On the Role of Architectural Styles in Improving the Adaptation Support of Middleware Platforms

Naeem Esfahani and Sam Malek

Department of Computer Science
George Mason University
{nesfaha2, smalek}@gmu.edu

Abstract. Modern middleware platforms provide the applications deployed on top of them with facilities for their adaptation. However, the level of adaptation support provided by the state-of-the-art middleware solutions is often limited to dynamically loading and off-loading of software components. Therefore, it is left to the application developers to handle the details of change such that the system's consistency is not jeopardized. In this paper, we present an approach that addresses the current shortcomings by utilizing the information encoded in a software system's architectural style. This information drives the development of adaptation patterns, which could be employed to enhance the adaptation support in middleware platforms. The patterns specify both the exact sequence of changes and the time at which those changes need to occur.

Keywords: Middleware, Adaptation Patterns, Architectural Styles

1 Introduction

The unrelenting pattern of growth in size and complexity of software systems that we have witnessed over the past few decades is likely to continue well into the foreseeable future. As software engineers have developed new techniques to address the complexity associated with the construction of modern-day software systems, an equally pressing need has risen for mechanisms that automate and simplify the management and modification of software systems after they are deployed, i.e., during run-time. This has called for the development of self-* (self-configuring, self-healing, self-optimizing, etc.) systems. However, the construction of such systems has been shown to be significantly more challenging than traditional, relatively more static and predictable, software systems.

Previous studies have shown that a promising approach to resolve the challenges of constructing complex software systems is to employ the principles of *software architecture* [7], [8]. Software architectures provide abstractions for representing the structure, behavior, and key properties of a software system. They are described in terms of *software component*s (computational elements), *connector*s (interaction elements), and their *configuration*s. A given software *architectural style* (e.g., *publish-subscribe*, *peer-to-peer*, *pipe-and-filter*, *client-server*) further refines a

vocabulary of component and connector types and a set of constraints on how instances of those types may be combined in a system [1].

Software architecture has also been shown to provide an appropriate level of abstraction and generality to deal with the complexity of dynamic adaptation of software systems [3]. This observation has led to research on *architecture-based adaptation*, which is the process of reasoning about and adapting a system's software at the architectural level [3], [6].

Architecture-based adaptation is often realized via the run-time facilities provided by an implementation platform, i.e., *middleware*. Unfortunately, the level of adaptation support provided by most state-of-the-art middleware solutions is limited to dynamically loading and offloading of software components. They do not consider the state or dependency among the system's software components. This is driven by the fact that, in the general case, component dependency relationships are application specific, and cannot be predicted a priori by the middleware designers.

The lack of advanced adaptation management and coordination facilities in the existing platforms forces the application developers to implement them on their own. Unfortunately, the status quo places significant burden on the application developers. The developers have to spend a significant amount of time understanding the underlying details of a middleware platform, before they can develop the required adaptation facilities. As a result, the theoretical advances [4], [10] for consistent and sound adaptation of a software system remain untapped, and the application developers rely on the rudimentary adaptation capabilities that the existing middlewares provide by default.

In this paper, we present an approach that attempts to alleviate these shortcomings. The approach relies on the information encoded in a software system's architectural style. More specifically, an underlying insight guiding our research is that a software system's architectural style reveals a lot about the dependency relationships among the system's software components. This information is utilized to identify *adaptation patterns*, which determine the recurring sequence of changes that need to occur for adapting a software system built according to a given style. An adaptation pattern ensures that the system is not left in an inconsistent state and the application's functionality is not jeopardized. We have realized the adaptation patterns on top of an existing middleware platform, called Prism-MW [5].

The paper is organized as follows. Section 2 provides the required background, while Section 3 motivates the work by summarizing the problems with existing approaches. Section 4 describes our overall approach. Section 5 describes the extraction of an adaptation pattern from a given architectural style. Section 6 provides an overview of the application of patterns in improving the capabilities of an existing middleware. Finally, the paper concludes.

2 Background and Related Work

For exposition purposes, we are going to use a simple application intended for routing incoming cargo to a set of warehouses. This application was first presented in the

seminal work on architecture-based adaptation [6]. We have reproduced its architecture in Fig. 1a. The architectural style of this application is C2 [9]. A software system built in the C2 style consists of layers, where *request* events travel upward, while *notification* events travel downward. *E*vents that are received are evaluated to determine if the component needs to process them.

In Fig. 2, the *Ports*, *Vehicles,* and *Warehouses* components are abstract data types (ADTs) that keep track of the state of shipping ports, transportation vehicles, and goods warehouses, respectively. The *Telemetry* component determines when cargo arrives at a port, and tracks the cargo from the time it is routed until it is delivered to the warehouse. The *Port Artist*, *Vehicle Artist*, and *Warehouse Artist* components are responsible for graphically depicting the state of their respective ADTs to the end-user. The *Router* component provides the end-user's last selected port, vehicle, and warehouse. The *Graphics* component renders the drawing requests sent from the artists using the Java AWT graphics package.

2.1 Challenges of Architecture-Based Adaptation

Typically, middleware support for architecture-based adaptation is realized in the form of adding, removing, and replacing software components. However, such changes could jeopardize the functionality of a software system, as they could leave the system in an inconsistent state. For instance, consider a scenario where we would like to replace the *Warehouses* component. Such capability may be realized by removing the old *Warehouses* and adding a new instance of it [6]. This solution, however, ignores the other components, such as *Telemetry*, which depend on the *Warehouses* for delivering their services.

Let us assume the end-user makes a "Route Warehouse" request using the *Graphics* component. Fig. 1b shows the interactions (events/messages) that would result in response to this request. If such a request is made while *Warehouses* component is being updated, and thus temporarily unavailable, it is processed by the *Router* and *Warehouse Artist*, but not the new *Warehouses* component. The effect of

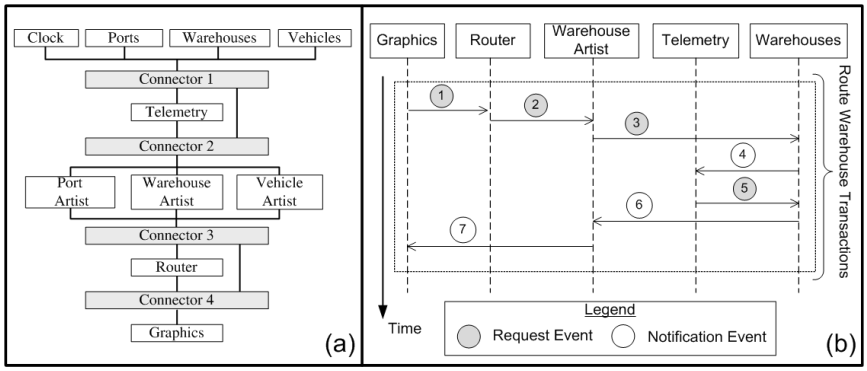

Fig. 1. Cargo-routing application: a) its C2 architecture; and b) a transaction for the cargo-routing application and the corresponding component dependencies.

this may manifest itself in the form of functional failure: the new *Warehouses* component may not receive event 3, resulting in the system to never respond to the user (i.e., events 4-7 do not occur).

At first blush it may seem that buffering events intended for *Warehouses* component would solve the problem. However, buffering by itself cannot address consistency issues that may arise. Consider the situation in which *Warehouses* component is replaced after it has replied with event 4, but before receiving event 5. In this case, it is possible for the old component to process request 3, and the new component to process request 5, assuming it is buffered for later processing. However, this may violate the component's interaction protocol (i.e., event 5 can be processed only after event 3 has, which would not be the case with the newly installed component). Since the new component may not have the correct state, the system may become inconsistent.

2.2 Change Management Model

A generally applicable solution to this problem was proposed by Kramer and Magee's seminal model of dynamic change management [3], which provides a separation of structural concerns from application concerns. Their work also identifies two possible states for a software component during the adaptation process. Each state defines how a component behaves during the corresponding phase of adaptation:

- *Active*: A component can start, receive, and process transactions.
- *Passive*: A component in this state will continue to receive and process transactions, but will not initiate any new transactions.

Quiescence is defined as the required property to adapt a component [4]. Quiescence implies that a component (1) is not currently involved in a transaction, (2) will not start any new transactions, and (3) no transactions have been or will be initiated by other components that require service from this node.

Based on this change management model, Gomaa and Hussein [2] suggest the development of reconfiguration patterns for software product lines. However, their approach does not consider transitive dependencies and their implications.

3 Research Problem

It is typically left to the application developers to implement the required change management and coordination facilities mentioned above. These facilities would provide the logic that ensures the system's consistency during adaptation (i.e., the order in which the various components are activated and passivated).

The implementation of these facilities is a major burden on the application developers for the following reasons: (1) *Identifying the component dependencies:* Determining the changes that need to occur in the system to place a software component in a particular adaptation state (i.e., active, passive) depends on the component dependencies. However, identifying transitive dependencies requires

understanding the details of the application logic, which defeats the purpose of treating components as black boxes and adapting a system at the architectural level. (2) *High complexity:* Realizing such facilities requires the development of complex state management and coordination logic. (3) *Lack of reuse:* Since each component has its own unique set of dependencies on other components, one component's state management logic cannot be easily reused by other software components that may need to be updated at run-time. (4) *High coupling:* Since the state management logic depends on the component dependency relationships, the resulting software is very fragile. That is as soon as the software evolves (e.g., components change the way they interact and use one another), the state management logic needs to be modified.

Traditionally, one method of reducing complexity and increasing the developer's productivity is to employ middlewares. The middleware engineers develop the frequently needed intra-component facilities (e.g., data marshalling, remote method invocation, service discovery), and provide them as reusable modules to any applications developed on top of the middleware. Unfortunately, employing the same approach in the context of adaptation is not feasible, since it needs inter-component analysis and the middleware designers cannot predict a priori which software components will be deployed on top of a middleware, how they will be configured, and what will be their dependencies. Therefore, modern middleware platforms do not provide change management facilities beyond simple dynamic addition and removal of components. This is precisely the research problem that we have aimed to solve in this paper through the use of knowledge embedded in architectural styles and the capabilities of a unique style aware middleware.

4 Approach

In light of the challenges mentioned above, currently three methods of adapting a software system are employed: (1) Query the component itself to provide information about its dependency relationships. This relates to the 1st problem in Section 3, i.e., violates the black-box treatment of components. Moreover, it hinders reusability of components developed in this manner. (2) Yank the old component and replace it with a new one. As exemplified using the Cargo Routing application in Section 2.1, this approach could leave the sytem in an inconsistent state. (3) Bring down (Passivate) the entire system before adapting it, and restart it afterwards. This approach clearly results in severe disruption in system's execution.

We propose a new approach that builds on the existing models of dynamic change management (recall Section 2). The key underlying insight guiding our research is that a software system's architectural style could reveal the dependency relationships among the components of a given system, even if the components are indirectly connected to one another. The dependency relationships are critical when adapting a software system, as they determine the impact of change on the system [4], [10].

We use the rules and constraints of an architectural style to infer the component dependencies for any software system built according to that style. An example of this can be seen in Fig. 1a. In a C2 software system it is generally true that components in

lower layer depend on components in higher layer. As a concrete example, take *Warehouse Artist* that depends on *Warehouses*.

The component dependencies are in turn used to determine a reusable sequence of changes that need to occur for placing a component in the appropriate adaptation state. Such a recurring sequence of changes, which are coordinated among the system's architectural constructs (e.g., components, connectors) is called an *adaptation pattern*. An adaptation pattern provides a template for making changes to a software system built according to a given style without jeopardizing its consistency.

An adaptation pattern for a given style is guaranteed to be generally applicable for systems built according to that style, since (1) quiescence is guaranteed to be reachable [4], and (2) applications built according to the style exhibit similar dependency relationships among their components.

5 Style-Driven Adaptation Patterns

In this section we describe the process of extracting adaptation patterns from an architectural style. For this purpose we have chosen the C2 style [9]. Note that while the overall approach is generally applicable to any style, the details of the patterns, their accuracy, and level of disruption due to adaptation directly depend on the characteristics of the style. The styles with rich properties and rules inevitably result in more interesting and effective patterns.

During normal operation a C2Component is *Waiting* to receive asynchronous request event from an associated C2Connector. If the event is not intended for the component, it returns to the *Waiting* state. Otherwise, it starts *Processing* the request and additional request and notification events are generated as needed. After the *Processing* has completed and the appropriate events are sent, the component returns to the *Waiting* state.

Adaptation of a software system requires its constituents (e.g., components, connectors) to coordinate the changes that need to occur. It is the responsibility of the adaptation module to track the adaptation state (e.g., active, passive) of the component and neighboring architectural constructs. This recurring coordination

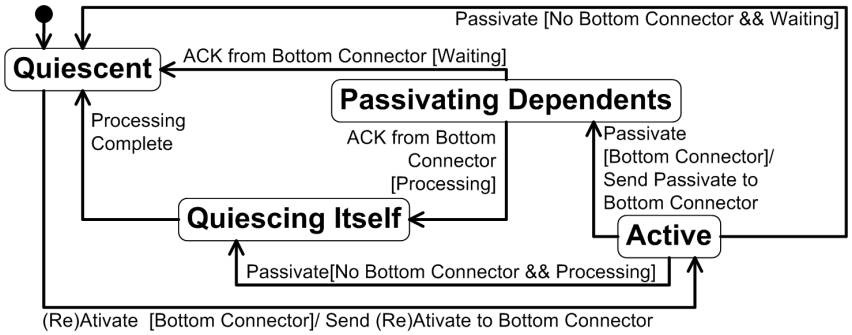

Fig. 2. Partial state chart of C2 adaptation pattern: A C2Component that is being adapted.

constitutes the adaptation pattern for an architectural construct in a given style.

An adaptation pattern may be expressed using statechart models (Fig. 2). Each pattern contains one or more statecharts that define the sequence of steps a component goes through during the adaptation process. In essence, each statechart describes the run-time behavior of a component type (e.g., Client in Client-Server, Publisher in Publish-Subscribe) provisioned by a style during the adaptation process.

The adaptation process requires a component that is to be updated to satisfy the quiescence property. The statechart in Fig. 2 presents the transitions that take an *Active* C2Component that is being adapted to satisfy the quiescence property. When in the *Active* state, the component processes any received events. The first step toward quiescing the component can take one of three paths. Let us first consider the scenario where the component has no bottom connector (i.e., no other component depend on it). In this case, either the component is currently processing or waiting (idle). If the component is waiting, then it simply transitions to *Quiescent*. If the component is processing, it starts *Quiescing Itself*, and waits. When the processing has completed, it transitions to *Quiescent*.

If the component has a bottom connector (i.e., other components depend on it), then the component sends a *Passivate* request to the bottom connector to passivate the dependent components. Once an ACK reply is received from the bottom connector, the component gets *Quiescent* if it is waiting, and starts *Quiescing Itself* if it is processing. In the latter case, the component eventually transitions to the *Quiescent* when the component has completed the work.

The pattern described above, while simple, codify the structural rules and constraints of C2 style into reusable logic that allows for consistent adaptation of any C2 software system. Due to space constraints we have just shown the adaptation patterns for the component being adapted.

6 Style-Aware Adaptation

We have leveraged the style-driven adaptation patterns described above to provide advanced run-time adaptation facilities in Prism-MW [5]. Prism-MW is an *architectural middleware*, which supports architectural abstractions by providing implementation-level modules (e.g., classes) for representing each architectural element, with operations for creating, manipulating, and destroying the element. These abstractions enable direct mapping between a system's software architectural model and its implementation. Prism-MW's core functionality provides the necessary support for developing arbitrarily complex applications, as long as one relies on the provided default facilities (e.g., event scheduling, dispatching, and routing). The developer can extend the core functionality as needed. Prism-MW provides three key capabilities that we have relied on to realize the proposed approach. It provides support for (1) basic architecture-level dynamism, (2) multiple architectural styles, and (3) architectural reflection. By codifying the adaptation patterns in Prism-MW, we have been able to provide significantly more advanced adaptation capabilities than that is currently offered by other middleware platforms.

7 Conclusion

Most state-of-the-art middleware solutions provide rudimentary support for dynamic adaptation of software systems. They lack the ability to handle the implications of replacing a software component. Therefore, the application developers are burdened with the responsibility of managing the adaptation process at the application-level. We have developed a new approach that addresses the current shortcomings. It leverages the rules and characteristics of an architectural style to determine adaptation patterns for software systems built according to that style. These patterns specify the required sequence of actions to put a software component in a state that can be adapted without jeopardizing the software system's consistency, and hence its functionality. In our future work, we plan to develop a catalog of adaptation patterns for commonly employed architectural styles. Such a catalog would be of great interest to both the software engineering and middleware community. We also plan to include the new patterns in the adaptation support of Prism-MW.

Acknowledgments. This work is partially supported by grant CCF-0820060 from the National Science Foundation.

References

1. Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures. Doctoral Thesis #AAI9980887 Univ. of California Irvine (2000).
2. Gomaa, H., Hussein, M.: Software reconfiguration patterns for dynamic evolution of software architectures. Working IEEE/IFIP Conference on Software Architecture. pp. 79-88 (2004).
3. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. Int'l Conf on Software Engineering. pp. 259-268, Minneapolis, MN (2007).
4. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE Trans. Softw. Eng. 16, 11, 1293-1306 (1990).
5. Malek, S., Mikic-Rakic, M., Medvidovic, N.: A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems. IEEE Trans. Softw. Eng. 31, 3, 256-272 (2005).
6. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. Int'l Conf on Software Engineering. pp. 177-186, Kyoto, Japan (1998).
7. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. Softw. Eng. Notes. 17, 4, 40-52 (1992).
8. Shaw, M., Garlan, D.: Software architecture: perspectives on an emerging discipline. Prentice-Hall, Inc. (1996).
9. Taylor, R.N., Medvidovic, N., et al.: A component- and message-based architectural style for GUI software. Int'l Conf on Software Engineering. pp. 295-304, Seattle, Washington (1995).
10. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. IEEE Trans. Softw. Eng. 33, 12, 856-868 (2007).