Utilizing Architectural Styles to Enhance the Adaptation Support of Middleware Platforms

Naeem Esfahani [1]   Sam Malek

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

Abstract

Context: *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.*

Objective: *We aim to change the status quo by providing the middleware facilities necessary to ensure the consistency of software after adaptation. We would like these facilities to be reusable across different applications, such that the middleware can streamline the process of achieving safe adaptation.*

Method: *Our approach addresses the current shortcomings by utilizing the information encoded in a software system's architectural style. This information drives the development of reusable adaptation patterns. The patterns specify both the exact sequence of changes and the time at which those changes need to occur. We use the patterns to provide advanced adaptation support on top of an existing architectural middleware platform.*

Results: *Our experience shows the feasibility of deriving detailed adaptation patterns for several architectural styles. Applying the middleware to adapt two real-world software systems shows the approach is effective in consistently adapting these systems without jeopardizing their consistency.*

Conclusion: *We conclude the approach is effective in alleviating the application developers from the responsibility of managing the adaptation process at the application-level. Moreover, we believe this study provides the foundation for changing the way adaptation support is realized in middleware solutions.*

Keywords: Software Architecture, Architectural Style, Adaptation Patterns, Middleware

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.)

[1] Corresponding author

systems [1]. However, the construction of such systems has been shown to be significantly more challenging than traditional, relatively more static and predictable, software systems [2].

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* [3–5]. 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*, and *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 [6].

Software architecture has also been shown to provide an appropriate level of abstraction and generality to deal with the complexity of dynamically adapting of software systems [7]. 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 [7,8].

Architecture-based adaptation is often realized via the run-time facilities provided by an implementation platform, such as *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 [8]. 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 [9,10] 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 [11]. 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. The middleware platform, called Prism-MW [12], possesses several unique characteristics that make it suitable for realizing the patterns presented in this paper. Most notably, it is a style-aware middleware, allowing it to reflect on the style of the running application and applying the appropriate patterns. Finally, we describe our experience with realizing the approach on top of Prism-MW and applying it to a real-world software system with more than 40 KSLOC in size.

The paper is organized as follows. Section 2 presents a case study, which is used throughout this paper for describing and evaluating this research. Section 3 provides the required background. Section 4 motivates the work by summarizing the

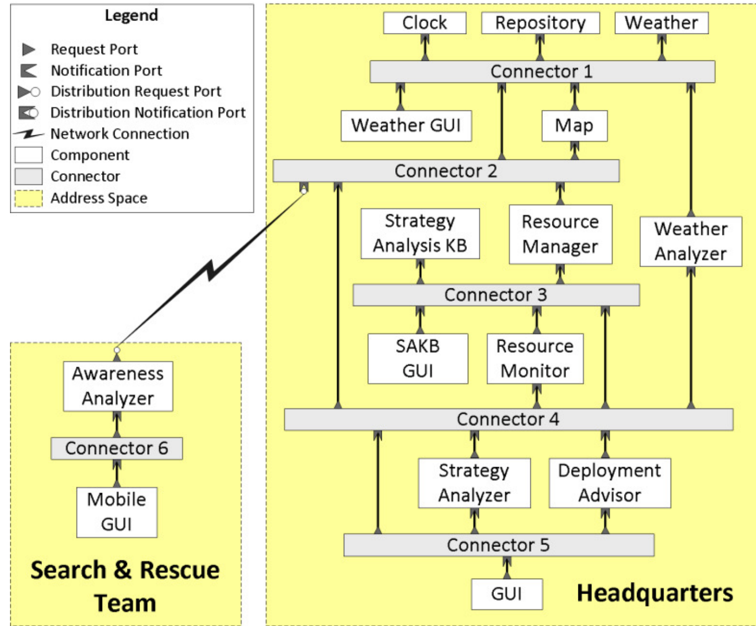


Fig. 1. The architecture of Emergency Deployment System (EDS) composed of Headquarters and Search and Rescue Team.

problems with the existing approaches. Section 5 describes our overall approach. Section 6 describes the extraction of adaptation patterns from two representative architectural styles. Section 7 describes the implementation of our approach on top of Prism-MW. Sections 8 presents the evaluation of the approach as well as our experiences. Finally, the paper concludes with an overview of the related work and outline of our future work.

2 Motivating Case Study

We use a software system that was previously developed in collaboration with a government agency to motivate the problem, describe our contributions, and evaluate the research. In this section, we provide an overview of its software architecture, including its architectural style, as well as its implementation on top of a middleware, and the challenges that we have faced in its runtime adaptation.

2.1 Description of Application

The software system, called *Emergency Deployment System (EDS)* [12], is intended for the deployment of personnel in emergency response scenarios. As depicted in Fig. 1, it is comprised of two types of subsystems: *Headquarters* and *Search & Rescue Teams*. Each *Headquarters* has several *Search & Rescue Teams,* which are equipped with smartphones and tablets running *Mobile GUI* component, providing the emergency crew with an intuitive interface to access the system's services. These teams, which are deployed in the incident area, provide a local assessment of situation using *Awareness Analyzer*.

Headquarters manages the *Search & Rescue Teams* and merges the received data into a coherent knowledge base layered on top of a *Map* and supported by a *Repository*. The *Headquarters* also provides *Weather* information managed by *Weather GUI* and a *Clock* for the system. The *Headquarters* allows the users to decide about deployment of the *Search &*

Rescue Teams and assignment of resources to them. *Weather Analyzer* provides the weather prediction, which along with other information is used for making the deployment decisions. *Headquarters* also keeps a *Strategy Analysis Knowledge Base*, which provides the domain specific policies with respect to different deployment strategies. This knowledge base is managed by *SAKB GUI*. A user uses the *GUI* of the *Headquarters* to accomplish four main goals: (1) track the resources using *Resource Monitor*, (2) distribute resources to the *Search & Rescue Teams* using *Resource Manager*, (3) analyze different deployment strategies using *Strategy Analyzer*, and finally (4) find the required steps toward a selected strategy using *Deployment Advisor*. Interested reader may find a more detailed description of EDS in [12].

2.2 Architectural Style

The architectural style of EDS is C2 [13]. In a C2 software system, a component at a given layer may *only* depend on components that are in layers above it. For instance, *Strategy Analyzer* in Fig. 1 depends on *Weather Analyzer*, but not on *Deployment Advisor* that is at the same level and *GUI* that is below it. C2 components asynchronously communicate by sending *request* events that travel up for invoking services of components above, and *notification* events that travel down for carrying responses to components below. Components in C2 style are required to communicate through connectors. However, a component may at most connect to one connector on its top and one on its bottom interface. For instance, in Fig. 1, *Clock* is connected to a connector on its bottom interface, *SAKB GUI* is connected to a connector on its top interface, and *MAP* is connected to a connector on both its top and bottom interfaces. On the other hand, a C2 Connector can be connected to multiple components and connectors on each side. A C2 connector broadcasts a request event received from the bottom to all of its interfaces on the top and any notification received from the top to all of its interfaces on the bottom.

2.3 Implementation

We have previously implemented EDS on top of Prism-MW. Prism-MW is an *architectural middleware* [12], which is a type of middleware that provides one-to-one mapping between architectural abstractions and their implementation counterparts. In other words, Prism-MW provides programming language constructs for realizing architectural concepts, such as components, connectors, configurations, etc. Most importantly, Prism-MW provides extensive support for software architectural styles. This is an important facet of Prism-MW that we have leveraged in realizing the research described here. Below, we provide an overview of a subset of EDS's implementation on top of Prism-MW to help the reader understand its usage. Later, in Section 7, we provide an overview of Prism-MW's design and its new enhancements to support the research described here.

Fig. 2a shows a subset of EDS for which the implementation fragments are provided below. Fig. 2b shows the *main* method that bootstraps the system. Here, we have abridged the code to improve its readability. Yet the code captures the essence of Prism-MW in its extensive support for architecture-based development. Intuitively, the code in Fig. 2b specifies descriptive architecture of the system as follows: (1) line 5 instantiates the *Architecture* object, called *arch*, which serves as a container for all of the constructs executing on a single host, (2) lines 8-10 instantiate the *C2Components* and C2*Connectors* that comprise the system, (3) lines 13-15 add them to the architecture, (4) lines 18-19 compose (*weld*) components and connectors into a configuration by connecting their *ports*, and finally (5) line 22 starts the architecture, which in turn triggers the execution of components comprising the architecture.

This example also demonstrates how event-based communication facility in Prism-MW can be used to enable interactions between the two components. As depicted in Fig. 2c, the *SAKB GUI* component creates and sends a request event to update a rule with two parameters in its payload (i.e., *ruleID* and *reliability* of the rule), in response to which in Fig. 2d, the *Strategy Analysis KB* component updates the corresponding rule and sends an acknowledgement back via a corresponding notification event.

In core Prism-MW, an event does not need to identify its recipient components. They are uniquely defined by the topology of the architecture and routing policies of the employed connectors [12]. This simple example shows how Prism-MW aids the programmer in building a system, such as EDS, in a manner that ensures compliance to its architectural specifications.

2.4 Challenges with Architecture-Based Adaptation

Typically, middleware support for architecture-based adaptation is realized in the form of adding, removing, and replacing software components. For instance, replacing a component in Prism-MW is achieved by a call to *remove* method followed by a call to *add* method on the existing architecture (recall *arch* in Fig. 2b). However, such changes could jeopardize the integrity of a software system; they could leave the system in an inconsistent state. For instance, consider a scenario where we would like to replace the *Map* component. This could be realized by removing the old *Map* component and adding a new instance of it [8]. This solution, however, ignores other components (e.g., *Strategy Analyzer*) that depend on *Map* for delivering their services.

Let us assume the end-user makes a "*Strategy Analysis*" request using the *GUI* component. Fig. 3 shows the interactions (events/messages) that result in response to this request. If such a request is made while the *Map* component is being replaced, and thus temporarily unavailable, it may be processed by the

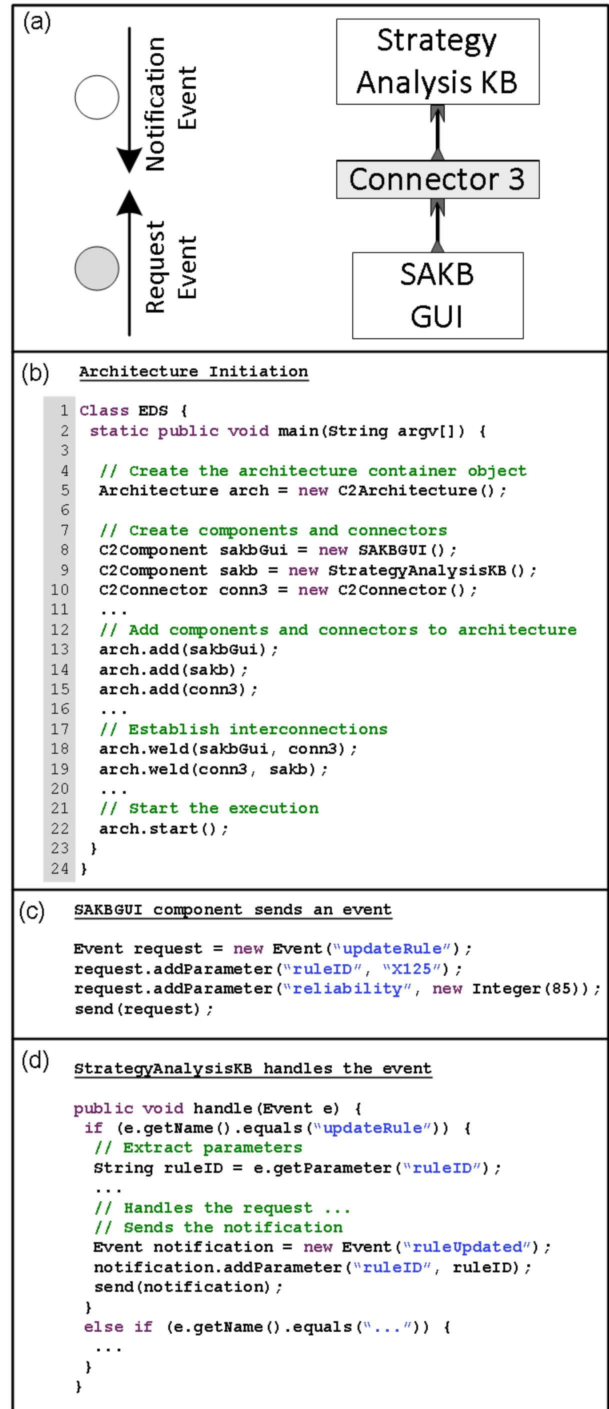


Fig. 2. Illustration of EDS implementation on top of Prism-MW: (a) the architecture of a subset of EDS; (b) Architecture initialization; (c) *SAKB GUI* sends a request event; and d) *Strategy Analysis KB* handles the request and responds.

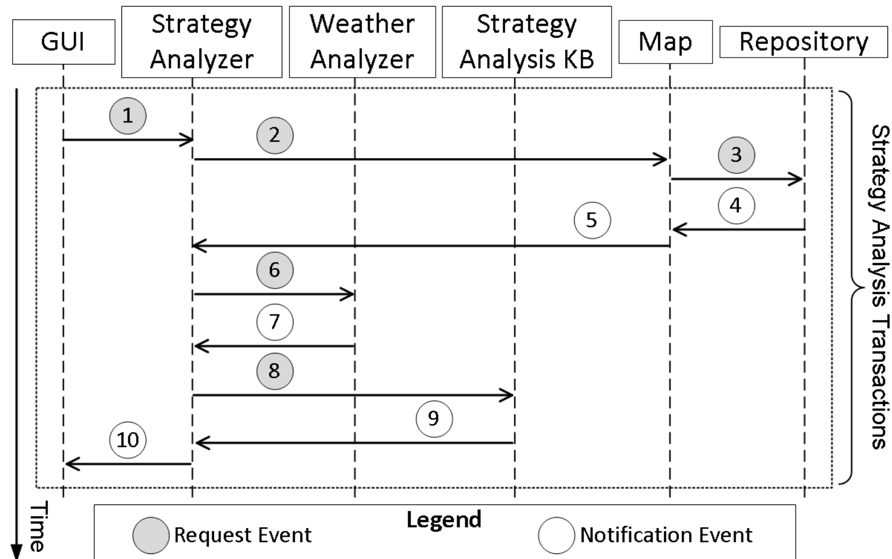


Fig. 3. The *strategy analysis* transaction and the corresponding component dependencies in EDS.

Strategy Analyzer and old *Map*, but not the new *Map*. The effect of this may manifest itself in the form of functional failure: the new *Map* may not receive event 2, resulting in the system to never respond to the user (i.e., events 3-10 do not occur).

At first blush it may seem that buffering events intended for the *Map* component would solve the problem. However, buffering by itself cannot address consistency issues that may arise. Consider the situation in which *Map* component is replaced after it has sent out event 3, but before receiving event 4. In this case, it is possible for the old *Map* to process event 2, and the new *Map* to process event 4, assuming it is buffered for later processing. However, this may violate *Map*'s interaction protocol (i.e., event 4 can be processed only after event 2 has, which would not be the case with the newly installed *Map*). Since the new *Map* may not have the correct state, the system may become inconsistent.

Note that even if the component is stateless, inconsistency problems may arise. This typically happens when a stateless component provides a reverse functionality. For instance, consider a stateless encryption component that stores/retrieves data from a file using two interfaces that are reverse of one another: *cipher* and *decipher*. Replacing this component with one that uses a different type of encryption algorithm in the middle of a transaction could break the system's functionality, since decipher interface cannot be used on data that was ciphered using the old component.

By the same reasoning, in the case of stateful components, even if there is support for state transfer (i.e., ability to extract and set the component's state externally), the issue of consistency cannot be fully addressed. The reason is that the consistency of a system depends not only on the component's internal state, but also the state of its interaction with other components.

3 Research Background

A solution to the problem of ensuring consistency during the runtime adaptation of software was proposed by Kramer and Magee's seminal model of dynamic change management, known as *quiescence* [9]. In this work, a *transaction* is defined as an exchange of information (e.g., message, event) between two components, initiated by one of the components. Each step

depicted in Fig. 3 corresponds to a transaction. A *dependent transaction* is defined as a two-party transaction whose completion may depend on the completion of other consequent transactions. Steps 1-10 in Fig. 3 form a dependent transaction.

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. The two states are, (1) *active*: A component can start, receive, and process transactions. (2) *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 of adapted component for a system to remain in a consistent state [9]. 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. For satisfying the quiescence property the component should be in the *passive* state. Moreover, *quiescence* implies that all the nodes that can initiate transactions on the updated component must be passive. This is known as the *passive set* [9]. By definition, quiescence also solves the situation in which there are cyclic or mutual dependencies between components.

For building the passive set only the static dependencies, which are the superset of all the dependencies, are considered. Some approaches (e.g., [10]) have considered dynamic dependencies for special kinds of components that can provide information about their dependencies in the future.[2] However, they still rely on static dependencies for a fall back as the reachability of a consistent state for adaptation is not guaranteed when only the dynamic dependencies are considered. Considering the dynamic dependencies can help in reducing the disruption caused by adaptation and we are studying them in a different thread of research. In this paper, however, we focus on static dependencies, and dynamic dependencies fall outside the scope of this research.

4 Research Problem

From a theoretical perspective, the approach presented above solves the problem of ensuring consistency during adaptation. However, applying this approach in real-world software systems remains a challenge. It is typically left to the application developers to implement the required change management and coordination facilities. 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. In event based systems, which are the focus of our work, component dependencies can be expressed in terms of

[2] This approach is often not practical. Detecting the dynamic dependencies in the future operation of a software component that behaves non-deterministically is undecidable over Turing machines as Halting problem can be reduced to it.

transactions. Two components depend on one another, if there is a (dependent) transaction between the two. However, identifying transactions, in particular dependent transactions, requires understanding the details of the application logic (e.g., Fig. 3), which defeats the purpose of treating components as black boxes and adapting a system at the architectural level. Identifying the dependency relationships in a large software system is very difficult.

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 with 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 facilities (e.g., data marshalization, remote method invocation, service discovery), and provide them as reusable modules to any application that is developed on top of the middleware. Unfortunately, employing the same approach in the context of adaptation is not feasible, since 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.

5 Approach Overview

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 first problem in Section 4, i.e., violates the black box treatment of components. Moreover, it hinders reusability of components developed in this manner. (2) Remove the old component abruptly and replace it with a new one. As exemplified using the EDS application in Section 2.4, this approach could leave the sytem in an inconsistent state. (3) Bring down the entire system before adapting it, and restart it afterwards. This approach results in severe disruption in system's execution. None of the existing approaches make use of advances in dynamic change management [9,10] and hence are not desirable.

We propose a new approach that builds on the quiescence model of dynamic change management (recall Section 3). The key underlying insight, established in our preliminary work [11] 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

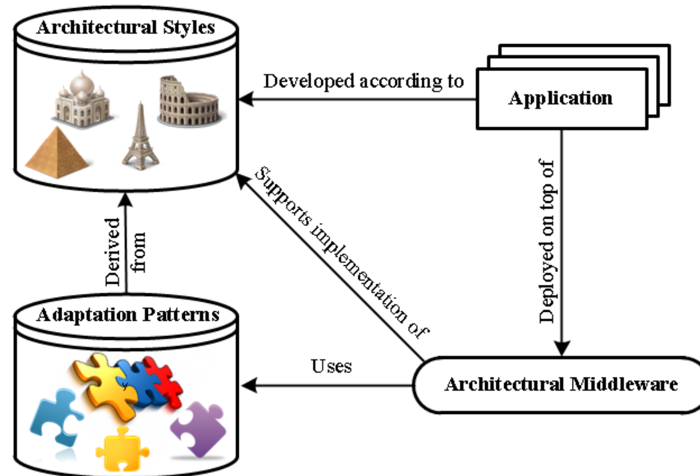


Fig. 4. Overview of the approach.

one another. The dependency relationships are critical when adapting a software system, as they determine the impact of change on the system [9,10].

An *architectural style* is a named collection of design decision, which constraint the design to a well-documented subset of possible choices [14]. Some of the most important properties of a given style are the allowable relationships among its components and connectors. If a given software system adheres to the rules and constraints of a style, we can infer the dependency relations automatically. To that end, we use the well-documented rules and constraints of a given architectural style to infer the component dependencies for the software system adhering to that style without needing additional specific information about the system.

An example of this can be seen in Fig. 1. Since we already now that the style of EDS is C2, without knowing the details of its application logic, we can derive dependency relationships between its components. For instance, since *SAKB GUI* is below *Strategy Analysis KB*, before doing any change to the latter, we need to make sure that the former is in a passive state. Note that this relationship is derived only from the style and topology of the architecture and without using any internal knowledge about the application. In other words, the proposed approach does not require availability of a detailed model of the component interactions in the system, such as that shown in Fig. 3, for reasoning about quiescence.

The component dependencies are in turn used to determine a reusable sequence of steps for placing a component of a given style 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 [9], and (2) applications built according to the style must comply with the dependency relationships established in that style.

We have enhanced an existing middleware platform with style-induced adaptation patterns discussed above. Unlike any existing solution, the middleware ensures the consistency of the adapted software system. Since each style acts according to a

predefined set of rules, which are enforced by the middleware, the adaptation capability can be reused for any software built according to that style. Therefore, the middleware alleviates the application developers from implementing the same complex and error-prone functionality. Our experience suggests that the proposed approach effectively alleviates the challenges mentioned in Section 4. The following sections describe the approach in more detail.

Fig. 4 depicts the relationships between the different components of our approach. An *application* is developed according to a style, which comes from a set of known *architectural styles*. For each style in this set, the related *adaptation pattern* is derived and maintained in a repository. An *adaptation pattern* is used to orchestrate the process of runtime change in the applications of the corresponding style. This way an *application* benefits from the reusable adaptation facilities provided by a *middleware* once it is deployed on top of it. Application developers can also extend the set of *architectural styles* and corresponding *adaptation patterns* to account for domain-specific styles. In the following section we describe the derivation of *adaptation patterns* for two complex and well-documented architectural styles.

6 Style-Driven Adaptation Patterns

In this section, we describe the process of extracting adaptation patterns for two representative and complex architectural styles, C2 [13] and D3 [15]. In extracting such patterns we assume that a given component has a single style. This does not prevent a software system from having a composition of different styles as long as the components are not shared among styles. Note that while the overall approach can be applied similarly 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.

6.1 Adaptation Patterns for C2 Style

During normal operation a *C2Component* receives asynchronous messages from an associated *C2Connector*. The C2 style defines how each of the architectural constructs can be connected: each *C2Component* must be connected to at least one and at most two *C2Connector*s, while a *C2Connector* can be connected to as many *C2Component*s and *C2Connector*s as required. A software system built in the C2 style consists of layers (similar to Fig. 1), where *request* events travel upward, while *notification* events travel downward [13]. *Request* and *notification* events received from bottom and top connectors are evaluated to determine which need to be processed (depicted in Fig. 5). If the event is not intended for the component, it returns to the *Waiting* state. Otherwise, the event is processed 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.

We chose C2 style [13] due to its intended use in dynamic settings [8], and its rich set of rules and constraints that form a superset of those in simpler styles (e.g., Client-Server). Moreover, any C2 software system typically consists of many dependent transactions, making it a suitable style for describing adaptation patterns.

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

An adaptation pattern could be expressed using statechart models (Fig. 6). 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. 6a presents the transitions that take an *Active* C2Component that is being adapted to satisfy the quiescence property. When in the *Active* state, component processes 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 components 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 transitions to either *Quiescent* if it is waiting, or to *Quiescing Itself* if it is processing. In the latter case, the component eventually transitions to the *Quiescent* when the component has completed the work. Note that according to Kramer and Magee [9] a component can only be in two states (i.e., active, passive). In our adaptation patterns, we are modeling additional states (e.g., *Quiescent*), which refer to the intermediary steps during the adaptation process, and not the state of a component.

Fig. 6b shows the transitions that take a C2Connector in the substrate (i.e., a connector that depends on the C2Component that is currently being adapted) from *Active* to *Passive* state. The first step a connector takes is to request all of its bottom components to become passivated. Once all of the connector's bottom components have become passivated, if it is currently waiting, it transitions to the *Passive* state. Otherwise, if the connector is busy handling (routing) messages, it transitions to the *Passivating Itself* state and waits for the job to finish before transitioning to the *Passive* state.

Similar to the previous two adaptation patterns, Fig. 6c shows the transitions that take a C2Component in the substrate (i.e., a C2Component that depends on the component currently being adapted) from *Active* to the *Passive* state.

These patterns ensure that once the component/connector transitions to the *Quiescence* state, all the components/connectors that may depend on it have also transitioned to *Passive* state. This means that the quiescence property holds for the component. By executing these patterns, the middleware waits until the component that is being adapted

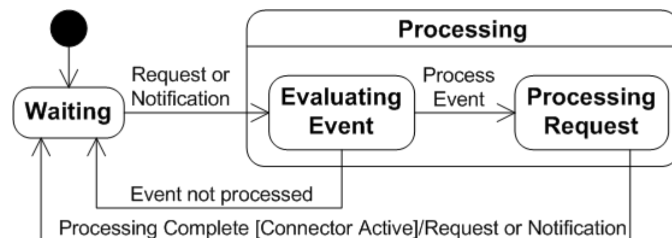


Fig. 5. Life cycle of a C2 Component during normal operation.

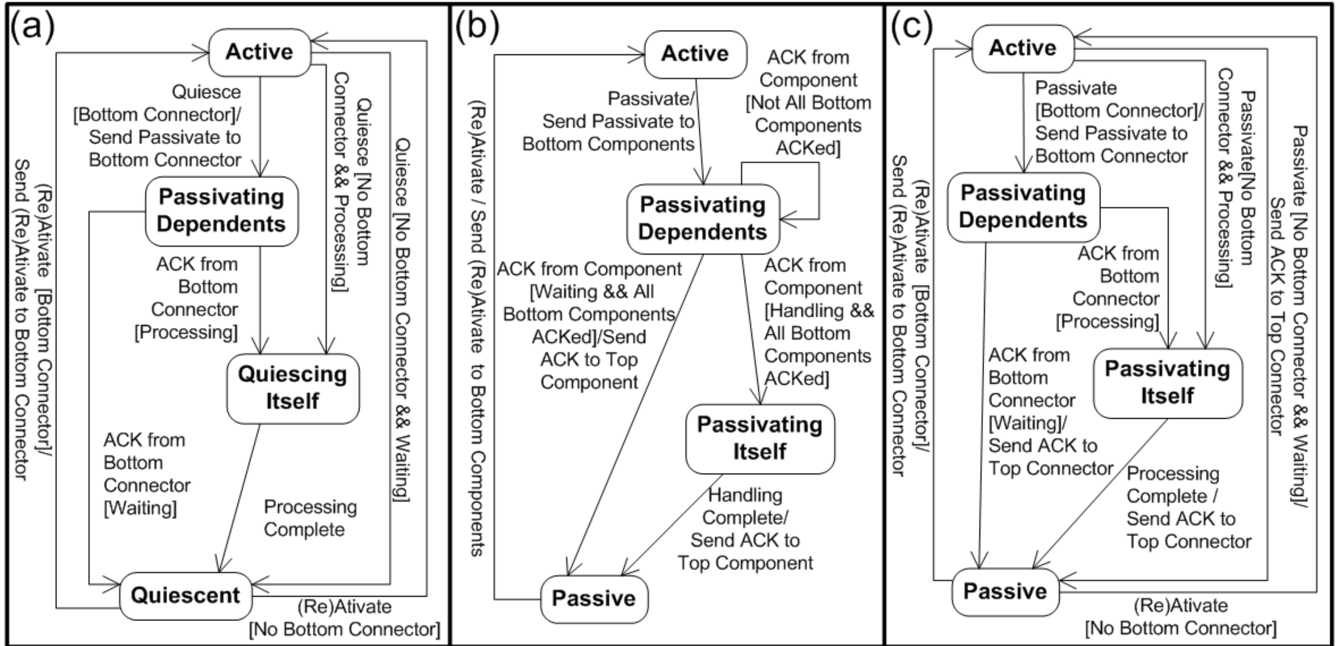


Fig. 6. The statecharts of C2 adaptation patterns: (a) A C2Component that is being adapted; (b) A C2Connector in the substrate that depends on the adapted C2Component; and (c) A C2Component in the substrate that depends on the adapted C2Component.

achieves the quiescence state before replacing it. We further detail the middleware's role in Section 7.

The patterns 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 do not show the adaptation patterns of other generic styles (e.g., Publish-Subscribe, Client-Server) that we have developed so far.[3] While the overall approach of developing adaptation patterns for other styles is the same as what has been presented above, our experience suggests that different styles often result in drastically different patterns.

6.2 Adaptation Patterns for D3 Style

In the previous section we described a generic architectural style. As another example, in this section we describe a *domain-specific software architectural style*, called D3 [15]. *Domain-specific software architecture* [16–18] aims to increase productivity and reuse in the construction of software systems for a particular application domain. It captures the best practices and experiences from the domain engineers in such a way that guides the future design of similar systems [19]. An effective method of representing this knowledge is to define domain-specific component/connector types, and rules that guide their interaction, in the form of a domain-specific software architectural style.

D3 [15] is an example of a domain-specific architectural style that targets the domain of Computer Supported Collaborative Design. Software systems developed for this domain facilitate concurrent construction, sharing, and

[3] Interested reader can access the growing repository of adaptation patterns at http://cs.gmu.edu/~smalek/AdaptationPatterns

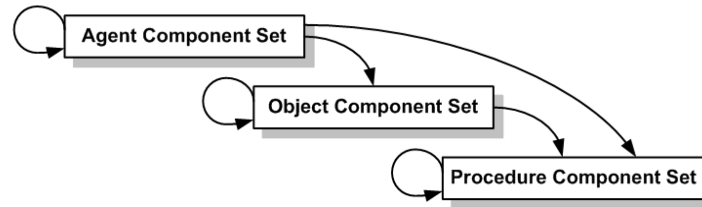


Fig. 7. Layering between component sets in D3 reproduced from Li et al.

synchronization of design artifacts among a distributed group of users. There are three types of components in this domain: (1) *Procedure* components are stateless functions that mainly perform commonly required transformations on the data; (2) *Object* components encapsulate data about design artifacts and the operations defined on top of them; and (3) *Agent* components are active entities which are used to help each designer to work with the system. As depicted in Fig. 7, the components are organized in three layers of sets. In each layer a component can depend on the components that are in the layer below or at the same layer. In other words, An Agent component can access (and depend on) other Agent, Object, and Procedure components; while an Object component can only access other Object and Procedure components; and finally a Procedure component can only access other Procedure components.

The components communicate with one another via a domain-specific connector, called *TriBus*. TriBus connector provisions asynchronous request and response communication. Furthermore, the TriBus connectors enforce the rules of layering and each connection between two components should go through it. To that end, components register themselves and request connections to other components using TriBus which plays the role of mediator between two components.

The communication between components does not go beyond short term request-response relationships except communication among Agents; Agents can have long living communication sessions, which are managed using style-specific messages, such as *INFORM*, *REQUEST*, *PROPOSE*, *ACCEPT*, and *DECLINE*. An Agent can voluntarily end any prior engagement with other Agent components at any time by sending a DECLINE message to them. However, it should wait for receiving the DECLINE message back from other Agents before finishing the session.

Fig. 8 depicts the adaptation pattern for the architectural constructs comprising D3 style. The Procedure component is one-time application of a function on data which makes the component stateless by definition. Therefore, the adaptation patterns for the Procedure component (see Fig. 8a and d) do not need to passivate the dependent components and simple queuing of received events suffices. Note that this does not contradict with the argument about stateless components in Section 2.4, as the Procedure components are applied one-time only and do not provide a reverse functionality. On the other hand, the patterns for stateful components (see Object and Agent) need to establish the passive set first (i.e., Fig. 8b, c, e and f).

Extra care should be taken for managing long living communication sessions between Agents. Before adapting an Agent, all the sessions in which the Agent is present should be finished. This is achieved through sending DECLINE message to all collaborating Agents (i.e., Fig. 8c and f). The information about active sessions is encapsulated in TriBus therefore broadcasting the DECLINE message is left to adaptation pattern of TriBus (see Fig. 8g), which is discussed further below. An Agent may receive DECLINE message from other Agents; the Agent should process the message and DECLINE back.

Note that receiving DECLINE message cannot happen after the Agent has received an ACK from TriBus, as all the sessions have already finished.

Fig. 8g shows the adaptation pattern for TriBus. Note that the pattern only manages the adaptation state related to the part of connector that deals with the connections between the component being adapted and its dependent components. In

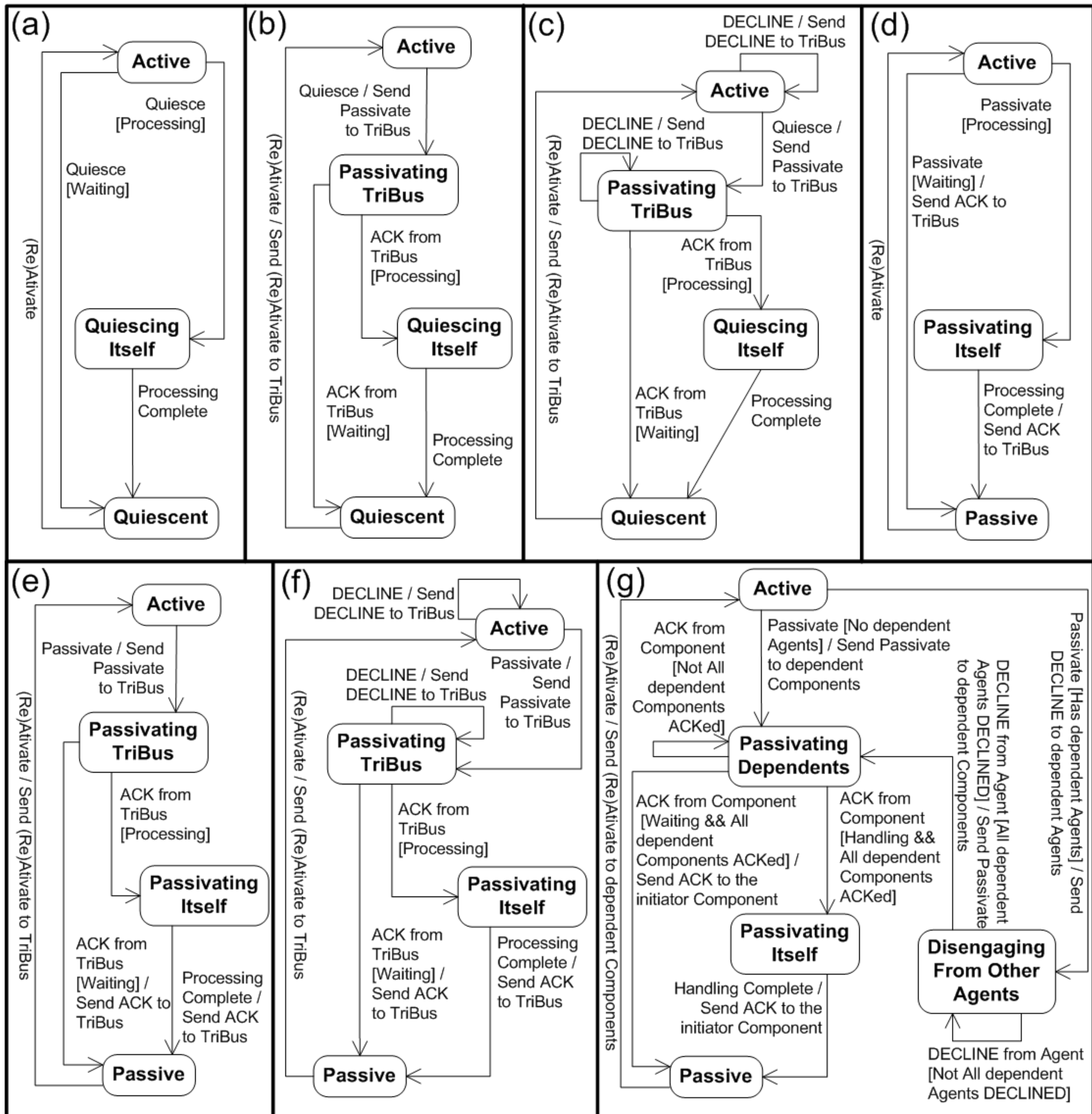


Fig. 8. The statecharts of D3 adaptation patterns: (a) A Procedure that is being adapted; (b) An Object that is being adapted; (c) An Agent that is being adapted; (d) A Procedure that depends on the adapted Component; (e) An Object that depends on the adapted Component; (f) An Agent that depends on the adapted Component; and g) The TriBus (Connector in D3 style) that is connected to the component being adapted or passivated.

other words, the other connections established through the same TriBus are unaffected. When the TriBus receives the Passivate request from a component, it looks into its registry to find the dependent components. If the component sending the Passivate request is an Agent component and there are other Agent components depending on it, TriBus sends them the DECLINE message and waits for them to acknowledge that by sending a DECLINE message back. When that is not the case (i.e., no other Agent component is dependent on the one being adapted), the pattern passivates all the dependent components. Afterwards, the TriBus transitions into Passive state and sends an ACK to the component that initiated the process (i.e., sent Passivate request to the TriBus). As in other patterns, when the TriBus is (Re)Activated, it also (Re)Activates all of the passivated components.

We next focus on the second contribution of our work, describing how the adaptation patterns described above can be used to advance the state of the art in middleware design.

7 Style-Aware Adaptation

We have leveraged the style-driven adaptation patterns described above to provide advanced run-time adaptation facilities in Prism-MW [12]. Prism-MW is an *architectural middleware*, which supports architecture-based development 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. As a result, in Prism-MW, the architectural models are completely synchronized with the implementation of the system, which alleviates the problem of architectural erosion. 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. In this section, we first provide an overview of these capabilities to familiarize the reader with this middleware. Afterwards, we provide a detailed description of the enhancements made to Prism-MW to realize style-aware adaptation of software systems. Finally, we conclude this section with a discussion of the performance consequences associated with using the revised version of the middleware.

7.1 Overview of Prism-MW

Fig. 9 shows the class diagram view of Prism-MW. The gray classes constitute the middleware core. *Brick* is an abstract class that represents an architectural building block. It encapsulates common features of its subclasses (*Architecture*, *Component*, *Connector*, and *Port*). *Architecture* records the configuration of its constituent components, connectors, and ports, and provides facilities for their addition, removal, and reconnection, possibly at system run-time. *Event*s are used to capture communication in the architecture. *Component*s perform computations in the architecture. The developer provides the application-specific logic by extending the component class. *Connector*s are used to control the routing of events among the attached components. Components and connectors can have an arbitrary number of attached *Port*s, which they use to attach to one another and interact. Every *Brick* in Prism-MW is associated with the *Scaffold* class, which provides a number of facilities for queuing, scheduling, monitoring, and routing of events in a distributed setting.

Prism-MW's core 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 first step a developer takes is to

subclass from the *Component* class for all components in the architecture to implement their application-specific methods. The next step is to instantiate the *Architecture* class and to define the needed instances of components, connectors, and ports. Finally, attaching component and connector instances into a configuration is achieved by using the *weld* method of the *Architecture* class.

Through the use of the *style-specific* classes shown in orange in Fig. 9, the middleware's default style-agnostic behavior can be modified. This feature of Prism-MW has been successfully employed to provide support for more than 20 different architectural styles, including C2, Client-Server, Publish-Subscribe, and Pipe-and-Filter [12]. For instance, a style-specific component, such as *Server*, *Publisher*, or *C2Component* can be constructed by sub-classing from the regular *Component* class and providing the style-specific logic. Similarly, as depicted in Fig. 9, other sub-classes of *Brick* (i.e., *Architecture*, *Connector*, and *Port*) can be extended to realize the proper stylistic behavior. Interested reader may refer to [12] for more details on Prism-MW.

7.2 Style-Aware Adaptation Support

In order to support the run-time changes to the application, Prism-MW components and connectors can be added, removed, and welded during the execution of the system. As discussed further below, we have enhanced the middleware's basic dynamism with the style-driven adaptation patterns to ensure the consistency of the system during such changes.

We have realized support for the adaptation patterns in Prism-MW through three new facilities: (1) management and enforcement of the adaptation state (e.g., passive, passivating dependents, quiescence) of components and connectors, (2)

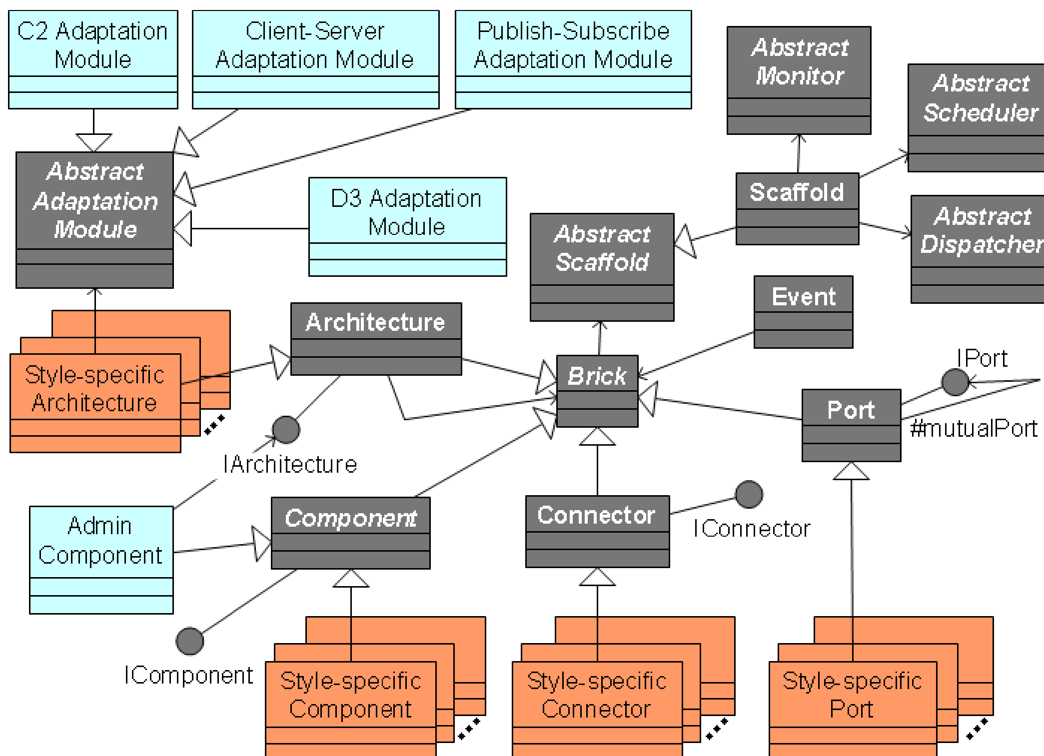


Fig. 9. Simplified UML Class diagram of Prism-MW: middleware core shown in gray, style-specific extensions shown in orange, and enhancements for style-aware adaptation support shown in blue.

realization of the adaptation patterns via several pluggable modules, and (3) execution of adaptation patterns and coordination of change management via meta-level components. We first describe each of these enhancements in detail below. We then discuss the implications of distribution on these facilities. Finally, we present an example to illustrate how the different parts of the middleware fit together to support style-aware adaptation of software.

7.2.1 Adaptation State

A *status* variable is associated with every *Brick* to determine the adaptation state of components and connectors. Recall from Fig. 9 that both Prism-MW's *Component* and *Connector* extend the *Brick*. The *Brick* abstract class also provides *setter* and *getter* methods for accessing the status of a component and connector. Note that while some existing middleware platforms may provide information about the "liveness" property of a software component (i.e., whether or not it is active), most cannot provide additional information, such as which components are in passive mode and which ones satisfy the quiescent property. As further described below, Prism-MW tracks and updates the adaptation state of each component in the system, alleviating the component's application logic from managing its adaptation state.

In Prism-MW, *Component* class is extended for realizing the application logic. Among other services, *Component* class provides the ability to send and receive events to the application logic. We have used the same mechanism to enforce the components behaving according to their adaptation status. The *Component* class provides a wrapper that prevents "rogue" application logic from initiating a new transaction when in the *Passive* state. This also allows the middleware to detect the "rogue" components that ignore the passivate command and keep the active state for unlimited time. There are different ways of dealing with such components. In some domains a simple time-out mechanism [20] can solve the problem. However, in risk-averse domains informing the end-user is the best option, as this may be the sign of a problem with the component's functionality and trustworthiness. Currently, Prism-MW could be configured to take either approach.

The third party components can also be wrapped using the *Component* abstract class in Prism-MW to enable control over them. This way Prism-MW is able to manage the component's lifecycle in the adaptation process. Prism-MW is also compatible with a third party component that can be used in different applications (e.g., services), as long as the component manages the separation between usages in different applications (e.g., state isolation) and provides a separate interface to Prism-MW for controlling the component in each usage.

7.2.2 Adaptation Module

The adaptation patterns have been realized as pluggable *Adaptation Modules*, depicted as blue classes in Fig. 9. A style-specific *Adaptation Module* needs to override and provide an implementation for two methods inherited from *Abstract Adaptation Module*: *establishPassiveSet* and *revive*. The first one returns the steps necessary for transitioning a component (connector) from active to passive, while the second one returns the same for achieving the reverse (i.e., from passive to active). These two methods codify state machines that correspond to the adaptation patterns, such as those described in Sections 6.1 and 6.2. Each step represents one of the transition labels in the statechart of adaptation pattern. Each step is realized as a triplet-tuple of the form <*condition*, *state change*, *action*>.

One of the goals in Prism-MW, which is also something we strived for in providing the style-aware adaptation support, is the extensibility of the middleware for the needs of application developers. One of the ways that developers can extend Prism-MW is the introduction of new architectural styles [12]. Our design also allows the developers to provide adaptation support for a new style by implementing the corresponding *Adaptation Module*. Thus, depending on the architectural style of a given application, different instance of *Adaptation Module* is associated with the *Architecture* object. There is no hidden feature for this as we use the same approach for providing the adaptation support for the styles currently supported in Prism-MW.

7.2.3 Admin Component

As shown in blue in Fig. 9, we have developed a meta-level component, called *Admin Component*, which is in charge of coordinating the execution of adaptation patterns. An *Admin Component* is instantiated with every instance of the middleware and coexists with the application-level components on the same architecture container. Unlike the other components, however, it has a pointer to the *Architecture* object. Through this pointer, the *Admin Component* is capable of reflecting on both the middleware and the running application. It can identify the architectural style of the running application, use the style-specific *Adaptation Modules*, access the application's architectural constructs, and use the *setter* and *getter* methods to set their adaptation state.

Admin Component provides three interfaces: *quiesce*, *passivate, and activate*. They are used to place a given component in the appropriate adaptation state before/after the *Architecture*'s *add*, *remove*, and *replace* methods are invoked. Given a component to be quiesced, the *Admin Component* uses the configuration of the architecture to determine the affected components and connectors (i.e., those that need to be passivated). It also uses the *Adaptation Module*'s *establishPassiveSet* to determine the sequence of steps necessary for placing those components and connectors in passive state. Executing these steps may result in further invocations of the *Adaptation Module*'s *establishPassiveSet* on new component. *Admin Component* executes the steps by directly invoking the components using the *status setter* and *getter* methods described in Section 7.2.1. A similar process is employed to activate the components.

7.2.4 Distribution

Support for distribution in Prism-MW is provided through *distribution ports* (see [12] for details). These ports allow communication between different address spaces (hardware hosts). For instance, Fig. 10 shows a distributed variation of the small C2 architecture shown in Fig. 2a, where the basic ports connecting *SAKB GUI* and *Connector 3* are changed to distribution ports.

Prism-MW uses the same basic mechanism for communication that spans address spaces as it does for local communication: A sending component or connector places its outgoing event on an attached port. However, instead of depositing the event on the local event queue, the distribution port deposits the event on the network. Once the event is propagated across the network, the distribution port on the recipient address space uses its internal thread to retrieve the incoming event and places it on its local event queue. This mechanism provides distribution transparency in the system and allows components in the middleware to behave as they would without distribution.

The *Admin Components* in different address spaces use distribution ports to coordinate the adaptation in a distributed setting. The local *Admin Component* sends an adaptation step that needs to execute remotely to its counterpart via the distribution ports. For instance, consider the situation in which *Connector 3* in Fig. 10 is being passivated, in turn requiring *SAKB GUI* to also be passivated. Since *Admin Component* of *Address Space "A"* cannot directly control *SAKB GUI*, it uses its distribution port to send a request to the *Admin Component* in *Address Space "B"* to *passivate SAKB GUI*. The resulting confirmation also traverses back through the same distribution ports.

7.2.5 An Example

We now illustrate the interactions among the different elements of the middleware in applying the C2 adaptation patterns (recall Fig. 6) on a subset of the EDS architecture depicted in Fig. 10. In this example, we consider a scenario in which *Strategy Analysis KB* in Fig. 10 is being updated with a newer version of the component and hence needs to transition to the quiescent state. Fig. 11 depicts the resulting interactions within the middleware constructs and with the application components/connectors for making this transition possible.

The scenario starts when a *quiesce* request for *Strategy Analysis KB* is received by the *Admin Component* of address space *"A"* (step 1). The *Admin Component* consults with the local *Adaptation Module* to determine the appropriate sequence of change management steps (steps 2-3). Per instructions received from the *Adaptation Module* and given the current configuration of the architecture, the *Admin Component* transitions the *status* of *Strategy Analysis KB* to *Passivating Dependents* (step 4) and recursively calls itself to *passivate* the bottom connector (step 5). This recursive call results in the *Admin Component* again consulting with the local *Adaptation Module* to determine the set of steps for passivating the connector (step 6-7). Per received instructions, the *Admin Component* then sets the status of the connector to *Passivating Dependents* (step 8). The *Admin Component* also sends a Prism-MW event using the distribution ports to the remote *Admin Component* running on address space *"B"* to passivate the *SAKB GUI* component (step 9). The remote *Admin Component* consults the *Adaptation Module* on host *"B"* to place *SAKB GUI* in the passive state and executes the necessary steps (steps 10-14). Through its pointer to the architecture, the remote *Admin Component* is able to determine that there are no additional components/connectors that depend on *SAKB GUI*. Following the confirmation that *SAKB GUI* is passive (step 15), the *Admin Component* proceeds with executing the remaining steps to place the connector in the passive state (steps 16-19) and

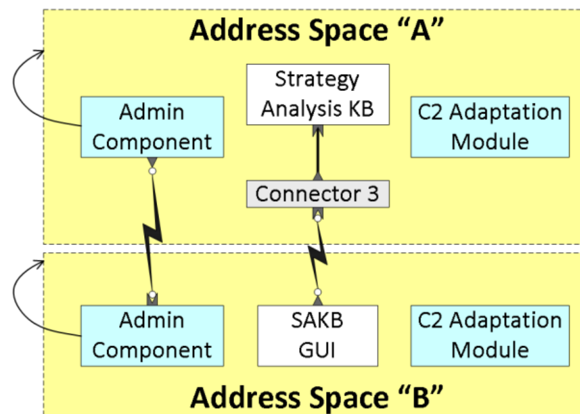


Fig. 10. Admin Components managing the adaptation in a distributed setting.

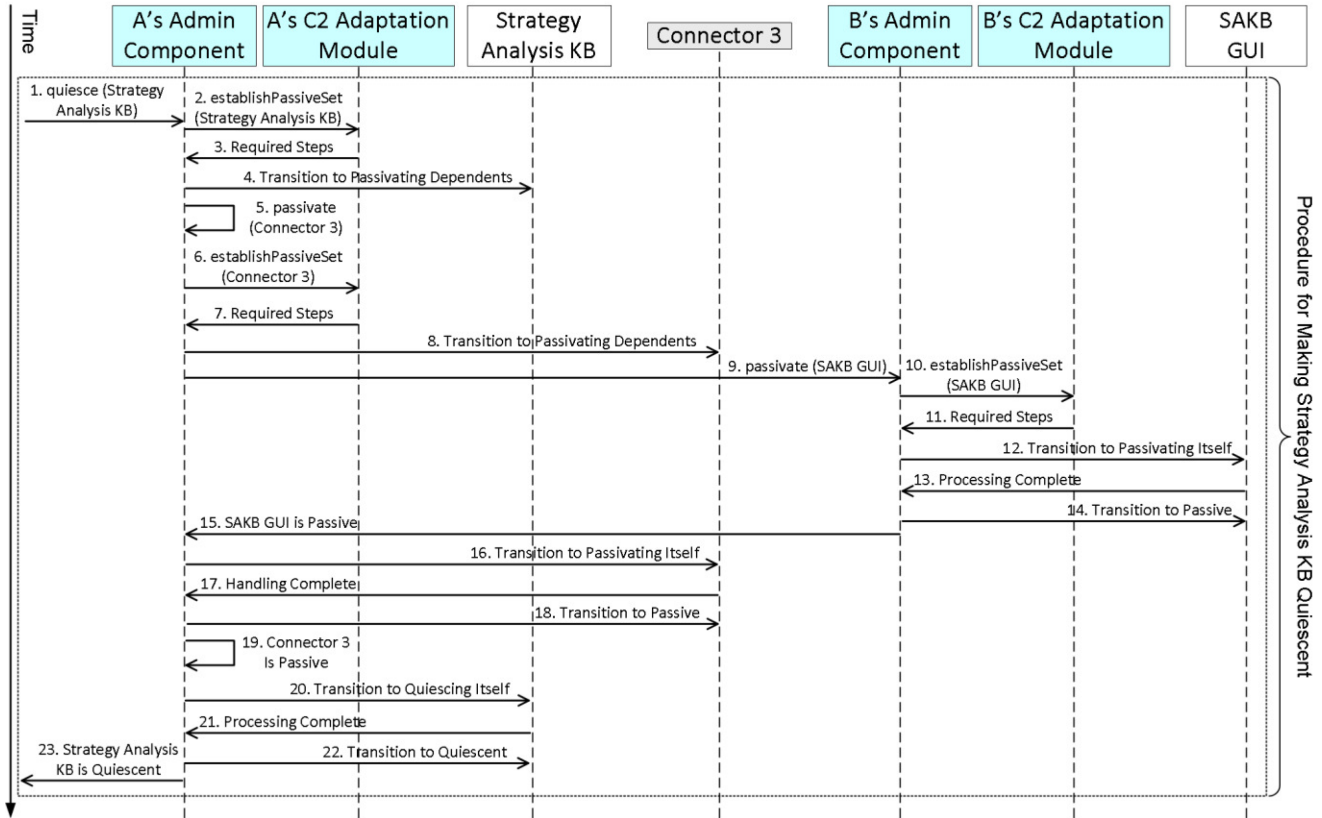


Fig. 11. Procedure for making *Strategy Analysis KB* in Fig. 10 quiescent, which involves communication between *Admin Components* of different address spaces.

subsequently placing the *Strategy Analysis KB* in the quiescent state (steps 20-23).

In summary, Fig. 11 shows the manifestation of the middleware's realization of the C2 adaptation patterns in this example. Indeed, there is a very close correspondence between the interactions depicted in Fig. 11 and the statecharts representing the C2 adaptation patterns in Fig. 6.

7.3 Efficiency of Adaptation

Adaptation patterns provide a reasonable compromise between abruptly replacing a component, which achieves fast adaptation at the expense of jeopardizing the system's consistency, and complete restart of the system, which is very slow, but provides the only other alternative to quiescent for ensuring consistent adaptation of the software. The time it takes to place a software component in quiescence is determined by the time it takes to establish the passive set for that component. This is represented as $Time(PS_c)$, where PS_c represents the set of components that need to be passivated for component c.

When two or more components need to be quiesced at the same time, a combination of three general cases may occur: (1) passive sets of components are completely disjointed, (2) some components' passive sets are completely subsumed in the others' passive sets, and (3) components' passive sets partially overlap. If adaptation actions (commands) corresponding to the adaptation patterns can proceed in parallel, then the time taken to place all of the components in quiescence is bounded by the time required to place the component that takes longest in quiescence. That is, for n components, we say $Time(PS_{1..n})$ =

max{$Time(PS_1)$, …, $Time(PS_n)$}. On the other hand, if the commands can only proceed sequentially, then the time taken to place all of the components in quiescence is bounded by the sum of the times required to quiesce components individually. That is, for n components, $Time(PS_{1..n}) = \Sigma_{i=1..n} Time(PS_i)$. Finally, note that since some components' passive sets may overlap, the actual time it takes to achieve quiescence is often smaller than the theoretical bound provided above.

In the current implementation of the approach in Prism-MW, putting multiple components in quiescence occurs sequentially. In practice, however, the time taken to actually adapt (replace) a set of components takes a bit longer than the time it takes to achieve quiescence, as there is also an additional delay associated with the API calls to remove the old components, and instantiate and start the new ones.

8 Evaluation

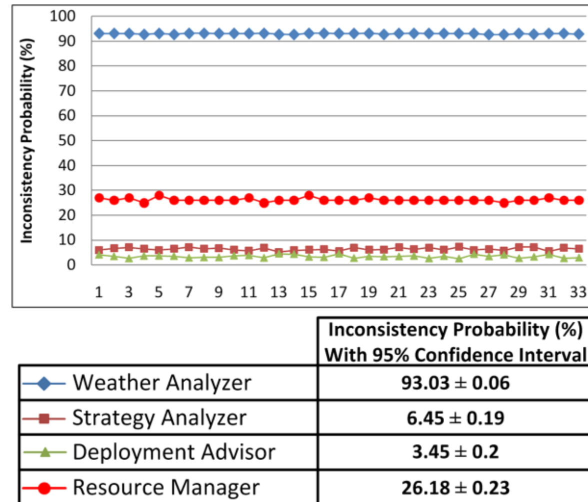
Our experiences with enhancing Prism-MW to provide adaptation support for several well-known architectural styles (C2, Client-Server, Publish-Subscribe, D3, and Pipe-and-Filter) and its application on two real-world software systems have been very positive. In this section, we first present the empirical results obtained in our controlled experiments in the laboratory, which demonstrate the benefits and characteristics of our approach, followed by a qualitative overview of our experience in the context of applying the approach in practice.

8.1 Benchmarks

We performed extensive benchmarks of the middleware in a controlled experimental setup in the laboratory to measure and evaluate our approach. These experiments were performed using the enhanced version of Prism-MW and on EDS to empirically evaluate the advantages of our approach versus the existing approaches enumerated in Section 5.[4] We do not consider the naive approach that assumes a component can provision its dependencies, since in an event-based system it breaks the information hiding and black box principles that form the fundamental premises of our work, as well as works by others in change management [9].

Our approach can be applied in a distributed setting by simply leveraging Prism-MW's distribution facilities [12]. However, for a fair comparison, we restricted the evaluation of adaptation patterns to a local system, i.e., the headquarters portion of EDS depicted in Fig. 1. This allowed us to ignore network delays as well as other factors associated with distribution that may influence the results. We experimented with three adaptation scenarios, each involving the replacement of one of the following four components: *Weather Analyzer*, *Strategy Analyzer*, *Deployment Advisor*, and *Resource Manager.* *Weather Analyzer* is a batch processing component that based on the *Weather* data provides weather related predictions. This component is one of the most computationally intensive components in the system. The other three components, on the other hand, are interactive and sporadically invoked. We applied each adaptation scenario in 33 different executions of the system and established a 95% confidence interval to obtain the results reported here.

[4] Interested reader may download the original version of Prism-MW from http://csse.usc.edu/~softarch/Prism/ and the version of Prism-MW enhanced with adaptation patterns from http://cs.gmu.edu/~smalek/AdaptationPatterns/



	Inconsistency Probability (%) With 95% Confidence Interval
Weather Analyzer	**93.03 ± 0.06**
Strategy Analyzer	**6.45 ± 0.19**
Deployment Advisor	**3.45 ± 0.2**
Resource Manager	**26.18 ± 0.23**

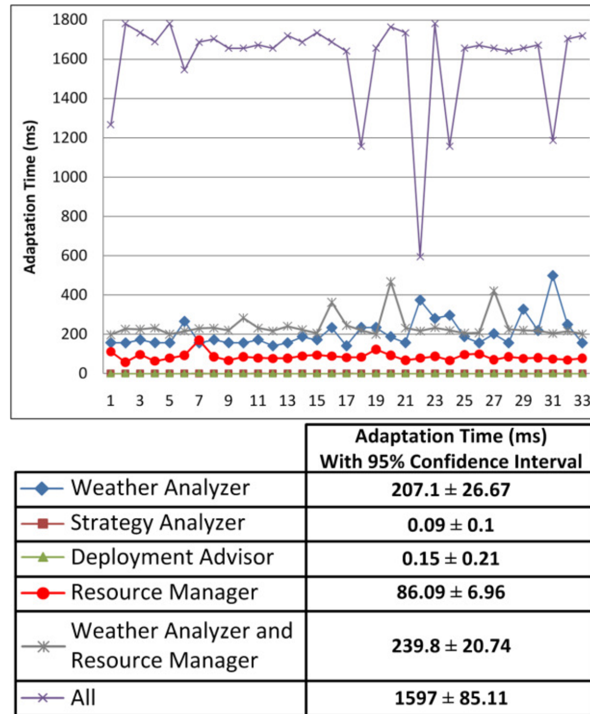
Fig. 12. Likelihood of inconsistency due to abrupt replacement of 3 EDS components in 33 different scenarios.

Fig. 12 shows the possibility of placing the system in an inconsistent state as a result of abruptly replacing the corresponding component in the 33 experiments. These experiments were performed on the version of Prism-MW without support for adaptation patterns. Each data point is a probability that is obtained through instrumentation of the system (e.g, utilization metrics, transaction traces) and represents the amount of time a component is busy in the measured period. As shown, replacement of the *Weather Analyzer* component is more likely to result in an inconsistency (i.e., on average 93%) than the other two components. The reason is that the *Weather Analyzer* is a batch processing component that is constantly utilized, while the other three are sporadically invoked as a result of interactions initiated by the user. The utilization of *Resource Manager* is also relatively high, as it provides services to several other components, and thus the inconsistency is more likely (i.e., on average 26%) if it is changed abrubtly. The possibility of inconsistency is significantly lower in the case of *Strategy Analyzer* and *Deployment Advisor*. However, they remain high enough to pose a significant risk to the correct operation of the system.

In comparison to the results of Fig. 12, applying the same adaptations in the 33 experiments using the version of Prism-MW with support for adaptation patterns showed no inconsistent behavior. This result is not surprising per se, since our approach is designed to ensure consistency through application of adaptation patterns and establishment of passive sets. However, the experiments highlight a key weakness of the modern middleware platforms and demonstrates the high likelyhood of functional inconsistency due to adaptation, in particular, in situations where the software is highly utilized.

Fig. 13 compares the adaptation time of our approach against the complete restart of the system, which is the only approach that can ensure consistent adaptation of the software. Here, we compare six adaptation scenarios with one another: four scenarios involve replacement of the individual components, one scenario involves replacement of two components (*Weather Analyzer* and *Resource Manager*), and finally the last scenario involves complete restart of the system, indicated as *All* in Fig. 13.

Replacing *Weather Analyzer* requires more time than the other three components. This is expected, due to the fact that *Weather Analyzer* is significantly more utilized (i.e., it is more likely to be actively involved in transactions) than the other three components, hence it is more likely for the middleware to have to wait for the current running transaction to finish



	Adaptation Time (ms) With 95% Confidence Interval
Weather Analyzer	**207.1 ± 26.67**
Strategy Analyzer	**0.09 ± 0.1**
Deployment Advisor	**0.15 ± 0.21**
Resource Manager	**86.09 ± 6.96**
Weather Analyzer and Resource Manager	**239.8 ± 20.74**
All	**1597 ± 85.11**

Fig. 13. Comparing the adaptation time of our approach against the complete restart of EDS in 33 different scenarios.

execution (and thus achieve quiescence) before replacing it. As one would expect, the results also corroborate that restarting *All* of the components takes significantly more time than using the adaptation patterns to achieve quiescence and adapt the components at runtime.

From Fig. 13 we can see the result of evaluating the patterns for adapting the *Weather Analyzer* and *Resource Manager* at the same time. Recall from Section 7.3 that in Prism-MW quiescing multiple components involves sequential application of adaptation patterns on those components. Therefore, the theoretical bound on the time it takes to put *Weather Analyzer* and *Resource Manager* components in quiescence at the same time is the sum of the times that would take to quiesce them individually. In this case, the theoretical bound at 95% confidence would be 207.1ms + 86.09ms = 293.19ms. However, from Fig. 13, we can see that in practice, the actual time it takes to apply the patterns, achieve quiescence, and adapt the components is significantly smaller, i.e., 239.8ms. As we mentioned in Section 7.3, this is due to the fact that the adapted components (i.e., *Weather Analyzer* and *Resource Manager*) are sharing several components in their passive sets (i.e., *Strategy Analyzer*, *Deployment Advisor*, and *GUI*). Therefore, once the middleware is done with putting one of the components in quiescence, some (if not all) of the members of the passive set of the other component are already passivated. This also has another good side effect. Due to passivation the load on the second component also decreases, hence, quiescence for the second component can be reached much faster compared to when the process starts from scratch.

Finally, note that unlike complete restart of the system, using our approach a significant portion of the system remained operational during the adaptation process. For instance, *Weather GUI* and *SAKB GUI* could be used to access a significant subset of system's capabilities on the *Headquarters* platform, while *Search and Rescue* devices could access the *Headquarters* services throughout the adaptation scenarios that do not affect those parts of the system.

8.2 Experience

Beyond the experimental results discussed in the previous section, we have had experience with applying the approach in a real-world software system, called MIDAS [3], which was developed previously on top of Prism-MW and in collaboration with Bosch Research and Technology Center engineers for their family of smart-spaces sensor network systems. We do not provide a detailed description of MIDAS and point the interested reader to [3]. MIDAS employs a heterogeneous domain specific architectural style, which incorporates characteristics from several other well-known styles, including SOA, Publish-Subscribe, and Pipe-and-Filter. This experience helped us make certain observations, which we summarize according to the four problems mentioned in Section 4 that have motivated this research:

- *High complexity*: Developing the patterns and implementing the extensions to Prism-MW to realize the patterns required a graduate student unfamiliar with Prism-MW an effort of approximately 1000 man-hour over a period of one year. The task required the developer to become familiar with 80% of the middleware's existing code. This experience shows that realizing such facilities requires thorough understanding of the middleware and poses a severe burden on the application developers. On the other hand, the middleware engineers (i.e., in our research, the graduate students that have developed this middleware), who are naturally most familiar with the middleware's implementation, can provide the same facilities rapidly and with only a few hours of programming.
- *Lack of reuse*: For the styles that were shared between MIDAS and EDS, we were able to reuse the same version of the middleware to consistently adapt both systems. This corroborates our assertion that since adaptation patterns for a given style are generally applicable to any software system built in that style, the realization of patterns in the middleware could be reused across different software systems.
- *High coupling*: Our experience with the evolution of MIDAS application over a period of six month, during which close to 20% of the 40KSLOC code base was changed, showed no impact on the middleware's adaptation logic. This was expected as the patterns were able to automatically determine the component dependency relationships based solely on the style, as opposed to the changes in the internal application logic. This experience shows that our approach provides a separation between the middleware's adaptation logic and the evolution of the application logic, as long as the evolution does not entail a change in the application's architectural style.
- *Identifying the component dependencies*: We have been able to develop an adaptation pattern for all the styles we have encountered so far, including C2, Client-Server, D3, Publish-Subscribe, MIDAS, and Pipe-and-Filter. Our experience has showed that different styles result in very different adaptation patterns. These patterns were all able to consistently identify the static component dependency relationships in both MIDAS and EDS system, and when required transition a component to the quiescence state. Finally, among the styles that we have looked at so far, we observed that those with a rich set of rules and constraints result in more interesting and detailed patterns, which in turn allow for refined and less disruptive adaptation support in the middleware.

9 Related Work

This work relates to four different areas of research: (1) patterns; (2) dynamic software adaptation; (3) models at runtime; and (4) middleware technologies. We have categorized the related work according to these areas as follows.

Patterns. The object oriented programming community has been studying solutions to recurring problems in software development. They have abstracted these solutions in terms of lessons learned and experiences at programming language level. The result is expressed as Problem-Solution pairs in terms of design patterns [21]. After successful adoption of design patterns, patterns in different parts of the software development lifecycle were also studied [22–27]. The main benefit of a pattern is the fact that it can represent a set of complex design and implementation choices as a single reusable abstraction [6,28]. Related to our adaptation patterns are the following approaches:

- Gomaa and Hussein [29] suggest the development of reconfiguration patterns for software product lines. Their approach does not consider dependent transactions and their implications. The reconfiguration patterns are also not realized on top an implementation platform.
- Ramirez and Cheng [30] introduced a set of design patterns for building dynamic adaptive software systems. The purpose of patterns proposed in their work is different from ours. Their patterns are at the level of software design, and are aimed to facilitate the design and construction of a self-adaptive software system. On the other hand, our patterns deal with ensuring the consistency of software during adaptation.
- Tyson et al. [31] developed a pattern, called *component pattern,* in which peer-to-peer overlay networks can be effectively developed for the purposes of adaptation. Their pattern can be encoded as a style in Prism-MW for building peer-to-peer overlay networks, and the style can be used to derive the adaptation patterns similar to what we have discussed in this paper. To that end, we believe their work is complementary to the research presented in this paper.

Dynamic software adaptation. Existing research on dynamic software adaptation falls into two categories: design concepts and implementation mechanisms. We elaborate on the most related works from both categories below. Design concepts provide the theoretical foundations for modeling and reasoning about dynamic changes in a software system. In Section 3 we provided an overview of a related research work [9] that has shaped the theoretical basis of our research. The following works are also related:

- Kramer and Magee [7] state that software architectures provide an appropriate level of abstraction for modeling and reasoning about dynamic adaptation. They define a three-layer model (component control, change management, and goal management) to address the challenges associated with the development of self-managed systems. Our work addresses some of the challenges associated with component control and change management layers.
- Garlan et al. [32] propose a methodology for architecture-based adaptation of software systems with a focus on the reusability issues. They recognize the importance of the knowledge expressed in architectural styles. However, they do not present an approach to codify this knowledge into reusable patterns. Moreover, their approach is neither realized in a middleware platform, nor does it ensure the system's consistency.

- Vandewoude et al. [10] propose *Tranquility* as a necessary condition for consistent adaptation of software systems. Tranquility builds on the notion of Quiescence proposed by Kramer and Magee, except it relaxes some of the constraints to achieve faster adaptation times. However, unlike Quiescence, Tranquility is not guaranteed to be reachable. Vandewoude et al. extended the Draco middleware [10] to provide support for *Tranquility.* However, Draco relies on software components to provide the middleware with not only a list of transactions they have already participated in the past, but also transactions they will participate in the future. This assumption breaks the black-box treatment of components, and is not practical, since it is not feasible for a third-party component to know a priori in which transactions it will participate. This is exactly the problem we have tried to address in our work.

Models at runtime. The fact that we maintain the architectural models of the system synchronized with the running code to avoid architectural drift and erosion is related to the work of models at runtime community [33]. The following approaches also have similar concepts to our work, but none deals with the notions of quiescence and functional consistency of the system during/after adaptation:

- Co-ev framework [34] keeps the architectural models of the system and its runtime implementation synchronized. This is crucial as the models are used to verify adaptations before they are applied to the system. Confirmed adaptations are then committed to both models and runtime implementation to keep them synchronized with one another. We achieve the same goal in Prism-MW, but in a different way. Prism-MW is an architectural middleware and provides implementation-level modules that correspond to the system's software architectural constructs. In other words, in Prism-MW, the model and code are indistinguishable from one another, and therefore, they can only be changed together, which avoids the problem of design erosion.
- Giese and Wagner [35] use triple graph grammar to synchronize models in different layers of abstraction. Triple graph grammar allows for bidirectional and non-destructive model transformation. This way any change in low-level models (e.g., implementation level) directly manifests itself in the high-level models as well. Since architectural models and running system are tightly coupled with each other in Prism-MW, any change would be applied on both the architecture and implementation of the system at the same time. In other words, the layers of abstraction in Prism-MW are constantly synchronized but not through model transformation.
- FIESTA framework [36] uses the state-of-the-art aspect oriented techniques to integrate new functionality into the architecture of the system. The notion of adaptation pattern discussed in this paper is different and complementary to this work, as it aims to provide functional consistency during adaptation, which is required no matter how the change is applied to the system.

Middleware. The implementation mechanisms deal with technologies and middleware solutions intended for adaptive and dynamic settings. None of the existing implementation technologies that we are aware of provides support for consistent adaptation of a software system based on its stylistic characteristics:

- C2 framework [8] is an architectural middleware intended for the development of software systems according to C2 style. It provides rudimentary support for adaptation in terms of simply adding and removing components, but does not ensure consistency of the system during such changes.

- MobiPADS [37] is a reflective middleware that supports active deployment of augmented services for mobile computing. MobiPADS supports dynamic adaptation in order to provide flexible configuration of resources and optimize the performance of mobile applications.
- OpenCom [38] is component-based systems-building technology for building low-level system software. It has a simple, efficient, minimal kernel, which provides a set of extension mechanisms. A subset of these extensions is the reflective extension, which provides generic support for inspecting, adapting and extending the structure and behaviour of systems at run-time. In the follow up work [31,39], the OpenCom technology has been used to build adaptive software systems.
- Fractal Component Model [40] is a hierarchical and reflective model for the development of component based software. A key concept in this model is *membrane*. Similar to *Component* abstract class in Prism-MW (recall Fig. 9), membrane plays the role of a wrapper for a Fractal component, which adds facilities beyond the functional behavior of the wrapped component. Similar to *Component* in Prism-MW, which is customized through extensions, membrane can also be customized. The membrane can contain several forms of controllers each of which provides different reflective features. *Life-Cycle controller* facilitates reconfiguration; for instance, it provides support for starting and stopping a component. However, these controls are at the component control layer [7], while the quiescence adaptation patterns are at the change management layer [7], which is one level of abstraction higher. An extension to Fractal Component Model is [41], which takes into account the integrity of adaptation. To that end, the reconfiguration is certified if only pre/post conditions on the new configuration are satisfied. The integrity constraints are very similar to constraints of a given style in Prism-MW, as they determine what is a valid architecture. The adaptation patterns described in this paper are different, as they specify the required steps for driving a software component in a given style to quiescence.
- ArchJava [42] is an extension to Java that unifies software architecture with implementation, ensuring that the implementation conforms to the architectural constraints. ArchJava currently has several limitations that would likely limit its applicability: Communication between ArchJava components is achieved solely via method calls, ArchJava is only applicable to applications running in a single address space, it is currently limited to Java and its efficiency has not yet been assessed.
- Aura [43] is an architectural style and supporting middleware for ubiquitous computing applications with a special focus on user mobility, context awareness, and context switching. Similarly to Prism-MW, Aura has explicit, first-class connectors. Aura also provides a set of components that perform management of tasks, environment monitoring, context observing, and service supplying. This suggests that the Aura style could be successfully supported using Prism-MW augmented with a set of Aura-specific extensions. This would eliminate the need for performing optimizations of Aura's current implementation support, which has to date only been tested on traditional, desktop platforms.
- MUSIC [44], which is based on MADAM [45,46], is an open-sources middleware for ubiquitous, mobile and context-aware systems. The goal of MUSIC is to allow software deployed on mobile devices to be adapted based on the contextual information, non-functional requirements (Quality of Service), and availability of services. Adaptations supported in MUSIC consist of reconfiguration of the system by changing its parameters or selecting alternative components. In a recent, MUSIC has been extended to support Service Oriented Architectures. MUSIC allows a service

and a local component to realize the same component type and then the application can adapt by selecting which one to use at runtime. Unlike our work, the consistency of system during adaptation has not been the focus of MUSIC.

10 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. By codifying these patterns in a style-aware middleware, we have been able to provide significantly more advanced adaptation capabilities than that is currently offered by other platforms.

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 realize our approach on middleware platforms widely used in the industry (e.g., J2EE Java Message Service, Corba, and Enterprise Service Bus). Given that the level of architectural support provided by these middlewares varies, they may have to be adjusted and extended appropriately to accommodate the work presented in this paper. Finally, our adaptation patterns currently cannot handle faults during the process of placing a component in quiescence. Enhancing the patterns to handle such cases would be another interesting avenue of future research.

11 Acknowledgments

This work is partially supported by grants CCF-0820060 and CCF-1217503 from the National Science Foundation and grant N11AP20025 from Defense Advanced Research Projects Agency. We would like to thank Ashirvad Naik and Michael Thimblin for their work on some of the preliminary adaptation patterns. We also would like to thank Hamid Bagheri for the constructive discussions that we had about this research.

12 References

[1] J.O. Kephart, D.M. Chess, The Vision of Autonomic Computing, IEEE Computer. 36 (2003) 41-50.

[2] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, et al., Software Engineering for Self-Adaptive Systems: A Research Roadmap, in: Software Engineering for Self-Adaptive Systems, LNCS Hot Topics, 2009: pp. 1-26.

[3] S. Malek, C. Seo, S. Ravula, B. Petrus, N. Medvidovic, Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support, in: Int'l Conf. on Software Engineering, Minneapolis, Minnesota, 2007: pp. 591-601.

[4] D.E. Perry, A.L. Wolf, Foundations for the study of software architecture, Softw. Eng. Notes. 17 (1992) 40-52.

[5] M. Shaw, D. Garlan, Software architecture: perspectives on an emerging discipline, Prentice-Hall, Inc., 1996.

[6] R. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Univ. of California Irvine, 2000.

[7] J. Kramer, J. Magee, Self-Managed Systems: an Architectural Challenge, in: Int'l Conf. on Software Engineering, Minneapolis, Minnesota, 2007: pp. 259-268.

[8] P. Oreizy, N. Medvidovic, R.N. Taylor, Architecture-based runtime software evolution, in: Int'l Conf. on Software Engineering, Kyoto, Japan, 1998: pp. 177-186.

[9] J. Kramer, J. Magee, The Evolving Philosophers Problem: Dynamic Change Management, IEEE Trans. Softw. Eng. 16 (1990) 1293-1306.

[10] Y. Vandewoude, P. Ebraert, Y. Berbers, T. D'Hondt, Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates, IEEE Trans. Softw. Eng. 33 (2007) 856-868.

[11] N. Esfahani, S. Malek, On the Role of Architectural Styles in Improving the Adaptation Support of Middleware Platforms, in: European Conf. on Software Architecture, Copenhagen, Denmark, 2010: pp. 433-440.

[12] S. Malek, M. Mikic-Rakic, N. Medvidovic, A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems, IEEE Trans. Softw. Eng. 31 (2005) 256-272.

[13] R.N. Taylor, N. Medvidovic, K.M. Anderson, J. E. James Whitehead, J.E. Robbins, A component- and message-based architectural style for GUI software, in: Int'l Conf on Software Engineering, Seattle, Washington, 1995: pp. 295-304.

[14] R.N. Taylor, N. Medvidovic, E.M. Dashofy, Software Architecture: Foundations, Theory, and Practice, Wiley, 2009.

[15] B. Li, G. Zeng, Z. Lin, A domain specific software architecture style for CSCD system, SIGSOFT Softw. Eng. Notes. 24 (1999) 59-64.

[16] A. Agrawala, J. Krause, S. Vestal, Domain-specific software architectures for intelligent guidance, navigation and control, in: IEEE Symp. on Computer-Aided Control System Design, Napa, California, 1992: pp. 110–116.

[17] B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morignot, M. Balabanovic, A Domain-Specific Software Architecture for Adaptive Intelligent Systems, IEEE Trans. Softw. Eng. 21 (1995) 288-301.

[18] W. Tracz, DSSA (Domain-Specific Software Architecture): pedagogical example, SIGSOFT Softw. Eng. Notes. 20 (1995) 49-62.

[19] R.N. Taylor, A. van der Hoek, Software Design and Architecture The once and future focus of software engineering, in: Int'l Conf on Software Engineering, Minneapolis, Minnesota, 2007: pp. 226-243.

[20] S. Fritsch, A. Senart, D.C. Schmidt, S. Clarke, Time-bounded adaptation for automotive system software, in: Int'l Conf on Software Engineering, Leipzig, Germany, 2008: pp. 571–580.

[21] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns, Addison-Wesley Reading, MA, 2002.

[22] F. Buschmann, K. Henney, D.C. Schmidt, Pattern-oriented software architecture: On patterns and pattern languages, John Wiley & Sons Inc, 2007.

[23] M. Fowler, Analysis Patterns: Reusable Object Models, Addison-Wesley Professional, 1996.

[24] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002.

[25] S.W. Ambler, Process Patterns: Building Large-Scale Systems Using Object Technology, Cambridge University Press, 1998.

[26] S.W. Ambler, More Process Patterns: Delivering Large-Scale Systems Using Object Technology, Cambridge University Press, 1999.

[27] M. Shaw, Some patterns for software architectures, Pattern Languages of Program Design. 2 (1996) 255–269.

[28] R.T. Monroe, A. Kompanek, R. Melton, D. Garlan, Architectural Styles, Design Patterns, and Objects, IEEE Softw. 14 (1997) 43-52.

[29] H. Gomaa, M. Hussein, Software reconfiguration patterns for dynamic evolution of software architectures, in: Working IEEE/IFIP Conf. on Software Architecture, Oslo, Norway, 2004: pp. 79-88.

[30] A.J. Ramirez, B.H.C. Cheng, Design patterns for developing dynamically adaptive systems, in: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, ACM, Cape Town, South Africa, 2010: pp. 49-58.

[31] G. Tyson, P. Grace, A. Mauthe, G. Blair, S. Kaune, A Reflective Middleware to Support Peer-to-Peer Overlay Adaptation, in: Int'l Conf on on Distributed Applications and Interoperable Systems, Springer-Verlag, Lisbon, Portugal, 2009: pp. 30-43.

[32] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, P. Steenkiste, Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure, IEEE Computer. 37 (2004) 46-54.

[33] Models@run.time, http://www.comp.lancs.ac.uk/~bencomo/MRT/. (n.d.).

[34] P. Sriplakich, G. Waignier, A.-F. Le Meur, Enabling Dynamic Co-evolution of Models and Runtime Applications, in: Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference, Turku, Finland, 2008: pp. 1116–1121.

[35] H. Giese, R. Wagner, Incremental Model Synchronization with Triple Graph Grammars, in: International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), 2006: p. 543--557.

[36] G. Waignier, A.-F. Meur, L. Duchien, FIESTA: A Generic Framework for Integrating New Functionalities into Software Architectures, in: European Conf. on Software Architecture, Springer Berlin Heidelberg, Madrid, Spain, 2007: pp. 76-91.

[37] A.T.S. Chan, S.-N. Chuang, MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing, IEEE Trans. Softw. Eng. 29 (2003) 1072-1085.

[38] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, et al., A generic component model for building systems software, ACM Trans. Comput. Syst. 26 (2008) 1-42.

[39] G. Coulson, P. Grace, G. Blair, L. Mathy, D. Duce, C. Cooper, et al., Towards A Component-Based Middleware Framework for Configurable and Reconfigurable Grid Computing, in: Int'l Wrkshp. on Enabling Technologies: Infrastructure for Collaborative Enterprises, IEEE Computer Society, Modena, Italy, 2004: pp. 291-296.

[40] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems, Softw. Pract. Exper. 36 (2006) 1257–1284.

[41] M. Léger, T. Ledoux, T. Coupaye, Reliable dynamic reconfigurations in a reflective component model, in: Int'l Symp. on Component Based Software Engineering, Prague, Czech Republic, 2010: pp. 74–92.

[42] J. Aldrich, C. Chambers, D. Notkin, ArchJava: connecting software architecture to implementation, in: Int'l Conf on Software Engineering, Orlando, Florida, 2002: pp. 187-197.

[43] J.P. Sousa, D. Garlan, Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments, in: Working IEEE/IFIP Conference on Software Architecture, Deventer, The Netherlands, 2002: pp. 29-43.

[44] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, et al., MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments, in: Software Engineering for Self-Adaptive Systems, LNCS, 2009: pp. 164–182.

[45] M. Mikalsen, N. Paspallis, J. Floch, E. Stav, G.A. Papadopoulos, A. Chimaris, Distributed context management in a mobility and adaptation enabling middleware (MADAM), in: Proceedings of the 2006 ACM Symposium on Applied Computing, ACM, Dijon, France, 2006: pp. 733-734.

[46] K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, et al., A comprehensive solution for application-level adaptation, Softw. Pract. Exper. 39 (2009) 385–422.