# Runtime Adaptation of Applications through Dynamic Recomposition of Components[⋆]

Arun Mukhija and Martin Glinz

Institut für Informatik
University of Zurich, CH-8057, Switzerland
{mukhija | glinz}@ifi.unizh.ch

**Abstract.** Software applications executing in highly dynamic environments are faced with the challenge of frequent and usually unpredictable changes in their execution environment. In order to cope with this challenge effectively, the applications need to adapt to these changes dynamically. CASA (Contract-based Adaptive Software Architecture) provides a framework for enabling dynamic adaptation of applications, in response to changes in their execution environment. One of the principle adaptation mechanisms employed in the CASA framework is dynamic recomposition of application components. In this paper, we discuss implementation issues related to the approach for dynamic recomposition of application components in CASA.

## 1 Introduction

A major challenge for software applications executing in highly dynamic environments (such as those in pervasive and ubiquitous computing scenarios) is the consistently changing execution environment of these applications. The changes in execution environment can be in the form of (i) changes in *contextual information* (user's location, identity of nearby objects or persons etc.), or (ii) changes in *resource availability* (bandwidth, battery power, connectivity etc.).

Contextual information refers to (purely) the *information* about the context of an application that may influence the service provided by the application (such as locational information, temporal information, atmospherical information etc.), in contrast to resources that form the *physical infrastructure* available to the application for providing this service (such as communication resources, data resources, computing resources etc.). A change in contextual information may present an opportunity for an application to adapt its behavior, in order to provide a more relevant service with respect to the changed contextual information. Similarly, a change in resource availability may require an application to change its resource consumption accordingly, necessitating an adaptation of the application's behavior.

[⋆] The work presented in this paper was supported (in part) by the National Center of Competence in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation.

*Proceedings of the 18th International Conference on Architecture of Computing Systems (ARCS 2005), Innsbruck, Austria, March 2005, pp. 124-138. ©2005 Springer.*

The existing approaches for dynamic adaptation of applications have focused mainly on runtime changes in resource availability. Most of these approaches try to adapt the lower-level services used by applications at the middleware level, and thereby influence the resource consumption due to these applications. Examples of adaptation of the lower-level services include modifying the quality or compression level of the data being transmitted over a communication channel in response to a change in the available bandwidth, changing the caching policy in response to a change in the available memory etc.

However, we argue that a dynamic change in application code should be provided as a means of application adaptation, *in addition* to the adaptation of the lower-level services at the middleware level, in order to effectively deal with the changes in execution environment.

This is because: (1) In response to a change in contextual information, a corresponding change in the functionality of an application is usually required, which typically requires a change in the application code. For example, if the contextual information related to a *Tourist Guide* application changes from *shopping mall* to *open-air cinema*, the application needs to provide relevant information about the weather conditions and show-timings, in place of the information about the availability of the items in the user's shopping list in the shopping mall. This kind of change in functionality requires a change in application code. (2) Even if small variations in resource availability can be handled by adapting the lower-level services, for large variations a change in application code is usually required. For example, consider a *Disaster Control* application transmitting the live video stream of an erupting volcano from a mobile node to a coordination center. For a small drop in the available bandwidth, an adaptive middleware may try to reduce the quality of the video transmitted, in order to save bandwidth. But for a significant drop in the bandwidth, it may be more apt for the application to send a textual description of the volcano (along with frequent images, if possible), rather than reducing the quality of the video beyond a threshold level. This kind of adaptation again requires a change in application code.

A runtime change in application code can be most primitively achieved by hardwiring the adaptation mechanism within an application (e.g. using programming constructs like if-else or switch-case etc.). However, this is a very tedious and limited solution to the problem. It makes the process of application development more complex, because the adaptation code is intertwined with the application code. Moreover, with this approach the adaptation policy cannot be changed during runtime, because of the hardwiring of the adaptation mechanism, posing a limitation to its usefulness for dynamic environments.

Recent approaches for dynamic weaving and unweaving of aspects, influencing the crosscutting functionality of an application such as security or persistence management, are a step in the right direction (the term *aspect* used in the sense of the aspect-oriented programming [6]). But, as the name indicates, these approaches are restricted to adapting the crosscutting functionality of an application. Whereas in practice, an adaptation of the core functionality of an application may be required *as well*, like in the examples of *Tourist Guide* and *Disaster Control* applications above.

Modern software applications are composed of components, where each component implements a subtask of the application (we will use the term *component* to refer to *application component* in this paper). In a component-based application development, the components encapsulate their implementation details, interact with each other only through their well-defined interfaces (using method calls), and generally follow the principle of *separation of concerns*. This makes it possible and convenient to alter the application code dynamically by recomposing the components at runtime.

CASA (Contract-based Adaptive Software Architecture) [1, 10] provides a framework for enabling dynamic adaptation of applications executing in dynamic environments. The CASA Runtime System monitors the changes in the execution environment of applications, and in case of significant changes carries out dynamic adaptation of applications. The adaptation policy of every application is defined in a so-called application contract. In order to meet adaptation needs of a broad and diverse set of applications, CASA supports the following adaptation mechanisms: dynamic change in lower-level services, dynamic weaving and unweaving of aspects, dynamic change in application attributes, and dynamic recomposition of components. The adaptation concerns are separated from the application, thereby reducing the complexity involved in developing adaptive applications. In this paper, we discuss implementation issues related to the approach for dynamic recomposition of components in CASA.

The rest of the paper is organized as follows. In Section 2, we give a brief overview of the CASA framework. In Section 3, we identify the key requirements for dynamic recomposition of components. In Section 4, we discuss implementation issues related to dynamic recomposition of components in CASA. In Section 5, we give an overview of related work. And in Section 6, we conclude the paper and indicate future direction of our work.

## 2 Overview of the CASA framework

Figure 1 shows the conceptual working of the CASA framework. Every computing node hosting adaptive applications is required to run an instance of the CASA Runtime System (CRS). The CRS is responsible for monitoring the changes in execution environment on behalf of these applications, and to adapt these applications as and when necessitated by a change in execution environment. The adaptation policy of every application is defined in a so-called application contract.

A three-step adaptation process is illustrated in Figure 1. Every time the CRS detects a change in the execution environment (step 1), it evaluates the application contracts of the running applications with respect to the changed state of the execution environment (step 2). If the CRS discovers a need for adapting certain applications, it carries out the adaptation of the affected applications, in accordance with the adaptation policies specified in the respective application contracts (step 3).

Application adaptation can be realized using one or more of the following adaptation mechanisms supported by CASA, depending on the adaptation needs of a specific application:

– *Dynamic change in lower-level services:* For a dynamic change in lower-level services used by applications, CASA can be integrated with any adaptive middleware for this purpose that supports external regulation of its adaptation strategy. Several reflection-based adaptive middleware fit in this category, such as Odyssey [11], QuO [15] etc.
– *Dynamic weaving and unweaving of aspects:* For dynamic weaving and unweaving of aspects, CASA relies on a flexible and efficient system for this purpose called PROSE [13].
– *Dynamic change in application attributes:* For a dynamic change in application attributes, the application needs to provide appropriate callback methods that can be called by the CRS at runtime.
– *Dynamic recomposition of components:* For dynamic recomposition of components, CASA follows an indigenous approach described in Section 4.

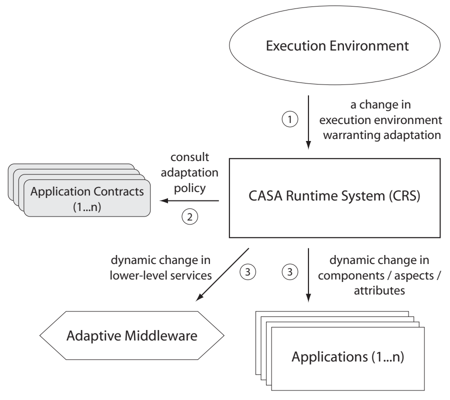
Execution Environment
1
a change in execution environment warranting adaptation
consult adaptation policy
Application Contracts (1...n)
2
CASA Runtime System (CRS)
dynamic change in lower-level services
3
3
dynamic change in components / aspects / attributes
Adaptive Middleware
Applications (1...n)

**Fig. 1.** Working of CASA

```
<app-contract name="App1">
   <context id="1">
      <params .../>
      <config id="1">
         <resources .../>
         <components>
            <binding handle="HC1" boundto="CdefA1"/>
            <binding handle="HC2" boundto="CdefG2"/>
            .
            .
         </components>
         <aspects .../>
         <callback .../>
         <llservices .../>
      </config>
      .
      .
      .
   </context>
   .
   .
   .
</app-contract>
```

**Fig. 2.** Application contract

An excerpt of an application contract is shown in Figure 2. The application contract is external to the application, and is specified using an XML-based language. This enables easy modification, extension, and customization of the adaptation policy at runtime. Moreover, it facilitates separating the adaptation concerns from the application.

The application contract is divided into <context> elements, where each <context> element represents a state of contextual information of interest to the application (the parameters characterizing this state are specified within

<params> element). Each <context> element in turn contains a list of alternative configurations of the application, suited to the particular state of contextual information. These configurations are listed in a special ordering that reflects their user-perceived preference. Each <config> element, representing a configuration, specifies the resource requirements of the configuration, the components and aspects constituting the configuration, the callback methods to be called for the configuration, and the lower-level services corresponding to the configuration. The detailed specification of an application contract is not described in this paper as it is not relevant to our discussion of the approach for dynamic recomposition of components, except the specification of <components> element which is discussed partially in Section 4.

Depending on the current state of the execution environment (contextual information and resources), the appropriate configuration from the application contract is selected and activated by the CRS. More details on the CASA framework can be found in [1, 10].

## 3 Requirements for dynamic recomposition of components

A *component composition*, or just *composition*, is a collection of components qualified to do the required application task under a specific state of the execution environment.

A primary and obvious requirement for application adaptation through dynamic recomposition of components is:
*Requirement 0:* An adaptive application needs to provide a number of alternative compositions for different states of the execution environment.

We can now define *dynamic recomposition of components* as changing between alternative compositions of an application at runtime.

Any two alternative compositions may vary in just a few components, while many other components remain the same across both compositions. When changing from one alternative composition to another, there may be some new components to be added and some old components to be removed.

*Dynamic replacement* of components is a special case of a dynamic removal of a component $A$ followed by a dynamic addition of a component $A'$, such that $A'$ is able to serve all those components that could be served by $A$, in an alike manner as $A$ itself.

If a component $A$ can be dynamically replaced by a component $A'$, then both $A$ and $A'$ must subscribe to the same component *contract* (the term *contract* used in the sense of the Design by Contract approach [9]). That is, the following two requirements need to be satisfied by $A$ and $A'$ (in CASA, dynamic replacements are bidirectional, i.e. if $A$ can be dynamically replaced by $A'$, then it automatically implies that $A'$ can also be dynamically replaced by $A$):
*Requirement 1:* Both $A$ and $A'$ must conform to the same interface, i.e. the method signatures of the publicly-accessible methods of $A$ and $A'$ must be the same.

*Requirement 2:* The pre and post conditions of the publicly-accessible methods of $A$ and $A'$, which must be satisfied for the interaction of these methods with their clients, must be the same. The pre and post conditions may also include certain non-functional assertions or constraints.

Next, we state a requirement for mapping the state of $A$ to the state of $A'$. For this purpose, we define the *persistent* state of a component as the state that needs to remain persistent in between its executions.

*Requirement 3:* A valid *persistent* state of $A$ when mapped to $A'$, using an appropriate state mapping function, must become a valid *persistent* state of $A'$.

The following two requirements pertain to the dynamic removal and dynamic addition of components.

*Requirement 4:* If a component $A$ is removed during dynamic recomposition, then it must be replaced dynamically by a component $A'$ or else all the components depending on $A$ must also be removed along with $A$.

*Requirement 5:* If a component $A'$ is added during dynamic recomposition, then the components on which $A'$ depends either must already be present or they must be added along with $A'$.

Requirements 4 and 5 are related to ensuring the *completeness* of alternative compositions.

Both *completeness* and *correctness* of every alternative composition, in terms of its ability to do the required application task under its corresponding state of the execution environment, need to be ensured by the application developer at the time of composing the alternative compositions.

The following two requirements are related to ensuring the *consistency* of the application.

*Requirement 6:* If a component $A$ is replaced dynamically by a component $A'$, then $A'$ must be able to continue the execution from where $A$ left.

*Requirement 7:* The integrity of the interactions among components must not be compromised due to dynamic recomposition.

Requirements 6 and 7 above help to protect the application from being in an inconsistent state as a result of the dynamic recomposition.

## 4 Implementation of dynamic recomposition of components in CASA

In this section, we discuss the implementation issues related to dynamic recomposition of components for the applications developed using object-oriented programming languages. In particular, we consider Java as a target language, because of its widespread use and popularity. However, we will try to keep our discussion as language-neutral as possible, so that the results are applicable for a wide range of object-oriented programming languages.

A dynamic recomposition implies adding / removing / replacing components dynamically. Dynamic replacement of components is of particular interest here, as it is more critical than simple addition or removal of components which is relatively straightforward to carry out. Hence we will focus on dynamic replacement of components in the following.

In principle, there are two possible strategies for dynamic replacement: Lazy replacement and Eager replacement. Below we briefly discuss the two.
*Lazy replacement:* In this strategy, once the decision for dynamic recomposition is taken, an already running component is allowed to complete its current execution before being replaced.
*Eager replacement:* In contrast to the lazy replacement strategy, here the execution of a running component is suspended once the decision for dynamic recomposition is taken, and the execution resumes again from the point where it was suspended, after the component is replaced.

Figure 3 illustrates lazy replacement (Figure 3a) and eager replacement (Figure 3b). In Figure 3, the horizontal axis represents the time line, and the vertical dashed line represents the time $T$ when the decision for dynamic recomposition is taken. In this example, the components $A$, $B$, $C$ and $D$ are to be replaced by the components $A'$, $B'$, $C'$ and $D'$ respectively as a result of dynamic recomposition (dark bars denote the execution of old components, and light bars denote the execution of new components). Only the components $A$ and $C$ are under execution at time $T$. In Figure 3a (representing lazy replacement) $A$ and $C$ are allowed to complete their execution before being replaced by $A'$ and $C'$ respectively. Whereas in Figure 3b (representing eager replacement), the execution of $A$ and $C$ is suspended at time $T$, they are replaced by $A'$ and $C'$ respectively, and the execution resumes again with $A'$ and $C'$.

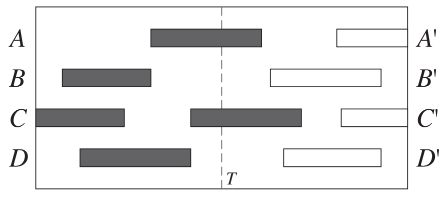

**Fig. 3a.** Lazy replacement strategy

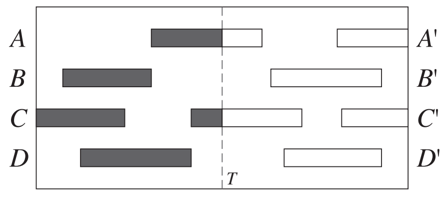

**Fig. 3b.** Eager replacement strategy

Since the eager replacement strategy is able to give a faster response to a change in execution environment than the lazy one, we decide in favor of eager replacement for CASA. However, as discussed later, it may not always be possible to use eager replacement, and thus sometimes lazy replacement may be the only option.

## 4.1 Dynamic replacement process

In terms of object-oriented programming, a component is essentially an instance of a class (with a restriction that, unlike normal class instances, components cannot have any externally-visible state). Thus, from an implementation point of view, replacing a component involves replacing the corresponding class definition of the instance. We will use the terms "component" and "class instance" interchangeably throughout the rest of this paper.

We now define an *adaptable class* as the one whose instances are dynamically replaceable (i.e. can replace, or be replaced by, instances of other classes dynamically). Additionally, we define a *set of alternative classes* as a collection of adaptable classes whose instances can dynamically replace each other. That is, all the adaptable classes that are members of the same set of alternative classes, and by implication the instances of these adaptable classes, conform to the requirements 1-3 identified in Section 3.

To ease our implementation process, we impose the following additional conditions:
(i) An instance $A$ of a class $C$ can be dynamically replaced by an instance $A'$ of a class $C'$ only if $C$ and $C'$ are members of the same set of alternative classes.
(ii) Any given composition may contain instances of only one of the adaptable classes from any given set of alternative classes. That is, no two instances in a given composition may be of different classes from the same set of alternative classes.

We use a variant of the Bridge pattern [2] for hiding the complexities of dynamic replacement from the application code. In particular, every set of alternative classes is associated with a unique *Handle* class. The *Handle* class conforms to the same interface as the adaptable classes in its associated set.

The *Handle* class acts as an *abstraction* that can be bound to any of the adaptable class *implementations* from its associated set of alternative classes at runtime (the terms *abstraction* and *implementation* used in the sense of the Bridge pattern [2]).

We know that (i) any given composition may contain instances of only one of the adaptable classes from any given set of alternative classes, and (ii) every set of alternative classes has a unique *Handle* class associated to it. Therefore, we can conclude that: for any given composition there is a unique adaptable class bound to any given *Handle* class.

The binding between a *Handle* class and its corresponding adaptable class for a given composition is represented as a part of the composition specification in the application contract (refer <binding> element within <components> element in Figure 2).

In order to provide a layer of transparency between the application code and the dynamic replacement process, wherever there is a need for creating an instance of an adaptable class in the application code, an instance of the corresponding *Handle* class is created instead. This *Handle* class instance is then linked to an instance of the adaptable class that is currently bound to the *Handle* class, at runtime (as explained below).

Let a set of alternative classes `S` consist of the adaptable classes `CdefA`, `CdefB` and `CdefC`, and the associated *Handle* class for the set `S` be `HC`. At any given time, `HC` will be bound to a unique adaptable class from the set `S`, depending on the currently active composition. However, this binding may change dynamically as a result of dynamic recomposition.

In the application code, when a new instance `objHC` of the *Handle* class `HC` is created, the constructor of `objHC` invokes the CRS (CASA Runtime System). The CRS gets the information about the adaptable class currently bound to `HC`, say `CdefA`, from the specification of the currently active composition, and returns the namespace location of the class `CdefA` back to the constructor of `objHC` (the CRS also registers `objHC` for future recompositions). The constructor of `objHC` then creates an instance of `CdefA`, say `objA`, and stores it internally as *active* adaptable class instance.

Although a *Handle* class conforms to the same interface as the classes in its associated set of alternative classes, it does not provide a *real* implementation for any of the methods in this interface. The methods of a *Handle* class instance simply forward the method calls invoked on them to the corresponding methods of the *active* adaptable class instance, and return the results as received from the latter. For example, if a method `foo()` is invoked on `objHC`, then `objHC.foo()` simply invokes the method `objA.foo()`, and returns the result as received from `objA.foo()`.

If there is a change in the binding between the *Handle* class `HC` and its corresponding adaptable class, due to dynamic recomposition, then the CRS passes the namespace location of the newly bound adaptable class, say `CdefB`, to all the instances of `HC` (including `objHC`). The instances of `HC` replace the old adaptable class instances with the instances of `CdefB` as *active* adaptable class instances (the details of this replacement are discussed next). The calls to an instance of `HC` will now be forwarded automatically to the new adaptable class instance in place of the old one. This way, the *Handle* class instances help to hide the details of dynamic replacement from the application.

Figure 4 illustrates the above example of dynamic replacement. In Figure 4a the *Handle* class instance `objHC` is linked to the old adaptable class instance `objA`, just before the dynamic replacement is carried out. And in Figure 4b, `objHC` is linked to the new adaptable class instance, say `objB`, just after the dynamic replacement is over. The external components (`extObj1`, `extObj2` and `extObj3`) are largely unaffected by this dynamic replacement, as their links to `objHC` remain undisturbed by the change.

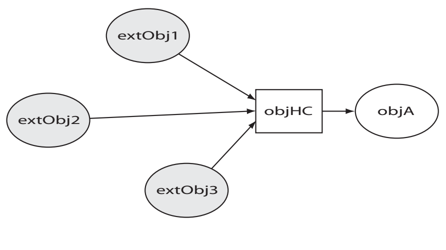

**Fig. 4a.** Before dynamic replacement

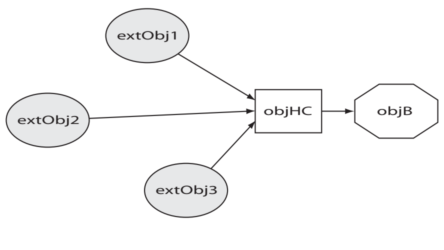

**Fig. 4b.** After dynamic replacement

Below we discuss the sequence of steps to be carried out by `objHC` when replacing `objA` with `objB` (as per the eager replacement strategy).

**Sequence of steps:**

1. Deactivate `objA`
2. Suspend the execution of `objA`
3. Create `objB`
4. Transfer the state of `objA` to `objB`
5. Activate `objB`

If `objA` is not running at the time of replacement then step 2 is not required. Below we discuss the implementation of the above-mentioned steps.
*Step 1: Deactivate* `objA`*:* First, on receiving an indication from the the CRS about dynamic replacement, `objHC` *deactivates* the reference to `objA`. This ensures that the calls made to `objHC` during the dynamic replacement process are not forwarded to `objA`, and rather wait within `objHC`.
*Step 2: Suspend the execution of* `objA`*:* Suspending the execution of `objA` implies suspending all the calls currently executing on `objA`. But before actually suspending a call executing on `objA`, it needs to be ensured that the execution of the call has reached a *safe* point where it can be resumed *correctly* by `objB`, at the end of dynamic replacement. And for this, the *safe* points need to be explicitly defined in the body of `objA` (more discussion on this follows later).

After deactivating the reference to `objA` (step 1), `objHC` sets a signal for the suspension of `objA`. At every *safe* point, each call executing on `objA` checks if a signal for the suspension of `objA` has been set. If such a signal is set, then an exception is thrown on this call, to be eventually caught by `objHC`. The information about the *safe* point where the call is suspended is also passed to `objHC` along with the exception. After catching the exception, `objHC` needs to take necessary actions like reinvoking the call on `objB` after the completion of the dynamic replacement process. This time the information about the *safe* point where the call was previously suspended is passed as an argument while reinvoking the call, to enable `objB` to resume the execution correctly. For this, the methods of `objB` should be able to accept an additional argument of the type `SafePoint` (during normal forwarding of calls by `objHC`, the value of this argument will be null).

This step is over when all the calls executing on `objA` have returned (either normally or after being suspended) to `objHC`.
*Step 3: Create* `objB`*:* After setting the signal for the suspension of `objA`, `objHC` creates an instance of the new adaptable class (passed by the CRS), i.e. `objB` (the creation of `objB` may take place while step 2 is still on, i.e. during the time all the calls executing on `objA` return to `objHC`).
*Step 4: Transfer the state of* `objA` *to* `objB`*:* Once all the calls executing on `objA` have returned to `objHC` (at the end of step 2), the state of `objA` is transferred to `objB` at the initiation of `objHC`.

For transferring the state, i.e. storing the state and loading the state, every dynamically replaceable component needs to provide appropriate `storeState` and `loadState` methods. This is because state parameters (names and types) may vary across the old and new components, which means that the semantic information necessary for state transfer can be provided by the respective components only. The `storeState` method of `objA` may need to convert its own component-specific representation of the state into a standard representation (standard for the corresponding set of alternative classes), which the `loadState` method of `objB` understands and may again convert into its own component-specific representation.

*Step 5: Activate* `objB`*:* Finally, `objHC` sets `objB` as *active* adaptable class instance (`objA` can now be garbage collected).

Now the execution can continue on `objB`.

**Discussion:** The description above assumes that the new component requires the state of the old component to be transferred to it, and also requires the information about the *safe* points where the calls were suspended, in order to continue the execution from where the old component left. However, in practice, either or both of these requirements can be relaxed, depending on the properties of the concerned components (on the other hand, it ultimately rests on the capability of the new component itself to continue the execution correctly, even if both these requirements are satisfied).

That is, in some cases, there may not be a need for passing the information about the *safe* points to the new component, e.g. if the state transferred to the new component provides enough information to resume the execution correctly. And in rather extreme cases, there may not be a need for transferring the state of the old component to the new component, e.g. if the new component is specifically designed to recover from a state loss, though it will most likely result in a degraded performance.

There can be some components that can be suspended abruptly, e.g. if the new component provides an entirely different functionality and is going to begin its execution from its initial point of execution (as typically in response to a change in contextual information). This means that every point of execution in the old component is in effect a *safe* point. From the implementation perspective, this implies that there is no need for explicitly defining *safe* points in such components, and the already executing calls can be simply suspended by throwing exceptions abruptly in step 2 above.

Queuing the new calls made during the dynamic replacement process within the *Handle* class instance, as well as the calls that were suspended and returned to the *Handle* class instance, and invoking these calls at the end of dynamic replacement, help maintain the integrity of the interactions among components.

Next we show that eager replacement may not be viable for some components, leaving lazy replacement as the only option.

Consider an eager replacement where the state of the old component needs to be transferred to the new component, and the old component is running at the time of replacement. One of the necessary conditions for ensuring the validity of this replacement is that the state transferred gets transformed into a reachable state of the new component. This will most likely not be possible at any random point of execution of the old component, but probably at some specific points. Such points of execution of the old component that ensure that the state transferred gets transformed into a reachable state of the new component are referred as *valid-change* points. If the state transferred at any random point of execution is ensured to get transformed into a reachable state of the new component, then it implies that every point of execution of the old component is a *valid-change* point.

We know that in the eager replacement strategy the state is transferred at the *safe* point where the last of the calls executing on the old component is suspended. Now to ensure a valid replacement, this *safe* point has to be a *valid-change* point. And since we cannot predict in advance at which *safe* point the last call will be suspended, we can say that every *safe* point has to be a *valid-change* point.

However, we argue that there is no guarantee that a *valid-change* point exists in an arbitrary component to be replaced (not counting the control points just before the initial point and just after the last point of execution, as they are not practically very helpful).

To support our argument, we refer to the results provided by Gupta et. al. [3] in the context of a runtime change in software version. They define a valid change as the one in which the state of the old software version gets transformed into a reachable state of the new software version. They also show that locating the points of execution where a valid change may be guaranteed is in general *undecidable*, and approximate techniques based on data-flow analysis and knowledge of application developer are required. This effectively implies that there may not exist any point of execution in the old software version that may guarantee a valid change.

This result can be directly extended to the case of dynamic replacement of components, to support our argument that there is no guarantee that a *valid-change* point exists in an arbitrary component to be replaced.

If a component does not contain any *valid-change* point, then the possibility of defining *safe* points in the component is automatically ruled out. This, in turn, renders eager replacement unachievable for such components, leaving lazy replacement as the only option.

With lazy replacement, the component to be replaced is certainly not running at the time of replacement, and thus the state to be transferred refers to the *persistent* state of the component, in contrast to the *transient* state for a component that is running at the time of replacement. From requirement 3 (Section 3), we know that the *persistent* state of the old component when transferred to the new component is automatically a reachable state of the new component.

For lazy replacement, the replacement process discussed before can be suitably modified in a straightforward manner. In any case, the implementation of either of the two strategies is localized within a *Handle* class instance and the corresponding dynamically replaceable components.

### 4.2 Performance evaluation

A prototype, based on the CASA framework, has been implemented in Java, and the results have been encouraging. We have been able to demonstrate the dynamic adaptation features of the CASA framework, at a minimal performance cost. A detailed overview of performance evaluation of the prototype is given in [4]. Below we present some of the indicative results.

During normal operation of an application, the only performance overheads are due to using an additional level of indirection when accessing a dynamically replaceable component through a *Handle* class instance, and for checking a signal for component suspension at every *safe* point within the component code. Both these overheads were found to be quite insignificant – in the order of a few micro seconds.

The performance overhead during dynamic replacement of components varied widely depending on the number of components to be replaced – for the test results, the values were 2-7 ms for a single component and 25-100 ms for twenty components, depending on the processor speed (assuming no delay for the calls executing on the old components to reach their respective *safe* points).

The overhead for the state transfer between components was found to be very small (in the order of a few micro seconds), while the size of the state to be transferred did not have much influence on the results.

The frequency of *safe* points in a component code has an obvious positive impact on the swiftness of dynamic replacement. Since the overhead due to each *safe* point during normal operation is negligible (a couple of micro seconds), it is recommended to define *safe* points quite frequently in every dynamically replaceable component, if possible.

## 5 Related work

Over the last few years, some approaches have been proposed for software adaptation using dynamic change in application components. Rasche and Polze [14] present an approach for dynamic reconfiguration of component-based applications for the Microsoft .NET platform. This approach uses a transaction-based component model to decide the appropriate timing and order for reconfiguration. However, dynamic reconfiguration here implies adding new components, removing old components, changing the connections among components, or changing the component attributes, while it does not provide means for dynamic replacement of components involving state transfer etc.

The Accord framework [7] enables a dynamic change in application behavior according to the rules associated with every application component. However, with this approach, the interactions between application components need to be defined in terms of rules associated with the corresponding components, in order for these interactions to be changeable at runtime by changing the corresponding rules. Since the number of potential interactions between application components can be quite large, the number of possible rules can be exponential, making the rule management quite complex and inducing performance overhead due to execution of all these rules at runtime.

Some more work has been done on runtime software evolution, which has a close bearing with the software adaptation using dynamic change in application components. Oreizy et. al. [12] provide a software architecture-based approach for runtime software evolution, and discuss dynamic recomposition of application components at the architecture level. In this approach, the components interact with each other only through the connectors that mediate all component communications. This makes it possible to alter a component composition by changing the component bindings of the connectors at runtime. The role of connectors here is similar to the role of *Handle* components in CASA, though in CASA only the dynamically replaceable components need to be accessed through *Handle* components.

Dynamic Java classes [8] provide a generic approach to support evolution of Java programs by changing their classes at runtime. This approach shares the same goals as our implementation approach. A drawback of this approach, however, is that it takes a much harder way of modifying the JVM to implement dynamic replacement of classes. Using a customized JVM may result in reduced portability, and may eventually restrict the usage of this approach. Similarly, the approach of dynamic C++ classes [5] allows a version change of a running C++ class. However, with this approach, once the version of a class has been changed, only the new instances created after the version change belong to the newer version. The already created instances belonging to the older version are either allowed to continue till they expire normally or they are destroyed abruptly, while no attempt is made to replace these instances with ones belonging to the newer version. Clearly, such an approach is not suitable for our purpose.

## 6 Conclusion and future work

The CASA framework enables dynamic adaptation of applications in response to changes in their execution environment. With a view to meet adaptation needs of a broad and diverse set of applications, the CASA framework supports dynamic adaptation at various levels of an application – from lower-level services to application code. In this paper, we discussed the implementation issues related to the adaptation of an application by recomposing its components dynamically, as supported in the CASA framework.

An underlying presumption in realizing application adaptation through dynamic recomposition of components is that the application provides alternative component compositions for different states of the execution environment. The cost of developing these alternative component compositions would be mitigated by the amount of reuse of the components constituting these compositions. We have also presumed that the correctness and completeness of alternative component compositions is ensured by the application developer at the time of composing these compositions. We envisage that appropriate tools to help ensure this would be available to the application developer.

In the near future, we intend to identify dynamic adaptation needs of different kinds of applications executing in dynamic environments. Based on this information, we will verify which of these adaptation needs are met effectively by our current approach and where modifications or extensions will be required.

## References

1. The CASA Project. http://www.ifi.unizh.ch/req/casa/
2. E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
3. D. Gupta, P. Jalote and G. Barua. A Formal Framework for On-line Software Version Change. *IEEE Transactions on Software Engineering*, 22(2), 1996.
4. A. Gygax. *Studying the Effect of Size and Complexity of Components on the Performance of CASA*. Internship Report, IFI, University of Zurich, 2004. http://www.ifi.unizh.ch/req/ftp/papers/casa-perf.pdf
5. G. Hjalmtysson and R. Gray. Dynamic C++ Classes: A lightweight mechanism to update code in a running program. *Proc. of USENIX Annual Technical Conference*, 1998.
6. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier and J. Irwin. Aspect-Oriented Programming. *Proc. of 11th European Conference on Object-Oriented Programming*, 1997.
7. H. Liu, M. Parashar and S. Hariri. A Component Based Programming Framework for Autonomic Applications. *Proc. of 1st International Conference on Autonomic Computing*, 2004.
8. S. Malabarba, R. Pandey, J. Gragg, E. Barr and J.F. Barnes. Runtime Support for Type-Safe Dynamic Java Classes. *Proc. of 14th European Conference on Object-Oriented Programming*, 2000.
9. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10), 1992.
10. A. Mukhija and M. Glinz. A Framework for Dynamically Adaptive Applications in a Self-organized Mobile Network Environment. *Proc. of ICDCS 2004 Workshop on Distributed Auto-adaptive and Reconfigurable Systems*, 2004.
11. B.D. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn and K.R. Walker. Agile Application-Aware Adaptation for Mobility. *Proc. of 16th ACM Symposium on Operating Systems Principles*, 1997.
12. P. Oreizy, N. Medvidovic and R.N. Taylor. Architecture-Based Runtime Software Evolution. *Proc. of 20th International Conference on Software Engineering*, 1998.
13. A. Popovici, T. Gross and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. *Proc. of 1st International Conference on Aspect-Oriented Software Development*, 2002.
14. A. Rasche and A. Polze. Configuration and Dynamic Reconfiguration of Component-based Applications with Microsoft .NET. *Proc. of 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2003.
15. J.A. Zinky, D.E. Bakken and R.E. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1), 1997.