Perimorph: Run-Time Composition and State Management for Adaptive Systems *

E. P. Kasten and P. K. McKinley

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
{kasten,mckinley}@cse.msu.edu

Abstract

This paper addresses a key issue that arises in run-time recomposition of software: the transfer of nontransient state between old components and their replacements. We focus on the concept of *collateral change*, which refers to the set of recomposition actions that must be applied atomically for continued correct execution. We describe Perimorph, a system that supports compositional adaptation of both functional and nonfunctional concerns by explicitly addressing collateral change. The operation of Perimorph is demonstrated through the implementation and testing of a 2D/3D digital elevation mapping application that supports recomposition and handoff among networked devices with varying capabilities.

Keywords: adaptive middleware, component-based design, collateral change, run-time composition, state management, compositional adaptation, mobile computing

1 Introduction

Distributed applications are pervasive in today's world. In part, driven by the expansion of the Internet and the desire for mobility, today's computer users require quality-of-service guarantees, security and flexibility from a multitude of platforms. Moreover, changing requirements, combined with a heterogeneous computing infrastructure and dynamic wireless network conditions, demand that distributed software be able to adapt to its environment.

Adaptations may require recomposition of functional aspects, which realize the imperative behavior of an application, and nonfunctional aspects, such as quality-of-service, fault tolerance and security. Adapting functional aspects is needed for upgrade of existing components or enhancement and extension of the primary function of a system. Such functional adaptation may correct problems or improve a system's ability to cope with decreasing network quality. Equally, nonfunctional aspects may require adaptation, possibly augmenting security and fault tolerance concerns, in response to a changing environment or application domain.

Two general approaches have been used to realize adaptive behavior in software. *Transformational* [1], or *parameter*, adaptation involves the modification of program variables that determine program behavior. As noted by Hiltunen and Schlichting [2], a prominent example of transformational adaptation is the manner in which TCP adjusts its behavior, through the values of variables associated with window management and retransmission timeouts, in response to perceived network congestion [3, 4]. In contrast, *compositional adaptation* [1] results in the exchange of algorithmic or structural parts of the system with ones that improve a program's fit to its current environment [2, 5–10]. Compositional adaptation can insert fault tolerant components, such as forward error correction filters, in response to an unreliable or lossy wireless connection [9, 10], or address nonfunctional concerns [11–13], such as hardening a system's resistance to attack under adverse conditions [14].

A key issue that arises in compositional adaptation is state management. Recomposition of algorithmic or

* 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 CDA-9617310, NCR-9706285, CCR-9912407, EIA-0000433, and EIA-0130724.

structural components at run-time requires the transfer of nontransient state between an old component and its replacement. While the state capture problem has been addressed in other contexts, such as checkpointing [15, 16], process or thread migration [16–18] and mobile agents [19, 20], the methods employed there generally are not directly applicable because they either incur too much overhead or do not support state transfer between different implementations of a component. Rather, recomposition involves state transfer as it relates to *collateral change*, which we define as the set of recompositions that must be applied to an application atomically for continued correct execution.

The main contribution of this paper is to propose a software design approach that facilitates compositional adaptation by explicitly addressing collateral change. This approach is intended to complement related research projects, such as those implementing dynamic component reconfiguration [9, 10, 14, 21–23] or dynamic aspect weaving [11–13, 24–28], by providing mechanisms that support state management during run-time recomposition. We have used this approach to construct a prototype system called Perimorph[1], which supports run-time recomposition of both functional and nonfunctional aspects of the system.

The remainder of this paper is organized as follows. Section 2 provides background on the state management problem and discusses issues relevant to compositional adaptation. Section 3 defines and discusses compositional adaptation with respect to a simple example, the "adaptive queue." Section 4 provides details on the design and implementation of the Perimorph system. Section 5 describes an adaptive digital elevation mapping application, designed using Perimorph, that supports recomposition and application handoff among networked devices with varying capabilities. Section 6 presents our conclusions and discusses future directions.

2 Approaches to State Capture

The problem of state capture has been explored in a variety of domains. Checkpointing, process or thread migration, and mobile agents all employ mechanisms that extract state from a running program, and restore it in some way. We briefly review these approaches and their relevance to our problem.

Storing a snapshot of a running program provides a level of fault tolerance for a program that executes over long periods [15]. Checkpointing allows an image of the running program to be stored in a file. If the program or machine should crash, this image can be used to restart the program as of its last checkpoint. Libckpt [15] provides a library that implements nearly transparent checkpointing. However, process level checkpointing does not allow transfer of state at the component level.

Process migration involves saving the process state and restoring it on another machine. Mobile agents [19, 29, 30] use state preservation and restoration to move from machine to machine. Several research groups [16, 17] have implemented mechanisms for migrating Java threads. Although threads are not complete processes, the granularity of state preservation exceeds that of a component. Virtualization [18] refers to the abstraction of physical resources, such as monitors, keyboards and mice, as virtual devices. The Computing Communities project [18] uses virtualization to allow process migration without loss of connectivity to physical resources. However, injection of state is typically back into an identical process or component, DACIA [21] is a system designed to provide modular construction and runtime reconfiguration and recomposition. Although DACIA provides mobility and state maintenance through the use of Processing and ROuting Components (PROCs) similar to Java serialization, transfer of state between dissimilar PROCs is not supported. Whereas, during adaptive recomposition, state injection may be into algorithmically or structurally dissimilar components.

The state capture problem has also drawn attention from the adaptive middleware community. DynamicTAO [14] and some agent systems [31] use a state transfer process similar to the memento pattern [32]. The memento pattern allows the state of an object to be captured, extracted, and later injected into the same object without violating encapsulation. Even if state transfer is allowed between different components, however, the exchange is often between data structures that are identical. Support for transfer of state between dissimilar components requires the conversion of the extracted memento into a form acceptable for injection into a new component.

3 Compositional Adaptation

Compositional adaptation [2, 8–11], or the ability to affect and modify a program while it executes, poses a unique problem with respect to state capture. Replacement of algorithmic or structural components at run-time requires that the original component be frozen, its nontransient state injected into its replacement, and the new component exchanged with the old. As a simple illustrative example, let us consider two implementations of a producer-consumer queue, one implementation using a fixed-length array and the other using a dynamically re-

[1] The term perimorph is borrowed from geology. A perimorph is a crystal that contains another crystal of a different type. We use it here as an allusion where crystal facets are considered to be components or factors of compositional structure.

sizeable vector. Both implementations provide the same operations, `put()`, `get()`, and `isFull()`. However, the vector version of `isFull()` operation will always return `FALSE` since the `put()` operation dynamically allocates the necessary structures for appending a new item to the queue.

State maintenance. Let us further consider how we might design a meta-level function, as found in reflective systems [33, 34], whose purpose is to transfer state from one implementation to another at run time. The meta-level cannot simply copy an array onto a vector byte-by-byte, nor can a `put()` operation, designed for one implementation, be used to append an item to a queue using the other implementation. Rather, the system must extract a normalized representation of the array-based queue and inject it into the vector-based queue. Such a state extraction and restoration scheme must somehow "understand" both array and vector implementations of a queue and be able to convert between them.

One approach is to replay operations. Observably, the state of the queue is the result of operations conducted using its interface. If all the operations executed on an array-based queue were recorded and then later replayed to a vector-based queue, injection of state could be accomplished by using a fast forward replay of these operations to the vector-based queue. The operations could be recorded by requiring that all queue operations be directed through a recording mechanism. For certain components, it might be possible to optimize the recorded information (for example, by recognizing that the sequence of queue operations `put`, `put`, `get` is equivalent to executing the last `put`). However, these types of optimizations are intrinsic to the type of component for which operational recording is desired. In the general case, recording and replaying all operations on an interface would be prohibitively expensive.

Another possible solution for transferring state in our example stems from the observation that both the array and vector-based queues are of the same abstract type. Typically, if two objects, such as integers, are of the same type, they can be assigned to one another. We could provide such assignment operators in our queue implementations and use them to transfer state. In its crudest form, however, each component would need to implement an assignment operation for all other components of the same type, which could be rather unwieldy for many components and implementations.

A better solution is is to enable state extraction to export a *normalized representation* of the component's state, understood by all other components of the same abstract type. The normalized state can then be assigned to an algorithmically or structurally dissimilar component. A component only needs to know how to code a normalized memento of its own state and how to decode a normalized state memento captured from another component. By using the memento pattern [32] in conjunction with normalization, an array-based queue can be assigned to a vector-based queue. This approach is used in Perimorph.

Reference update. Regardless of the method used to capture state, the issue of reference update during component exchange must be addressed. As shown in Figure 1, when one component is exchanged for another it is necessary to update the references that point to the old component such that they refer to the new one. Doing so is necessary to ensure that the program continues to execute correctly.

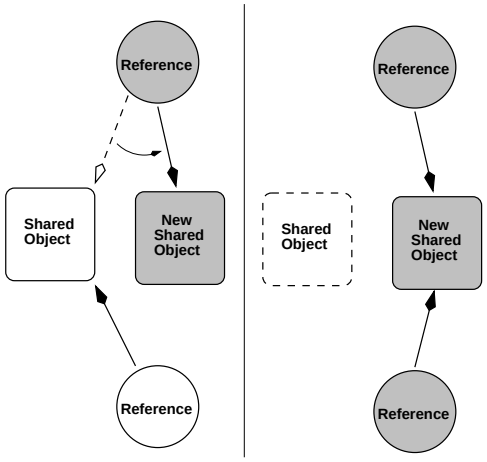

Figure 1. Object reference update problem. Left, component replacement executed using a particular reference. Right, desired result of a component replacement.

An approach used in many recomposable systems [11, 14, 18, 21] is to introduce a level of indirection such that the constituent objects can be decoupled. This method allows an application to access an object in a consistent way but the access method is independent of the object's implementation. As such, the implementation of an object can be modified without changing how an application refers to it. For instance, wrappers can decouple an application by allowing indirect reference to a component through pointers to its wrapper [11]. The wrapped component can be transparently replaced without updating references that refer to the wrapper. A dynamically recomposable system must enable the decoupling of an application such that components can be recomposed. Moreover, decoupling can eliminate the necessity of updating an application's references to shared objects in the face of component exchange. Perimorph decouples components through the use of proxies, which enable the application to invoke component operations while allowing transparent replacement of the proxied component.

Collateral change. *Collateral change* refers to modifications applied to a system that transpire at the same time and in response to some other system modification. For example, to convert a queue from an array implementation to a vector implementation requires both the replacement of the array with a vector and the modification of the `put()`, `get()` and `isFull()` operations such that they use a vector instead of an array. Moreover, these modifications must all happen while the queue is "frozen" such that the entire recomposition transpires atomically. Otherwise, operations on the queue would be inconsistent. Collateral change also affects the recomposition of nonfunctional concerns, such as concurrency control. The dynamic addition of a mutex to control concurrent use of a queue by producer and consumer threads would require changes to both the `put()` and `get()` operations such that the mutex would be locked when these operations are invoked and released when each thread is finished. Continued operation of the queue depends on these changes happening collaterally.

4 Perimorph

Perimorph is designed to enable an application designer to quantify and codify collateral changes, as related to compositional adaptation. The key concept used in perimorph is composition of *factors*, which represent modifications that can be applied to component operations. Each set of collateral changes can be codified as a *factor set* that contains factors and nontransient data structures shared between the factors. Perimorph also uses repositories, called *stores*, to provide a well known structure and interface for manipulating and recomposing an application. Moreover, Perimorph provides a meta-level view of the base-level application composition while supporting runtime recomposition. Perimorph is implemented in Java.

Factor sets are related to aspect-oriented programming (AOP) [12, 13, 35–37]. Specifically, an aspect may comprise one or more factor sets. However, rather than addressing cross-cutting concerns and disentangled code, as in AOP, factors together with factor sets provide constructs for capturing collateral change and nontransient state. In other words, sets of collateral changes represent the *factoring* of an application such that recomposition is defined in terms of viable sets of modifications. All factors that are members of the same factor set must be applied atomically. Applying nonviable changes to an application usually results in program failure.

Component construction. Figure 2 shows the relationship of a component, factors, and factor sets in Perimorph. Components are identified by a name (a Java String) given to them when they are created. Interface sets are added to components and contain operation signatures defining the interfaces implemented by a component. As an example, we have used Perimorph to implement an "adaptive queue" as a solution to the issues described in the previous section. The adaptive queue has an interface set consisting of the signatures `put(Item)`, `get()` and `isFull()`. Operations comprise an interface signature and zero or more factors. Factors are attached to an interface signature forming the body of an operation.

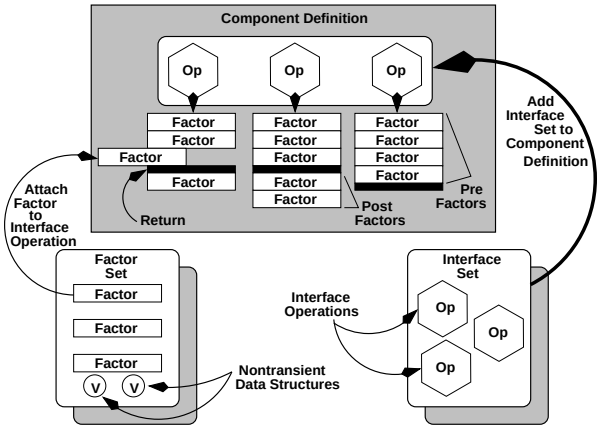

Figure 2. Relationship of factors, factor sets and a component definition.

Factors are attached to an interface operation as either *pre-* or *post-* factors. Pre-factors are executed before a *return* and post-factors are executed following a *return*. Pre-factors implement the operation body, while post-factors provide post operation processing. Any pre-factor can trigger a return, preventing the execution of subsequent pre-factors and jumping to post-factor processing. Equally, any post-factor can trigger completion of an operation. Post-factors allow the completion of functions begun by pre-factors. For instance, a pre-factor may lock a mutex to control concurrent access to a component. A post-factor could unlock the mutex, ensuring that other threads are allowed continued access.

Data structures defined within factor sets represent nontransient state. When factors from one factor set are replaced by another, nontransient state needs to be extracted from the old set and injected into the new. The transfer of state is completed using `getState()` and `setState()` factor set methods that extract and inject a normalized state memento. Factor set data structures are shared by all factors belonging to the same factor set.

References and invocations. Proxies represent components in the base-level, allowing the application to invoke component operations while decoupling components and providing the base-level with a consistent view of the program's structure. Proxies are used in place of base-

level component references. For example, in the adaptive queue, both the producer and consumer hold a proxy, instead of a reference, for the queue component. When the control thread recomposes the array-queue as a vector-queue, it is unnecessary to update these proxies, since the next invocation of a `put()`, `get()` or `isFull()` operation, will retrieve the vector-queue, instead of the array-queue, from the `ComponentStore`.

Execution of a component operation is depicted in Figure 3. An application invokes a component operation by calling a proxy's `invoke()` method and specifying an operation signature, such as `put(Item)`, as a parameter. Using the component's name, the required component is located in the `ComponentStore`, and the specified operation is retrieved from the component's interface set. Factors, previously attached to the operation signature, are invoked one after the other until operation execution is complete. Finally, control is returned to the base-level caller.

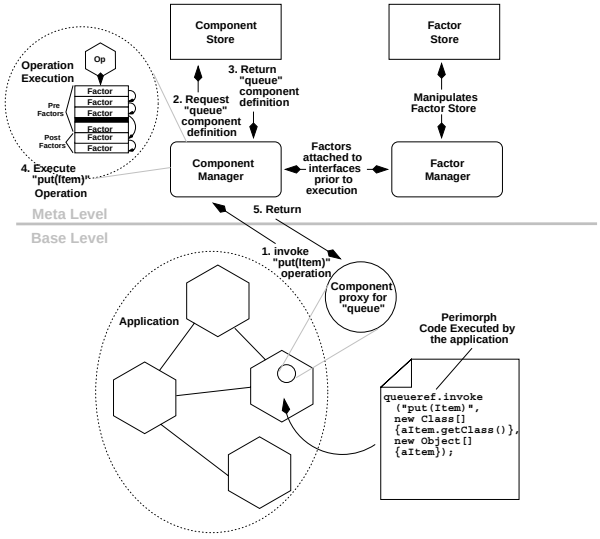

Figure 3. Executing a component operation.

Recomposition. Recomposing a component involves adding, deleting or replacing factors. Both functional and nonfunctional factors can be added, removed or replaced, allowing the entire function of a component to be changed or augmented. For instance, an array-based queue can be replaced with a vector-based queue. Nonfunctional concerns, such as concurrency controls or security, can be added and removed as needed. Reference update is automatic as recomposition operates on the component definition, leaving all component proxies alone. Separating the definition of a component from the references to it obviates the need to update object references scattered throughout the code, simplifying recomposition significantly.

Figure 4 diagrams the Perimorph adaptive queue. Functional concerns are defined by the array and vector-based queue factor sets, shown at the bottom of the figure. Recomposing a queue using a vector, requires the exchange of factors from the array-based factor set with those of the vector-based factor set. Moreover, the nontransient state of the array-based factor set must be transferred to the vector-based factor set. Two nonfunctional concerns are also implemented. Tracing, as defined by the trace factor set, prints informational messages about calls to the queue interface. Thread concurrency controls, defined by the mutex factor set, prevent the producer and consumer threads from operating on the queue simultaneously.

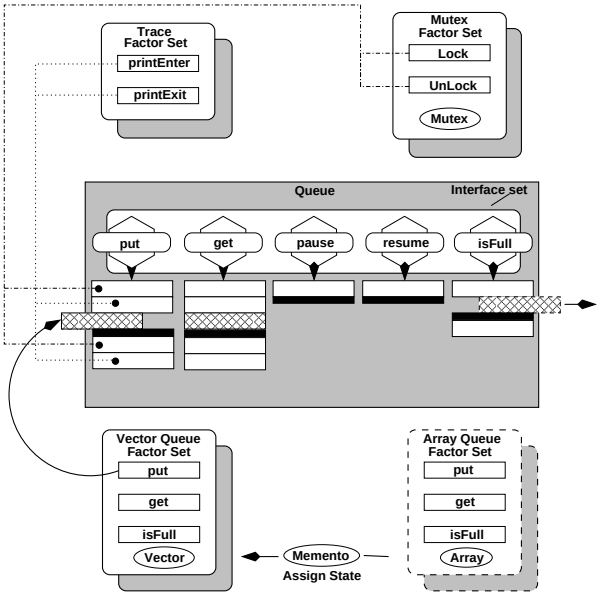

Figure 4. Composition of the adaptive queue showing several factor sets.

Activation and deactivation. Factor sets can be activated or deactivated as they are put into or removed from use. Activation and deactivation automates the process of initialization and shutdown of factor sets, such as those defining graphical interfaces or using threads. Reference counts are kept for all factor sets such that the system can determine when factors are attached to component interfaces. When the reference count drops to zero, the `FactorManager` calls the factor set's `deactivate()` method. When the reference count first rises above zero, the `activate()` method is called. A designer needs only to implement these methods for factor sets that require activation or deactivation; for other factor sets they can simply be left as empty methods.

5 Example: Mapping Application

In addition to the adaptive queue application, we used Perimorph to implement a digital elevation model (DEM) [38] mapping program. The DEM format is a common data format used by the United States Geological Survey (USGS) and other organizations for recording geographical elevation information. We developed our mapping application using Perimorph such that a 2D viewer can be recomposed into a 3D viewer at run time. Such recompositions are useful during handoff between dissimilar devices. For instance, a palmtop, due to limited memory, processing power and display capability, might use only the 2D viewer. However, upon arriving at the office, a user may handoff the application to a workstation that can easily present a three-dimensional map. With Perimorph, the viewer can dynamically be transformed into a 3D viewer without loss of application state.

Figure 5 shows a two-dimensional representation of Mount St. Helens after eruption in 1980. This representation uses different colors to indicate changes in elevation. Typically, the lighter the color the greater the elevation. Initially, the mapping application comprises factors implementing a 2D viewer. Figure 6 depicts the factors recomposed during conversion to a 3D display. Upgrading the map requires modification of the functional concerns of both the map plotter and map window components. The map plotter paints the map on the map window. Depending on whether the map plotter and map window are composed using the two or three-dimensional factor set determines how map data will be displayed. Nontransient state, comprising DEM map data, is assigned from the two-dimensional to the three-dimensional factor set during factor exchange.

Figure 5. 2D map prior to recomposition.

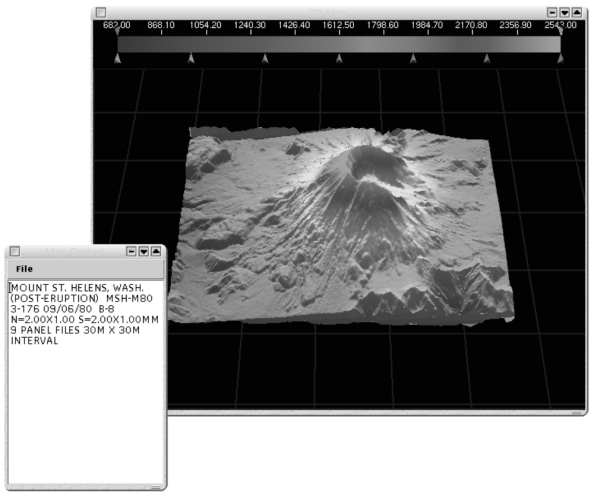

Figure 7. 3D map following recomposition.

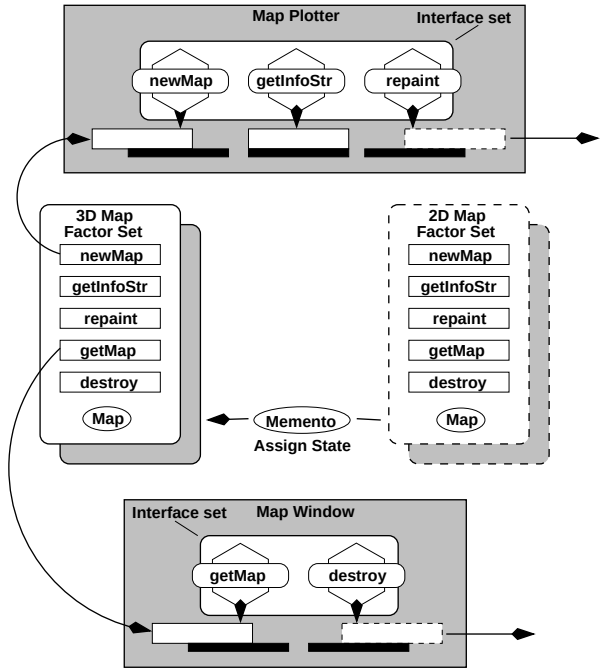

Figure 6. Recomposition of the DEM mapping application. Recomposition of both the map plotter and map window components is required. Operations on these components are called by the map control which does not require any change.

Figure 7 shows a three-dimension map following dynamic, run-time recomposition. Proper initialization and construction of the GUI components require the coding of `activate()` and `deactivate()` factor set methods, which were left as empty methods for the adaptive queue. Besides dynamic reconfiguration, constructing applica-

tions with Perimorph enables other state-related functionality. For example, both the adaptive queue and the elevation mapping application can be captured at any point in their execution and stored on disk or sent over the network to another machine. A state memento for an entire application can be constructed by saving the contents of the Perimorph stores and the nontransient state of all factor sets. This memento can be serialized and stored on disk or sent over a network. When the application is restarted, Perimorph requests a reload, deserializing these stores. References to components are reestablished as the application requests references from the `ComponentManager`. Thus, Perimorph applications can easily support checkpointing and distributed handoff in addition to run-time recomposition. Moreover, composition can be adjusted following handoff, allowing adaptation to new environmental conditions, such as reduced memory or a smaller physical display.

6 Summary and Future Work

In this paper, we argued the need to address collateral change in compositional adaptation. We described Perimorph, a system that supports dynamic, run-time recomposition of both functional and nonfunctional concerns. The system allows transparent reconfiguration of components. Factor sets provide a construct for capturing how collateral change affects system recomposition. Nontransient state is defined at the factor set scope and can be assigned between factor sets of equivalent abstract type using state normalization in conjunction with the memento pattern. In particular, we built both a simple, illustrative adaptive queue and a digital elevation mapping application, demonstrating the usefulness of Perimorph constructs.

Further study is needed on how best to factor adaptive systems with respect to collateral change and automate state transfer. We intend to extend Perimorph in several ways. First, language constructs that ease a programmer's burden when designing and implementing adaptive applications are desirable. Constructs that provide a high level of abstraction for systems, like Perimorph, can further improve a software designer's ability to understand and construct applications supporting adaptation, and help bring adaptive software into the mainstream. Second, examining how platforms, such as the Java virtual machine, can be modified to better enable dynamic recomposition and improve meta-level representation of running applications might lead to improved function and performance. Third, systems that support collateral change and dynamic composition require formal methods and software engineering principles to verify correctness and guide design and implementation of dynamically recomposable systems.

References

[1] B. Tekinerdogan and M. Aksit, "Adaptability in object-oriented software development workshop report," in *Proceedings of the 10th Annual European Conference on Object-Oriented Programming (ECOOP)*, (Linz, Austria), July 1996.

[2] M. A. Hiltunen and R. D. Schlichting, "Adaptive distributed and fault-tolerant systems," *International Journal of Computer Systems Science and Engineering*, vol. 11, pp. 125–133, September 1996.

[3] Information Sciences Institute University of Southern California, "RFC 793: Transmission control protocol." `http://www.faqs.org/rfcs/rfc793.html`, September 1981.

[4] J. F. Kurose and K. W.Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*. Boston, Massachusetts, USA: Addison Wesley, 2001.

[5] N. Venkatasubramanian, "Safe 'composability' of middleware services," *Communications of the ACM*, vol. 45, pp. 49–52, June 2002.

[6] M. Aksit, L. Bergmans, and S. Vural, "An object-oriented language-database integration model: The composition-filters approach," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, (Utrecht, Netherlands), pp. 372–395, June 1992.

[7] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting, "Constructing adaptive software in distributed systems," in *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, (Mesa, Arizona, USA), pp. 635–643, April 2001.

[8] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr, "Building adaptive systems using ensemble," Tech. Rep. TR97-1638, Department of Computer Science, Cornell University, Ithaca, New York, USA, July 1997.

[9] P. K. McKinley and U. I. Padmanabhan, "Design of composable proxy filters for heterogeneous mobile computing," in *Proceedings of the Second International Workshop on Wireless Networks and Mobile Computing*, 2001.

[10] E. P. Kasten, P. K. McKinley, S. M. Sadjadi, and R. Stirewalt, "Separating introspection and intercession to support metamorphic distributed systems," in *Proceedings of the 22nd International Conference on Distributed Computing Systems ICDCS'02*, (Vienna, Austria), July 2002. to appear.

[11] L. Bergmans and M. Aksit, "Composing crosscutting concerns using composition filters," *Communications of the ACM*, vol. 44, pp. 51–57, October 2001.

[12] E. Avdičaušević, M. Mernik, M. Lenič, and V. Žumer, "Experimental aspect-oriented language - AspectCOOL," in *Proceedings of 17th ACM Symposium on Applied Computing, SAC 2002*, (Madrid, Spain), pp. 943–947, 2002.

[13] E. Truyen, W. Joosen, and P. Verbaeten, "Run-time support for aspects in distributed system infrastructure," in *Proceedings of the First AOSD Workshop on Aspects,*

Components, and Patterns for Infrastructure Software (ACP4IS'2002), (Enschede, Netherlands), 2002.

[14] F. Kon, M. Romàn, P. Liu, J. Mao, T. Yamane, L. C. M. aes, and R. Campbell, "Moinitoring, security, and dynamic configuration with the dynamicTAO reflective ORB," in *Proceedings IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, (New York, New York, USA), pp. 3–7, April 2000.

[15] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," in *Proceedings, Usenix Winter 1995 Technical Conference*, (New Orleans, Lousiana, USA), pp. 213–223, January 1995.

[16] S. Bouchenak and D. Hagimont, "Pickling threads state in the java system," in *33rd International Conference on Technology of Object-Oriented Languages (TOOLS)*, (St. Malo, USA, France), June 2000.

[17] M. Ma, C. Wang, and F. Lau, "Delta execution: A preemptive java thread migration mechanism," *Cluster Computing*, vol. 3, no. 2, pp. 83–94, 2000.

[18] S. Zhang, M. Khambatti, and P. Dasgupta, "Process migration through virtualization in a computing community," in *13th IASTED Conference on Parallel and Distributed Computing Systems (PDCS2001)*, (Dallas, Texas, USA), August 2001.

[19] S. Fünfrocken, "Transparent migration of java-based mobile agents," in *Proceedings of Second International Workshop on Mobile Agents 98*, (Stuttgart, Germany), pp. 26–37, September 1998.

[20] T. Watanabe, A. Noriki, and K. Shinbori, "Towards a substrate for reliable mobile agent systems," in *Workshop on Reflective Middleware*, (Palisades, New York, USA), April 2000.

[21] R. Litiu and A. Prakash, "Dacia: A mobile component framework for building adaptive distributed applications," in *Principles of Distributed Computing (PODC) 2000 Middleware Symposium*, (Portland, Oregon, USA), July 2000. also appeared as Technical Report CSE-TR-416-99, Department of EECS, University of Michigan, Dec 1999.

[22] D. Alexander, M. Shaw, S. Nettles, and J. Smith, "Active bridging," in *Proceedings ACM SIGCOMM 1997*, (Cannes, France), September 1997.

[23] P. Costanza, "Dynamic object replacement and implementation-only classes," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2001): 6th International Workshop on Component-Oriented Programming (WCOP 2001)*, (Budapest, Hungary), June 2001.

[24] F. Akkawi, A. Bader, and T. Elrad, "Dynamic weaving for building reconfigurable software systems," in *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, (Tampa Bay, Florida, USA), October 2001.

[25] E. Bruneton and M. Riveill, "Reflective implementation of non-functional properties with the JavaPod component platform," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2000): Workshop On Reflection and Metalevel Architectures*, (Sophia Antipolis and Cannes, France), June 2000.

[26] A. Popovici, T. Gross, and G. Alonso, "Dynamic weaving for aspect-oriented programming," in *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, (Enschede, The Netherlands), pp. 141–147, ACM Press, 2002.

[27] B. Redmond and V. Cahill, "Supporting unanticipated dynamic adaptation of application beh aviour," in *Proceedings of the 16th European Conference on Object-Oriented Programming*, Malaga, Spain: Springer-Verlag, June 2002. volume 2374 of Lecture Notes in Computer Science.

[28] A. Oliva and L. E. Buzato, "The design and implementation of Guaraná," in *Proceedings 5th USENIX Conference on Object-Oriented Technol ogies and Systems*, (San Diego, California, USA), May 1999.

[29] A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R. Singh, "Mobile agent programming in Ajanta," in *Proceedings of the 19th International Conference on Distributed Computing Systems*, (Austin, Texas), pp. 314–322, 1999.

[30] G. Karjoth, D. B. Lange, and M. Oshima, "A security model for aglets," *IEEE Internet Computing*, pp. 68–77, July–August 1997.

[31] E. Kendall, P. M. Krishna, C. Pathak, and C. Suresh, "Patterns of intelligent and mobile agents," in *Proceedings of teh 2nd International Confernece on Autonomous Agents (AGENTS-98)*, (New York, New York, USA), pp. 92–99, May 1998.

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

[33] P. Maes, "Concepts and experiments in computational reflection," in *Proceedings of the ACM Conerfence on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 147–155, December 1987.

[34] J. des Riviéres and B. C. Smith, "The implementation of procedurally reflective languages," in *Conference Record of the 1984 ACM Symposium on LISP and functional programming*, (Austin, Texas, USA), pp. 331–347, 1984.

[35] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting started with AspectJ," *Communications of the ACM*, vol. 44, pp. 59–65, October 2001.

[36] "ActiveJ." `http://aspectj.org/servlets/AJSite`.

[37] F. J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier, "Aspectix: A quality-aware, object-based middleware architecture," in *Proceedings of the 3rd IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, (Kracow, Polland), September 2001.

[38] "Rocky mountain mapping center: Elevation program." `http://rmmcweb.cr.usgs.gov/elevation/`.