Composing Adaptive Software

Compositional adaptation enables software to modify its structure and behavior dynamically in response to changes in its execution environment. A review of current technology compares how, when, and where recomposition occurs.

Philip K. McKinley
Seyed Masoud Sadjadi
Eric P. Kasten
Betty H.C. Cheng
Michigan State University

Interest in adaptive computing systems has increased dramatically in the past few years, and a variety of techniques now allow software to adapt dynamically to its environment. Two revolutions in the computing field are driving this development. First is the emergence of *ubiquitous computing*,[1] which focuses on dissolving traditional boundaries for how, when, and where humans and computers interact. For example, mobile computing devices must adapt to variable conditions on wireless networks and conserve limited battery life. Second is the growing demand for *autonomic computing*,[2] which focuses on developing systems that can manage and protect themselves with only high-level human guidance. This capability is especially important to systems such as financial networks and power grids that must survive hardware component failures and security attacks.

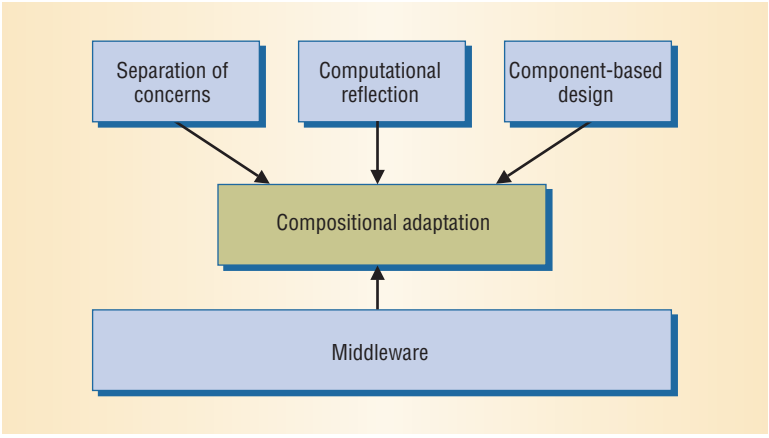

Figure 1. Main technologies supporting compositional adaptation: separation of concerns, computational reflection, and component-based design.

There are two general approaches to implementing software adaptation. *Parameter adaptation* modifies program variables that determine behavior. The Internet's Transmission Control Protocol is an often-cited example: TCP adjusts its behavior by changing values that control window management and retransmissions in response to apparent network congestion. But parameter adaptation has an inherent weakness. It does not allow new algorithms and components to be added to an application after the original design and construction. It can tune parameters or direct an application to use a different existing strategy, but it cannot adopt new strategies.

By contrast, *compositional adaptation* exchanges algorithmic or structural system components with others that improve a program's fit to its current environment. With compositional adaptation, an application can adopt new algorithms for addressing concerns that were unforeseen during development. This flexibility supports more than simple tuning of program variables or strategy selection. It enables dynamic recomposition of the software during execution—for example, to switch program components in and out of a memory-limited device or to add new behavior to deployed systems.

Dynamic recomposition of software dates back to the earliest days of computing, when self-modifying code supported runtime program optimization and explicit management of physical memory. However, such programs were difficult to write and debug. Several new software tools and technologies now help address these problems. Given the increasing pace of research in compositional adaptation, we offer a review of the supporting technologies, proposed solutions, and areas that require further study.

Published by the IEEE Computer Society

Middleware and Adaptation

Much recent research in adaptive software focuses on middleware—the layers of services separating applications from operating systems and network protocols.

Douglas Schmidt decomposes middleware into four layers,[1] shown in Figure A:

- *Host-infrastructure middleware* resides atop the operating system and provides a high-level API that hides the heterogeneity of hardware devices, operating systems, and—to some extent—network protocols.
- *Distribution middleware* provides a high-level programming abstraction, such as remote objects, enabling developers to write distributed applications in a way similar to stand-alone programs. Corba, DCOM, and Java RMI all fit in this layer.
- *Common middleware* services include fault tolerance, security, persistence, and transactions involving entities such as remote objects.
- *Domain-specific middleware* services are tailored to match a particular class of applications.

Most adaptive middleware is based on an object-oriented programming paradigm and derived from popular middleware platforms such as Corba, Java RMI, and DCOM/.NET.

Many adaptive middleware approaches work by intercepting and modifying messages. Figure B shows the flow of a request-reply sequence in a simplified Corba client-server application. This application comprises two autonomous programs hosted on two computers connected by a network.

Assume that the *client* has a valid Corba reference to the *servant* object. The client request to the servant goes first to the *stub*, which represents the Corba object on the client side. The stub marshals the request and sends it to the client *object request broker*. The client ORB sends the request to the server ORB, where a *skeleton* unmarshals the request and delivers it to the servant. The servant replies to the request, by way of the server ORB and skeleton. The client ORB will receive the reply and dispatch it to the client.

In recent years, numerous studies have addressed the issue of how middleware can adapt to dynamic, heterogeneous environments to better serve applications.[2,3] Middleware traditionally hides resource distribution and platform heterogeneity from the application business logic. Thus it is a logical place to put adaptive behavior that is related to crosscutting concerns such as QoS, energy management, fault tolerance, and security policy.

References

1. D.C. Schmidt, "Middleware for Real-Time and Embedded Systems," *Comm. ACM*, June 2002, pp. 43-48.
2. *Comm. ACM*, special issue on adaptive middleware, June 2002, pp. 30-64.
3. *IEEE Distributed Systems Online*, special issue on reflective middleware, June 2001; http://dsonline.computer.org/0105/features/gei0105.htm.

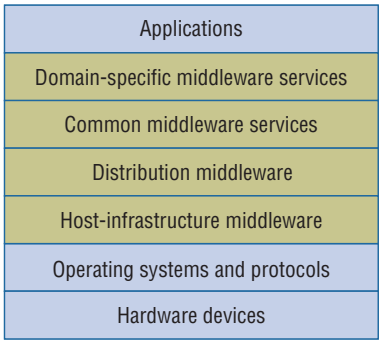

Figure A. Four-layer decomposition of middleware to bridge the gap between an application program and the underlying operating systems, network protocols, and hardware devices.

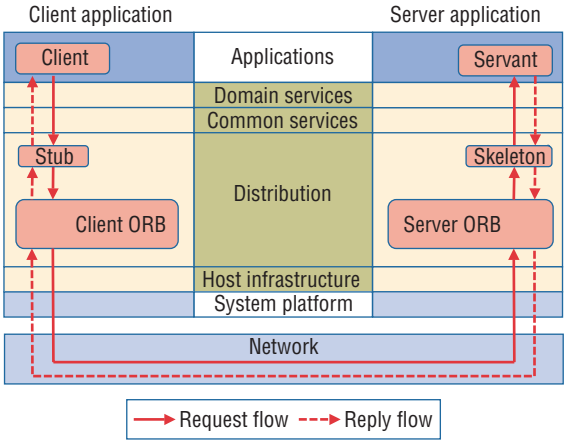

Figure B. Corba call sequence for a simplified client-server application.

ENABLING TECHNOLOGIES

At the core of all approaches to compositional adaptation is a level of indirection for intercepting and redirecting interactions among program entities. Figure 1 shows three technologies—separation of concerns, computational reflection, and component-based design—that we consider as key to reconfigurable software design. Programmers can use these technologies to construct self-adaptive systems in a systematic and principled—as opposed to ad hoc—manner.[3]

In addition, the widespread use of middleware in distributed computing has been a catalyst for compositional adaptation research. Middleware provides a natural place to locate many types of adaptive behavior, as the "Middleware and Adaptation" sidebar describes.

Separation of concerns

Separation of concerns[4] enables the separate development of an application's functional behavior—that is, its business logic—and the code for

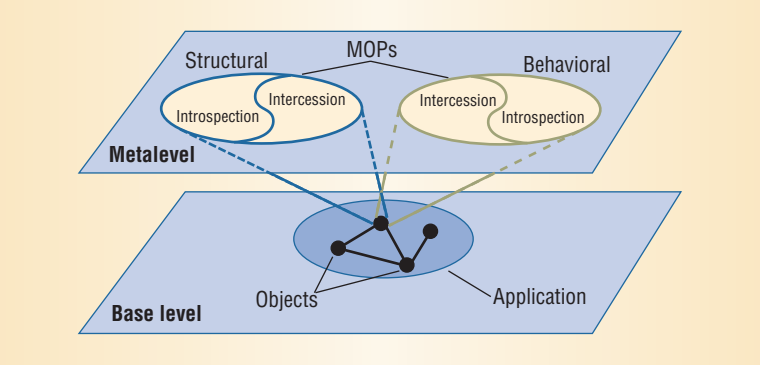

Figure 2. Metalevel understanding collected into metaobject protocols.

crosscutting concerns, such as quality of service (QoS), energy consumption, fault tolerance, and security. An application cannot implement a crosscutting concern at a single program location; instead, it must add the code at many places. Separating crosscutting concerns from functional behavior simplifies development and maintenance, while promoting software reuse.

Separation of concerns has become an important principle in software engineering.[5] Presently, the most widely used approach appears to be aspect-oriented programming.[6] AOP provides abstraction techniques and language constructs to manage crosscutting concerns. The code implementing these concerns, called *aspects*, is developed separately from other parts of the system. In AOP, *pointcuts* are sets of locations in the code where the developer can weave in aspects. Pointcuts are typically identified during development. Later, for example during compilation, the developer uses a specialized compiler, called an *aspect weaver*, to combine different aspects with an application's business logic to create a program with new behavior. An example is the AspectJ compiler. AOP proponents argue that disentangling crosscutting concerns leads to simpler software development, maintenance, and evolution.

AOP is important to dynamic recomposition because most adaptations are relative to some crosscutting concern, such as QoS. AOP enables these concerns to be isolated from the rest of the program. However, in traditional AOP the compiled program is still tangled. To support dynamic recomposition, the programmer needs a way to maintain this separation at runtime.

Computational reflection

Computational reflection refers to a program's ability to reason about, and possibly alter, its own behavior.[7] Reflection enables a system to reveal selected details of its implementation without compromising portability.

Reflection comprises two activities: *introspection* to let an application observe its own behavior, and *intercession* to let a system or application act on these observations and modify its own behavior. In a self-auditing distributed application, for example, software "sensors" could use introspection to observe and report usage patterns for various components. Intercession would allow the system to insert new types of sensors, as well as components that implement corrective action, at runtime.

As Figure 2 shows, a reflective system (represented as base-level objects) and its self-representation (represented as metalevel objects) are causally connected, meaning that modifications to either one will be reflected in the other.

A metaobject protocol (MOP) is an interface that enables "systematic" introspection and intercession of the base-level objects. MOPs support either structural or behavioral reflection.[3] *Structural reflection* addresses issues related to class hierarchy, object interconnection, and data types. As an example, a metalevel object can examine a base-level object to determine what methods are available for invocation. Conversely, *behavioral reflection* focuses on the application's computational semantics. For instance, a distributed application can use behavioral reflection to select and load a communication protocol well suited to current network conditions.

A developer can use reflective services that are either native to a programming language—such as Common Lisp Object System (CLOS), Python, or various Java derivatives—or provided by a middleware platform. When combined with AOP, reflection enables a MOP to weave code for crosscutting concerns into an application at runtime. However, dynamically loading and unloading adaptive code requires the target software modules to exhibit a "plug-and-play" capability.

Component-based design

The third major technology supporting compositional adaptation is component-based design. *Software components* are software units that third parties can independently develop, deploy, and compose.[9] Popular component-based platforms include COM/DCOM, .NET, Enterprise Java Beans, and the Corba Component Model.

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. To provide dynamic recomposition, a component-based framework must support late binding, which enables coupling of compatible components at run-

Table 1. Example research projects, commercial packages, and standard specifications that provide compositional adaptation.

Project	Institution/Organization
Language-based projects	
AspectJ	Xerox Palo Alto Research Center
Composition filters	Universiteit Twente, The Netherlands
Program Control Language (PCL)	University of Illinois
Open Java	IBM Research
R-Java	University Federal de São Carlos, Brazil
Kava	University of Newcastle, UK
Adaptive Java	Michigan State University
Transparent Reflective Aspect Programming in Java (TRAP/J)	Michigan State University
Middleware-based projects	
Domain-specific services layer:	
Boeing Bold Stroke (BBS)	Boeing
Common services layer:	
CorbaServices	Object Management Group
Quality objects (QuO)	BBN Technologies
Adaptive Corba Template (ACT)	Michigan State University
Interoperable Replication Logic (IRL)	University of Rome, Italy
Distribution layer:	
.NET remoting	Microsoft
Open ORB and Open COM	Lancaster University, UK
The ACE ORB (TAO) and Component Integrated ACE ORB (CIAO)	Distributed Object Computing Group
DynamicTAO and Universally Interoperable Core (UIC)	University of Illinois
Orbix, Orbix/E, and ORBacus	Iona Technologies
Squirrel	University of Kaiserslautern, Germany
AspectIX	Friedrich-Alexander University, Germany
Host infrastructure layer:	
Java virtual machine (JVM)	Sun Microsystems
Common Language Runtime (CLR)	Microsoft
Iguana/J	Trinity College, Dublin
Prose	Swiss Federal Institute of Technology
Adaptive Communication Environment (ACE)	Distributed Object Computing Group
Ensemble	Cornell University
Cross-layer projects	
Distributed Extensible Open Systems (DEOS)	Georgia Institute of Technology
Grace	University of Illinois

time through well-defined interfaces used as contracts. In addition, to provide consistency with other applications, a component-based framework must support coexistence of multiple versions of components.

By enabling the assembly of off-the-shelf components from different vendors, component-based design promotes software reuse. Moreover, mechanisms for maintaining a program's component structure after the initial deployment, when combined with late binding, facilitate compositional adaptation.

Middleware and other factors

In addition to the three main technologies supporting dynamic recomposition, many other factors have contributed to the growth in this area. Perhaps the most important is middleware's increasing role in distributed computing. Middleware provides a layer that developers can exploit to implement adaptive behavior. Indeed, many approaches to compositional adaptation are realized in various middleware layers.

Other technologies important to adaptive software design include software design patterns, mobile agents, generative programming, adaptive programming, and intentional programming.[5]

COMPOSITIONAL ADAPTATION TAXONOMY

Researchers and developers have proposed a wide variety of methods for supporting compositional adaptation. Table 1 lists several research projects, commercial software packages, and standard specifications that support some form of compositional adaptation. The list is by no means exhaustive. Rather, it includes projects that exemplify the

Table 2. Software recomposition techniques.

Technique	Description	Examples
Function pointers	Application execution path is dynamically redirected through modification of function pointers.	Vtables in COM, delegates and events in .NET, callback functions in Corba
Wrappers	Objects are subclassed or encapsulated by other objects (wrappers), enabling the wrapper to control method execution.	ACE, R-Java, PCL, QuO, TRAP/J
Proxies	Surrogates (proxies) are used in place of objects, enabling the surrogate to redirect method calls to different object implementations.	ACT, AspectIX
Strategy pattern	Each algorithm implementation is encapsulated, enabling transparent replacement of one implementation with another.	DynamicTAO and UIC
Virtual component pattern	Component placeholders (virtual components) are inserted into the object graph and replaced as needed during program execution.	ACE and TAO
Metaobject protocol	Mechanisms supporting intercession and introspection enable modification of program behavior.	Open Java, Kava, TRAP/J, Open ORB, Open COM, Iguana/J
Aspect weaving	Code fragments (aspects) that implement a crosscutting concern are woven into an application dynamically.	AspectJ, Composition Filters, TRAP/J, AspectIX, Iguana/J, Prose
Middleware interception	Method calls and responses passing through a middleware layer are intercepted and redirected.	ACT, IRL, Prose
Integrated middleware	An application makes explicit calls to adaptive services provided by a middleware layer.	Adaptive Java, Orbix, Orbix/E, ORBacus, BBS, CIAO, Iguana/J, Ensemble

distinctions in a taxonomy we have developed based on how, when, and where software composition takes place. We have applied the taxonomy to many additional projects.[10]

How to compose

The first dimension of our taxonomy addresses the specific software mechanisms that enable compositional adaptation. Table 2 lists several key techniques with brief descriptions and examples. Mehmet Aksit and Zièd Choukair[8] provide an excellent discussion of such methods.

All of the techniques in Table 2 create a level of indirection in the interactions between program entities. Some techniques use specific software design patterns to realize this indirection, whereas others use AOP, reflection, or both. The two middleware techniques both modify interaction between the application and middleware services, but they differ in the following way: Middleware interception is not visible to the application, whereas integrated middleware provides adaptive services invoked explicitly by the application.

We use the term *composer* to refer to the entity that uses these techniques to adapt an application. 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—an aspect weaver, a component loader, a runtime system, or a metaobject. Indeed, autonomic computing promises that, increasingly, composers will be software components.

When and where the composer modifies the program determines the *transparency* of the recomposition. Transparency refers to whether an application or system is aware of the "infrastructure" needed for recomposition. For example, a middleware approach to adaptation is transparent with respect to the application source code if the application does not need to be modified to take advantage of the adaptive features. Different degrees of transparency (with respect to application source, virtual machine, middleware source, and so on) determine both the proposed solution's portability across platforms and how easily it can add new adaptive behavior to existing programs.[10]

When to compose

Second, we differentiate approaches according to when the adaptive behavior is composed with the business logic. Generally speaking, later composition time supports more powerful adaptation methods, but it also complicates the problem of ensuring consistency in the adapted program. For example, when composition occurs at development, compile, or load time, dynamism is limited but it is easier to ensure that the adaptation will not produce anomalous behavior. On the other

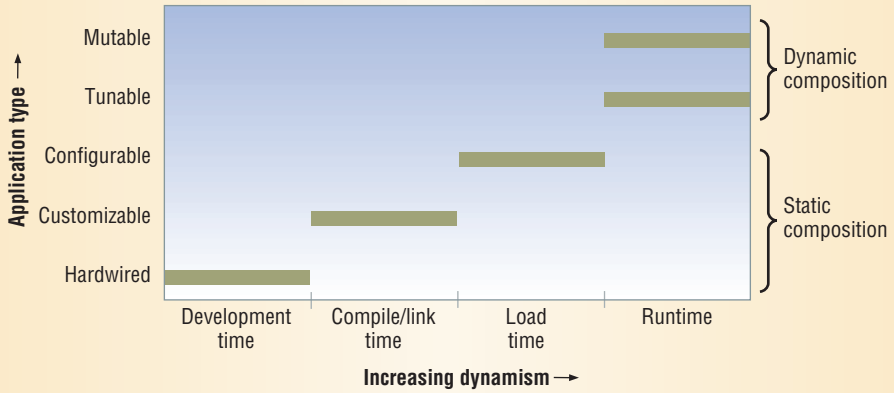

Figure 3. Classification for software composition using the time of composition or recomposition as a classification metric.

hand, while runtime composition is very powerful, it is difficult to use traditional testing and formal verification techniques to check safety and other correctness properties. Figure 3 illustrates the use of composition time as the classification metric for adaptive applications. The vertical axis lists application types that implement either static or dynamic composition. *Static* composition methods take place at development, compile, or load time, whereas *dynamic* composition refers to methods that a composer can apply at runtime.

Static composition. If an adaptive program is composed at development time, then any adaptive behavior is *hardwired* into the program and cannot be changed without recoding.

Alternatively, a developer or user can implement a limited form of adaptation at compile time or link time by configuring the application for a particular environment. For example, aspect-oriented programming languages such as AspectJ enable weaving of aspects into programs during compilation. Aspects might implement an environment-specific security or fault-tolerance policy. Such *customizable* applications require only recompilation or relinking to fit to a new environment.

Configurable applications delay the final decision on the algorithmic units to use in the current environment until a running application loads the corresponding component. For example, the Java virtual machine (JVM) loads classes when a Java application first uses them. Although we consider load-time composition a type of static composition, it offers more dynamism than other static methods. When the application requests the loading of a new component, decision logic might select from a list of components with different capabilities or implementations, choosing the one that most closely matches the current needs. For example, if a user starts an application on a handheld computer, the runtime system might load a minimal display component to guarantee proper presentation.

Other load-time approaches work by dynamically modifying the class itself as it is loaded. For example, to provide runtime monitoring and debugging capabilities, Kava enables the JVM to modify the bytecode as it loads a class.

Dynamic composition. The most flexible approaches to compositional adaptation implement it at runtime. A composer can replace or extend algorithmic and structural units during execution without halting and restarting the program. We differentiate two types of approaches according to whether or not the composer can modify the application's business logic.

Tunable software prohibits modification of code for the business logic. Instead, it supports fine-tuning of crosscutting concerns in response to changing environmental conditions, such as dynamic conditions encountered in mobile computing environments. An example is the fragment object model used in AspectIX, which enables runtime tuning of a Corba application's distribution behavior.

In contrast, *mutable software* allows the composer to change even the program's imperative function, enabling dynamic recomposition of a running program into one that is functionally different. For example, in the OpenORB middleware platform, all objects in the middleware and application code have reflective interfaces, so at runtime the reflective application can change virtually any object in any way, including modifying its interface and internal implementation. While very powerful, in most cases the developer must constrain this flexibility to ensure the system's integrity across adaptations.

Where to compose

The final dimension in which we compare approaches to compositional adaptation centers on where in the system the composer inserts the adaptive code. The possibilities include one of the middleware layers (see Figure A in the "Middleware and Adaptation" sidebar) or the application code itself. In this survey, we do not discuss changes to the operating system; however, we note that operating system extensibility is an active research area. Moreover, adaptations in cross-layer frameworks

Introducing adaptive behavior in higher middleware layers enables portability across virtual machines.

such as DEOS and Grace involve the cooperation of the operating system, middleware, and application.

Middleware layers. Projects involving compositional adaptation at the host-infrastructure middleware layer generally fall in one of two groups. One approach is to construct a layer of adaptable communication services. ACE is an early example that used service wrappers and C++ dynamic binding to support adaptable interprocess communication and event handling services. Ensemble provides a layered architecture that enables a distributed application to select a particular communication protocol.

The second approach is to provide a virtual machine with facilities to intercept and redirect interactions in the functional code. For example, JVM and common language runtime (CLR) facilitate dynamic recomposition through reflection facilities provided by the Java language and .NET platform, respectively. R-Java supports metaobjects by adding a new instruction to the Java interpreter, while Prose and Iguana/J use aspect weaving to add behavioral reflection to the standard JVM. In general, approaches in this category are very flexible with respect to dynamic reconfiguration in that they allow new code to be introduced at runtime. However, they use customized virtual machines to provide transparency to the application, which may reduce portability.

Introducing adaptive behavior in higher middleware layers—distribution, common services, and domain-specific services—enables portability across virtual machines. These approaches typically involve middleware components that intercept messages associated with remote method invocations and redirect or modify them in a manner that accounts for current conditions. For some frameworks, the application developer constructs explicit calls to adaptive middleware services. Examples include Orbix, Orbix/E, ORBacus, CIAO, and Boeing Bold Stroke. QuO uses wrappers around Corba stubs and skeletons to gain control of the call sequence, whereas IRL and ACT use Corba portable interceptors to do so. Portable interceptors serve as "generic" hooks that a composer can use at runtime to load other types of interceptors. Since a user can load a portable interceptor using a command-line parameter, this approach enables the composer to integrate adaptive components into the program without modifying either the application or the middleware code.

Application code. Although middleware approaches support transparent adaptation, they apply only to programs that are written against a specific middleware platform. A more general approach is for developers to implement compositional adaptation in the application program itself.

Two main techniques are available. The first is to program all or part of the application code using a language that directly supports dynamic recomposition. Some languages, such as CLOS or Python, provide support inherently, while others have been extended to support adaptation. For example, Open Java, R-Java, Handi-Wrap, PCL, and Adaptive Java all extend Java to include new keywords and constructs that enhance the adaptive code's expressiveness. However, this approach requires the developer to use these features explicitly in constructing the program.

The second technique is to weave the adaptive code into the functional code. AspectJ and Composition Filters weave adaptive behavior into existing applications at compile time. In contrast, tools such as TRAP/J use a two-step approach to enable dynamic recomposition. In the first step, an aspect weaver inserts generic interception hooks, in this case implemented as aspects, into the application code at compile time. In the second step, a composer dynamically weaves new adaptive components into the application at runtime, and a metaobject protocol uses reflection to forward intercepted operations to the adaptive components. This approach offers a way to add adaptive behavior to existing applications transparently with respect to the original code. Such a capability is important as users expect legacy applications to execute effectively across an increasingly diverse computing infrastructure.

KEY CHALLENGES

Despite many advances in mechanisms to support compositional adaptation, the full potential of dynamically recomposable software systems depends on fundamental advances on four other fronts.

Assurance

Recomposable software design requires a programming paradigm that supports automated checking of both functional and nonfunctional system properties.[11]

To help ensure the adapted system's correctness, developers must first certify all components for correctness with respect to their specifications. They can obtain this certification either by selecting com-

ponents that have already been verified and validated offline using traditional techniques, such as testing, inspection, and model checking, or by generating code automatically from specifications. The certification can include nonfunctional requirements, such as security and performance, as well as functional requirements.

Second, techniques are needed to ensure that the system still executes in an acceptable, or *safe*, manner during the adaptation process. Our group and others are using dependency analysis to address this problem. In addition, developers can use high-level contracts[12] and invariants to monitor system correctness before, during, and after adaptation.

Security

Whereas assurance deals primarily with system integrity, security addresses protection from malicious entities—preventing would-be attackers from exploiting the adaptation mechanisms. In addition to verifying component sources, an adaptive software system must protect its core from attackers. Various well-studied security mechanisms are available, such as strong encryption to ensure the confidentiality and authenticity of messages related to adaptation.

However, the system must also hide adaptation management from would-be intruders and prevent them from impeding or corrupting the adaptation process. A comprehensive approach to this problem must ensure the integrity of the data used in decision-making and conceal the adaptive actions, perhaps by obscuring them within other system activities.

Interoperability

Distributed systems that can adapt to their environment must both adapt individual components and coordinate adaptation across system layers and platforms. Software components are likely to come from different vendors, so the developer may need to integrate different adaptive mechanisms to meet an application's requirements. The problem is complicated by the diversity of adaptive software approaches at different system layers. Even solutions within the same layer are often not compatible.

Developers need tools and methods to integrate the operation of adaptive components across the layers of a single system, among multiple computing systems, and between different adaptive frameworks.

Decision making

Adaptive systems respond to a dynamic physical world. They must act autonomously, modifying software composition to better fit the current environment while preventing damage or loss of service. Decision-making software uses input from software and hardware sensors to decide how, when, and where to adapt the system. Interactive systems may even require the decision maker to learn about and adapt to user behavior.

The system must also hide adaptation management from would-be intruders.

Some researchers have constructed software decision makers using rule-based approaches or control theory. Others have designed decision makers whose actions are inspired by biological processes, such as the human nervous system and emergent behavior in insect species that form colonies.

These approaches to decision making in adaptive software have been effective in certain domains, but environmental dynamics and software complexity have limited their general application. More extensive research in decision making for adaptive software is needed. Future systems must accommodate high-dimensional sensory data, continue to learn from new experience, and take advantage of new adaptations as they become available.

Many of the mechanisms for compositional adaptation are available now, and we expect their use to increase as programmers become more familiar with adaptive software technologies and society comes to expect computer systems to manage themselves. There is a potential downside, however, in the lack of supporting development environments. Compositional adaptation is powerful, but without appropriate tools to automatically generate and verify code, its use can negatively impact—rather than improve—system integrity and security.

The computer science community must build development technologies and tools, well grounded in rigorous software engineering, to support compositional adaptation. This foundation will raise the next generation of computing to new levels of flexibility, autonomy, and maintainability without sacrificing assurance and security. ■

Acknowledgments

We express our gratitude to the many individuals who have contributed to this emerging area of study. Discussions with researchers associated with many of the projects listed in Table 1 have greatly improved our understanding of this area. We also thank the faculty and students in the Software Engineering and Network Systems Laboratory at

Michigan State University for their contributions to RAPIDware, Meridian, and related projects.

This work was supported in part by National Science Foundation grants CCR-9901017, CCR-9912407, EIA-0000433, EIA-0130724, and ITR-0313142, and by the US Department of the Navy, Office of Naval Research, under grant no. N00014-01-1-0744.

Further information

Our group is participating in compositional adaptation research through two projects: RAPIDware (www.cse.msu.edu/rapidware) addresses adaptive software for protecting critical infrastructures, and Meridian (www.cse.msu.edu/meridian) addresses automated software engineering for mobile computing. Among other artifacts, these projects produced ACT, Adaptive Java, and TRAP/J. The technical report on our taxonomy is a "living document" available through the RAPIDware URL.

References

1. M. Weiser, "Hot Topics: Ubiquitous Computing," *Computer*, Oct. 1993, pp. 71-72.
2. J.O. Kephart and D.M. Chess, "The Vision of Autonomic Computing," *Computer*, Jan. 2003, pp. 41-50.
3. G.S. Blair et al., "An Architecture for Next-Generation Middleware," *Proc. IFIP Int'l Conf. Distributed Systems Platforms and Open Distributed Processing* (Middleware 98), Springer, 1998, pp. 191-206.
4. D.L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Comm. ACM*, Dec. 1972, pp. 1053-1058.
5. K. Czarnecki and U. Eisenecker, *Generative Programming*, Addison-Wesley, 2000.
6. G. Kiczales et al., "Aspect-Oriented Programming," *Proc. European Conf. Object-Oriented Programming* (ECOOP), LNCS 1241, Springer-Verlag, 1997, pp. 220-242.
7. P. Maes, "Concepts and Experiments in Computational Reflection," *Proc. ACM Conf. Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA), ACM Press, 1987, pp. 147-155.
8. M. Aksit and Z. Choukair, "Dynamic, Adaptive, and Reconfigurable Systems Overview and Prospective Vision," *Proc. 23rd Int'l Conf. Distributed Computing Systems Workshops* (ICDCSW03), IEEE CS Press, May 2003, pp. 84-89.
9. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed., Addison-Wesley, 2002.
10. P.K. McKinley et al., "A Taxonomy of Compositional Adaptation," tech. report MSU-CSE-04-17, Dept. Computer Science and Engineering, Michigan State Univ., 2004.
11. N. Venkatasubramanian, "Safe 'Composability' of Middleware Services," *Comm. ACM*, June 2002, pp. 49-52.
12. A. Beugnard et al., "Making Components Contract Aware," *Computer*, July 1999, pp. 38-45.

Philip K. McKinley *is a professor in the Department of Computer Science and Engineering at Michigan State University. His research interests include adaptive middleware, mobile computing, pervasive computing, distributed systems, and group communication. McKinley received a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of the IEEE Computer Society and the ACM. Contact him at mckinley@cse.msu.edu.*

Sayed Masoud Sadjadi *is a PhD candidate in the Department of Computer Science and Engineering at Michigan State University. His research interests include adaptive software, middleware, pervasive computing, autonomic computing, and sensor networks. Sadjadi received an MS in software engineering from Azad University at Tehran. He is a student member of the IEEE Computer Society and the ACM. Contact him at sadjadis@cse.msu.edu.*

Eric P. Kasten *is a PhD candidate in the Department of Computer Science and Engineering and a software developer in the National Superconducting Cyclotron Laboratory, both at Michigan State University. His research interests include autonomic computing and learning algorithms for adaptable software. Kasten received an MS in computer science from Michigan State University. He is a member of the IEEE Computer Society and the ACM. Contact him at kasten@cse.msu.edu.*

Betty H.C. Cheng *is a professor in the Department of Computer Science and Engineering at Michigan State University. Her research interests include formal methods for software engineering, component-based software development, object-oriented analysis and design, embedded systems development, and visualization. Cheng received a PhD in computer science from the University of Illinois at Urbana-Champaign. She is a senior member of the IEEE Computer Society and a member of the ACM. Contact her at chengb@cse.msu.edu.*