready Package generator, a feature that has been extensively used in the Gridification of Java applications. At the moment of generating the WrapperLevel and Aspect classes, users may decide to generate an adapt-ready package. This package is essentially a JAR file which contains TRAP/J runtime classes and the adapt-ready classes that, run in conjunction with the original application, provides all of the mechanism necessary to provide dynamic adaptation at startup time or runtime.

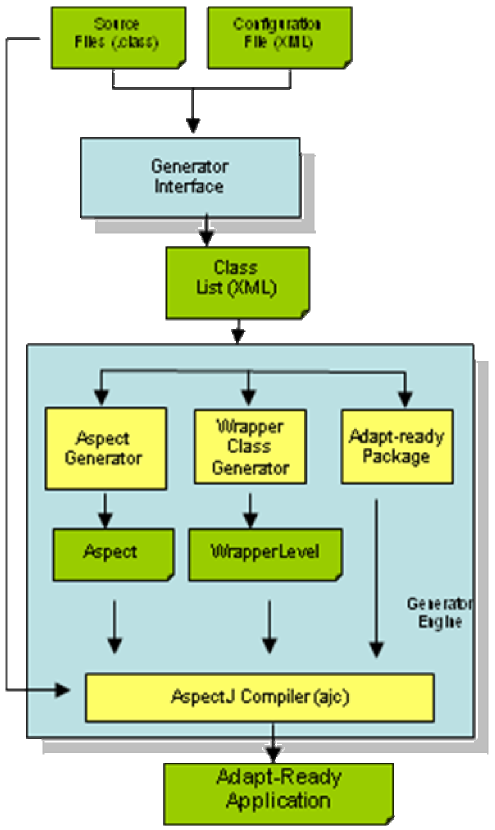


Figure 4. TRAP/J v2.1 Compile time model

4.2 Runtime Model

TRAP/J v2.1 provides many improvements over its predecessor. By comparing Figure 5 and Figure 2 one can observe that the two outer layers are identical in both TRAP/J and TRAP/J 2.1. The difference is that in TRAP/J 2.1 the two middle layers from the previous version were merged into one layer, the Wrapper Level. Basically, the Wrapper Level encapsulates the functionality of both the Meta Level and Base Level from the original version of TRAP/J. Even though this runtime model reduces the levels of indirection, it alone does not guarantee a significant reduction of the critical execution path. The main factor that plays a key role is the smart delegate insertion mechanism that TRAP/J 2.1 uses. In contrast to the previous version, which simply adds each inserted delegate to the end of a list data structure, TRAP/J 2.1 is designed to determine exactly which methods does the delegate provide adaptation for upon insertion. Therefore, every time a delegate is inserted to adapt one of the adaptable classes, this information is stored in the respective Wrapper Level class, and thus made readily accessible to the adapt-ready application. By using this smart delegate insertion mechanism, we are paying the cost of the additional time that it takes to insert a delegate. But this cost is worth paying because the additional knowledge that it provides the adapt-ready application with, allows for it to run with a minimal loss of performance with respect to the original application. Figure 6 shows the reduced critical execution path in TRAP/J 2.1: when a method from the original application is invoked, the call is redirected to the Wrapper Level method. At this point the adapt-ready application is capable of knowing whether a delegate with adaptable behavior has been provided, by querying the information that is saved by the smart delegate insertion mechanism. The difference between the execution paths of TRAP/J and TRAP/J 2.1 lies here. In the second version the adapt-ready application only makes expensive reflective calls after it knows that adaptive behavior is present. When no adaptive behavior is provided, the adapt-ready application is promptly aware of this and, therefore, does not incur additional overhead. Thus, performance loss is greatly reduced.

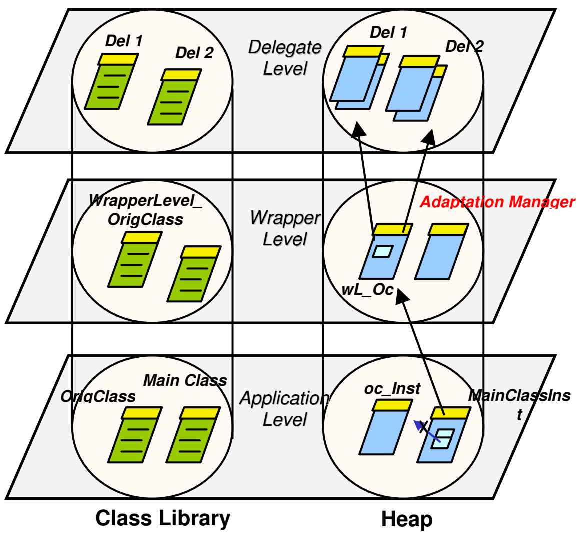


Figure 5. Runtime model TRAP/J v2.1

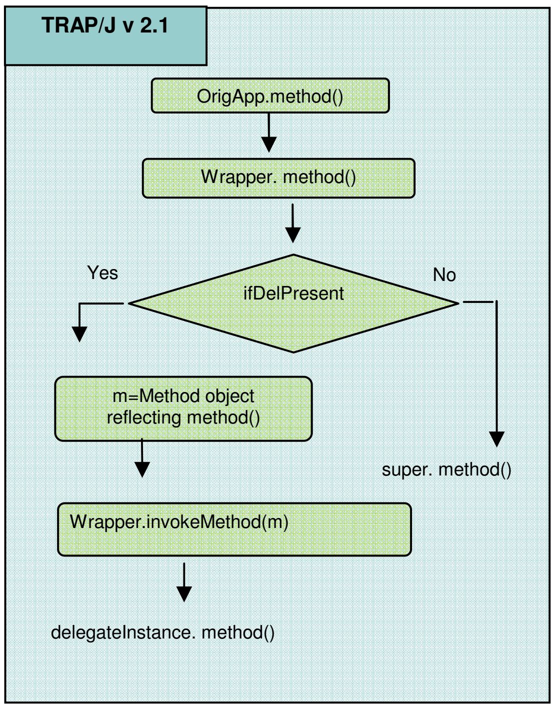


Figure 6. TRAP/J v2.1 execution path

The adapt-ready application is managed at runtime by a singleton component, the Adaptation Manager. As you can see in figure 5, it communicates closely with the Wrapper Level to load and insert delegates. In addition, it always keeps track of the current status of the adapt-ready application.

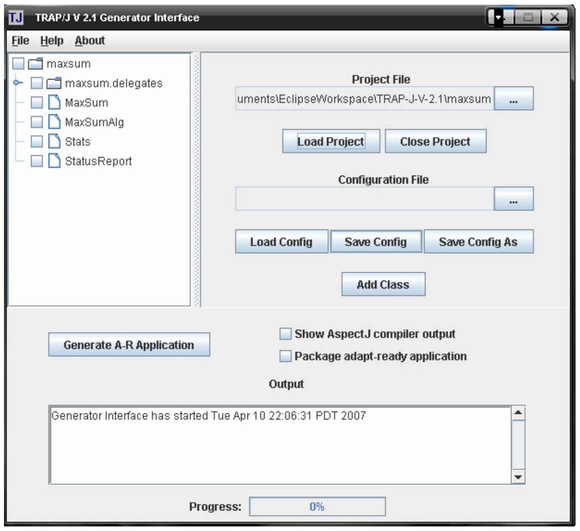
The Adaptation Manager is the link between the web based composer (see figure 8) and the adapt-ready application. In other words, it allows users to retrieve the current status and to provide the adaptive behavior through the composer interface.

4.3 Usability

In terms of usability, TRAP/J 2.1 features a user friendly generator and composer interface. The generator interface is equipped with an adapt-ready package generator and a build-in AspectJ compiler.

4.3.1 Generator Interface

Figure 7 shows TRAP/J v2.1 Generator interface. It is a simple user interface that allows users to browse their current file system for the original application project file. From a tree view display of the project, users can choose which classes they want to make adaptable from either the original application or a standard java library class. An important thing to note here is that any class in Java could be made adaptable except, immutable class (e.g. java.lang.String, java.lang.Integer). This imposes no restriction other than the expected behavior: an immutable class is always expected to behave the same way. Once the user selects the classes he wants to make adaptable, he clicks on the generate button. Users can also load previously saved configuration files that contains all of the classes selected if the user decides to regenerate a similar adapt-ready application, as a snapshot of a preciously saved configuration. After selecting all the classes, user can invoke the Generator Engine to generate the adapt-ready application. This provides a significant improvement over the command line counterpart in first implementation.



Project File: uments\EclipseWorkspace\TRAP-J-V-2.1\maxsum

```
Generator Interface has started Tue Apr 10 22:06:31 PDT 2007
```

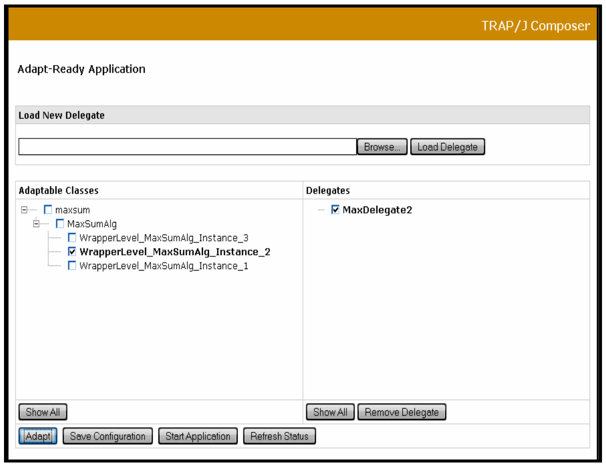
Progress: 0%

Figure 7. TRAP/J v2.1 Generator Interface

In addition, the Generator automates the process of manually compiling the adapt-ready application. In previous version of TRAP/J, users must manually compile the generated adapt-ready application using the AspectJ (ajc) compiler. This process is now done automatically as part of the generation process as TRAP/J 2.1 makes the necessary calls to the ajc. Another convenient feature is that TRAP/J 2.1 can create a TRAP/J runtime library which essentially contains the core runtime classes weaved with the generated classes. This allows users to deploy TRAP/J easily to any machine. This has been successfully tested in grid environments where TRAP/J was used to gridify (adapt) applications otherwise not suited to run on a grid. With this approach, however, the AspectJ compiler must be called in order to weave together TRAP/J runtime with the original application.

4.3.2 Composer Interface

TRAP v2.1 is structured with a client-server architecture, which basically allows multiple remote users to upload delegates (.class) files into the adapt-ready application. TRAP/J incorporates a simple web server implementation called NanoHTTPD, which serves the composer interface that interacts with the user. As we can see in figure 8, the Composer shows the current status of the adapt-ready program including all classes and instances being adapted, and delegates which have been uploaded. The adapt-ready application can be run in two modes: the "run" mode allows a user to adapt the currently running adapt-ready application by browsing and loading delegates remotely. This interface depicts a clear status of the currently running application by showing the name of the application, the classes that were made adaptable, all instances currently created, and all loaded delegates.



Adaptable Classes:
- [ ] maxsum
  - [ ] MaxSumAlg
    - [ ] WrapperLevel_MaxSumAlg_Instance_3
    - [✓] WrapperLevel_MaxSumAlg_Instance_2
    - [ ] WrapperLevel_MaxSumAlg_Instance_1

Delegates:
- [✓] MaxDelegate2

Figure 8. TRAP/J v2.1 Composer Interface

On the other hand, the application can be run in "paused" mode. In this case, the user may load default behavior for the entire class, and start the application with this new behavior already inserted. When the interface is launched is "paused" mode, the button "Start Application" becomes enabled which notifies users they can begin to execute the application. The effect of specifying behavior of the entire class is that all instances of the application will behave in comparable fashion. Through the Composer Interface, users can choose to load, remove, adapt, and unadapt any delegates for either the entire class or particular instances.

5. Case Study

We will demonstrate and explain the execution of TRAP/J v2.1 based on a Maximum Subsequent Sum (MSS) program.

We used three different algorithms with different orders (see table 1) that all perform a MSS calculations based on an input file of randomly generated numbers. For each algorithm, we have developed a delegate class. We will use TRAP/J to adapt MSS for both instances and classes at startup time and at runtime.

Instance	Algorithm	Performance	Time

	Speed		(sec)
1	Fast	O(N)	0.0
2	Medium	$O(N^2)$	13.5
3	Slow	$O(N^3)$	30.4

Table 1. MSS algorithms and running times results

As we can see in figure 8, MaxSum creates three instances of MaxSumAlg, a class that implements a Maximum Subsequent Sum algorithm. All instances read from the same data structure of randomly generated numbers. We have adapted all instances with different algorithms at both runtime and startup time. Initially, the application runs with a slow algorithm, then we provided the adaptation to instances 2 and 3 at runtime with the faster algorithms. We can also decide to unadapt all instances and they will all go back to their initial behavior.

6. Future Work

There are other approaches that could be taken to provide Transparent Shaping. One approach uses Attribute-Oriented Programming. Actually, TRAP.NET uses a similar approach where adaptation occurs at the level of methods. TRAP.NET is essentially a plug in for Visual Studio and acts as a preprocessor that reads and interprets custom attributes and generates the adapt-ready application executable. A similar approach could be used on Java, as well. With the advent of J2SE 5.0 Annotations API, methods and classes could be tagged with attributes that indicate which methods/classes a user wants to make adaptable. To realize this, a tool called XDoclet, an open source Java tool, allows for custom attributes to be specified within the Java code as Javadoc tags as figure 9 shows. Similarly, we could make TRAP/J an Eclipse plug-in preprocessor that reads and parses these annotations from which it generates an adapt-ready application.

```
/**
* @TRAP/J: makeAdaptable(true)
*/
public class MyClass {

    . . .


    /**
    * @TRAP/J: makeAdaptable(true)
    */
    public void myMethod() {

    }


}
```

Figure 9. Custom attributes used in adaptation

Another interesting extension is a trend called safe adaptation. In grid computing, for instance, we have used TRAP/J to adapt a matrix multiplication application at startup time using hyper matrix multiplication algorithm. The challenge is further increased when adapting this kind of application at runtime. Changing among blocking algorithms at runtime requires TRAP/J to transition among algorithms without loss of intermediate calculations.

7. Summary

TRAP/J v2.1 promises to deliver very close performance to the original application. Original method calls will be faster than TRAP/J since the system will determine whether there’s a delegate method present before obtaining the reflective information; otherwise, any calls to any method for which there is no delegate, the original method is called without suffering any performance overhead (Refer to figure 5 for detailed information). Moreover, TRAP/J v2.1 will eliminate the exhaustive, unnecessary searching from the previous version when invoking a delegate method; the reflective information again is obtained only when a delegate for that method has been loaded.