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 of the system. 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* [18], or *parameter*, adaptation involves the modification of program variables that determine program behavior. As noted by Hiltunen and Schlichting [9], 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 [10]. In contrast, *compositional adaptation* [18] results in the exchange of algorithmic or structural parts of the system with ones that improve a program's fit to its current environment [3, 6, 9, 11, 16, 20]. Compositional adaptation can insert fault tolerant components, such as forward error correction filters, in response to an unreliable or lossy wireless connection [11, 16], or address nonfunctional concerns [5, 19], such as hardening a system's resistance to attack under adverse conditions [13].

A key issue that arises in compositional adaptation is state management. Recomposition of algorithmic or 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, process or thread migration and mobile agents, the methods employed there generally are not directly applicable because they either in-

* 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.

cur 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 [4, 11, 13, 14, 16] or dynamic aspect weaving [2, 5, 17, 19], 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 defines and discusses compositional adaptation with respect to a simple example, the "adaptive queue." Section 3 provides details on the design and implementation of the Perimorph system. Section 4 describes an adaptive digital elevation mapping application, designed using Perimorph, that supports recomposition and application handoff among networked devices with varying capabilities. Section 5 presents our conclusions and discusses future directions.

2. Compositional Adaptation

Compositional adaptation [5, 9, 11, 16, 20], 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 resizeable vector. Both implementations provide the same operations, `put()`, `get()`, and `isFull()`. However, the vector `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 [7, 15], 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 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.

A better solution 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 (i.e. implements a queue). 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 [8] 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. 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. An approach used in many recomposable systems [5, 13, 14, 21] introduces a level of indirection such that the objects that comprise an application 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. 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

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.

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.

3. Perimorph

Perimorph is implemented using Java and enables an application designer to quantify and codify collateral changes, as related to compositional adaptation, in terms of factor sets. Perimorph 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 run-time recomposition.

Component construction. Figure 1 shows the relationship of a component, *factor sets* and *factors*. Factors 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. 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. For instance, 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.

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.

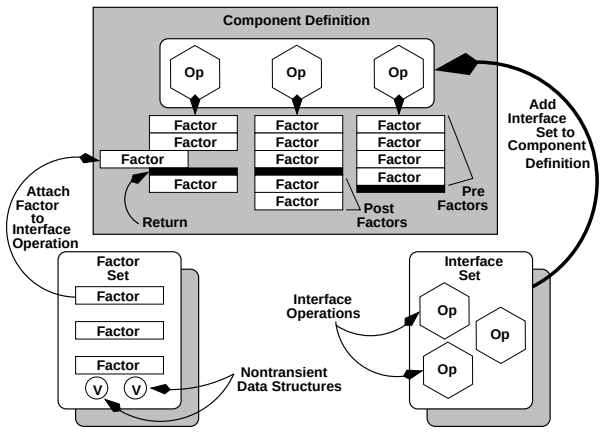

Figure 1. Relationship of factors, factor sets and a component definition.

Aspect Oriented Programming (AOP) aspects [12, 19] and factor sets are related in that an aspect may comprise one or more factor sets. However, Rather than focus on designing a system in terms of 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.

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 2. 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.

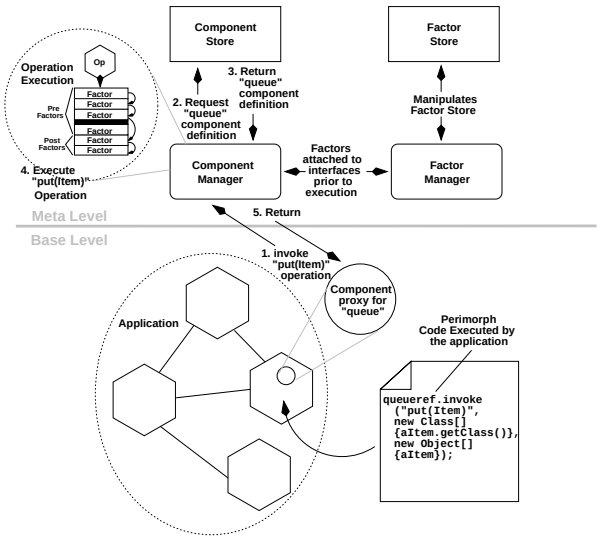

Figure 2. Executing a component operation.

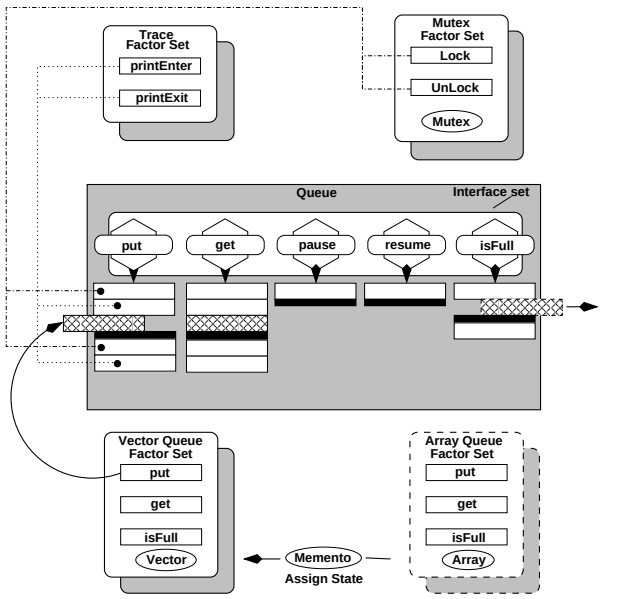

Figure 3. Composition of the adaptive queue showing several factor sets.

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 3 diagrams the Perimorph adaptive queue, introduced in Section 2. 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.

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.

4. Example: Mapping Application

In addition to the example adaptive queue application, we used Perimorph to implement a digital elevation model (DEM) [1] 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 4 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 5 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 the 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 4. 2D map prior to recomposition.

Figure 6 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 applications 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

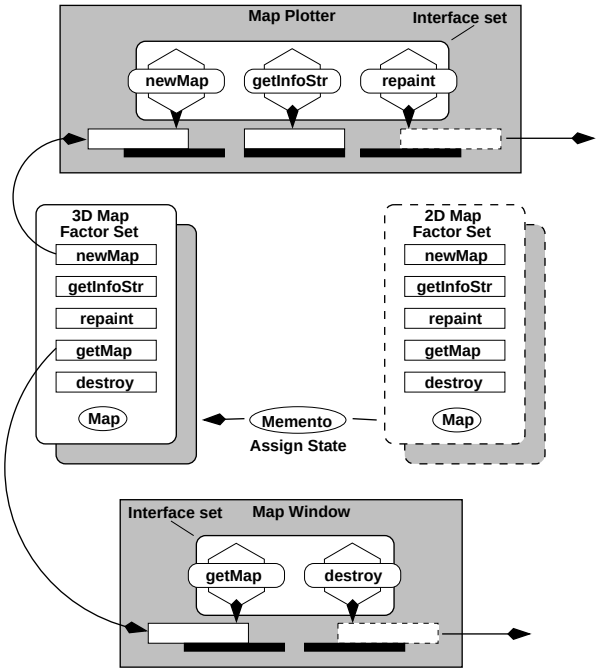

Figure 5. Recomposition of the DEM mapping application. Recomposition of both the map plotter and map window components is required. Operations on these components are called by the map control which does not require any change.

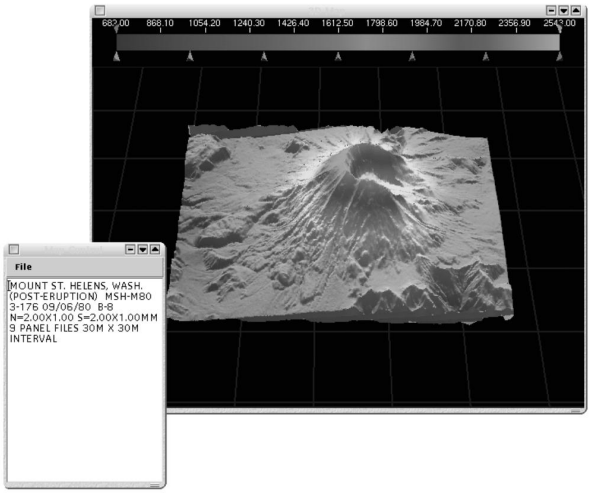

Figure 6. 3D map following recomposition.

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.

5. Conclusions

In this study, we designed a system, Perimorph, 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.

Further study is needed on how best to factor adaptive systems with respect to collateral change and automate state transfer. Constructs that provide a high level of abstraction for systems, like Perimorph, can further improve a software designer's ability to understand and build applications supporting adaptation. Moreover, 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] Rocky mountain mapping center: Elevation program. `http://rmmcweb.cr.usgs.gov/elevation/`.

[2] 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.

[3] 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)*, pages 372–395, Utrecht, Netherlands, June 1992.

[4] D. Alexander, M. Shaw, S. Nettles, and J. Smith. Active bridging. In *Proceedings ACM SIGCOMM 1997*, Cannes, France, September 1997.

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

[6] 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)*, pages 635–643, Mesa, Arizona, USA, April 2001.

[7] 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*, pages 331–347, Austin, Texas, USA, 1984.

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

[9] M. A. Hiltunen and R. D. Schlichting. Adaptive distributed and fault-tolerant systems. *International Journal of Computer Systems Science and Engineering*, 11(5):125–133, September 1996.

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

[11] 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.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[13] F. Kon, M. Romàn, P. Liu, J. Mao, T. Yamane, L. C. Magalhaes, 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)*, pages 3–7, New York, New York, USA, April 2000.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.