Toward Architecture-based Context-Aware Deployment and Adaptation

Ning Gui[a,b,*], Vincenzo De Florio[a,*], Hong Sun[a], Chris Blondia[a]

[a] *PATS group, University of Antwerp, Belgium and IBBT, Ghent-Ledeberg, Belgium*
[b] *Central South University, Changsha, China*

Abstract

Software systems are increasingly expected to dynamically self-adapt to the changing environments. One of the main adaptation mechanisms is dynamic recomposition of application components. This paper addresses the key issues that arise when context knowledge is used to steer the run-time (re)composition process so as to match the new environmental conditions. In order to integrate such knowledge into this process, A Continuous Context-Aware Deployment and Adaptation (ACCADA) framework is proposed. To support run-time component composition, the essential runtime abstractions of the underlying component model are studied. By using a layered modeling approach, our framework gradually incorporates design-time as well as run-time knowledge into the component composition process. Service orientation is employed to facilitate the changes of adaptation policy. Results show that our framework has significant advantages over traditional approaches in light of flexibility, resource usage and lines of code. Although our experience was based on the OSGi middleware, we believe our findings to be general to architecture-based management systems using reflective component models.

Keywords: Adaptive middleware, context-specific knowledge, run-time composition, Service-oriented architecture.

1. INTRODUCTION

Software systems today are increasingly expected to dynamically self-adapt to accommodate for changing environments-resource variability, changing user needs, and system faults. However, mechanisms that support self-adaptation

*Corresponding author
Email addresses: `ning.gui@ua.ac.be` (Ning Gui), `vincenzo.deflorio@ua.ac.be` (Vincenzo De Florio)

currently are hard-wired within each application. These approaches are often highly application-specific, static in nature, and tightly bound to the code. Furthermore, the localized treatments of application adaptation could not effectively deal with those complex environments in which many multi- influencing applications coexist.

In order to deal with the adaptation problem outside the single application scope, architecture-based adaptation frameworks are proposed in (Oreizy et al., 1999; Garlan et al., 2004) to handle cross system adaptation. Rather than scattering the adaptation logics in different applications and representing them as low-level binary code, architecture-based adaptation uses external models and mechanisms in a closed-loop control fashion to achieve various goals by monitoring and adapting system behavior across application domains. A well-accepted design principle in architecture-based management includes using component-based technologies to develop management system and application structure (Kon et al., 2005; Anderson et al., 2003; Sicard et al., 2008).

However, in traditional approaches, applications are constructed well before run-time. Design-time knowledge for an application in the structure is largely lost after this off-line application constructing process. Without this knowledge, it is nearly impossible for external engines to effectively change run-time applications' structure with the assurance that the new configuration would perform as intended. On the other hand, this off-line approach makes it very hard to integrate possible external context[1] knowledge into the application adaptation process as that knowledge can only be available well after an application was built.

During our research on run-time adaptation, we observed that in order to achieve effective architecture-based adaptation frameworks, three important prerequisites must be fulfilled. First, when building application, those practices of rigid location and binding between component instances should be replaced with run-time, context-specific composition. Second, selected design-time information must be exposed in some form and those constraints in building application must be made explicitly verifiable during run-time. Third, since different contexts have radically different properties of interest and require dynamic modification strategies, it is critical that the architectural control model and modification strategies could be easily tailored to various system contexts.

Our framework tackles these problems from different perspectives. A run-time application construction methodology is proposed to provide a continuum between the design and run-time process. An architecture-based management framework structure is designed to facilitate the integration of context-specific adaptation knowledge. In order to support run-time component composition, a declarative component model with uniform management interface and meta-

[1]By context, we refer to (Dey et al., 2001) and define it as "any information that characterizes a situation related to the interaction between humans, applications and the surrounding environment."

data based reflection is proposed. By adopting a service-oriented architecture-based implementation, our framework provides efficient mechanisms for adapting to specific context requirements. The effectiveness of our architecture is demonstrated from both a qualitative and a quantitative point of view. Simulation results show the soundness of our implementation in term of lines of code, memory usage and adaptation capabilities. Compared to our previous work (Gui et al., 2009), apart from extending considerably the contents and enhancing simulation use cases, this paper introduces explicit conflicts resolution management and provides reflective service that exposes the system adaptation process. These improvements enable our platform to synthesize a coherent global adaptation plan from possible conflicting solutions.

The rest of the paper is organized as follows. Section 2 exposes our design methodology and a context-specific management framework as well as those challenges we faced in realizing this framework. Section 3 presents the structure of our management framework as well as the component model and construction process. Section 4 focuses on two scenarios on how context knowledge can be used for adaptation and to resolve composition ambiguity. The ideas exposed in this paper have been validated from different viewpoints in Section 5. We discuss related work in Section 6, and we conclude in Section 7.

2. Architecture-based Adaptation

Architecture-based adaptation was introduced to deal with cross system adaptation. In principle, such external control mechanisms provide a more effective engineering solution with respect to internal mechanisms for self-adaptation because they abstract the concerns of problem detection and resolution into separable system modules (Garlan et al., 2004). However, they still lack of systematic support for multi-context knowledge integration. An important contribution of this paper is the design and development of architectural principles and design patterns to integrate different context-specific knowledge into architecture-based adaption framework.

In order to better support run-time component composition, firstly, we need to re-consider the design methodology in application composition.

2.1. Context-specific application construction methodology

Traditional approaches treat components only as design-time artifacts. Intensive studies have been carried out on designing languages to specify properties of a component, e.g. OMG IDL (OMG Committee, 2007) and Architecture Analysis and Design Language (AADL) (Feiler et al., 2006). Those languages are used to design and verify the application construction plan. Model-driven design tools are used to parse such descriptions and automatically generate auxiliary glue code. Later, that glue code, together with component implementations,

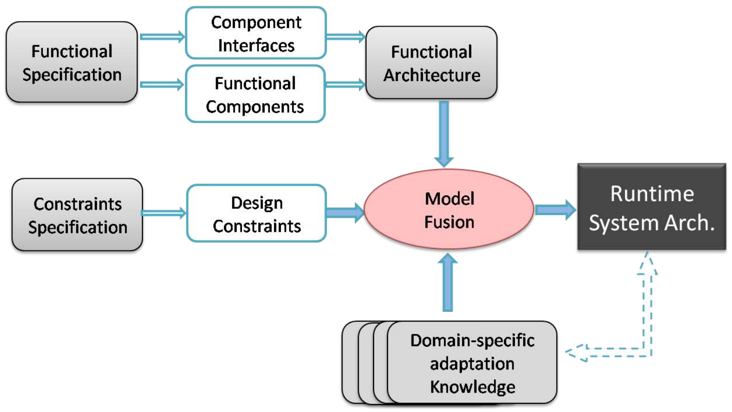


Figure 1: Context-specific application construction flow.

is compiled into a static entity and deployed with little capability of further changes.

The current trend on component model is rather towards making components as run-time manageable entities (Kasten and McKinley, 2004; Coulson et al., 2008; Hall and Cervantes, 2004). However, such approaches maintain little design-time knowledge on the run-time entities. Lack of such knowledge makes it hard to have accurate and correct run-time adaptation.

The methodology. In order to more effectively deal with run-time component composition, we propose a new methodology to explicitly incorporate context-specific knowledge into the software composition & adaptation process. The new architecture design & composition flow, depicted in Fig. 1, represents a procedure that tries to incorporate the functional design information with context concerns in composing the run-time software architecture. Depending on the employed design languages and corresponding tools, the compliance with the functional interface is enforced during the design process. However, unlike the traditional approaches, in which a component's functional knowledge is lost during this compiling process, in this case the design time information is explicitly exposed and maintained.

As an application is constructed during run-time, in order to achieve correct and accurate adaptation, a set of constraints must be maintained. In this process, constraints from three main aspects should be evaluated: 1) The functional dependence constraints must be satisfied. 2) A component's non-functional constraints must be guaranteed: this information includes, for instance, requirements for CPU speed, screen size or that some property value fall a certain range. 3) Context-specific knowledge, which specifies the domain related information and adaptation strategy should also hold valid after adaptation process.

The dashed arrow between run-time architecture and context knowledge blocks

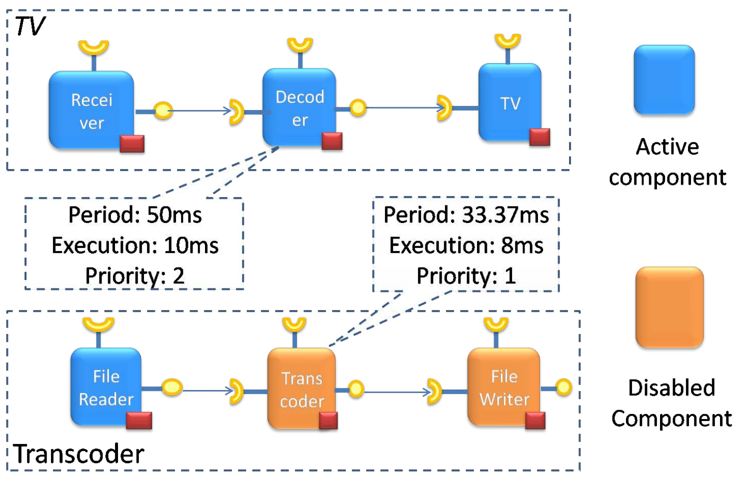


Figure 2: QoE adaptation demonstration.

shows that managed applications are continuously restructured and evolved according to context knowledge and adaptation strategy. The combined knowledge enables automatically run-time verification for constraints from various aspects—functional dependence, non-functional constraints and context specific considerations. This allows the system to change the software structure according to its hosting environment and without violating constraints from these three aspects.

Service-oriented Approach. According to the Mazeir Salehie's recent review (Salehie and Tahvildari, 2009) of 12 projects on self-adaptive systems, all of the projects use certain level of external control approach, which supports separation of the adaptation mechanism from the application logic. However, many of their implementation designs lack of systematic support for a dynamically changing context. The source of this problem lies partially within their approaches in implementation—the adaptation modules are finely designed and statically linked with system run-time. As the adaptation module is tightly coupled with other system run-time services, it is very hard to change the system adaptation strategy.

In order to achieve more re-usability and flexibility, our framework is designed according to the Service-oriented model. Each module is designed and implemented as a service provider. Modules implement and register their interfaces into the system service registry. Thanks to such loosely coupled structure, candidate adaptation modules can be easily interchanged during system run-time. By doing so, many existing and/or future more sophisticated context adaptation policies can be plugged into our framework.

2.2. Motivating Example

To better illustrate all the complexities in introducing the context knowledge into the application composition process, we make use of an example scenario that will be revisited several times throughout the course of this paper.

Today we are surrounded by an ever increasing number of networked equipment that is reachable anywhere. These smart devices are equipped with open systems that can be harnessed to do something for you for temporal or long-term basis. The open-system approach implies that a set of new applications will be installed into the host devices without a thorough system analysis.

As an example, let us consider a set-top box device with open platform support. The basic application of such device is TV processing. In brief, this application will received streaming video data from a remote server, decode this data and send the output it to the TV port. As an open system, the set-top box can also install certain applications to enhance its usability. For example, a user can install a new application which transcodes recorded High Definition TV stream to iPhone format for later display on his/her mobile devices. Figure 2 shows the simplified component graph for those two applications, which will be further studied in later sections.

As a typical multi-task system, if a user starts those two applications, a set-top box will try to execute the two applications simultaneously no matter whether the set-top device has enough resources. If that is not the case, this may eventually lead to transient timing problems of TV decoding task including missing frames, data overflows etc. These kinds of time breaches can result in poor video quality and bad user experience.

Context-specific knowledge, however, can help the architecture automatically determine which actions should be taken according to the current context. One possible strategy is to disable the computationally intensive transcoding component and reserve enough system resources for the TV application. This is because a user normally prefers to give highest priority to those applications that matter their experience most. Figure 2 shows the snapshot of component states after such adaptation. When the user turns off the TV application, the architecture senses the change and enables the stopped transcoding application.

Let us suppose there are two transcoder components to provide transcoding service with different CPU usage and transcoding quality as shown in Fig. 3. Such ambiguity cannot be solved by the functional dependence as the two components provide the same service interface. With context knowledge, the system can determine when to choose the best component to use. For instance, when the TV application is enabled and the system is lacking adequate resources, making use of this context knowledge, one adaptation strategy is to replace the transcoder with lower execution time (high CPU usage) with a low CPU usage component. Context knowledge can effectively determine which combination is the best for the current context.

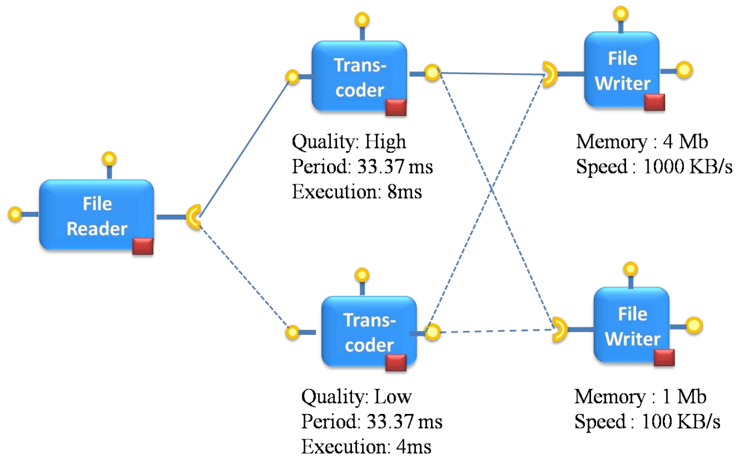


Figure 3: Using context knowledge in solving composition ambiguity.

2.3. Challenges in system design

The above case study shows that, without context specific knowledge, it is very hard for application-based adaptation to effectively deal with conflicts across application domains. Architectural-based context-specific adaptation allows system-wide adaptation to be orchestrated according to the current context knowledge. By using such an approach, context knowledge is explicitly expressed and plays an indispensable role for adaptations across different applications.

While attractive in principle, this raises a number of research and engineering challenges.

First, the ability to handle a wide variety of systems and changing contexts must be addressed. Since different systems and different contexts may have radically different properties of interest, as well as strategies for adaptation, the architectural control model and modification strategies should be easily changed to the specific environment.

Second, the complexity to cope with the dynamic availability of the components needs to be solved with minimal cost. Each application is composed by managed components during run-time, thus it is very important to provide services such as components' state monitoring, life-cycle control and reference management between components. Application developers should be rid of the burden to code for component dynamicity support, for instance, manually coding to support component departure or arrival.

Third, in architecture-based adaptation, applications are composed, configured, and reconfigured during system run-time. While flexible in nature, two types of problems appear: how to determine the composability between two components and how to deal with the composition ambiguity when more than one candidate component exists. Many approaches, such as Seinturier et al. (2006) and Hall and Cervantes (2004), focus on solving the composability problem. In fact, the validity of composability can be effectively solved by matching a language based interface type with tagged information. However, as for the ambiguity problem, so far there is no effective solution, for much of such ambiguity can only be clarified with run-time context knowledge. For instance, video player applications for mobile devices should be composed with those components that match the target mobile phone's characteristics such as screen size, CPU, and available memory.

In order to cope with such challenges, A framework for Continuous Context-Aware Deployment and Adaptation (ACCADA) is proposed here. Its key modules are designed and introduced in the following section.

3. ARCHITECTURAL FRAMEWORK

We adopt a standard view of software architecture that is typically used today at design time to characterize an application to be built. Specifically, an application is represented as a graph of interacting computational elements. Nodes in the graph, called components, represent the system's principal computational elements (software or hardware) and data stores. The connections represent the pathways for interaction between the components. Additionally, the component may be annotated with various properties, such as expected throughputs, latencies, and protocols of interaction.

Rather than treating components only as design-time artifacts, in our framework, applications are run-time composed from a set of managed component instances. Context-specific adaptation can be achieved by dynamic (re)composing application components according to context knowledge.

3.1. Architecture-based management framework

Figure 4 shows our ACCADA architecture for adaptation. As illustrated in the figure, our approach makes use of an extended control loop, consisting of five basic modules : *Event Monitor*, *Adaptation Actuator*, *Structural Modeler*, *Context-Specific Modeler* and *Context Reasoner*. ACCADA uses an abstract architectural model to monitor a running system's run-time properties, evaluate the model for (functional as well as context-specific) violations, and—if a problem occurs—performs global and component-level adaptations on the running system. The *Event Monitor* module observes and measures various system states.In our current implementation, the system is provided with Event Monitors for the most popular of basic monitoring information: adding/removing components, registering/modifying/unregistering services from the service registry, memory footprint and CPU usage monitoring, etc. The user can also provide their own customized Event Monitor to collect specific events of interest. When a significant event takes place, the corresponding Event Monitor sends notifications to trigger a new round of the adaptation process

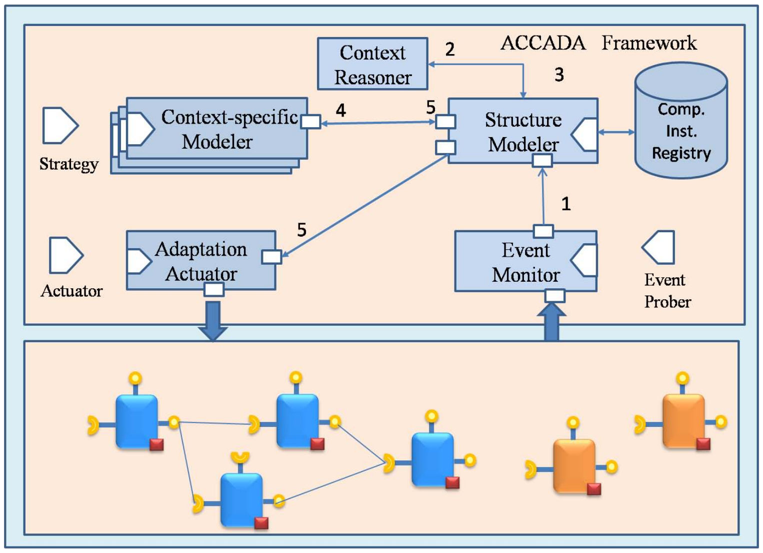


Figure 4: ACCADA framework: Management Layer (upper layer) and managed application components (lower layer).

The Adaptation Actuator carries out the actual system modification. The actual action set is tightly related to the component implementation. From our developing experience, in order to achieve effective architecture-based adaptation, the basic set of actions should include component life-cycle control, attribute configuration, and component reference manipulation.

The above two modules provide an interface to manage the installed component instances and form the ACCADA Management Layer (discussed in Section 3.4). The other three modules constitute what we call the Modeling Layer which is in charge of building the system architectural model according to the changing context (discussed in Section 3.3).

As discussed in Section 2.1, building a software system architecture model is not a trivial endeavor — it includes handling design-time knowledge such as interfaces or constraints as well as run-time aspects on environmental changes. We assign the management of these two aspects to two different modules, using the Divide and Conquer principle so as to more effectively deal with different requirements—software architecture management and context-specific knowledge integration. One module, the Structural Modeler, manages functional dependencies between installed components—checking whether the required and provided interfaces are compatible—and maintains the application's software architecture. This module is comparably stable as it is only determined by the component implementation model and will not change with context. So, it is designed as the core module in our system. In addition to such functional dependency management, it also provides reflection service(Section 3.4) to expose system internal status and adaptation steps performed for the validation and verification of system modelers.

The other set of modules constitute the Context-specific Modeler. As our system needs to cope with dynamically changing environments, rather than going for the traditional approach of "one modeler for all possible contexts", our framework supports more than one Context-specific Modeler specifically designed for different contexts in the system. However, in order to avoid conflicting policies, at any specific time, only one such modeler will be utilized. By reasoning upon various system metrics, *Context Reasoner* is designed to determine the most appropriate Context-specific Modeler to date.

3.2. Requirements for Component Model

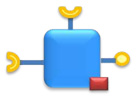
In ACCADA, a component represents a single unit of functionality and deployment. In order to achieve architecture-based run-time composition, a component model with the following attributes is needed:

Uniform management interface. As components are individually installed and configured by the system service, it is very important that a component could be managed through a uniform management interface. With this interface, components are reified in a uniform way for configuration, state monitoring and management. This approach enables system management services such as Event Monitor and the Adaptation Actuator to be designed in a generic way and used across multiple application domains. The following interface:

```
interface GeneralManagement {
    void instantiate(ComponentContext context);
    void destroy(ComponentContext context);
    void setProperty(String name, Property a);
    Property getProperty(String name);
    Enumeration < Property > getProperties();
};
```

supports life-cycle control and get/set component properties. It can be accessed via two different approaches—either accessing directly or mediated through an architectural layer which, apart from performing the requested actions, also coherently updates the status of the global software architecture. The first method is typical of application-based approaches to building software architectures, while the later characterizes architecture-based application constructions. In order to maintain a coherent and accurate global configuration, it is vital that this uniform management interface can only be accessed through architecture-exposed methods.

Component description and introspection. In order to support different types of components, a component model should be able to describe the component's distinguished stable characteristics. These features include a component's provided



```
<? xml version="1.0" encoding="UTF-8"?>
<drt:component name="decoder" desc="this is TV decoder"
enabled=”true”>
<implementation bincode="ua.pats.demo. TV.decoder"/>
<inport name="compressimage" interface="RTAI.SHM"
Type="byte" size ="80000” optional=”false”
bind="setimageSHM" unbind="unsetimageSHM"
/>
<outport name="decodedimage" interface="RTAI.SHM"
       type= "byte" size="600000"/>
<property name=" masterclock " type="string"
value="audio" />
<context-specific language=”UA.PATS.Language.SRDF”>
  <Priority> 2 </Priority>
  <ExecutionTime> 8000 </ExecutionTime >
  <Period>33300</Period>
  <Deadline> 12000 <Deadline>
</context-specific >
</drt:component>
```

Figure 5: Sample component configuration.

and required interfaces, component's properties as well as other constraints, for example, the type of CPU that is required. In order to expose this information, interface-based introspection and reflection are used in Fractal (Salehie and Tahvildari, 2009) and OpenCom (Michael et al., 2001; Coulson et al., 2008).In the latter, a general interface is designed for such knowledge discovery.

Instead of following these approaches, we use a concise management interface to control and capture the components' run-time state, while meta-data is applied to expose component-specific knowledge. Compared to interface-based introspection, meta-data based approach provides designers with a more light-weight and explicit solution. Interface-based component knowledge discovery means that a component's feature can only be known when it is initialized. As compared, meta-data approach enables components to be identified prior to their initialization; furthermore, this reduces the burden of component developers to implement those introspection interfaces, as meta-data already provides much design-time structural information. Those meta-data can be naturally abstracted from application design, validated in the verification process, and then reused during the run-time composition process. In this approach, a component design knowledge actually winds through its whole life-cycle. The ACCADA framework can dynamically discover and retrieve a component's structural characteristics as soon as they are installed into the system.

Figure 5 shows an excerpt of such meta-data. This component model is based on our previous work in the Declarative Real-time component model (DRCom) (Gui et al., 2008a). Due to page limits, we will not introduce the definition of each element.

3.3. Modeling Layer

As already mentioned, modeling the whole system architecture and making accurate adaptation decisions is a very complex process. That is especially true in our framework, as not only functional dependencies but also the context knowledge are considered in the composition process. These two aspects have been kept separated and assigned to what we call Structural Modeler and Context-specific Modeler, described in what follows.

3.3.1. Structural Modeler

As the application is constructed, configured and reconstructed during system run-time, how to derive the functional and structural dependency among components becomes one of the key problems in run-time component composition.

The Structural Modeler consists of several processes, the most important of which are:

Dependence Compatibility Check. This component first checks all the installed components dependence relationships. We say that a component is "structure-satisfied" when all its required interfaces (Receptacles) have corresponding provided interfaces. A component can only be initialized when it is structure-satisfied. This also guarantees component initialization orders. Depending on the adopted component model, different policies may be employed, such as the interface based matching(method call oritened) —used in the model of Declarative Service and Fractal model—or port-based matching(data communication oriented) as it is the case in DRCom model.

Specifically, in DRCom model, meta-data is used to describe the communication port between real-time tasks. Figure 5 shows a component that needs a data port called "compressimage".

Such a function is quite important for run-time composition as it provides a general matching service for possible functionally compatible components. This functions is indispensable in maintaining application architecture during system configuration changes.

Maintenance of application architecture (Reference update). As components will be installed and uninstalled during run-time, the issue of reference update during component exchange must be addressed. When one component is exchanged for another it is necessary to update the references that point to the old component such that they refer to the new one. Doing so is necessary to ensure that the program continues to execute correctly. For example, when a component is disabled, the modeler will first check whether another component with the same functional attributes exists. If such a candidate is successfully found, the modeler will repair the references between components to change the old references with the new one, and then destroy the invalid connections. Otherwise, all components which depend on this disabled component will also be disabled. All of these adaptations are performed during run-time without disabling the whole application. By having the system managing the run-time reference update, an application's architecture integrity can be preserved even in the face of configuration changes. This also facilitates the adaptation developers, as they do not need to maintain the applications' structure.

The Structural Modeler offers the following interface to provide its computation results of structure-satisfied components.

```
public interface StructureModelerResolver
 {     public List<ComponentConfiguration>
                resolveSatisfied(List ManagedComponentConfigurations);
 }
```

Such an interface takes a list containing all managed component configurations. Each time the Structural Modeler is inquired, it will check whether all the required interfaces from one managed component have their corresponding service provided. After this process, the modeler returns a list of all the components that satisfied functional constraints, denoted as satisfiedComponentConfigurations. All these components will be tagged by the Structural Modeler as "enabled". The rest of disabled component configurations will be computed by removing the satisfied components from the list of all managed components. This is done by executing `enabledComponentConfigurations.removeall(satisfiedComponentConfigurations)`.

In order to minimize overhead, the Structural Modeler does not propose all possible configuration — which normally includes components states as well as all connections between components, to the Context-Specific Modeler because the number of configurations will grow very fast with the number of managed components. Instead, it only sends a list of all structure-satisfied components to the Context-Specific Modeler. After Context-Specific Modeler and Structural Modeler collectively determine the status of each managed component, Structural Modeler repairs the reference after adaptation. This can effectively reduce adaptation execution time

Many run-time composition approaches, such as Servicebinder (Cervantes and Hall, 2004) and Perimorph (Kasten and McKinley, 2004), provide a similar layer to manage the references between components. However, lacking context information integration, this functional layer itself could not solve conflicts when several functional configurations are available. Indeed, different configurations could match different environmental conditions or platform characteristics. Such kind of ambiguity can only be handled with context knowledge.

3.3.2. Context-specific Modeler

As the Structural Modeler deals with the functional related constraints in building and maintaining the software architecture, the Context-specific Modeler deals with constraints related to the knowledge of context. All components that satisfy functional requirements will be further evaluated by context knowledge. As a result, the modeler will build a context-specific architecture model using its knowledge and adaptation strategy. This model will be checked periodically and/or upon request. If a constraints violation occurs, it determines the course of action and delegates such actions to the adaptations execution module.

```
public interface ContextResolver {
      public List<ComponentConfigurations>
            resolveSatisfied(List structuralenabledCC);
      public List<ComponentProperty>
       resolveProperties (List enabledComponentConfigurations);
}
```

Each Context Modeler will need to implement the ContextResolver interface which has two methods: A) resolveSatisfied takes all structure-satisfied components that were computed by the Structural Modeler and returns all the enabled components resolved by this modeler; all the components not in this list are set to "disabled"; B) resolveProperties returns a list of Properties and their new value. System run-time will invoke these two methods to get adaptation plans from selected modeler and send results to the Model Fusion module. This interface will be registered in the OSGi service registry together with their attributes which specify their target application domains. Here, *structuralenabledCC* is a list of all structure-satisfied component configurations, which provides a snapshot of current system global state. A component configuration contains information such as component description, reference cache, and instance references, therefore a Context Modeler can use that knowledge to track system changes such as the addition/removal of components or changes in properties.

In ACCADA, several context modelers with different context adaptation knowledge can be installed simultaneously. They must implement the same context modeler service interface with different attributes describing their target concerns. Such concerns could be for instance prolonging the mission life in case of low batteries, or maximizing the user experience when watching movies on a given mobile device. Service orientation enables the architecture to support different or future unpremeditated adaptation strategies. Another benefit from this approach is that one modeler instance just needs to deal with a fraction of all possible adaptation concerns. Compared to "one size fits all" approaches, our solution makes the modeler very concise, easy to implement and consuming fewer resources. By switching the context-specific modeler, the system architecture model as well as the adaptation behavior can be easily altered, which could be beneficial in matching different environmental conditions. For any Context-specific modeler, three adaptation actions can be taken: enable, disable, and setProperty. This design choice of using simple atomic actions simplifies the model fusion rule set (described in Section 3.3.4). At the same time, these adaptation actions can be used as a "base" to compose more complex adaptation actions (for instance, replacing a component can be realized by "enable new one" and " disable old one". Here, which Context-specific Modeler is to be used is determined by the Context Reasoner.

3.3.3. Context Reasoner

As several Context-specific modelers may co-exist at the same time, only one of them will be selected according to current system context. Context Reasoner and Context-specific modelers are implemented through the Sponsor-Selector pattern (Riehle et al., 1998). Context Reasoner selects the best match from a dynamically changing set of candidate Context-specific Modelers.

One simple interface is designed to return the best matched reference:

```
public interface Contextreasoner
{ public Resolver findCurrentAdaptor();
}
```

According to different system requirements, the reasoning logic may be as simple as e.g. using CPU status as decision logics, or as complex as using a semantic reasoning engine or some other artificial intelligence approach. By separating three kinds of responsibilities—knowing when a modeler is useful, selecting among different modelers, and using a modeler—it enables the software system to integrate new modelers, and new knowledge about modelers, at run-time in a way that is transparent to the user. In current implementation, a simple rating strategy based on context matching constraints as in (Fujii and Suda, 2009) is used to check the similarity of the current context with system resource constraints tagged with each Context-specific modeler, such as CPU, memory, disk space and user's preference. Such a rating based scheme is by no means the only selection strategy that our framework allows. Benefited from our service-oriented architecture, a third party developer can install their own implementation of Context Reasoners. Existing context-aware technologies, e.g., CroCo (Pietschmann et al., 2008), SOCAM (Gu et al., 2005) can be utilized to develop a customized Discovery service.

As Structural Modeler and Context-specific Modeler collectively build the system global architecture model, Context Reasoner helps the software system to adjust architecture model according to context switch. However, these two modelers may come out with different adaptation plans which need to be handled in the Model Fusion modules.

3.3.4. Model Fusion

In our framework, system architecture models are divided into different aspects to avoid possible conflicts. In fact the Structural Modeler only deals with functional parts while *Context-Specific Modeler* deals with the ambiguity that can only be solved by context knowledge. However, although they capture different features of a system, these two modeling processes may conflict with each other's structure or behavior, thus, the process of combining these two models is vital for system correctness. We call this process Model Fusion. The following rules are used:

a. A component can only be enabled when it satisfies both modelers' constraints. If so, it is marked as "enabled".
b. If a component is set to "disabled" by any of these two modelers, the final action will be "disabled".
c. A component's property will be set when it is in "enabled" state.
d. Then, the Structural Modeler proceeds to building the software architecture by using the remaining active components.

As it can be seen, model fusion module plays a key role in the system correctness. The rule-set specified here is built by the guideline of guarantying application functional dependence as first priority. It is by no means the only rule set. Some ill-designed Context-Specific Modeler can also create inconsistency during adaptation. In order to check the system's correctness and effectiveness, certain reflection service is supported in the design of our system run-time.

3.4. Reflective adaptation service

As our system supports future contextual modelers to be added into the adaptation process, some of them may give invalid tactics and strategies which may lead the system into an adverse state. In the model fusion module, a set of rules are provided to explicitly fuse two modeler solutions into one modeler; however, there is no guarantee that the fused modeler will perform as intended. How to evaluate the effectiveness and efficiency of the adaptation route computed by one fused modeler is still one of the major issues in our system design.

As system performance can only be evaluated according to the system's current context optimization goal, it is very unlikely to have a general policy to deal with so many domain/context-specific requirements. Rather than providing a solution for one specific context, our software system provides services to exposing a snapshot of the system internal state. For instance, the enabled components list, currently used *Context-Specific Modeler*, and the adaptation actions proposed by each modeler can be exposed and e.g. logged for future analyses. The system will also send asynchronous events to the external listeners when certain adaptation actions are taken. Third party programs can selectively register these interfaces and listen to those actions that might be important to them. Such information can help system users evaluate the effectiveness of various modelers as well as the Context Reasoner. For instance, a "Ping-Pong" effect among several Context Modelers in a short time might indicate an ill-designed Context Reasoner.

3.5. Management Layer

This layer provides an abstract interface to manage the interactions between modeling layer and component instances. It consists of two main elements: Event Monitor and Adaptation Actuator. Event Monitor tracks installed components' state changes as well as probes the measured attributes across different system components. The value of a managed component's attribute can be retrieved via the getProperty() methods. Depending on the particular system, Event Monitor can be tailored to specific attributes. For instance, in the DR-Com model—whose service scenario is embedded real-time applications—it is important to get real-time task information such as the components' execution time, number of overruns, etc.

The Adaptation Actuator implements the atomic adaptation actions that the system can take. This can include actions that manage a component's life-cycle state—start, stop, pause, resume—as well as property manipulations via function setProperty(...), for example, changing the computation task's priority, period, etc. The uniform management interface simplifies the design of the actuator as the actions can be taken in a general way. It can be easily reused in different component models. To avoid synchronization problems, a queue based task dispatch mechanism is designed to perform such atomic adaptation actions.

3.6. General adaptation process

The above six key modules residing in the modeling and management layers are orchestrated so as to form an external control loop across different application domain. When a significant change has been detected, the modeling layer is notified to check whether existing constraints are being violated. Algorithm 1 describes the general adaptation process.

```
Algorithm 1: General adaptation process
Requires: An architecture-based management system
          with context-specific adaptation logic
Ensure:   Keep constraints satisfied in the face of changes, both
      functional as non-functional, through Context-specific knowledge
1. A system change triggers adaptation process
2. Structural Modeler gets the set of satisfied
   components in terms of functional dependence
3. Context reasoner returns Context-specific modeler's reference
4. The selected Context-specific Modeler builds an adaptation plan
5. Structural modeler merges two adaptation plans
6. The Adaptation Actuator executes the adaptation plan
```

In our framework, adaptation is carried out step-wise. The adaptation process showed here is only one round of adaptation for the whole adaptation process. Any adaptation actions taken by different modelers might change other component status. For instance, in Fig. 2, after the Transcoder is disabled, the system will raise a Service.Unregistered event for Transcoding service. This event will raise another round of adaptation. In Section 4.2, a practical example will be given to demonstrate a whole application constructing process.

4. CASE STUDIES

This section describes a case study addressing Quality of Experience (QoE) based adaption (Alben, 1996). The basis of this case study is the system described in Section 2.2. The goal of an automatic adaptation is in this case to maximize user's QoE by always protecting those applications with highest impact on the user experience. We examine this context knowledge aided adaptation and the process to execute the adaptation plan.

4.1. QoE based adaptation strategy

As we discussed in Section 2.2, applications may interfere with each other while competing for system resources—in this case, the TV and transcoding applications could not run simultaneously due to lack of system resources. In order to maximize user's TV watching experience, the resources demanded by the TV application should be guaranteed. Thus, the context tactics can be expressed as the following invariant: at any time, enough resources must be available to the TV application.

When these two applications run simultaneously, the Event Monitor notices that decoder module's overruns are increasing. It notifies the Modeling Layer for possible changes. In this case, the Structural Modeler will not find any structural violation as both applications are functionally well constructed. Thus the satisfied functional model is sent to the Context-Specific Modeler.

As described in Alg. 2, the Context-Specific Modeler checks any violation in terms of context-specific constraints. In this case the number of overruns is checked and the violations of such constraint are detected. Then, the Context-Specific Modeler tries to disable the most costly components (except those related to the TV application) until TV decoding task overruns stop increasing.

```
Algorithm 2: Guarantee Quality of TV application
Requires: Installed components' CPU usage information with
          context-specific adaptation logic
Ensure: Allocate enough CPU time to the TV application
 1. If Decoder.getProperty(overrun) increase $\geq$ Max_threshold
 2.     for all cmp in SatisfiedComponents do
 3.        if component not from TV application
 4.           cmp.getProperty(CPUconsumption) from cmp attributes
 5.           record the cmp with highest CPU usage.
 6.        end if
 7.     end for
 8.     remove cmp from enabled component list(ecl)
 9. end if
10. send ecl back to Structural Modeler
```

After this adaptation process, other actions may also be triggered as a consequence of these adaptation actions. Here, after Transcoder is disabled, the File writer will be disabled by the Structural Modeler, as it functionally depends on the Transcoder components. After these state changes, the reference between enabled components will also be updated by the Structural Modeler as described in Section 3.3.1. The component state after adaptation is shown in Fig. 2, in which different colors correspond to different component states.

4.2. Context-aware application composition

In this section, we will show how the Context-Specific Modeler helps to select the most appropriate components by using the application shown in Fig. 3. In order to install such an application, the system will operate as follows:

1. Install all 5 components.
2. Structural Modeler will find that File Reader is structure-satisfied, thus Context-Specific Modeler will set it to “enabled”. Then “File Reader” will be enabled and will register its provided service interface into system registry via system run-time;
3. As the File Reader registers its functional interface, it triggers a service.registered event. The Structural Modeler now finds two structure-satisfied Transcoders and proposes them to the Context-Specific Modeler. Depending on current context adaption rules, the Context-specific modeler will select the Transcoder with high quality and set it to “enabled”. Then its service interface (TranscodedStream) will be registered in the registry.
4. Registration of new service will trigger another round of adaptation. Two File writer components which both require the transcoding component provided service will then become “structure-satisfied”, so the Context-Specific Modeler can now determine which one of them is to be enabled.
5. After the selected File writer is enabled, no new component state change will be issued by either Structural Modeler or Context-specific modeler. This round of Adaptation finishes.

This scenario shows how the Context-Specific Modeler is used to reduce the composition ambiguities. Our stepwise based adaptation can also effectively reduce the composition complexities compared with approaches from Fujii and Suda, T. (Fujii and Suda, 2009). In this approach, a modeler proposes all the possible combinations (connections and components status) to the system run-time and select one configuration using its customized learning or rule-based algorithms. This approach will have scalability problems as all possible combination might grow exponentially with the number of managed components. For instance, for this application, 4 different combinations need to be verified while only two steps of selection are needed for selection in our approach.

5. IMPLEMENTATION and SIMULATION

In this section, we will discuss our implementation to achieve a context-specific architecture-based adaptation. This framework has been validated both from a qualitative and a quantitative point of view including such concerns as implementation complexity, adaptation flexibility, memory usage, etc.

5.1. System implementation

Our implementation[2] uses the OSGi framework as our underlying development platform. OSGi (OSGi Committee, 2008) technology serves as the platform for universal middleware ranging from embedded devices to server environments. Equinox, a popular, free, open source OSGi Platform developed by the Eclipse organization, is used as our basic development platform. In the current state, our implementation focused on providing a light-weight implementation for local applications managements. In principle, our framework can be extended to distributed adaptation environments by using e.g. R-OSGi (Remote OSGi) support. However, issues related to that extension, such as coordination between modelers, data ordering, synchronization etc. are still to be tackled and are the matter for our future work.

Modules	**Sub-modules**	**Lines of code**	**Binary size** (bytes)
Monitoring	Reflections of code	142	2353
	Monitoring	354	7407
Structural Modeler	Functional constraints	200	15230
	Reference management	249	5328
Adaptation actuator	Dispose management	459	11782
	Instance management	280	6714
Context-specific Modeler	Simple CPU constraint adaptor	108	3798
Context Reasoner	Simple context match	90	2620
Parsing	Model classes	1329	2353
	Parser class	1450	36000
Auxiliary code		500+	

Table 1: Lines of code for key architectual modules.

For the component model, we use the DRCom model proposed in our previous work (Gui et al., 2008a,b). DRCom was originally designed for the construction of dynamically configurable & reflective real-time systems.

The DRCom model has a basic management interface enabling a uniform and coherent way to control a component's life-cycle and attributes. Each component is associated with a meta-data file which enlists its abstractions—for instance, interface for communication, attributes as well as reference constraints.

[2]Source code can be download at https://sourceforge.net/projects/s-transformer/

Each component is tagged with task related information for configuration and validation.

As discussed in Section 3.1, this system is implemented via five key modules. The lines of code of each of the implemented modules are shown in Table 1. Our framework also provides such mechanisms as deployment support and version control by simply reusing the OSGi system service, which leads to a lean and quite concise implementation.

One of the basic services in our system is Meta-data Parsing. This module parses the meta-data and stores it in the form of meta-data objects. A simple component meta-data language is defined to describe component characteristics. This component model designs an extensible XML format that supports future more complex description languages: Due to page limits, here we will not go into details. Clearly the complexity of Context Reasoner and Context-Specific Modeler is highly implementation specific, thus the lines of code listed here are just the simple implementations for our TV scenario described in Section 4.

5.2. Adaptation to different context

In the traditional approach towards application-based adaptation, in order to achieve adaptation matching different context requirements, developers normally need to reprogram the whole adaptation architecture. There are, to name but a few, modules for detection, modules for component management, adaptation logic as well as the execution modules.

	Application adaptation	**ACCADA**
Adaptation logic	Prefixed	Change in runtime
Context knowledge integration	Static/Internal	Flexible/ Architectural
Implementation complexity	High	Low
Multi-context support	NA or static	Yes and flexible
Context-specific adaptor implementation	Complex	Concise
Separation of design concern	Mixed	Yes
Level of adaptation	Inside specific application	Across several applications

Table 2: Application-based vs. Architecture-based Adaptation.

However, during context changes, only the adaptation strategy should be altered to express the context-specific knowledge and all the remaining modules can be kept largely unchanged. Without the burden to implement software maintenance tasks, a context-specific adaptor can be implemented very concisely. For instance, our adaptation to guarantee the QoE of the TV application can be implemented in less than 120 lines of codes. On the other hand, an ad-hoc approach would require re-implementing a new version of a basic component management run-time (in our case, about 2000 lines). Thus, programmers can focus on adaptation logic rather than having to take care of those low level details. Table 2 shows the comparison between application specific adaptation approaches and our framework.

Certain component frameworks provide tools to help programmers to automatically generate auxiliary codes. Examples include Juliac[3] —a Fractal toolchain backend, which generates Java source code corresponding to the application architecture specified by the designer. Such code includes membrane source code, a framework glue code and a bootstrapping code. In the following section, we compare our approach with Juliac's.

5.3. Comparison with Juliac

In the Juliac approach, ADL language is used to generate the glue code and the codes for introspection. The simplest "hello world" example uses two components, Client and Server. The Client will try to invoke the service exposed interface to print the "hello world" string. The reason for selecting Juliac for comparison is that it is a typical application based development platform. It is readily available and well documented and also based on the Java language.

	Application size	**Lines of code** (business)	**Lines of code** (generated)
Juliac	95.7 KB	100	3500
ACCADA	4.7 KB	140	0

Table 3: Comparing ACCADA with Juliac (Lines of code).

Table 3 shows that, for such simple application with only two functional components, the business code is about 100 lines, including import and interface definitions. With Juliac, about 3500 lines of Java codes will be generated. Such code mainly contains the glue code and basic system run-time. In comparison, in ACCADA, no process for off-line auxiliary code generation is needed, as it dynamically manages component's reference together at run-time. In our framework, an application mainly contains its business code, simple and easy to manage.

5.4. Architecture performance

To evaluate the performance of our implementation, we instrumented a test to measure the time for fetching, parsing, reference management, and configuring.

[3]Available at http://fractal.ow2.org/

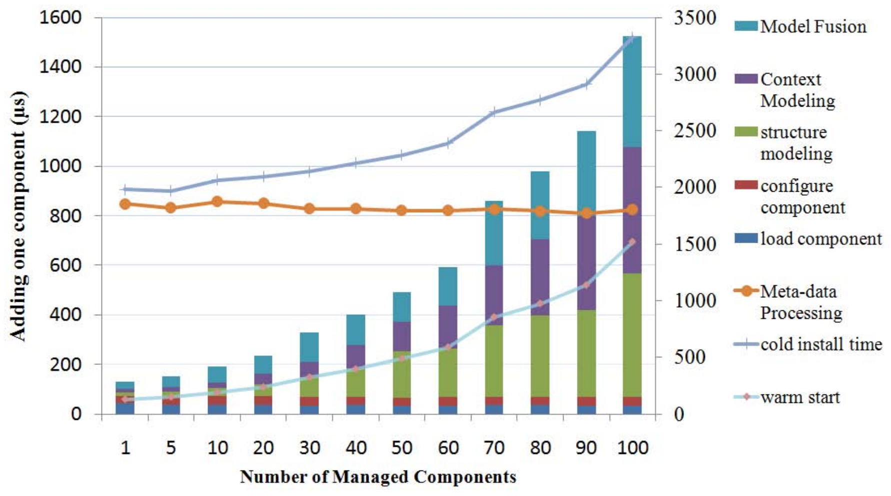


Figure 6: Framework performance on adding one component.

We focused on the time for installing a single component as we vary the number of components managed by the framework. Here, each component has one "in port", one "out port", and one attribute. The size of each component is the same—20.6 KB.

Here, as we manage almost 100 components, we use a different Context Modeler with respect to the one described in Section 4. The modeler performs a simple admission control policy:

$$\sum \frac{\text{Execution time}}{\text{Execution period}} < 1 \quad \text{for all enabled components.} \tag{1}$$

The arrival component will only be enabled when (1) holds true. In order to best test framework's performance, all these component execution tasks remain disabled during the experiment. Here, as we are only interested in the framework performance, in the newly installed component, the component initialization time is not counted as it may vary with the implementations.

For the hardware platform, we use a HP NC6400 laptop with 1.66 GHz dual core T5500 CPU, 3GB RAM and 80 GB 7200RPM HDD. The JVM we adopted is Sun JAVA 1.6.0.2 SDK on Windows.

Installing a new component normally consists of five main steps: component loading, meta-data processing, structural modeling, context modeling, and model fusion. Figure 6 shows the absolute times spent in each step of the process. Each value is the arithmetic mean of 300 runs of the experiment. In order to better illustrate the trend of different steps, we use two Y direction axes in expressing data. Values in stacked column use the main Y axis (left) and those values in marked lines use the secondary Y axis (right). The time scale used in both axes is microseconds (μs).

As we can see from Fig. 6, component installation time grows with n, the number of system managed components. This is mainly due to the fact that two key elements—component loading time and meta-data processing time, which counting for more than 60% of the total processing time—keep comparably stable when n grows. In contrast, the other three key elements, the structural modeling, context modeling and actuation will increase approximately linearly with n. The structure modeling process mainly deals with matching composability between installed components, which has computational complexity $\mathrm{O}(n)$. The context modeling process will check whether the new component can satisfy the resource requirements. This process is also $\mathrm{O}(n)$ (stateless implementation, no optimization). Here, the model fusion process includes time for post-processing the modeling results from two modeling processes.

These operations, implemented by using the set operations from the Java Vector class, grow gradually with the number of managed components (n). For instance, the average execution time of the "addall" operation is $\mathrm{O}(n \log n)$ while the worst execution time is $\mathrm{O}(n^2)$.

As most of the installation time results from the large meta-data processing task, we optimized the installation process by parsing a component's meta-data prior to its usage (without initiating the component). We call this a "warm-start". Compared to normal "cold" installation process, this approach can greatly reduce a component's response time—about 2000μs meta-data processing time can be spared. Of course, this comes at the cost of memory usage. Each installed component will need about 4K memory to store component's information.

Simulation results show that our framework scales well when the number of managed components grows to a hundred of components. However, the context modeling time confines to the simple algorithm described here. Other more complex reasoning policies may not perform well when the number of reasoned components grows. This is highly policy dependent which is out of the scope of this paper. The Context Reasoner also exhibits similar characteristics.

5.5. Simulation results

In order to validate this framework from both a qualitative and a quantitative point of view, we implemented the scenario described in Section 2.2 and the adaptation algorithm as described in Section 4. Hardware and software configuration is the same as in the previous section. In order to support real-time task, the software system is run on the Fedora 7 kernel 2.6.23 with Real-Time Application Interface (RTAI) version 3.6—a real-time patch for Linux (E. Bianchi, 2006).

As shown in Fig. 2, all six modules' execution parts are implemented as periodic tasks. The Decoding component of TV application is implemented with a real-time task with period 33.37ms with priority 2 (the smaller the value, the higher the priority), execution time of about 8ms and deadline of 12ms. The execution part of Transcoder module is implemented as a real-time task with period 50ms, priority 1, execution time of approximately 10ms and a deadline of 30ms. The schedule policy used by the underlying RTAI system is FIFO. In order to show the interference between these two components only, all other components use priority 6 and make use of asynchronous communication only, so that we can focus on the performance of the two coding modules.

As the video decoding execution time may vary according to the contents of video streams, in order to more clearly demonstrate the mutual influence among applications, we substituted the decoding function with a mock function using comparably constant CPU time for each round of calculation.

We performed 6000 observations for the execution time of Decoding Module after system enters a stable state. Figure 7 shows the execution time distribution with and without the Context-Specific Modeler. The time scale for execution is μs. In the former case, the execution time of the decoder is typically around 8000μs, while the longest execution time is about 8180μs when the context knowledge is used (as this disables the transcode module). In contrast, if no context-specific knowledge is available, the system will try to run these two applications simultaneously.

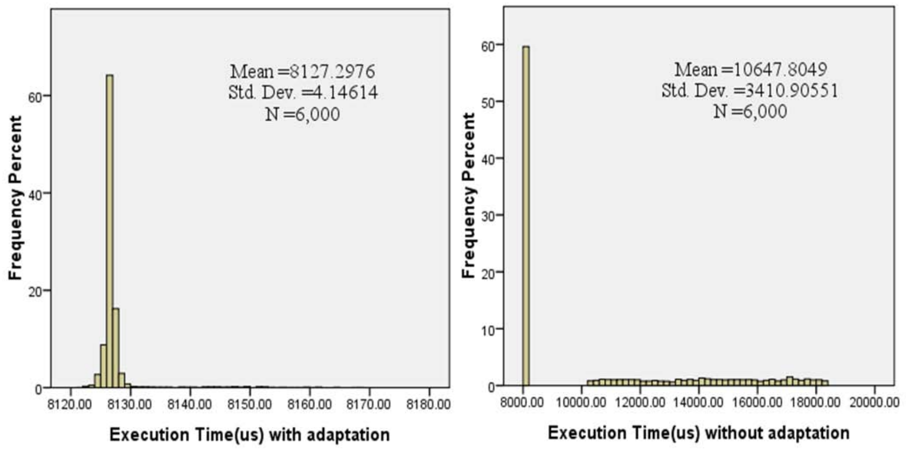


Figure 7: Execution time (Adaptation vs. no-Adaptation).

The standard deviation of the execution time is in this case 3410.9μs, much higher than 4.15μs—the one with adaptation. The jitter of decoder task's execution time is very big, as about 31.3% (1880/6000) of the runs exceed the deadline specified in the component's meta-data (12ms). This can result in a large number of lost frames.

As can be seen from Fig. 7, with context-specific adaptation knowledge, the decoding modules can achieve much better performance in term of mean execution time and standard deviation.

6. RELATED WORK

Our ACCADA framework builds on our earlier work on the DRCom component model. Compared to this earlier work, our framework exhibits a richer and more coherent set of features to support context-specific knowledge. Towards the requirements of context-specific run-time component composition, a complete new modeling layer is introduced and implemented.

JMX (Sun Microsystems, 2008) is a Java technology that supplies tools for managing and monitoring applications, system objects, devices (e.g. printers) and service-oriented networks, in which the lower layer is composed of components, called MBeans, representing the Java objects to manage. According to the requirements presented in this paper, the main limitation of JMX is that it does not have a complete component model. In particular, the dependencies between the managed objects are not exposed and managed. Thus an application can only be constructed via its own property methods.

SmartFrog (Anderson et al., 2003) is a framework for the management of configuration-driven systems. It defines a system as a collection of software components with certain properties (i.e., attributes). The framework provides mechanisms for describing these component collections and for deploying and managing their life cycle. The SmartFrog component model defines the interfaces that a software component should implement (or that can be provided by a management adapter). However, the approach lacks support on how to manage the context-specific adaptation. Furthermore, the description of components is static and cannot support non-functional properties and constraints.

Sicard et al. (2008) identify novel requirements for reflective component models for architecture-based management systems. The construct layer is designed for meta-data checkpoint and replication. Interfaces and processes for self-repairing are defined, e.g. life-cycle management, setter/getter interface as well as meta-data based configuration. A faulty component can be repaired by restoring its state and all the meta-data information outside of the component instance. However, this approach does not have a clear definition and separation between system services. Its hardwired architecture makes it difficult to reuse the framework across different contexts.

Garlan et al. (2004) propose a general architecture-based self-adaptation framework. The Rainbow framework uses software architectures and a reusable infrastructure to support self-adaptation of software systems. The use of external adaptation mechanisms allows the explicit specification of adaptation strategies for multiple system concerns and domains. However, their approach lacks of component composition support which is also important in building applications.

In order to deal with component dynamicity, Cervantes and Hall (2004) propose a service-oriented component based framework for constructing adaptive component-based applications. The key part of their framework is the Service Binder which automatically controls the relationships between components. Our approach mimics theirs in dealing with component's dynamicity. However, the static nature of their adaptation policy and resolving process limits the usage in changing environments.

Kasten et al. propose the Perimorph framework to achieve run-time composition and state management for an adaptive system (Kasten and McKinley, 2004). It enables an application designer to quantify and verify collateral changes in terms of factor sets. However, due to lack of a clearly defined component model, it is hard to extend their approach to cross applications adaptation. Their approach also does not consider how to integrate the context adaptation knowledge.

In order to handle complex dependencies between components, Kon et al. (2005) propose an integrated architecture for managing dependencies in distributed component-based systems. Their architecture supports automatic configuration and dynamic resource management in distributed heterogeneous environments. Their approach provides the explicit context-knowledge support in dealing with component dependence. However, how to support the context changes is not specified in their approach.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have described our approach to continuous context-aware deployment and adaptation. We have shown in particular how to integrate context-specific knowledge in run-time component composition. By designing a uniform management interface, our architecture provides a unified programming model over a wide range of components. The design time knowledge is maintained as meta-data and reused during run-time component composition. A service-oriented model is used in implementing architecture basic modules thus achieving a more flexible system architecture. This framework is easily configured to fit with different contexts. Furthermore, our approach has been compared with other application-based adaptation frameworks with respect to aspects such as lines of code and memory usage. Although our experience was done based on the OSGi middleware, we believe our findings to be general to architecture-based management systems using reflective component models.

Although we showed an integrated approach to providing context-specific run-time composition, areas remain open for further research:

Firstly, the application is globally constructed, which means that when the number of installed components increases, the performance of adaptation decreases accordingly. In order to optimize the reasoning process, we are working to add "scopes" in component description. By partitioning component instances into blocks, component searching and reason scope can be greatly reduced.

Second, as our system enables future context-specific reasoners to be added into adaptation process, some of them may give invalid tactics and strategies which may bring the system into adverse state. How to prevent our framework from indiscriminately accepting this plan is a part for our current research. We are working on designing an auditing module which tests and evaluates the possible effects of the outcome and prevents spiteful actions from being taken.

Acknowledgment

We express our gratitude to the anonymous reviewers, whose suggestions and remarks greatly enhanced the quality of our article.

Alben, L., 1996. Quality of Experience. ACM Interactions 3, 11–15.

Anderson, P., Goldsack, P., Paterson, J., 2003. Smartfrog meets lcfg: Autonomous reconfiguration with central policy control. Usenix Association Proceedings of the Seventeenth Large Installation Systems Administration Conference, 213–222.

Cervantes, H., Hall, R. S., 2004. A framework for constructing adaptive component-based applications: Concepts and experiences. Component-Based Software Engineering 3054, 130–137.

Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., Sivaharan, T., 2008. A generic component model for building systems software. ACM Transactions on Computer Systems 26 (1).

Dey, A. K., Abowd, G. D., Salber, D., 2001. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction 16 (2-4), 97–+.

E. Bianchi, L. Dozio, P. M., 2006. RTAI programming guide.

Feiler, P. H., Lewis, B. A., Vestal, S., 2006. The SAE architecture analysis and design language (AADL) a standard for engineering performance critical systems. 2006 IEEE Conference on Computer-Aided Control System Design, Vols 1 and 2, 302–307.

Fujii, K., Suda, T., 2009. Semantics-based context-aware dynamic service composition. ACM Transactions on Autonomous and Adaptive Systems 4 (2).

Garlan, D., Cheng, S. W., Huang, A. C., Schmerl, B., Steenkiste, P., 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37 (10), 46–+.

Gu, T., Pung, H. K., Zhang, D. Q., 2005. A service-oriented middleware for building context-aware services. Journal of Network and Computer Applications 28 (1), 1–18.

Gui, N., De Florio, V., Sun, H., Blondia, C., 2008a. A framework for adaptive real-time applications: the declarative real-time OSGi component model. In: The 7th Workshop on Adaptive and Reflective Middleware (ARM 2008). Leuven, Belgium.

Gui, N., De Florio, V., Sun, H., Blondia, C., 2008b. A hybrid real-time component model for reconfigurable embedded systems. In: ACM symposium on Applied computing. Fortaleza, Ceara, Brazil, pp. 1590–1596.

Gui, N., Florio, V. D., H.Sun, C.Blondia, 2009. Accada: A framework for continuous context-aware deployment and adaptation. The 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009) Volume 5873/2009, 325–340.

Hall, R. S., Cervantes, H., 2004. Challenges in building service-oriented applications for OSGi. IEEE Communications Magazine 42 (5), 144–149.

Kasten, E. P., McKinley, P. K., 2004. Perimorph: Run-time composition and state management for adaptive systems. 24th International Conference on Distributed Computing Systems Workshops, Proceedings, 332–337.

Kon, F., Marques, J. R., Yamane, T., Campbell, R. H., Mickunas, M. D., 2005. Design, implementation, and performance of an automatic configuration service for distributed component systems. Software-Practice & Experience 35 (7), 667–703.

Michael, C., Gordon, S. B., Geoff, C., Nikos, P., 2001. An efficient component model for the construction of adaptive middleware.

OMG Committee, 2007. CORBA component model v.4.0.

Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., Wolf, A. L., 1999. An architecture-based approach to self-adaptive software. IEEE Intelligent Systems & Their Applications 14 (3), 54–62.

OSGi Committee, 2008. OSGi service platform, release 4.1.

Pietschmann, S., Mitschick, A., Winkler, R., Meissner, K., 2008. CroCo: Ontology-based, cross-application context management. Third International Workshop on Semantic Media Adaptation and Personalization, Proceedings, 88–93.

Riehle, D., Buschmann, F., Martin, R. C., 1998. Pattern languages of program design 3. Software patterns series. Addison-Wesley, Reading, Mass.

Salehie, M., Tahvildari, L., 2009. Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems 4 (2).

Seinturier, L., Pessemier, N., Duchien, L., Coupaye, T., 2006. A component model engineered with components and aspects. Component-Based Software Engineering, Proceedings 4063, 139–153.

Sicard, S., Boyer, F., De Palma, N., 2008. Using components for architecture-based management. 2008 30th International Conference on Software Engineering: (ICSE), Vols 1 and 2, 101–110.

Sun Microsystems, 2008. Java(TM) Management Extensions (JMX TM).