ACCADA: A Framework for Continuous Context-Aware Deployment and Adaptation

Ning Gui Vincenzo De Florio Hong Sun Chris Blondia

PATS group, University of Antwerp, Belgium
and IBBT, Ghent-Ledeberg, Belgium

{ning.gui, vincenzo.deflorio,chris.blondia}@ua.ac.be

Abstract. Software systems are increasingly expected to dynamically self-adapt to the changing environments. One of the principle adaptation mechanisms is dynamic recomposition of application components. This paper addresses the key issues that arise when external context knowledge is used to steer the run-time (re)composition process. 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 done based on the OSGi middleware, we believe our findings to be general to other architecture-based management systems.

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

1 INTRODUCTION

Software systems today increasingly operate in changing environments and with diverse user needs, resulting in the continued increasing complexity for managing and adapting these systems. As a consequence, software systems are increasingly expected to dynamically self-adapt to accommodate resource variability, changing user needs, and system faults. However, mechanisms that support self-adaptation currently are hardwired within each application. These approaches are often highly application-specific, static in nature, and tightly bound to the code. Being static, such mechanisms can hardly cope with dynamic context changes. 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 single application scope, architecture-based adaptation frameworks are proposed in [1] [2] to handle the cross system adaptation. Rather than scatter the adaptation logics in different applications and represent 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 consists in using a component-based technology to develop management system and application structure [3-6] .

However, in traditional approaches, design-time knowledge for application structure is largely lost during the off-line application construction process. Without this knowledge, it is nearly impossible for external engines to effectively change a application' structure with the assurance that the new configuration would perform as intended. On the other hand, Integration of external context knowledge[1] to application becomes very difficult 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 framework, 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 and those constraints 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-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 both from qualitative and a quantitative point of view. Simulation results show the soundness of our implementation in term of line of code, memory and adaptation capabilities.

The rest of the paper is organized as follows. Section 2 exposes our design methodology and a context-specific management framework and those challenges in realizing this framework. Section 3 presents the structure of our management framework as well as the component model and construction process. The ideas exposed in this paper have been validated by a set of comparison from different aspects in Section 4. Related work is discussed in Section 5, and we conclude in Section 6.

2 Architecture-Based Adaptation

Architecture-based adaptation is proposed to deal with cross system adaptation. In principle, such external control mechanisms provide a more effective engineering

[1] By context, we refer to [7] and define it as "any information that characterizes a situation related to the interaction between humans, applications and the surrounding environment."

solution with respect to internal mechanisms for self-adaptation because they abstract the concerns of problem detection and resolution into separable system modules[2]. However, systematic support for multi-context knowledge integration is largely missing. An important contribution of this paper is the designing and developing architectural principles and design patterns to integrate different context-specific knowledge into architecture-based adaption framework. We design the context-specific application methodology to better support run-time component composition,

2.1 Context-specific application construction 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 Figure 1, represents a procedure which tries to incorporate the functional design information with context concerns in compositing 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 traditional approach, 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.

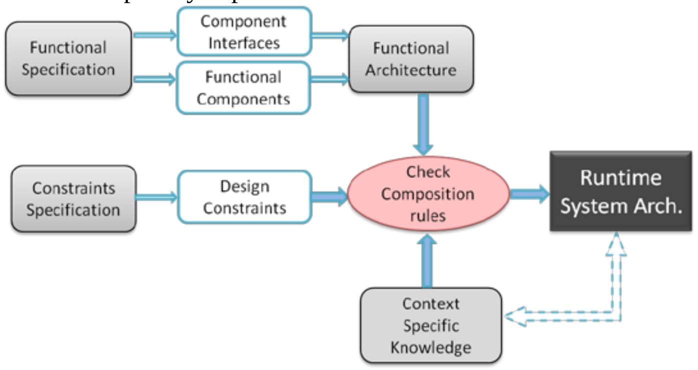

Fig. 1. Context-specific Application Construction Flow

As an application is constructed during run-time, in order to achieve correct and pointed adaptation, a set of constraints must be maintained. In this process, three main aspects constraints 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 be within a certain range. 3) context-specific knowledge, which specifies the domain related information and adaptation strategy should also hold valid after adaptation process.

As the dashed arrow points out, a managed application is continuously restructured and evolved according to context switches. The combined knowledge enables automatically run-time verification for constraints from various aspects which allows the system to change the software structure according to its hosting environment and without violating constraints from these three aspects.

2.2 Motivation 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 which can be harnessed to do something for you for temporal or long-term base. The open-system approach implies that a set of new applications will be installed into the host devices without thoroughly system analysis.

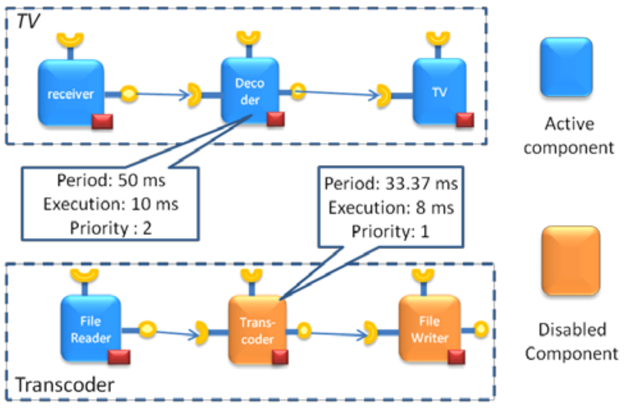

Fig. 2. QoE adaptation Demonstration

As an example, let us consider a Set-top device with Open platform support. The basic application of such device is TV processing. In brief, this application will received streaming video data from remote server, decode this data and output it to the TV port. As an open system, Set-top can also install certain applications to enhance its usability. For example, a user can install a new application which transcode 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 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 possibly transient timing problems of TV decoding task including missing frame, 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 can choose to disable the computationally intensive transcoding component and reserve enough system resources for 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.

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.

In our framework, applications are run-time composed from a set of managed component instances. Context-specific adaptation is achieved by dynamic (re)composing application components according to context knowledge.

3.1 Architecture-based management framework

Figure 3 shows our ACCADA architecture for adaptation. As can be clearly seen from that picture, 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 Selector*. 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) violation, 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. It sends notifications to trigger a new round of the adaptation process. The possible source of adaptation may include, for example, a new component being installed or the CPU or memory utilization reaching a status that may have significant effect on the existing system configuration. It could also be a simple Timer that triggers periodically at certain time intervals. 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 lifecycle 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 builds the system architectural model according to the changing contexts(Section 3.3)

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 environment changes. By using the Divide and Conquer principle, we assign the management of these two aspects to two different modules to more effectively deal with two different requirements – software architecture management and context-specific knowledge integration.

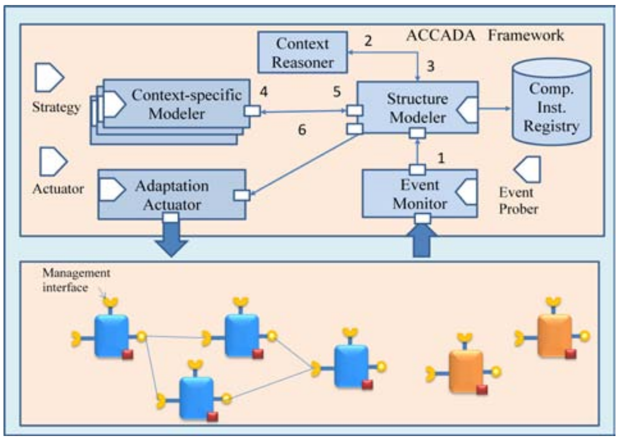

Fig. 3. ACCADA framework

One module, the *Structural Modeler*, manages functional dependences between installed components – checking whether the required and provided interfaces are compatible – and maintains 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 those functional dependence managements, the traces of all the adaptation steps performed is also monitored and exposed for later analysis. The other set of modules are the *Context-specific Modeler*. In order to cope with dynamically changing environments, rather than going for the traditional approach of "one adaptor for all possible contexts", our framework supports more than one *Context-specific Modelers* specifically designed for different contexts in the system. By reasoning upon various system metrics, *Context Selector* is designed to determine the most appropriate *Modeler* to date.

Service-oriented Approach: In order to achieve more reusability 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, a candidate service provider can be easily interchanged during system run-time. In doing so, many existing and/or future more sophisticated context adaptation policies can be plugged into our framework.

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 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 management interface supports lifecycle 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. 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. Our previous work provides the detailed design of such interface.

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 and required interfaces, component's properties as well as other constraints, for example, the type of CPU that is required. Interface-based introspective and reflective component models are proposed in Fractal [8] and OpenCom [9, 10], in which a general interface is designed for such knowledge discovery.

Instead of following these approaches, a concise management interface is used to control and capture the components' run-time state, while meta-data is applied to expose component-specific knowledge. Compared to the interface-based introspection, it provides designers with a more light-weighted and explicit solution. Compared to Interface-based approach, 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 lifecycle. The ACCADA framework can dynamically discover and retrieve a component's structural characteristics as soon as they are installed into the system.

In our previous work in the Declarative Real-time component model (DRCom), a simple declarative language is designed. Please refer [11]for details.

3.3 Modeling Layer

As already mentioned, modeling the whole system architecture and making pointed adaptation decisions is a very complex process. That is especially true in our framework, as not only functional dependences 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 relationship. A component can only be initialized when all its required interfaces (Receptacles) have corresponding provided interfaces.. This also guarantees component initialization orders. According to different component model, different policies may be employed, such as the interface based matching – used in the model of Declarative Service and Fractal model – or data communication matching as it is the case in DRCom model.

Such function is quite important for run-time composition as it provides a general matching service which is indispensable in maintaining application architecture during system configuration changes.

Maintenance of Application architecture (Reference update): As component 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 firstly 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 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.

Many run-time composition approaches, such as Servicebinder [12] and Perimorph [13], provide similar layer to manage the references between components. However, without context information integration, this functional layer itself could not solve conflicts when several functional configurations are available. 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 on request. If a constraints violation occurs, it determines the course of action and delegates such actions to the adaptations execution module.

In ACCADA, several context modelers with different context adaptation knowledge can be installed simultaneously. They implement the same context modeler service interface with different attributes describing their target concerns. Such concerns could be e.g. prolonging mission life in case of low batteries, or maximizing 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" approach, our solution makes the modeler very concise, easy to implement and consuming fewer resources. By switching 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. Here, which Context-specific Modeler is to be used is determined by the Context Selector .

3.3.3 Context Selector

As several Context-specific Reasoners may co-exist in a specific time, only one of them will be selected according to current system context. It will return an active context modeler "best matching" the current environment. According to different system requirements, the reasoning logic may be as simple as 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, new modelers can be integrated into software system in a way that is transparent to users. One simple interface is designed to return the best matched reference:

```
public interface ContextSelector
   { public ContextAdaptor findCurrentFitAdaptor();  }
```

3.4 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. Adaptation Actuator implements the atomic adaptation actions that the system can take. This can include actions that manage a component's lifecycle state – start, stop, pause, stop – as well as properties manipulations via setProperty(…), for example, changing the computation task's priority, period… The uniform management interface simplifies the design of the actuator as the actions can be taken in a general way and can be easily reused to different component models

3.5 General adaptation process

The above five 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 Selector returns Context-specific modeler's reference
4. The selected Context-specific Modeler builds an adaptation plan
5. Structure modeler merges two adaptation plan
6. The Adaptation Actuator executes the adaptation plan

4 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.

4.1 System implementation

Equinox, a popular, free, open source OSGi Platform developed by the Eclipse organization, is used as our basic development platform. In current state, our implementation focused on providing a light-weight implementation for local applications managements. However, it can be generally extended to distributed environment via using R-OSGi (Remote OSGi) support. We use slightly revised DRCom model[11, 14]. It was originally designed for the construction of dynamically configurable & reflective real-time systems.

Table 1. Lines of code for Architecture-based adaptation

	Functions	Line of code	Binary size (byte)
Monitoring	Reflections of code	142	2353
	Monitoring	354	7407
Parsing	Model class	1329	2353
	Parser class	1450	36000
Structural Modeler	Functional constraints	200	15230
	Reference management	249	5382
Adaptation executor	Dispose management	459	11782
	Instance management	369	8795
	Meta-function Invoking	280	6714
Context-specific adaptation	Plug-in constraint adaptor	108	3798
Context Reasoner	Simple context match	90	2620
Auxiliary code		500+	

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

One of the basic services in our system is the 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 Selector and Context-specific Modeler is highly implementation specific, thus the lines of code listed here are just the simple adaptation algorithm for our TV scenario described in section 4.

4.2 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.

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

	Application adaptation	ACCADA Framework
Adaptation logic	Prefixed	Change in runtime
Context knowledge Integration	Static/Internal	Flexible/Architecture
Implementation Complexity	High	Low
Multi-context support	NA or static	Yes and flexible
Context-specific Adaptor implementation	Complex	Concise
Separation of design concerns	Mixed	Yes
Level of Adaptation	Inside specific Application	Across several applications

However, during context changes, only the adaptation strategy should be altered to express the context-specific knowledge. Without the burden to support software maintenance, 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 need re-implement 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 2- a Fractal [8] toolchain backend, which generates Java source code corresponding to the application architecture specified by the designer. In the following section, we compare our approach with Juliac's.

4.3 Adaptation Complexity

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. 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. In

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

comparison, in ACCADA, no process for off-line auxiliary code generation is needed. An application mainly contains its business code, simple and easy to manage.

Table 3. Line of codes

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

Resource consumption: we executed this application with different number of instances. With the Juliac approach, the memory consumption will increase considerably with the number of applications, while in our framework it will increase of about 42Kbyte for each installed application. For each installed component, about 13Kbyte memory are needed to store the parsed meta-data information and reference relations. This overhead is comparably small with respect to the more than 430Kbyte memory required by the Fractal model. This discrepancy comes from the different models employed. Each Juliac application has to carry a full set of system run-time, with the increasing number of application; the overhead from the basic system service can be intolerably high. In contrast, ACCADA framework is designed to support a set of managed components and is decoupled from application business logics. No matter how many applications are deployed, only one set of basic services is needed.

In this test, we used equinox, a general purpose OSGi platform. Other implementation, such as Concierge OSGi [15], achieves memory consumption less than 200Kbytes memories. In other words, by simply changing OSGi implementation, the resource consumption can be further reduced.

4.4 Architecture performance

To evaluate the performance of the Automatic Configuration Service, we instrumented a test to measure the time for fetching, parsing, reference management, and configuring. We focused on the time for installing a single component as we vary the number of managed components 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. We use a Dell D630 laptop with 2.2 GHz dual core T7500 CPU, 2GB RAM and 80 GB 7200RPM HDD. The JVM we adopted is JAVA 1.6.0.2 SDK on Linux.

Here, we use a different Context Modeler with the one described in section 4. It checks following constraints (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 remains disabled during the experiment. Here, in the newly installed component, the component initialization time is not counted as it may varied according to different implementations.

$$\sum \frac{\text{Execution time}}{\text{Period}} < 1 \text{ for all enabled components} \quad (1)$$

Installing a new component normally consists of five main steps: component loading, meta-data processing, structural modeling, context modeling, and actuation. Figure 4 shows the absolute times spent in each steps. Each value is the arithmetic mean of 250 runs of the experiment. In order to better illustrate the trend of different steps, we use

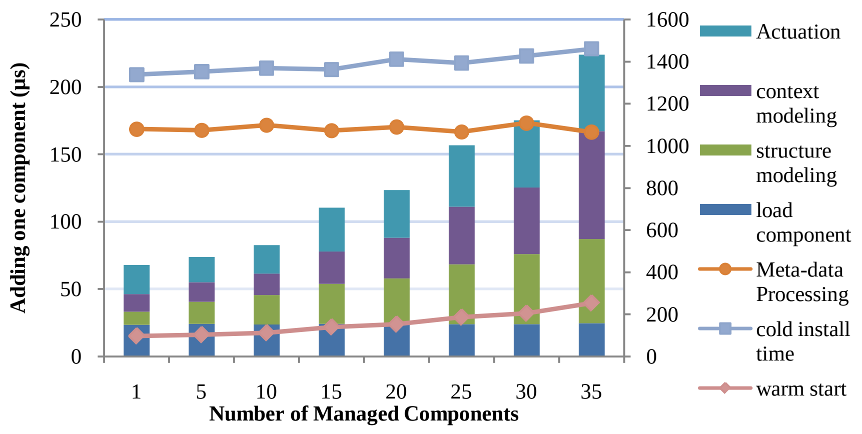

Fig. 4. Framework Performance on adding one component

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 micro-seconds (µs).

With the number of managed components grows, component installation time grows very slowly. It mainly due to the fact that two key elements –component loading time, meta-data processing time which count more than 80% total time, keep comparably stable when component number *n* grows. In contrast, the other three key elements, the structural modeling, context modeling and actuation will increase lineally with n as it has computing complexity O (n). Context modeling process which checks whether the new component can satisfy the resource requirements also have complexity of O (n) (stateless implementation, no optimization). Here, the actuation process also includes time for post-processing the modeling results from two modeling processes so it changes with the system scales. 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". This approach can effectively reduce a component response time – from 1000 µs to about 200µs.

Simulation results show that our framework scales well when the number of managed components grows. 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 managed components grows. This is highly policy dependent and is out of the scope of this paper. The Context Selector also has similar characteristics.

5 RELATED WORK

There is a substantial body of literature on reconfigurable middleware systems. Compared to on our earlier work on the DRCom component mode, our framework exhibits a richer and more coherent set of features to support context-specific knowledge and provides a systematic approach to integrate such knowledge.

SmartFrog [4] is a framework for the management of configuration-driven systems. The framework provides mechanisms for describing these component collections and for deploying and managing their life cycle. However, there also lack of support of how to support the context-specific adaptation and the description of component is static and could not support non-functional properties and constraints.

Sylvain etc. identifies novel requirements on reflective component models for architecture-based management systems [5].The construct layer is designed for the meta-data checkpoint and replication. A faulty component can be repaired by restore its state and all the meta-data information outside of the component instance. However, their approach does not have clear definition and separation between system services. The hard-wired architecture makes it very hard to reuse their framework across different contexts.

Garlan etc. propose a general architecture-based self-adaptation framework [2]. 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 [12] propose a service-oriented component based framework for constructing adaptive component-based applications. The key part of the framework is the Service Binder which automatically controls the relationship between components. Our approach mimics theirs in dealing with component's dynamicity. However, our approach can provide more flexible adaptation compared to its static resolving policy.

Kasten etc. propose the Perimorph framework to achieve run-time composition and state management for adaptive system [13] It enables an application designer to quantify and codify collateral changes in terms of factor sets. However, due to lack of a clear defined component model, it hard to extern their approach to cross applications adaptation. Their approach also doesn't consider how to integrate the context adaptation knowledge.

In order to handle the complex dependence between components, Kon etc. propose an integrated architecture for managing dependencies in distributed component based systems [3]. The architecture supports automatic configuration and dynamic resource management in distributed heterogeneous environments. However, how to support the context changes is not specified in their approach.

6 CONCLUSION 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 the uniform management interface, the 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. Service-oriented model is used in implementing architecture basic modules thus achieving more flexible system architecture. This framework is easy to be configured to fit with different contexts.

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.

References

1. Oreizy, P., et al., *An architecture-based approach to self-adaptive software.* Ieee Intelligent Systems & Their Applications, 1999. **14**(3): p. 54-62.
2. Garlan, D., et al., Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer, 2004. 37(10): p. 46-+.
3. Kon, F., et al., Design, implementation, and performance of an automatic configuration service for distributed component systems. Software-Practice & Experience, 2005. 35(7)
4. Anderson, P., P. Goldsack, and J. Paterson, SmartFrog meets LCFG: Autonomous reconfiguration with central policy control. Usenix Association Proceedings of the Seventeenth Large Installation Systems Administration Conference, 2003
5. Sylvain, S., B. Fabienne, and P. Noel De, Using components for architecture-based management: the self-repair case, in Proceedings of the 30th international conference on Software engineering. 2008, ACM: Leipzig, Germany.
6. Costa, P., et al., The RUNES middleware for networked embedded systems and its application in a disaster management scenario. Fifth Annual IEEE International Conference on Pervasive Computing and Communications, 2007
7. Dey, A.K., G.D. Abowd, and D. Salber, A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction, 2001. 16(2-4): p. 97-+.
8. Seinturier, L., et al., A component model engineered with components and aspects. Component-Based Software Engineering, Proceedings, 2006. 4063: p. 139-153.
9. Coulson, G., et al., A generic component model for building systems software. Acm Transactions on Computer Systems, 2008. 26(1): p. -.
10. Michael, C., et al., An Efficient Component Model for the Construction of Adaptive Middleware, in Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg. 2001, Springer-Verlag.
11. Gui, N., et al. 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.
12. Hall, R.S. and H. Cervantes, Challenges in building service-oriented applications for OSGi. Ieee Communications Magazine, 2004. 42(5): p. 144-149.
13. Kasten, E.P. and P.K. McKinley, Perimorph: Run-time composition and state management for adaptive systems. 24th International Conference on Distributed Computing Systems Workshops, Proceedings, 2004
14. Gui, N., et al. A Hybrid real-time component model for reconfigurable embedded systems. in ACM symposium on Applied computing. 2008. Fortaleza, Ceara, Brazil.
15. Rellermeyer, J.S., Concierge: A Service Platform for Resource-Constrained Devices. Operating systems review, 2007. 41(3): p. 245.