An Aspect-Oriented Approach to Dynamic Adaptation

Z. Yang, B. H. C. Cheng*, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, P. K. McKinley
Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
{yangzhe1,chengb,stire,sowellje,sadjadis,mckinley}@cse.msu.edu.

ABSTRACT

This paper presents an aspect-oriented approach to dynamic adaptation. A systematic process for defining where, when, and how an adaptation is to be incorporated into an application is presented. Specifically, the paper presents a two-phase approach to dynamic adaptation, where the first phase prepares a non-adaptive program for adaptation, and the second phase implements the adaptation at run time. This approach is illustrated with a distributed conferencing application.

1. INTRODUCTION

Software is considered to be *dynamically adaptive* if conditions in the executing environment cause new code to be introduced at run time to achieve new behavior not previously possible with the original code. In the context of self-healing systems, dynamic adaptation may be needed to correct erroneous behavior, add code to implement a new security policy, provide fault-tolerance, or improve performance. Unfortunately, the software development community currently lacks design methods and tools for rigorously constructing dynamically adaptive software. This paper proposes a systematic approach for preparing an existing program for adaptation and defining dynamic adaptations.

Extending an existing application to support dynamic adaptation is complicated. Consider, as a running example, an existing online conferencing application, where participants may enter and leave a collaborative conferencing session, and consider that we now want to add secure communication capabilities. Moreover, we want to be able to change the security characteristics dynamically at run time. A major challenge in effecting such a change is to make the existing application *adapt-ready*, that is, to extend the program so that new security logic can be loaded and unloaded at run time. Such an extension could affect many lines of code that implement network communication. Moreover, it is difficult to codify *a priori* the precise conditions under which an adaptation should occur and what responses are appropriate. For example, in order to protect a conferencing session against eavesdroppers, an offending condition occurs when participants from untrusted hosts attempt to communicate during the session, and a dynamically adaptive response is for the conferencing group members to continue their session using encryption.

A key insight to systematically making programs adapt-ready is recognizing that the *concerns* that tend to warrant dynamic adaptation (e.g., security, QoS, fault-tolerance) are cross-cutting in nature. This paper explores the hypothesis that aspect-oriented programming (AOP) [11], a technique for separating and composing cross-cutting concerns, can be leveraged to systematically extend a non-adaptive program into one that is adapt-ready. Specifically, AOP's facility for defining aspects may be used to introduce an *adaptation infrastructure*. For example, consider adding secure communication to an insecure program that communicates using sockets. Infrastructure is needed to intercept invocations of a send (or receive) operation and to enable processing of the data prior to (immediately after) invoking the operation. In this case, the adaptation infrastructure comprises wrappers for calls to the send and receive operations, as well as code that manages the dynamic insertion and removal of packet-processing filters.

A second insight is to encapsulate the logic for adapting the run time behavior of the program into an *adaptation kernel*. An adaptation kernel is an engine for firing *adaptation rules*, each of which comprises a *condition* under which an adaptation should occur and an *action* that indicates the appropriate adaptive response. We use aspects to weave calls (hereafer called *traps*) to the adaptation kernel into the application program, thereby completely separating the application program from the code dealing with adaptation.

This paper describes a two-phase process that makes use of AOP to support dynamic adaptation. First, at development time, we use aspects and weaving to extend an existing program with adaptation infrastructure and entry points into the adaptation kernel. At run time, the adaptation kernel checks the condition of each adaptation rule to determine if an adaptation should be performed and executes the corresponding actions if the condition is satisfied. Thus, the second phase in our approach is to encode the adaptation logic in terms of conditions and actions and to implement these as `condition` and `action` classes, from which objects in the adaptation kernel are allocated. Because these condition and action classes can be loaded and unloaded at run

*Please contact this author for all correspondences.

WOSS '02, Nov 18-19, 2002, Charleston, SC, USA.

time, the adaptation kernel itself is adaptive. We applied our approach to a case study in which we used the AOP language AspectJ [10] to extend a network conferencing application into one that dynamically adapts to incorporate new security features as needed.

The remainder of the paper is organized as follows. Section 2 introduces the AOP-based approach to dynamic adaptation and gives a detailed explanation of the example conferencing application. Section 3 overviews dynamic adaptation projects most relevant to our approach. Section 4 briefly summarizes this work and overviews future investigations.

2. AOP-BASED DYNAMIC ADAPTATION

This section describes the two-phased approach to dynamic adaptation using AOP. The first phase, occurring at development time, identifies points in the original program at which adaptation may occur; it also determines what types of supporting software (i.e., infrastructure) is needed to enable adaptation at those points. The second phase takes place at run time and encompasses the activities surrounding the actual adaptation, including the checking of conditions for adaptation and the insertion or removal of code for adaptation purposes. Both phases are illustrated by our sample conferencing application. For clarity, we begin by reviewing relevant aspect-oriented development terms and presenting the general architecture for our approach to dynamic adaptation.

2.1 Background and motivation for AOP

In our study, we used AspectJ, a compile-time, Java-based AOP language [10]. AspectJ extends Java with three new programming features: pointcuts, advice, and aspects. A *pointcut* declaratively specifies a collection of program statements, specifically method invocations, using regular-expression pattern matching, which is further constrained by type checking. For example, pointcut `receive`, depicted in Figure 1, specifies the collection of calls to the `receive` method on objects of class `MulticastSocket`. The match is further constrained to occur only inside methods of class `ConfDisplay` (line 5) because it is required that the current object (i.e., the object pointed to by `this`) is of type `ConfDisplay`. Moreover, the variables `cd`, `ms`, and `dp` are bound to the current object, the socket object, and the arguments to the `receive` method respectively. These bound variables will be available for use in the code for the aspect that weaves in new code at this pointcut. A similar pointcut is defined for `send`.

Pointcuts enable the precise definition of the program points at which dynamic adaptation may occur. A program is made adapt-ready by extending it with new code that (1) introduces and exploits adaptation infrastructure, and/or (2) traps into the adaptation kernel. This extension is performed, not by modifying the original program, hereafter called the *core program*, but rather by weaving in an aspect that makes the program adapt-ready with respect to a particular concern. Specifically, an aspect comprises code fragments, called *advice*, which can be woven in and around statements that are quantified using pointcuts. In our approach, advice comprises code that provides infrastructure support, such as the ability to insert/remove filters, that can then be used to perform adaptation-specific processing, such as dynamically inserting filters to encrypt or decrypt

```
1  pointcut receive(
2          ConfDisplay cd,
3          MulticastSocket ms,
4          DatagramPacket dp):
5     this(cd)
6     && target(ms)
7     && args(dp)
8     && call(public *
9        *..MulticastSocket.
10       receive(DatagramPacket));
11 );
12
13 pointcut send(
14         ConfClient cc,
15         MulticastSocket ms,
16         DatagramPacket dp):
17    this(cc)
18    && target(ms)
19    && args(dp)
20    && call(public *
21       *..MulticastSocket.
22       send(DatagramPacket));
23 );
```

Figure 1: Definition of the *send* and *receive* pointcuts

packets to provide secure communication, and/or calls into the adaptation kernel. It is these traps into the adaptation kernel that cause the program to dynamically adapt. Having developed such an aspect, we compile the core program with AspectJ's source-to-source compiler to produce the adapt-ready program.

2.2 Architecture of adaptive programs

Figure 2 illustrates the architecture of adaptive programs developed using our approach. The roundtangle encapsulates the process address space for the given adapt-ready program and its corresponding adaptation kernel. The *BehaviorAdaptor* refers to a contiguous code fragment that replaces a method invocation M in the core program with code that (1) pre-processes and transforms (in or in-out) data to M; (2) postprocesses (out or in-out) data parameters and return values from M; and (3) traps into the adaptation kernel in order to enable dynamic adaptation. When the program traps into the adaptation kernel, the *Adaptation Manager* checks the rule base for conditions that are satisfied for that concern. Once a given condition is satisfied, two possible actions can be taken. Either new code can be loaded to add new behavior to the adapt-ready program, or new adaptation rules (i.e., *condition-action* pairs) can be added to the adaptation kernel, thereby changing the adaptation logic for that aspect. The stack within the adaptation kernel indicates that each different adaptation concern might give rise to a different adaptation manager and rule base, where these two components are tightly coupled for a given concern. The shaded boxes indicate the elements that are not defined during development time, but may be loaded at run time depending on the actions in the rule base.

To explore these ideas, we extended a non-adaptive conferencing application with support for secure communication. The original (non-adaptive) application allows a user to join and leave a conferencing group and to send textual information to other online users in the same group. The application was built using multicast sockets (i.e., the Java class `MulticastSocket`) to communicate over the network.

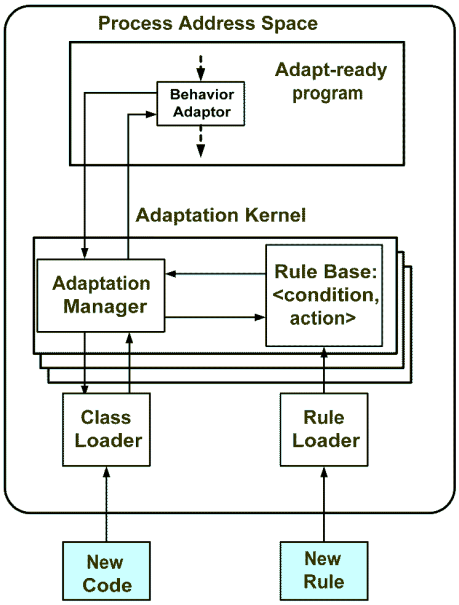

Figure 2: Dynamically adaptive program at run time

In the remainder of this section, we use this example to describe the main steps of our proposed methodology.

2.3 Phase I: Making a program adapt-ready

To make a core program into one that is adapt-ready, the designer must develop BehaviorAdaptors that extend the core program with adaptation infrastructure and traps into the adaptation kernel. These BehaviorAdaptors are encapsulated into an aspect that is then woven into the core program. In the conferencing application, the adaptation infrastructure comprises packet-filtering logic that is wrapped around network operations, specifically invocations of `send` and/or `receive` methods on objects of class `MulticastSocket`. BehaviorAdaptors wrap these invocations with code that redirects any outgoing data through a sequence of *datagram filters*, each of which is an active object that consumes datagram packets in order to produce new datagram packets. A similar filtering process is applied to incoming data.

Figure 3 illustrates the structure of our conference application after it has been extended with this filtering infrastructure. Whereas in the core code, the conferencing application (`Conf` in the figure) interacts directly with `MulticastSocket` objects, the adapt-ready `Conf` interacts instead with a sequence of filters, hereafter referred to as a *filter chain*. Filter chains encapsulate one or more datagram filters that are arranged in sequence and that eventually communicate directly with a `MulticastSocket` object. We designed the filter-chain infrastructure to allow datagram filters to be inserted and removed dynamically from these sequences as execution conditions change. Each pair of filters is interposed with a `FilterBuffer` object (depicted by circles in Figure 3), which enables each filter to run in its own thread and also simplifies the dynamic replacement of a running filter. Example filters might add forward error correction or encryption capability.

Figure 4 depicts a portion of the `FilterChainController` aspect, which transforms a program that interacts directly with multicast sockets into one that interacts with filter chains. Lines 6-11 introduce the filter chains that are depicted in Figure 3 as private data members of the core classes `ConfClient` and `ConfDisplay` (which are classes used by `Conf`). After weaving this aspect into the core code, objects of class `ConfClient` (respectively `ConfDisplay`) will contain a new attribute called `filterChain` that the original designer of these core classes never imagined needing. Additionally, both core classes (`ConfDisplay` and `ConfClient`) are extended with the operation `getFilterChain()` (not shown in figure) that returns a reference to these `filterChain` attributes.

The advice in Figure 4 is executed **around** each invocation that matches the `receive` pointcut. This advice means that each invocation of `receive` on a multicast socket object in the core code is effectively replaced with the line:

```
cd.getFilterChain().receive(ms, dp);
```

This BehaviorAdaptor statement retrieves a reference to the receiver filter chain (using the `getFilterChain` operation introduced by the aspect) and then invokes the receive method on this object. Each call to `receive(ms,dp)` on the filter-chain object returns a packet that has passed through all of the filters in the chain (e.g., one of the packets produced by filter R1 in Figure 3).

Using aspects and weaving, we are able to extend a core program with code that introduces and exploits adaptation infrastructure (in this case filter chains and datagram filters). In addition, an aspect can weave in traps into the adaptation kernel. In our study, we placed these traps in the filter-chain code, as opposed to the aspect, in order to allow multiple filters to be inserted depending on the environment conditions for different adaptation concerns. Regardless of how the traps are introduced, a program cannot adapt dynamically without them. For clarity, we use aspects, such as `FilterChainController`, to weave in infrastructure support for dynamic adaptations for different concerns, such as security and QoS. In the next section, we describe how adapt-ready code traps into the adaptation kernel and how this kernel accomplishes dynamic adaptation.

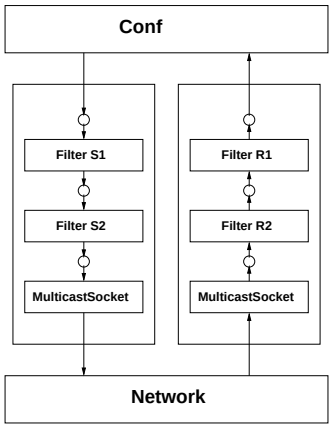

Figure 3: Architecture of Dynamically Adapted Conferencing Application

2.4 Phase II: Achieving Dynamic Adaptation

```
1  public aspect FilterChainController {
2  //
3  //  filter-chain objects that trap to
4  //  AdaptationManager
5  //
6  private SenderFilterChain edu.msu.cse.
7         sens.conf.ConfClient.filterChain
8                                  = null;
9  private RecverFilterChain edu.msu.cse.
10        sens.conf.ConfDisplay.filterChain
11                                 = null;
12
13
14 // receive pointcut definition elided
15 ...
16 // wrapper for core program receive opn
17 //
18  around( ConfDisplay cd,
19              MulticastSocket ms,
20              DatagramPacket dp)
21        :receive(cd, ms, dp)
22 { cd.getFilterChain().receive(ms,dp); }
23 }
```

Figure 4: Elided Aspect Code

Dynamic adaptation is achieved via the adaptation kernel, which is a loose federation of concern-specific adaptation managers that are explicitly invoked to check execution conditions and perform (concern-specific) adaptations as appropriate. To help clarify the role of an adaptation manager, consider the scenario where a developer wants to add a rule for dynamic adaptation for our running example. Two major steps need to be performed: (1) develop an *action class* that describes how an encryption filter should be inserted into a filter chain; (2) develop a rule that describes the conditions under which the adaptive action should take place and create an *adaptation action class* that describes how the action class (from the first step) should be loaded (e.g., following a URL pointer, loading a file from a directory, etc.). Therefore, an adaptation action class, using the context, implements the dynamic adaptation, and an action class contains the code that is actually dynamically loaded (e.g., encryption filter insertion). The adaptation manager then monitors the conditions for the rules and performs the corresponding actions according to the specifications in the adaptation action class. It suffices to understand the operation of one adaptation manager; thus we shall concentrate on the structure and operation of a concern-specific adaptation manager, `EncryptionAM`.

Figure 5 depicts the high-level structure of an adaptation manager. The abstract class `AdaptationManager` maintains a collection of 0 or more rule schemas, which collectively play the role of the rule base in Figure 2. Briefly, a *rule schema* is an object that encapsulates an encoding (usually a string) of an adaptation condition and an encoding (again typically a string) of an *adaptation action class*, which when loaded and instantiated can be *performed* to accomplish a dynamic adaptation. For example, one security feature that we implemented monitors the packets received during a conference to detect communication with insecure hosts. This detection is encoded as a condition in an instance of `ConditionSpecificRS` that matches the IP address of an incoming packet against a pattern of secure hosts. If the address does not match (i.e., the condition is satisfied), the adaptive action, coded as a subclass of `Action`, called `InsertDESFilters` (not shown due to space constraints) is to begin encrypted communication by adding a `DESEncoder` filter to the sender filter-chain and a `DESDecoder` filter to the receiver filter-chain. This `InsertDESFilters` *action*, i.e., the instantiation and subsequent insertion of filters into the filter-chains, is performed by running the `handle` method of an `EncryptionAction` object, whose class is a subclass of `ConcernSpecificAction`. Therefore, action objects are instantiated as required from adaptation action classes that are represented (usually as a string) inside a rule schema object.

Class `AdaptationManager` provides a method called `evaluate` that systematically iterates through each of these rule schemas to check conditions and perform actions. When we speak of traps into the adaptation kernel, we mean the invocation of the `evaluate` operation on a given adaptation manager. The `evaluate` operation is implemented using the *template method pattern*, which means it is a concrete method that invokes abstract *hook* operations, which are defined in derived classes [8]. As the figure illustrates, a concern-specific adaptation manager derives from `AdaptationManager` and provides methods for hook operations that set the context (`setCondContext`) and perform the necessary processing to carry out the action (`performAction`, such as follow a URL to an address that contains new code to be inserted).

At *development time*, we developed an aspect (`FilterChainController`) that transforms a program that interacts with sockets into one that interacts with filter chains, thereby enabling the insertion of filters for secure network interaction. This aspect definition contains adaptation infrastructure code, such as filter insertion methods, and may also contain traps into the adaptation kernel for `FilterChainController`[1]. At *compile time*, we wove this aspect definition into the core program to yield an adapt-ready program containing BehaviorAdaptors replacing the original `MulticastSocket send` and `receive` with respective calls in filter chains.

Finally, at *run time*, the adapt-ready program is instantiated and initialized to yield a *dynamically adaptive process* (denoted by the roundtangle). Depending on the conditions and actions in the adaptation kernel, numerous executable processes are possible, as depicted by the multiple arrows emanating from the run time roundtangle. Each BehaviorAdaptor in the running program uses a filter-chain object to "trap" to the concern-specific adaptation manager to determine which adaptation rules have been satisfied and which corresponding adaptation actions should be performed. In this particular example, the `RecverFilterChain` object traps to the `EncryptionAM` when a message has been received in order to determine if any of the encryption-based rules have been satisfied. If the message has been received from an unacceptable host, then `DESEncoderFilter` and `DESDecodeFilter` are dynamically inserted into the respective filter-chain sequences. In this manner, an insecure conferencing application is transformed into a dynamically adaptive, more secure application. Moreover, the dynamic adaptation code is encapsulated into a separate part of the program, making the conferencing application easier to develop and maintain.

[1]Though, as explained earlier, we chose to place these traps inside the code for the filter-chain classes for this particular concern.

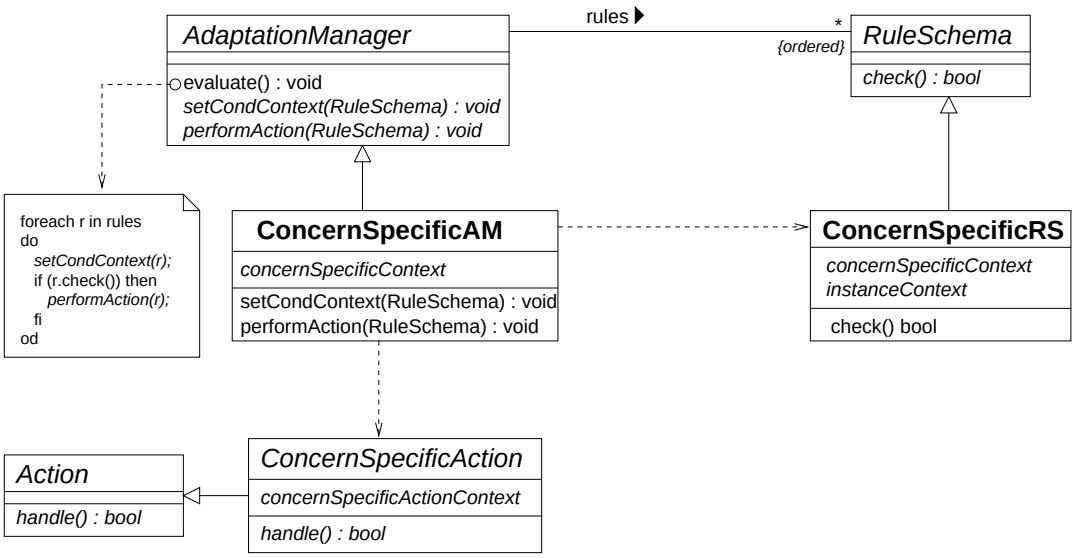

Figure 5: Adaptation manager structure

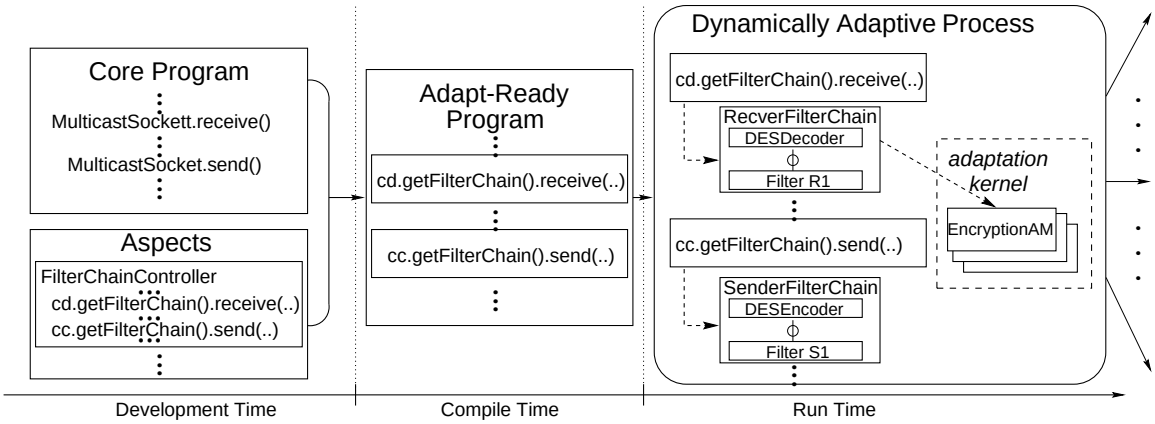

Figure 6: Achieving Dynamically Adaptive Code

3. RELATED WORK

For clarity and discussion purposes, we have identified three broad categories for adaptation, all centered around the different types of relationships between the original program and the adaptations.

The first category, termed *Static-Adapt*, refers to the approaches where the actual executable code does not change throughout the lifetime of a process: Several approaches yield programs with the *Static-Adapt* property. For example, approaches to dynamic adaptation based on programming-language extensions (e.g., Adve *et al.* [1] for C++ and Java and Mezini *et al.* [13] for Java) fall into the *Static-Adapt* category almost by default. In addition, the adaptive-component architecture of Chen, Hiltunen, and Schlicting [6] falls into the *Static-Adapt* category, as do Aksit's composition filters [4]. While all of these approaches are adaptive, they do not involve the loading or unloading of code at run time in ways that were not known at development time.

Another useful category of adaptive programs, termed *Trace-Adapt*, are those that add, modify, or remove code at run time (via loading or unloading classes) but that never modify or remove the *original* program statements, thereby providing *traceability* back to the original program. The approach presented in Section 2 yields programs that satisfy the *Trace-Adapt* property, as we disallow unloading any of the application's original classes. The Hadas project [3], which is closely related to ours, also falls into the *Trace-Adapt* category. A Hadas component comprises two sections, the *fixed* and the *extensible*. The extensible section contains the mutable portion of the components and can be changed according to the run-time environment, and the fixed section corresponds to the original program. Welch [15], on the other hand, uses user-defined dynamic Java-class loaders, in order to dynamically adapt a class with new code. However, these adaptations are always additive, which means that the original program is preserved. Other projects achieve adaptability via computational reflection [12], which is a technique that requires *meta-level* language support to enable a program to observe the behavior of itself during execution. It is interesting to note, however, that most of these approaches (e.g., Kasten *et al.* [9], Blair *et al.* [5], David *et al.* [7]) use reflection in such a way that the resulting programs still

satisfy the *Trace-Adapt* property.

Beyond the realm of adaptive programs that preserve the original source code, we can consider approaches such as the Kava [16] bytecode rewriting system, Akkawi's Dynamic Weaver Framework [2], and Iguana [14], a reflection-based system to associate new behavior with existing methods in the application. These systems are designed to modify the original program, thus we call them *Full-Adapt*. Once the elements of the original program are modified or removed, verification and validation tasks for the adapted program become less defined, since we lose traceability to the original program and its requirements.

Depending on how much *a priori* knowledge a developer has about changes in the environment and the level of flexibility needed to change an application's functionality to respond to those changes, different types of adaptation might be warranted.

While other approaches [7, 14] have used an AOP approach to dynamic adaptation, both use aspects to only define the pointcuts for adaptation, and use reflection to achieve the dynamic adaptation, where [7] falls into the *Static-Adapt* and [14] falls into the *Full-Adapt* category. In contrast, we are able to use aspects to encapsulate all code related to achieving an adaptation, and the run-time configurable adaptation kernel is used to manage the dynamic adaptations.

4. CONCLUSIONS

The aspect-oriented approach to dynamic adaptation that we presented provides a technique to fully separate the application code from the dynamic adaptation concern. Furthermore, the adaptation kernel not only supports run time changes to a program's functionality that differs from that of the core program, but it also supports the dynamic adaptation of the adaptation kernel itself. Currently, we have two major limitations with this approach. First, adaptation is centered around the pointcut definitions included in the aspect specification. Because our aspects were at a low-level, i.e., multicast send/receive, we were able to support multiple adaptation managers for a given aspect, ranging from security to QoS since all of them involved some type of communication between objects that could be affected by inserting filters. A second (related) limitation is the lack of a generic aspect definition, thus requiring a given pointcut to be tied closely with a class of adaptations. We are investigating generic adaptation aspect definitions to address limitations to the current approach. Future work will also explore developing more fine-grained formal property specifications of dynamic adaptation. Finally, we have discussed only adaptation related to security, but the proposed methods can be used to support self-healing and adaptation in other cross-cutting concerns.

Acknowledgements

The authors greatly appreciate the input provided by G. Kiczales during the early stages of this work. This work has been supported in part by NSF grants EIA-0130724, EIA-0000433, CCR-9984726, CCR-9912407, CCR-9901017, CDA-9700732, Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, Defense Advanced Research Projects Agency (DARPA) and by the Air Force Research Laboratory under Contract No. F30602-00-2-0618. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or the United States Air Force.

5. REFERENCES

[1] V. Adve, V. V. Lam, and B. Ensink. Language and compiler support for adaptive distributed applications. Technical Report UIUCDCS-R-2001-2220, University of Illinois at Urbana-Champaign, 2001.

[2] F. Akkawi and et al. Dynamic weaving for building reconfigurable software systems. In *Proc. of OOPSLA2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, Florida, October 2001.

[3] I. Ben-Shaul, O. Holder, and B. Lavva. Dynamic adaptation and deployment of distributed components in Hadas. *IEEE Transactions on Software Engineering*, 27(9):769–787, September 2001.

[4] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of ACM*, 44(10):51–57, October 2001.

[5] G. Blair, G. Coulson, and N. Davies. Adaptive middleware for mobile multimedia applications. In *Proceedings of the 8th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 259–273, 1997.

[6] W.-K. Chen, M. A. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 635–643, Phoenix, AZ, Apr 2001.

[7] P.-C. David and et al. Two-step weaving with reflection using AspectJ. In *Proc. of OOPSLA2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, Florida, October 2001.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] E. P. Kasten, P. K. McKinley, S. M. Sadjadi, and R. E. K. Stirewalt. Separating introspection and intercession in metamorphic distributed systems. In *Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCS'02)*, Vienna, Austria, July 2002.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

[11] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[12] P. Maes. Concepts and experiments in computational reflection. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 147–155, New York, NY, 1987. ACM Press.

[13] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In M. Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer Academic Press, 2000. University of Twente, The Netherlands.

[14] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behavior. In *Proc. 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, Malaga, Spain, 2002.

[15] I. Welch and R. Stroud. Dynamic adaptation of the security properties of applications and components. In *ECOOP Workshop on Distributed Object Security*, Brussels, Belgium, 1998.

[16] I. Welch and R. Stroud. Kava – using bytecode rewriting to add behavioural reflection to Java. In *Proc. USENIX Conference on Object-Oriented Technology*, 2001.