

# Transparent Shaping of Existing Software to Support Pervasive and Autonomic Computing

S. Masoud Sadjadi  
School of Computer Science  
Florida International University  
Miami, Florida 33199  
sadjadi@cs.fiu.edu

Philip K. McKinley  
Dept. of Computer Science  
Michigan State University  
East Lansing, Michigan 48824  
mckinley@cse.msu.edu

Betty H.C. Cheng  
Dept. of Computer Science  
Michigan State University  
East Lansing, Michigan 48824  
chengb@cse.msu.edu

## ABSTRACT

The need for adaptability in software is growing, driven in part by the emergence of pervasive and autonomic computing. In many cases, it is desirable to enhance existing programs with adaptive behavior, enabling them to execute effectively in dynamic environments. In this paper, we propose a general programming model called *transparent shaping* to enable dynamic adaptation in existing programs. We describe an approach to implementing transparent shaping that combines four key software development techniques: aspect-oriented programming to realize separation of concerns at development time, behavioral reflection to support software reconfiguration at run time, component-based design to facilitate independent development and deployment of adaptive code, and adaptive middleware to encapsulate the adaptive functionality. After presenting the general model, we discuss two specific realizations of transparent shaping that we have developed and used to create adaptable applications from existing programs.

## Categories and Subject Descriptors

D.3.4 [PROGRAMMING LANGUAGES]: Processors—*Code generation*

## General Terms

Design, languages, reliability.

## Keywords

Dynamic adaptation, middleware, program families.

## 1. INTRODUCTION

A software application is *adaptable* if it can change its behavior dynamically (at run time) in response to transient changes in its execution environment or to permanent changes in its requirements. Recent interest in designing adaptable software is driven in part by the emergence of pervasive computing and the demand

for autonomic computing [1]. *Pervasive computing* promises anywhere, any time access to data and computing resources with few limitations and disruptions [2]. The need for adaptability in pervasive computing is particularly evident at the “wireless edge” of the Internet, where software in mobile devices must balance conflicting concerns such as quality-of-service (QoS) and energy consumption in responding to variability of conditions (e.g., wireless network loss rate). *Autonomic computing* [3] refers to self-managed, and potentially self-healing, systems that require only high-level human guidance. Autonomic computing is critical to managing the myriad of sensors and other small devices at the wireless edge, but also in managing large-scale computing centers and protecting critical infrastructure (e.g., financial networks, transportation systems, power grids) from hardware component failures, network outages, and security attacks.

Developing and maintaining adaptable software are nontrivial tasks. An adaptable application comprises *functional* code that implements the business logic of the application and supports its imperative behavior, and *adaptive* code that implements the adaptation logic of the application and supports its adaptive behavior. The difficulty in developing and maintaining adaptable applications is largely due to an inherent property of the adaptive code, that is, the adaptive code tends to *crosscut* the functional code. Example crosscutting concerns include QoS, mobility, fault tolerance, recovery, security, self auditing, and energy consumption. Even more challenging than developing new adaptable applications is enhancing *existing* applications, such that they execute effectively in new, dynamic environments not envisioned during their design and development. For example, many non-adaptive applications are being ported to mobile computing environments, where they require dynamic adaptation.

This paper proposes a new programming model, called *transparent shaping*, that supports the design and development of adaptable programs from existing programs without the need to modify the existing programs’ source code directly. We argue that *automatic* generation of an adaptable program from a non-adaptable one is important to maintaining program integrity, not only because it avoids errors introduced by manual changes, but because it provides traceability for the adaptations and enables the program to return to its original behavior if necessary. Our approach to implementing transparent shaping combines four key technologies: *aspect-oriented programming* to enable separation of concerns at development time, *behavioral reflection* to enable software reconfiguration at run time, *component-based design* to enable independent development and deployment of adaptive code, and *adaptive middleware* to help insulate application code from adaptive functionality. To demonstrate the effectiveness of this approach, we

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DEAS 2005, May 21, 2005, St. Louis, Missouri, USA.

Copyright 2005 ACM 1-59593-039-6/05/0005 ...\$5.00.

describe two realizations of transparent shaping that we have developed and used to create adaptable applications.

The remainder of this paper is organized as follows. Section 2 discusses the four main components of our approach. Section 3 provides an overview of transparent shaping and describes its relationship to program families [4]. Sections 4 and 5, respectively, describe two realizations of transparent shaping; one is middleware-based and the other is language-based. Section 6 discusses how transparent shaping complements other research in adaptive software. Section 7 presents conclusions and identifies several directions for future research.

## 2. BASIC ELEMENTS

Our approach to transparent shaping integrates four key technologies: separation of concerns, behavioral reflection, software components, and middleware. In this section, we briefly review each technology and its role in transparent shaping.

*Separation of concerns* [5] enables the separate development of the functional code from the adaptive code of an application. This separation simplifies development and maintenance, while promoting software reuse. Moreover, since adaptation often involves crosscutting concerns, this separation also facilitates transparent shaping. In our approach, we use aspect-oriented programming (AOP) [6, 7], an increasingly common approach to implementing separation of concerns in software. While object-oriented programming introduces abstractions to capture commonalities among classes in an inheritance tree, crosscutting concerns are scattered among different classes, thus complicating the development and maintenance of applications. Conversely, in AOP the code implementing such crosscutting concerns, called *aspects*, is developed separately from other parts of the system. Later, for example during compilation, an *aspect weaver* can be used to weave different aspects of the program together to form a program with new behavior. Predefined locations in the program where aspect code can be woven are called *pointcuts*.

In traditional AOP, after compilation the aspects are tangled (via weaving) with the functional code. To facilitate dynamic reconfiguration, transparent shaping needs a way to enable separation of concerns to persist into run time. This separation can be accomplished using *behavioral reflection* [8], the second key technology for transparent shaping. Behavioral reflection enables a system to “open up” its implementation details at run time [9]. A reflective system has a self representation that deals with the computational aspects (implementation) of the system, and is *causally connected* to the system. The self representation of a reflective system is realized by metaobjects residing in the meta-level, which is separated from the actual system represented by objects in the base level. By incorporating crosscutting concerns associated with the system as part of its self representation, the resulting code at run time is not tangled and can be reconfigured dynamically. When combined with AOP, behavioral reflection enables dynamic weaving of crosscutting concerns into an application at run time [10].

The third major technology that supports transparent shaping is component-based design. *Software components* are software units that can be independently developed, deployed, and composed by third parties [11]. Well-defined interface specifications supported in component-based design enable adaptive code to be developed independently from the functional code, and potentially by different parties, using the interface as a contract. Component-based design supports two types of composition. In static composition, a developer can combine several components at compile time to produce an application. In dynamic composition, the developer can add, remove, or reconfigure components within an application at

runtime. When combined with behavioral reflection, component-based design enables a “plug-and-play” capability for adaptive code to be incorporated with functional code at run time that facilitates development and maintenance of adaptable software.

Finally, in many cases it is desirable to hide the adaptive behavior from the application using middleware. Traditionally, *middleware* is intended to mask the distribution of resources across a network and hide differences among computing platforms and networks [12]. As observed by several researchers [13], however, middleware is also an ideal place to incorporate adaptive behavior for many different crosscutting concerns. *Adaptive* middleware enables dynamic reconfiguration of middleware services while an application is running, adjusting the middleware behavior to environmental changes dynamically. Our approach to transparent shaping uses adaptive middleware in two ways. In the first, transparent shaping adds adaptive behavior to a middleware platform already supporting the application. In the second, transparent shaping is used to weave adaptive middleware itself into an application.

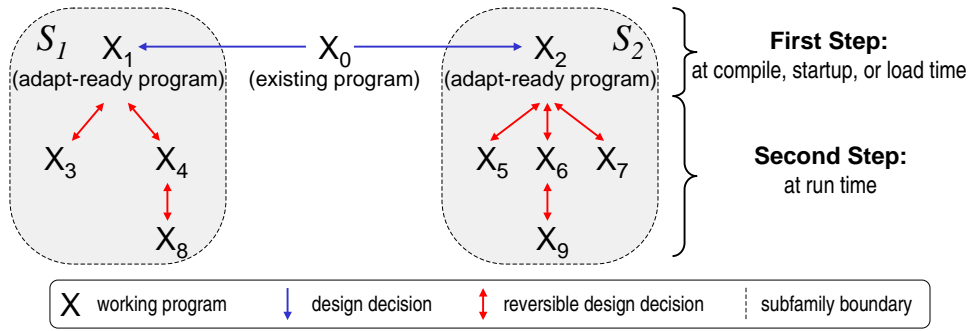
## 3. GENERAL APPROACH

By generating adaptable programs from existing applications, transparent shaping is intended to support the reuse of those applications in environments whose characteristics were not necessarily anticipated during the original design and development. Therefore, the challenge in transparent shaping is finding a way to produce adaptable programs that share the business logic of the original program and differ only in the new adaptive behavior.

As illustrated in Figure 1, one way to formulate this problem is using *program families*, a well-established concept in the software engineering community. A program family [4] is a set of programs whose extensive commonalities justify the expensive effort required to study and develop them as a whole, rather than individually. In short, transparent shaping can be viewed as producing a family of adaptable programs from an existing non-adaptable program. The adaptable program comprises the original program code that remains fixed during program execution, and adaptive code that can be replaced with other adaptive code dynamically. Replacing one piece of adaptive code with another piece of adaptive code converts an adaptable program into another adaptable program in the corresponding family. This conversion is possible in this programming model, because the adaptive code is not tangled with the functional code. We use the term *composer* to refer to the entity that performs this conversion. The composer might be a human – a software developer or an administrator interacting with a running program through a graphical user interface – or a piece of software – a dynamic aspect weaver, a component loader, a runtime system, or a metaobject.

Our approach to transparent shaping produces adaptable programs in two steps. In the first step, an *adapt-ready* program [14] is produced at compile, startup, or load time using static transformation techniques. An adapt-ready program is a program whose behavior is initially equivalent to the original program, but which can be adapted at run time by insertion or removal of adaptive code at certain points in the execution path of the program, called *sensitive joinpoints*. To support such operations, the first step of transparent shaping weaves interceptors, referred to as *hooks*, at the sensitive joinpoints, which may reside inside the program code itself, inside its supporting middleware, or inside the system platform. Example techniques for implementing hooks include aspects (compile time), CORBA portable interceptors [15] (startup time), and bytecode rewriting [16] (load time).

In the second step, executed at run time, the hooks in the adapt-ready program are used by the composer to convert the adapt-ready



**Figure 1: A transparent shaping design tree illustrating a family of adaptable programs produced from an existing program, which is the root of this tree. Children of the root are adapt-ready programs. Other descendants are adaptable programs.**

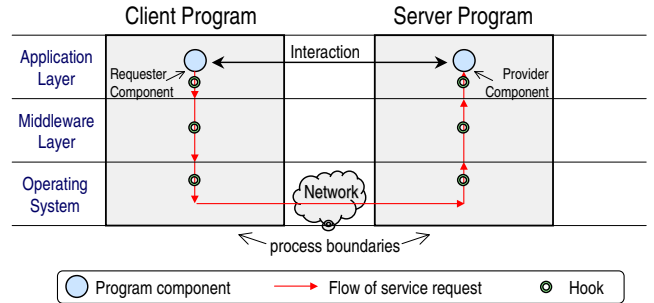
program into an adaptable program in the corresponding subfamily, as executing conditions warrant. Adapt-ready programs derived from the same existing program differ in their corresponding sensitive joinpoints and hooks. We note that the available hooks in an adapt-ready program limit its dynamic behavior. In other words, each adapt-ready program can be converted to a limited number of adaptable programs in the corresponding family. The adaptable programs derived from an adapt-ready program form a subfamily (e.g.,  $S_1$  and  $S_2$  in Figure 1).

We use Figure 1 to describe a specific example. Consider an existing distributed program ( $X_0$ ) originally developed for a wired and secure network. To enable this program to run efficiently in a mobile computing environment, the first step of transparent shaping can be used to produce an adapt-ready version of this program ( $X_1$ ), which has hooks intercepting all the remote interactions. At run time, if the system detects a low quality wireless connection, the composer can insert adaptive code for tolerating long periods of disconnection into the adapt-ready program (producing  $X_4$  from  $X_1$ ). Later, if the user enters an insecure wireless network, the composer can insert adaptive code for encryption/decryption of the remote interactions into the program (producing  $X_8$  from  $X_4$ ). Finally, when the user returns to an area with a secure and reliable wireless connection, the composer can remove the adaptive code for both security and connection-management to avoid unnecessary performance overhead due to the adaptive code (producing  $X_4$  from  $X_8$  and  $X_1$  from  $X_4$ , respectively).

We identify three approaches to realize transparent shaping that differ according to the placement of hooks (see Figure 2): (1) hooks can be incorporated inside an application program itself, (2) inside its supporting middleware, or (3) inside the system platform (operating system and network protocols). A number of projects on cross-layer adaptation address the last case [17–19]. In this paper, we consider only the first two cases, where the hooks are incorporated either inside the middleware or inside the application. Next, we describe two concrete realizations of each type of transparent shaping. The first, described in Section 4, is a middleware-based approach that uses CORBA portable interceptors [15] as hooks. The second, described in Section 5, uses a combination of aspect weaving and metaobject protocols to introduce dynamic adaptation to the application code directly. Both realizations adhere to the general model described above.

#### 4. MIDDLEWARE-BASED APPROACH

The first realization of transparent shaping we consider is the *Adaptive CORBA Template (ACT)* [20, 21], which we developed to

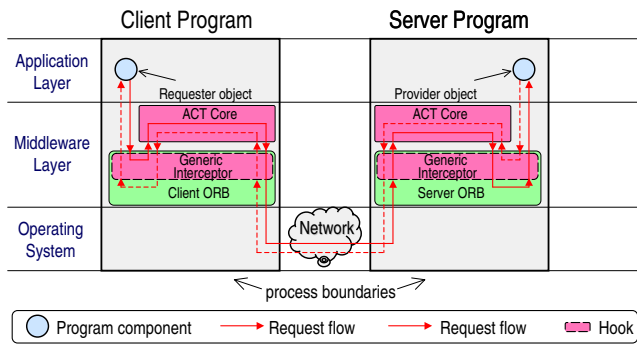


**Figure 2: Alternative places to insert hooks.**

enable dynamic adaptation in existing CORBA programs. ACT enhances CORBA ORBs to support dynamic reconfiguration of middleware services transparently not only to the application code, but also to the middleware code itself. As a realization of transparent shaping, ACT produces an adapt-ready version of an existing CORBA program by introducing a hook at startup time to intercept all CORBA remote interactions. Specifically, ACT uses CORBA portable interceptors [15], which can be incorporated into a CORBA program at startup time using a command-line parameter. Later at run time, these hooks can be used to insert adaptive code into the adapt-ready program, which in turn can adapt the requests, replies, and exceptions passing through the ORBs. In this manner, ACT enables run-time improvements to the program in response to unanticipated changes in its execution environment, effectively producing other members of the adaptable program family dynamically. To evaluate the performance and functionality of ACT, we constructed a prototype of ACT in Java, called *ACT/J*, a language-specific realization of ACT.

Figure 3 shows the flow of a request/reply sequence in a simple CORBA application using ACT. For clarity, details such as stubs and skeletons are not shown. ACT comprises two main components: a generic interceptor and an ACT core. A *generic interceptor* is a specialized request interceptor that is registered with the ORB of a CORBA application at startup time. The *client* generic interceptor intercepts all outgoing requests and incoming replies (or exceptions) and forwards them to its ACT core. Similarly, the *server* generic interceptor intercepts all the incoming requests and outgoing replies (or exceptions) and forwards them to its ACT core. A CORBA application is called *adapt-ready* if a generic interceptor is registered with all its ORBs at startup time.

In a series of case studies [20, 21], we have used ACT/J to sup-



**Figure 3: ACT configuration in the context of a simple CORBA application.**

port dynamic adaptation in three different ways. First, we used ACT/J to accommodate changing conditions of a wireless network infrastructure into existing CORBA applications developed originally for wired networks [21]. Specifically, we were able to adapt a CORBA image retrieval application to tolerate long periods of disconnection if used in a wireless network (note: the original application would crash if used by itself in a wireless network). Second, we used ACT/J to integrate the QuO framework [22] into existing CORBA applications dynamically and transparently with respect to the application source code [20]. *QuO* [22] is a powerful adaptive framework developed by BBN Technologies that supports dynamic adaptation for CORBA and Java RMI applications. Finally, we used ACT/J to integrate new adaptive code into extant QuO applications transparently [20], enabling interoperation of QuO with other adaptive frameworks. Our experimental results show that the overhead introduced by the ACT/J infrastructure is negligible, while the adaptations offered are highly flexible.

As a middleware-based realization of transparent shaping, ACT can be used to produce families of adaptable programs from existing CORBA programs, without the need to modify or recompile their source code. Using the generic interceptor as a hook inside middleware at startup time, ACT enables independent development and deployment of adaptive code from the application code at run time. In ACT, adaptive code are realized as software components (rules and proxies) that can be deployed inside the ACT core dynamically. By allowing dynamic insertion and removal of such adaptive code, ACT enables dynamic conversion of an adapt-ready CORBA program to different adaptable programs in its corresponding program subfamily.

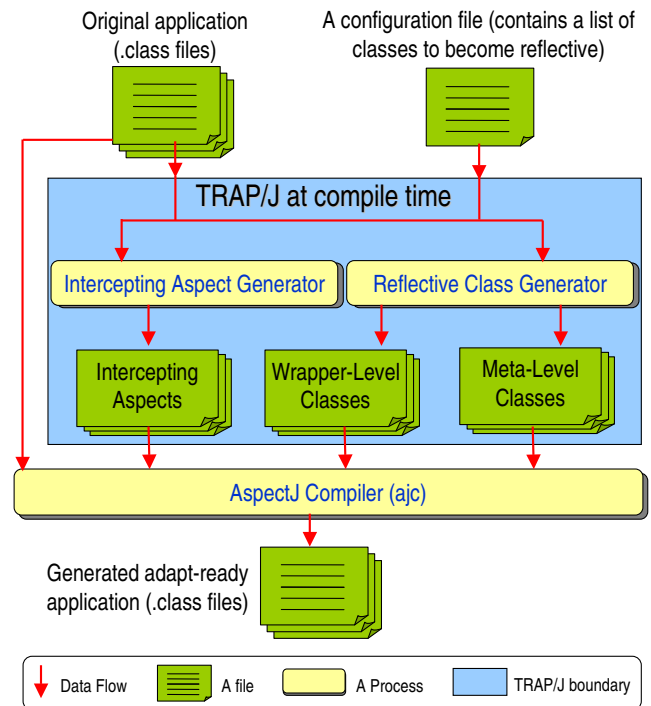
## 5. LANGUAGE-BASED APPROACH

Although transparent shaping can be realized by incorporating hooks inside middleware, as in ACT, many programs do not use middleware explicitly. In this section, we discuss TRAP (Transparent Reflective Aspect Programming) [23], a language-based realization of transparent shaping that supports dynamic adaptation in existing programs developed in class-based, object-oriented programming languages. TRAP uses generative techniques to create an adapt-ready application, without requiring any direct modifications to the existing programs.

With TRAP, the developer selects at compile time a subset of classes in the existing program that are to be reflective at run time. We say a class is *reflective* at run time if its behavior (e.g., the implementation of its methods) can be inspected and modified dynamically. Since many object-oriented languages, such as Java and

C++, do not support such functionality inherently, TRAP uses generative techniques to produce an adapt-ready program with hooks that provide the reflective facilities for the selected classes. As the adapt-ready program executes, new behavior can be introduced to the program by insertion and removal of adaptive code via interfaces to the reflective classes.

We developed TRAP/J, a prototype instantiation of TRAP for Java programs [23]. The operation of the first step, converting an existing Java program into an adapt-ready program, is depicted in Figure 4. We assume that the `.java` source files of the original application are not available. The application compiled class files (`.class` files) and a configuration file containing a list of class names (the ones selected to be reflective) are input to an Aspect Generator and a Reflective Class Generator. For each class name in the list, these generators produce one aspect, one wrapper-level class, and one meta-level class. Next, the generated aspects and reflective classes, along with the original application compiled class files, are passed to the AspectJ compiler (`ajc`) [24], which weaves the generated and original application code together to produce an adapt-ready application. The second step occurs at run time, when new behavior can be introduced to the adapt-ready application using the wrapper- and meta-level classes (also referred to as the adaptation infrastructure). Specifically, the interface of the meta-level class includes services that enable methods of the wrapper-level class to be overridden at run time with new implementations, called *delegates*.



**Figure 4: TRAP/J operation at compile time.**

For example, in an earlier study [23] we developed a delegate that effectively allows selected Java sockets in an existing program to be replaced with adaptable communication middleware components called *MetaSockets*. A *MetaSocket* is created from existing Java socket classes, but its structure and behavior can be adapted at run time in response to external stimuli such as dynamic wireless channel conditions. Specifically, data sent or received on the socket is passed through a pipeline of filters. A *MetaSocket* itself

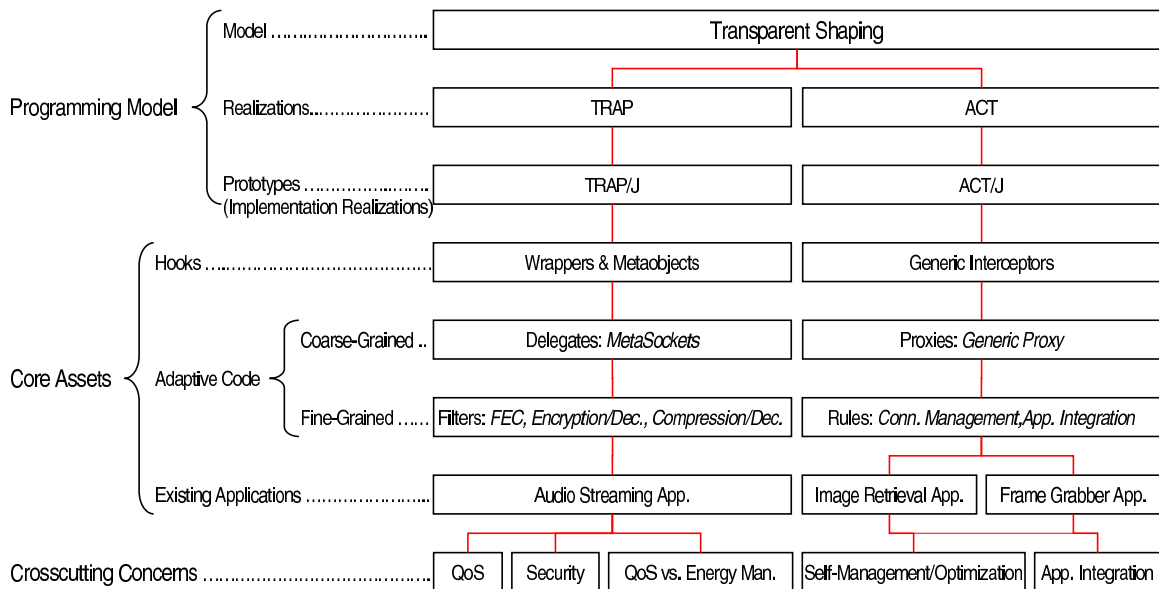


Figure 5: Transparent shaping contributions and artifacts.

can be reconfigured dynamically in its filter pipeline. The filter pipeline can be reconfigured dynamically, that is, filters can be inserted and removed, in response to changes in changing conditions. Moreover, the filter components can be developed by third parties and can be independent of the functional code of an application. Using TRAP/J and MetaSockets, we demonstrated how to transform existing network applications into adaptive applications that can better tolerate dynamic conditions on wireless networks [23].

In summary, TRAP enables production of adaptable program families from existing programs developed in class-based, object-oriented programming languages. Using the wrapper- and meta-level classes as hooks instrumented inside the application code at compile time, TRAP enables separate development and deployment of adaptive code in existing programs at run time. In TRAP, pieces of adaptive code are realized as delegates that can be inserted into and removed from an adapt-ready program dynamically, thereby converting the adapt-ready program to adaptable programs in its corresponding program subfamily.

## 6. DISCUSSION

Figure 5 summarizes our completed tasks related to transparent shaping. We have implemented and tested ACT/J and TRAP/J, as described above. We have also developed several core assets for supporting transparent shaping, including examples of hooks, adaptive code, and existing applications. The hooks in TRAP/J are pairs of wrappers and meta classes, which are generated by TRAP/J generators automatically. In ACT/J, there is only one hook, the generic portable interceptor, which can be reused in any CORBA program. Adaptive code in TRAP/J is realized by delegates. A reusable delegate using MetaSockets and filters is provided. A generic proxy was developed for ACT/J that can be used in any existing CORBA application. The generic proxy can receive any CORBA request and can adapt it using adaptive code realized by rules.

We are currently addressing several other aspects of transparent shaping. To support existing programs developed in C++, members of our group have already implemented TRAP/C++ [25] using

compile-time meta-object protocols supported by Open C++ [26]. In addition, we are investigating the development of TRAP/C#. To support CORBA programs developed using C++ ORBs, we plan to develop ACT/C++. We are also investigating techniques to support the insertion of hooks for adaptation into the operating system kernel [19], the third case mentioned earlier.

Transparent shaping complements other work in adaptive software, particularly adaptive middleware. Figure 6 depicts this relationship, according to Schmidt’s four layer middleware taxonomy [27]. Please note that the frameworks mentioned inside the transparent shaping boundary can be incorporated into existing applications transparently, while the ones outside this boundary require explicit calls from the application source code. As in our work with TRAP/J and MetaSockets, transparent shaping can enable existing non-adaptive applications to take advantage of adaptive host-infrastructure middleware services such as MetaSockets. Also, using our ACT/J framework, transparent shaping can enable existing CORBA applications to take advantage of adaptive common middleware services such as QuO. In addition, we note that many adaptive frameworks developed by other groups can be used to support transparent shaping. Examples include Composition Filters [28], RNTL ARCAD [10], Interoperable Replication Logic [29], FTS [30], TAO Load Balancing [31], Iguana/J [32], Prose [33], Guaranà [34], Eternal [35], and Rocks/Racks [36]. In [1], we provided a summary of these techniques.

Finally, we note that transparent shaping has potential impact beyond supporting adaptation in individual programs, for example, to support application integration [37]. To integrate two existing heterogeneous applications, possibly developed in different programming languages and targeted to run on different platforms, one needs to convert data and commands between the two applications on an ongoing basis. Transparent shaping offers a solution to this problem, without the need to modify application source code directly. In preliminary studies [37], we have proposed several alternative architectures and showed how transparent shaping can support interoperability, via Web services, for Java RMI, CORBA, and .NET applications. As a proof of concept, we have conducted

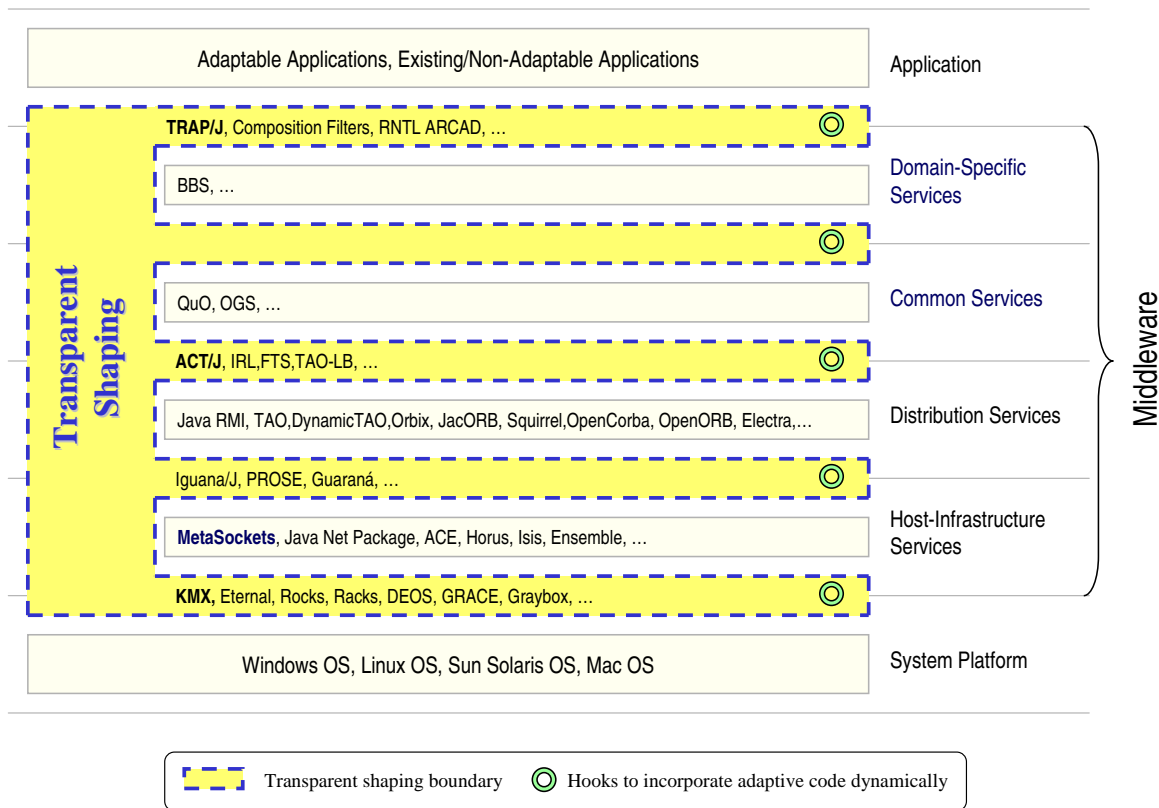


Figure 6: Relationship of transparent shaping to other contributions.

a case study that demonstrates the use of transparent shaping in the integration of an image retrieval application developed in CORBA with a frame grabber application developed in .NET.

## 7. CONCLUSIONS AND FUTURE WORK

Transparent shaping supports reuse of existing programs in new, dynamic environments even though the specific characteristics of such new environments were not necessarily anticipated during the original design of the programs. In particular, many existing programs, not designed to be adaptable, are being ported to dynamic wireless environments, or hardened in other ways to support pervasive and autonomic computing. We propose an approach to transparent shaping based on the concept of program families and demonstrate how automated methods can be used to transform a program into another member of the same family. Our approach integrates four key technologies: aspect-oriented programming, behavioral reflection, component-based programming, and adaptive middleware. We highlighted two different realizations of transparent shaping, ACT and TRAP, and showed how they realize the general adaptive programming model. In addition to our work on other realizations of transparent shaping, as well as application integration, we are also addressing several other aspects of transparent shaping: coordination of adaptive behavior across system layers and among different systems, formal techniques to ensure that adaptations leave the system in a consistent state [38], preventing adaptation mechanisms from being exploited by would-be attackers, and constructing “product lines” of adaptable software.

**Acknowledgements.** We express our gratitude to the faculty and students in the Software Engineering and Network Systems Laboratory at Michigan State University for their feedback and their insightful discussions on this work. This work was supported in part by the U.S. Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, and in part by National Science Foundation grants CCR-9912407, EIA-0000433, EIA-0130724, ITR-0313142, EIA-0000433, and CCR-9901017.

## 8. REFERENCES

- [1] Philip K. McKinley, Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *IEEE Computer*, pages 56–64, July 2004.
- [2] M. Weiser. Ubiquitous computing. *IEEE Computer*, 26(10):71–72, October 1993.
- [3] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.
- [4] David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, March 1976.
- [5] Peri Tarr and Harold Ossher, editors. *Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001 (W17)*, May 2001.
- [6] 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.

- [7] *Communications of the ACM, Special Issue on Aspect-Oriented Programming*, volume 44, October 2001.
- [8] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Languages (OOPSLA)*, pages 147–155. ACM Press, December 1987.
- [9] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of Metaobject Protocols*. MIT Press, 1991.
- [10] Pierre Charles David, Thomas Ledoux, and Noury 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, October 2001.
- [11] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.
- [12] David E. Bakken. *Middleware*. Kluwer Academic Press, 2001.
- [13] *Proceedings of the Middleware'2000 Workshop on Reflective Middleware (RM2000)*, New York, April 2000.
- [14] Z. Yang, B. H.C. Cheng, R. E. K. Stirewalt, J. Sowell, S. Masoud Sadjadi, and P. K. McKinley. An aspect-oriented approach to dynamic adaptation. In *Proceedings of the ACM SIGSOFT Workshop On Self-healing Software (WOSS'02)*, November 2002.
- [15] Object Management Group, Framingham, Massachusetts. *The Common Object Request Broker: Architecture and Specification Version 3.0*, July 2003.
- [16] Geoff A. Cohen, Jeffrey S. Chase, and David Kaminsky. Automatic program transformation with JOIE. In *1998 Usenix Technical Conference*, June 1998.
- [17] S. Adve, A. Harris, C. Hughes, D. Jones, R. Kravets, K. Nahrstedt, D. Sachs, R. Sasanka, J. Srinivasan, and W. Yuan. The Illinois GRACE project: Global resource adaptation through cooperation, 2002.
- [18] Distributed extensible open systems (the DEOS project), 2004. Georgia Institute of Technology - College of Computing.
- [19] F. Samimi, P. K. McKinley, S. Masoud Sadjadi, and P. Ge. Kernel-middleware cooperation in support of adaptive mobile computing. In *the Second International Workshop on Middleware for Pervasive and Ad-Hoc Computing*.
- [20] S. Masoud 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, March 2004.
- [21] S. Masoud Sadjadi and P. K. McKinley. Transparent self-optimization in existing CORBA applications. In *Proc. of the International Conference on Autonomic Computing (ICAC-04)*, pages 88–95, New York, NY, May 2004.
- [22] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.
- [23] 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.
- [24] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
- [25] Scott D. Fleming, Betty H.C. Cheng, R.E. Kurt Stirewalt, and Philip K. McKinley. An approach to implementing dynamic adaptation in c++. In *Proceedings of the first Workshop on the Design and Evolution of Autonomic Application Software 2005 (DEAS'05)*, in conjunction with ICSE 2005, St. Louis, Missouri, May 2005. To appear.
- [26] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. *Lecture Notes in Computer Science*, 707, 1993.
- [27] Douglas C. Schmidt. Middleware for real-time and embedded systems. *Communications of the ACM*, 45(6), June 2002.
- [28] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of ACM*, (10):51–57, October 2001.
- [29] 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)*, pages 109–118, Osaka, Japan, October 2002.
- [30] Erez Hadad. *Architectures for Fault-Tolerant Object-Oriented Middleware Services*. PhD thesis, Computer Science Department, The Technion - Israel Institute of Technology, 2001.
- [31] Ossama Othman. The design, optimization, and performance of an adaptive middleware load balancing service. Master's thesis, University of California, Irvine, 2002.
- [32] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, June 2002.
- [33] A. Popovici, T. Gross, and G. Alonso. Dynamic homogenous AOP with PROSE. Technical report, Department of Computer Science, Federal Institute of Technology, Zurich, 2001.
- [34] Alexandre Oliva and Luiz Eduardo Buzato. The implementation of Guaraná on Java. Technical Report IC-98-32, Universidade Estadual de Campinas, September 1998.
- [35] 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)*, July 1999.
- [36] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking*, pages 95–106, September 2002.
- [37] S. Masoud Sadjadi. *Transparent Shaping for Existing Software to Support Pervasive and Autonomic Computing*. PhD thesis, Department of Computer Science, Michigan State University, East Lansing, United States, August 2004.
- [38] 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.