Evolution Self-Adaptive Services using Planning-Based Reflective Middleware

Frank Eliassen¹, Eli Gjørven², Viktor S. Wold Eide³, and Jørgen Andreas Michelsen¹
¹University of Oslo
P.O. Box 1080 Blindern
N-0314 Oslo, Norway
frank, jorgenam@ifi.uio.no
²Simula Research Laboratory
P.O. Box 134
N-1325 Lysaker, Norway
elij, viktor@simula.no

ABSTRACT

Self-adaptive systems often use a middleware-based approach where adaptation mechanisms and policies are separated and externalized from the application code. Such separation facilitates the independent application of discovery, utility-based and context-aware evaluation, and selection of alternative implementations of a given service.

Here we argue that the QuA middleware is also able to support certain forms of evolution of adaptive systems. Since in QuA new implementation alternatives or updated versions of software are automatically discovered and considered during service planning, evolution both during run time and load time is supported. Experimental results from evolving a state-of-the-art adaptive media streaming application using our middleware are also presented.

1. INTRODUCTION

Systems are increasingly expected to adapt themselves to changing requirements and environments with minimum human interaction. Self-adaptation includes the ability to self-configure automatically and seamlessly according to higher-level policies specified by users. More often systems are also expected to proactively seek to upgrade themselves by finding and applying new or updated versions of software. This poses great challenges to application and system developers. As software evolve over time, existing adaptive behavior may not be applicable to new versions, or it may cause inconsistencies or failures in the running system.

Recent works on self-adaptive systems use a middleware-based approach where adaptation mechanisms and policies are separated and externalized from the application code [6]. However, complete separation is not possible, since developers of adaptation systems must make assumptions about the application subject to adaptation such as its architecture. As applications evolve, these assumptions may no longer be true.

The above suggests there is a need for a comprehensive methodology for the development of adaptive systems that are not only open for evolution, but also to the dynamic discovery, analysis and use of both initial and updated versions of software by self-managing systems. Our work may be seen as a contribution towards such a methodology.

In our work we keep a strict separation between a system’s behavior and its implementation. Accordingly we define evolution as “the process of developing services that have their origin in pre-existing services, and are created by modifications of preexisting services, but are distinguishable by their behavior from their preceding services”.

We separate two types of evolution. An evolution process where a service $S_{\text{orig}}$ is modified into a new service $S_{\text{new}}$ is substitutional if the type $T_{\text{new}}$ of $S_{\text{new}}$ conforms with the type $T_{\text{orig}}$ of $S_{\text{orig}}$. Otherwise, we call the evolution process non-substitutional. Our notion of substitutability is similar to behavioral subtyping for components in [8] a major difference being that our approach also covers evolution of QoS properties and QoS management functions associated to services.

In the QuA project [10], we are investigating planning-based adaptation, which means that applications are specified by their behavior, and planned, instantiated and maintained by the middleware in such a way that the behavioral requirements are satisfied throughout the application life-time. We characterize an application by the services it provides to its users. We define a service as an entity capable of computing a subset of output events, in response to a set of input events to a composition of components [11]. The subset of output events can be seen as specifying the “work done” by the service while the set of inputs as specifying the “work request”. A service may be as simple as to calculate the sum of two numbers, or more complex, such as delivering video and audio content to users during a video conference.

Central to this middleware is mirror-based service reflection, which supports introspection and intercession on a service through all the phases of its life-cycle, including pre-runtime [7]. Each service is represented by a service mirror, which is an object reflecting the service behavior and implementation. Mirror-based reflection supports the planning framework in reasoning about services both before and during their execution. Service mirrors can be advertised to and obtained from a middleware broker service.

This paper describes how the principles of the QuA middleware facilitates the continuous process of substitutable evolution of adapt-
able services. Through mirror-based reflection and planning, adaptation and evolution mutually contributes to each others success; Evolution by providing adaptation with new and improved adaptation alternatives and adaptation by ensuring that new adaptation alternatives of an adaptable service are integrated into the system in the best possible way. We also describe some initial experiments from evolving a state-of-the-art adaptive media streaming application using our middleware. Our experience with this application suggests that planning-based middleware constitutes a highly promising approach to integrating evolution into self-adaptive systems.

The rest of this paper is organized as follows. In Section 2 we describe some related research. In Section 3 we describe the main principles of mirror-based reflective middleware and the use of service mirrors to reflect the evolution of a service. In Section 4 we address the principles underlying planning-based middleware and how service planning supports both service adaptation and evolution. Section 5 outlines the implementation of the QuA middleware. Some initial validation of our approach is addressed in Section 6 that describes the evolution of a state-of-the-art adaptive media streaming application using our middleware. Section 7 offers conclusions and outlook to further work.

2. RELATED WORK

The middleware presented in this paper has similarities with architecture-based approaches to self-adaptation, where the system is modelled as a composition of components and their interconnection [9]. For example, the Rainbow project[6] introduces a system architectural model in its system model in order to evaluate and adapt a running system. Rainbow extends architecture-based adaptation by adopting architectural styles, which characterizes a family of systems related by shared structural and semantic properties. Such approaches focus on reuse of adaptation properties, mechanisms and strategies between systems, based on architectural commonalities that can be expressed in shared architectural styles. However, we focus on evolution of the adaptive behavior of components, how to represent this knowledge, and how it can be exploited automatically by the middleware and its adaptive mechanisms.

The product families community uses similar ideas to produce system variants with different properties based on a common architecture [13]. Our approach can be seen as a mechanism for dynamically generating a product suitable for a particular context. A major difference is that different variants of the product are discovered, evaluated and selected during runtime by the middleware in response to context changes, a process we call planning.

In the area of run-time dynamic evolution, several research projects focus on the problem of safe live updates, i.e. modifications that are applied to a program at run-time. For example in the SEESCOA project, a component-based run-time environment supporting run-time adaptation of stateful components participating in embedded applications[12]. Applying such mechanisms for changing from one software version to another assumes that the decision to make the change has already been made. Our planning-based middleware also provides a solution for this decision-making problem.

Reflective middleware such as OpenCOM[1] naturally supports evolution, since reflective facilities may selectively expose aspects of the system, without exposing its implementation. In OpenCOM, each component supports a reflective API that provides access to both structural and behavioral meta-information. Consequently, reflection is tightly coupled with running instances, and cannot be used, for example, to reflect on the architecture of a service that has not yet been instantiated. In the domain of reflective programing languages, object mirrors have been proposed as an improvement of traditional reflective facilities such as Java reflection and the Smalltalk reflective features [2]. The meta-level functionality is implemented separately from base-level objects using meta-objects called mirrors. We exploit mirror-based reflection in order to enable reflection on services that have not yet been instantiated, which is vital during planning.

3. MIRROR-BASED MIDDLEWARE

In our approach to middleware support for self-adaptation, services are specified by their behavior (service type and quality requirements), and then planned, instantiated, and maintained by middleware services in such a way that the behavioral requirements are satisfied throughout the service life-time. The middleware reflective API provides access to a service's meta level both before and during the running phase of the service. A service's meta-level reflects what is known about the behavior and implementation as well as the instantiation of the service.

3.1 Mirror-Based Reflection

Several reflective component models, such as OpenCOM [4] and Fractal [3], adopt the principle that components provide certain reflective interfaces. In contrast, we adopt the principles of mirror-based reflection where access to the meta-level is provided as a middleware service, like traditional middleware services such as service instantiation and binding.

The service mirror provides reflection on both static and dynamic aspects of a service. Conceptually, a service mirror consists of a Behavior part, an Architecture part and an Interfaces part. Figure 1 illustrates these three parts.

The Behavior part reflects the type of the service. This includes the functional behavior of the service in terms of its provided interfaces, and the non-functional (qualitative) behavior of the service in terms of a set of QoS dimensions. A service behavior definition may be refined by adding qualitative requirements and preference statements on the QoS dimensions in the form of a utility function. This is illustrated in the figure as the Quality part of the behavior part. Utility functions allow one to specify the degree of desirability of different adaptation alternatives of an adaptable service. Utility is defined as a function of the achieved QoS of an implementation alternative (as described by a mirror) for a given state of the operational environment. Adaptation policies specified as utility of service level attributes are robust to software evolution in the sense that predominately such software changes can be absorbed without modification to the adaptation policy.

The Architecture part contains the name of a so called Blueprint that encapsulates a service implementation conforming to the func-
tional behavior specified by the service mirror. Additionally, it contains a service mirror representing a platform, or virtual machine, able to interpret this blueprint and instantiate services from it. For example, if the blueprint contains one or more Java classes, the platform must run on a Java virtual machine.

The architecture part also associates the architecture with its QoS predictors, if any have been defined. The predictor functions provide, as the name hints, predicted QoS-values according to the QoS-dimensions defined by the service type of the mirror, and facilitate automated reasoning about the non-functional properties of the service ahead of run-time. Finally, the architecture is associated with its context dependencies, if any. A service mirror whose context dependencies are not satisfied by the current context, can be discarded.

As indicated in the figure, a service mirror may be a recursive tree structure where each mirror in the tree may define zero or more dependencies to other services, also represented by mirrors. A QoS-predictor may depend on predictor functions of these dependencies. Hence, automatic reasoning about the quality delivered by the top-level service is achieved by recursively parsing the QoS-predictions.

The interfaces part of a service mirror contains references to the run-time objects that provide the interfaces specified by the service type.

This service mirror facilitates run-time mapping between service types, alternative service implementations and running service instances, which can be used by planning and adaptation mechanisms as described in the next sections.

### 3.2 Evolving Services Using Mirrors

The planning framework of our middleware manages substitutional evolution as defined in Section 1. As the separation of substitutional and non-substitutional evolution is based on type conformance, we assume type conformance based on the substitutability of both functional and non-functional behavior. For the functional behavior part of a service type, a combination of the interface subtype relationship and the behavioral subtype relationship for components suggested in [8], would satisfy the requirement for substitutability. For the non-functional part of the behavior specification of a service type, we require that the subtype has at least the QoS dimensions defined for the supertype. When a subtype is used in a situation where its supertype is expected, the additional functional operations and QoS dimensions that the subtype potentially may define, will not be used. New QoS dimensions, however, may indirectly still have an influence on adaptation management decisions on the supertype service, through dependencies between new and old QoS dimensions. We do not assume any particular type checking system, but as will be discussed in Section 4 type conformance must be explicitly declared to the service broker in order to be discovered by the middleware services. This is the responsibility of the developer.

Based on the above requirements to type conformance, we discuss below how the middleware presented in this paper can manage the different types of service evolution.

First, substitutional evolution can be achieved by service evolution that adds interfaces to an existing type, or extends an interface type of the service type with additional operations such that the operations of the interface supertype is syntactically and behaviorally preserved. This may be useful if extending the functional behavior of a service with new interfaces or operations, while still supporting the behavior of the existing service type.

Second, substitutional evolution can also be achieved by service evolution that adds QoS dimensions to an existing service type. This may be useful when introducing behavior that functionally conforms to the existing type, but that can be more precisely described and specified if the set of QoS dimensions are extended. Naturally, substitutional evolution that adds both interfaces and QoS dimensions are also supported as a combination of the two.

Third, we can achieve substitutional evolution by modifying an existing service implementation so that the resulting service implements the same type as the preceding service, but has a different non-functional behavior.

A variant of the third type of evolution is to only redefine the QoS predictors of an existing service implementation. This can be beneficial if experience gained over time when using the existing service, gives a better understanding of the service’s non-functional behavior.

Finally, non-substitutional evolution means creating a new service using an existing one, and then advertising it with a type that is not (declared) conformant with the type of the preceding service. Our middleware allows such evolution to take place, but at the middleware level (i.e., in the broker) no relationship between the evolved service and the preceding service will exist. We believe non-substitutional evolution can be supported by planning-based middleware through some suitable extensions of the planning framework, but the details of this is future work.

### 4. Planning-Based Middleware

The task of service planning, is to plan the initial configuration or the reconfiguration of a service. The planner is responsible for evaluating alternative service mirrors in order to find and select the service implementation with highest utility that satisfies the functional and qualitative specifications of a service request.

The service planner uses a broker to find alternative service mirrors that specify service types conforming to the type specified by the input mirror. While evaluating the alternative mirrors, the service planner seeks to recursively resolve any specified dependencies. We say that a service mirror is fully resolved if it has an architecture with no dependencies, or if it has an architecture whose dependency mirrors are all fully resolved. Otherwise, we say that the mirror is partially resolved. The input to service planning is a mirror that is partially resolved.

When a mirror is fully resolved, the QoS provided by the mirrored service may be recursively calculated using QoS predictors, and the utility can be calculated using the utility function in the behavior specification of the root mirror of the mirror tree. Finally, the planner ranks the alternative fully resolved mirrors based on their utility. The result of service planning is normally the fully resolved mirror ranked highest by the planner.

Service planning is triggered during initial service instantiation as a result of a client requesting a service conforming to a particular behavior, or later during service runtime when the need for service adaptation is detected. The latter is caused by context changes such as varying resource availability or user needs.

Planning-based adaptation means to go back to the planning phase and reconsider the architecture that was selected for the currently running service, or one or several of its resolved dependencies, a process we call replanning. This leads to the following generic adaptation algorithm:

1. Get (a copy of) the service mirror that represents the service to be adapted
2. Unresolve (some or all of) the dependencies of the mirror (policy dependent) and perform service replanning on the now partially resolved mirror
3. If the result of service replanning imply changes to the running service (policy dependent), determine these changes by comparing the service mirror of the running service and the new service mirror that resulted from the step 2.

4. Bring the running service into safe state and enforce the changes (if any) on the base-level objects

Since a service mirror may be advertised to the middleware at any time, also during service run time, the generic adaptation algorithm defined above inherently supports the dynamic discovery and use of new versions at run-time, also while performing adaptations. In order for an implementation to be considered as substitutable with a running service, the type of the new version must conform to the type of the running service.

5. THE QUA IMPLEMENTATION

QuA has been designed as a small research component architecture that can be deployed to a wide range of computers. The QuA architecture is based on planning and mirror reflection. It provides a set of mirror operations (plan, instantiate, etc.) that can be used to instantiate and maintain services, and the method `reflectOn` which returns the service mirror of a running service. We note that the `reflectOn` method operation implements step 1 in the generic adaptation algorithm in Section 4.

An instance of the QuA platform consists of a small core that may be extended with specialized, domain specific services (figure 2). A QuA capsule represents the local runtime environment that a QuA platform instance depends on. A capsule hosts one or more repositories, where blueprints referred to by mirrors can be stored and obtained. The QuA platform associates each client thread with a service context that contains references to the capsule that provides the service’s local runtime environment, and to the planner and broker that is used to plan and replan that service.

The Service Planner component implements service planning as outlined in Section 4. It is invoked by the QuA component as a result of a call to one of its mirror operations (e.g., `plan`) and uses the Broker component to find alternative services mirrors to resolve dependencies. The Broker will return all service mirrors that specify a type conforming to the input argument type and that satisfies additional property constraints (if any).

The task of the Adaptation Manager is to detect context changes that require adaptation and generally perform step 2 of the generic planning algorithm. The Adaptation Manager requests the Configurator to adapt the running service providing the service mirror of the running service and the new mirror that is the target for the reconfiguration, as arguments. The Adaptation Manager is a pluggable service that allows different adaptation management algorithms to be explored within the same architecture. Our current implementation of the Adaptation Manager always requests replanning of the complete running service upon detecting any context change, and always initiates a reconfiguration as a result of replanning. In our current work we are addressing more advanced and scalable solutions.

The Adaptation Manager depends on the availability of a context component that is responsible for gathering, storing, aggregating, reasoning about, and providing context information. However, in this work focusing on service planning, we use a simplified approach to context management using a context simulator in our experiments.

The responsibility of the Configurator is to perform step 3 and 4 in the generic adaptation algorithm using the QuA mirror operations to instantiate and bind components, and internal logic to transfer state between components that are interleaved in the service architecture. For the latter we assume all components implement a configuration interface that allows the Configurator to bring components into safe state, get and set state and resume activity.

The current distribution model is based on centralized planning of distributed services. Distributed services are implemented by instructing remote capsules to host components, and to create bindings between these remote components, or between remote components and local components. Since a mirror refers to the platform on which the service it represents should run, it may refer to a particular local or remote capsule. In the latter case, a binding to the remote capsule is created during service instantiation, and the capsule is instructed to instantiate the component and return a remote reference to the instance. Like other components, the platform is represented by a service mirror. Any capsule can be made available to other peers by advertising a binding service that can be used to create a remote binding to the capsule.

6. EVOLUTION EXPERIMENT

The application domain of real-time media streaming is challenging and hence a reasonable choice for validating our middleware. The application, the Personal Media Service (PMS) can be viewed as an in-house personal proxy service for delivery of multimedia content. Any media content accessible from home is by the PMS made accessible from anywhere, assuming Internet connectivity. A major task of the PMS is to adapt the content of the media stream to the capabilities of the client device and the Quality of Service dimensions defined by the platform.

The PMS is implemented and evolved by exploiting the support for substitutional evolution by the QuA middleware. In the following we provide details of this evolution story including how the QuA middleware features can be used to automatically manage the software evolution process.

6.1 The PMS Live Streaming Version

The original PMS application was developed to support a user watching a live video while on the move. Being self-adaptive, it was designed to continuously monitor variations in network bandwidth and adapt itself in order to maintain the best possible user experience. This version was based on existing software for multidimensional scalable video streaming [5].

The original PMS offers a service type `WatchTV`. The functional behavior of `WatchTV` is to stream live video content to a client device. The QoS dimensions defined by the `WatchTV` type are `temporal-`, `luminance-`, and `chrominance quality`, denoted by `t, y, `.
Table 1: Dimensional utility functions

<table>
<thead>
<tr>
<th>Layers</th>
<th>User 1</th>
<th>User 2</th>
<th>User 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td>0.9</td>
<td>0.7</td>
<td>0.4</td>
</tr>
<tr>
<td>+2</td>
<td>0.8</td>
<td>0.6</td>
<td>0.3</td>
</tr>
<tr>
<td>+1</td>
<td>0.7</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>base</td>
<td>0.6</td>
<td>0.4</td>
<td>0.1</td>
</tr>
<tr>
<td>null</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Figure 3: PMS live streaming version

![Diagram of PMS live streaming](image)

Table 2: Utility function for combinations of temporal displacement \((d)\) in seconds and time scale ratio \((r)\), denoted \(K_{dc}(d, r)\)

<table>
<thead>
<tr>
<th>(d)</th>
<th>User 1</th>
<th>User 2</th>
<th>User 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt; 0 - 10&gt;</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>(&lt; 10 - 60&gt;</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>(&lt; 60 - )</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Figure 4: PMS time-shifted version, storage configuration

![Diagram of PMS time-shifted version](image)

and \(c\) respectively. The video signal is encoded in a number of layers. Scalability in a video quality dimension is achieved by selecting the number of layers to receive. A base layer provides the lowest quality. Several enhancement layers, each building on the previous, allows for fine grained selectivity of the quality received and the bandwidth needed to carry the video data.

The overall utility for the WatchTV type is defined as a weighted sum of dimensional utility functions. A dimensional utility function measures user satisfaction in one of the QoS-dimension. The weights of the overall utility functions correspond to the relative level of importance of each QoS-dimension as preferred by the user.

Table 1 illustrates the dimensional utility functions for three different users, defined as a set of coefficient values, each specifying the utility value for a quality layer of a QoS-dimension. The overall utility is given by equation 1.

\[
U(t, y, c) = W_K(t) + W_K(y) + W_K(c)
\]  

(1)

The architecture of the PMS is illustrated in Figure 3. The application is published to the middleware as a number of mirrors. There is one mirror for the WatchTV type which contains the architectural composition shown in Figure 3. For each component type in the composition one mirror is published, except for the type TransConfig for which it is published as many mirrors as there are different quality level configurations of the video stream.

A TransConfig component is used by a transcoder component (with type Trans) to configure the video quality level to be used in the transcoding. Only the mirror with type WatchTV has a QoS-predictor. This QoS prediction depends on the video quality level of the TransConfig component. The context dependency of each mirror with type TransConfig is based on measured bandwidth requirements and expresses the minimum bandwidth needed to support the corresponding video quality.

The contextual changes which trigger service (re)planning are changes in available network bandwidth and user preferences.

6.2 The Time-shifted PMS Version

In this updated version the PMS was extended to cover the case when the available bandwidth is insufficient for streaming even the lowest quality acceptable to the user. New adaptive behavior was added, causing the PMS to pause until the network again becomes usable and then resume. During pause the frames are buffered in order to resume streaming once the connection is reestablished. Such buffering introduces delay, a time-shift. Once the network becomes accessible, the delay is steadily reduced by streaming the video slightly faster from the buffer compared to the buffer arrival rate. Such time scaling results in a higher frame rate at the client device while catching up with the stream arriving at the PMS.

This enhancement of the PMS corresponds to the first and second type of evolution described in Section 3. In this case, the WatchTV service type is extended to a new service type WatchTV-TS with two additional QoS dimensions, temporal displacement and time scale ratio. The temporal displacement dimension represents the number of seconds introduced by time-shifting. The time scale ratio is the speed of the presented video relative to the speed of the original stream. Different users may have different preferences regarding this newly introduced behavior. To account for this, the utility function for the PMS live streaming version, equation 1, was extended to take into account these two additional QoS dimensions.

The new utility function is given by equation 2, where \(d\) denotes temporal displacement and \(r\) denotes time scale ratio.

\[
U(t, y, c, d, r) = W_K(t) + W_K(y) + W_K(c) + W_K(d, r)
\]  

(2)

Examples of different user preferences for the added QoS dimensions are illustrated in Table 2. The first user prefers no time scaling in the live streaming case and speedup in the time-shift case. In addition to the live-stream architecture shown in Figure 3, the time-shift version of the PMS introduced two additional architectural alternatives, depicted in Figures 4 and 5: (1) redirecting the stream to a buffer instead of the client device, and (2) streaming from the buffer to the client device.

In order to make the new service available to the middleware, mirrors for the three alternative architectures need to be announced all with the new type, WatchTV-TS, and correspondingly new QoS-predictors and utility function that conform with the extended QoS model. As can be seen from the figure, some new component types were needed in order to evolve the PMS to include such time-shift capability. One service mirror for each of these is also announced to the middleware except for the RetrieverConfig type for which there are as many mirrors as there are different configurations of a Retriever component. RetrieverConfig components are used to configure Retriever components with the time scale ratio to use when retrieving video data from the buffer.

For the storage architecture, the QoS prediction is zero in all dimensions. For the time-shift architecture the QoS prediction depends on the video quality level of the TransConfig and RetrieverConfig components used in the configuration of the service. Additional context dependencies of this version of the PMS specify for any mirror of a time-shift configuration that the current temporal displacement must be greater than zero.

If the WatchTV-TS type is declared conformant with the WatchTV type, the mirrors with the WatchTV-TS type will potentially be selected when planning the WatchTV type. However, the additional
QoS dimensions of the WatchTV-TS type will not be taken into account when planning a WatchTV service since a utility function based on the WatchTV type will not take the additional QoS dimensions as arguments. After replanning, the users will functionally be using the WatchTV-TS service as a WatchTV service (functional substitutability). On the other hand the user may still experience the QoS behaviour of a WatchTV-TS service (time shifting).

Context changes that cause the Adaptation Manager to trigger service (re)planning, are changes in available network bandwidth, changes in user preferences, and whenever the buffer has been drained and hence the temporal displacement becomes zero.

6.3 The Enhanced Time-shifted PMS Version

While the time-shifted PMS only supports time scale ratios 1.0 (no scaling) and 1.2 (playing the video a bit faster), this enhanced version supports more time-shift configurations, some having time scale ratio less than 1.0 and others greater than 1.0. This allows streaming to continue in situations where bandwidth availability drops below what is necessary for live streaming, by using a time scale ratio less than 1.0. Some users may prefer such a slowdown for a shorter period of time, instead of a pause. As an example, the third user in Table 2 accepts both slowdown and speedup. This user also expresses that reducing the temporal displacement is preferred, and even more so as the temporal displacement increases.

The new version was realized by creating new mirrors for the type RetrievalConfig for the new configurations. Extended context dependencies were defined for the new mirrors to allow time-shift configurations with time scale ratio of 1.0 or less, even when temporal displacement is zero. Additionally, a new mirror for the type WatchTV-TS has to be announced with a QoS predictor that considers the now varying time scaling ratio. Otherwise this new mirror is identical to the mirror with the time-shift architecture described in the previous subsection.

7. CONCLUSIONS

This paper has described how the QuA planning-based reflective middleware facilitates the continuous process of evolution of adaptable services. Central to this approach is mirror-based service reflection, that supports the planning framework in reasoning about services both before and during their execution. Through mirror-based reflection and planning, adaptation and evolution mutually contribute to each others success. The middleware automatically handles substitutional evolution through its service planning mechanisms. Substitutional evolution means that implementations of a supertype and its subtype can always be used interchangeably in any architecture where the supertype is expected.

To validate our approach, we have described the initial implementation and subsequent evolution of a state-of-the-art adaptive media streaming application using our middleware. Our experience with developing and evolving this application strongly suggests that our planning-based middleware is able to adequately support substitutional evolution of self-adaptive services.

In the future we plan to gain more experience with both substitutional and non-substitutional evolution using the QuA middleware by undertaking a more systematic study of evolution support.

8. REFERENCES


