Transformer: an adaptation framework supporting contextual adaptation behavior composition

Ning Gui1,3 Vincenzo De Florio2 and Tom Holvoet3

1 Department of Computer Science, Zhejiang Sci-Tech University, China
2 Performance Analysis of Telecommunication Systems group, University of Antwerp, Belgium and iMind, Ghent-Ledeberg, Belgium
3 Distrinet Lab, University of Leuven, Belgium

SUMMARY

As software systems today increasingly operate in changing and complex environments, they are expected to dynamically adapt to the changing environments sometimes with multiple co-existing adaptation goals. This paper proposes an adaptation framework to facilitate adaptation with multiple concerns by using reusable and composable adaptation modules. Rather than using one-size-fits-all approach, in this framework, system global adaptation behavior is generated by contextually fusing adaptation plans from multiple adaptation modules. In order to handle possible conflicts from multiple adaptation plans, supports for conflict detection and resolution are provided.

Following the framework design principles, a supporting middleware is implemented based on the service-oriented component model. Adaptation behaviors are realized as individually deployable adaptation components. A strategy called normalized context matching degree is proposed to rate and select applicable adaptation components. Possible conflicts in dealing with multiple adaptation concerns are resolved by using the semantics of actuators and context conditions. This middleware is also designed to be readily reconfigurable to support new features. Case studies and experiment results show that our framework exhibits significant advantage over traditional approaches in light of flexibility and reusability of the adaptation modules, with little complexity and performance overhead. Copyright © 2012 John Wiley & Sons, Ltd.

KEY WORDS: Dynamic behavior composition; architecture-based adaptation; context-awareness; service-oriented middleware

1. INTRODUCTION

With the proliferation of mobile and ubiquitous computing, software systems today increasingly operate in changing environments. As a consequence, more and more software systems are expected to dynamically self-adapt to accommodate for resource variability changing user needs and system faults — in most cases with different and changing combinations of these factors. Among the many approaches to support adaptation, architecture-based adaptation is extensively used, e.g. in [1, 2], to handle cross system adaptation by using external control models and adaptation mechanisms. By taking into account a holistic view of what happens inside each application and across multiple application domains, architecture-based adaptation provides application reconstruction and reconfiguration functions with minimal or no human intervention [3].

*Correspondence to: Computer Science Department, ZSTU Hangzhou, China, email: ninggui@gmail.com

Copyright © 2012 John Wiley & Sons, Ltd.
Prepared using speauth.cls [Version: 2010/05/13 v3.00]
A well-accepted design principle in architecture-based management is to use component-based technology to develop the management systems and application structures [2, 4, 5, 6]. Following such principle, in many of the approaches in the state-of-the-art, applications are automatically (re-)constructed and adapted to their changing contexts during run-time by taking compositional adaptation and/or parameter-based adaptation actions [7]. Several adaptive systems have already been proposed to provide such software adaptation support [8, 2, 9, 4, 5].

As pointed out in [10], for a changing environment, not only the adaptive software needs to be changed to reflect new context requirements, the system adaptation behavior itself should evolve to reflect these changes — denoted as adaptation behavior evolution. Furthermore, as software systems are used in more and more complex environments, the adaptation module should be designed to respect multiple adaptation concerns — for instance achieving both self-optimization and self-healing behaviors.

However, existing adaptation solutions are not prepared to support adaptation behavior evolution [11, 5]. Their focus is normally on providing proprietary solutions tailored for one particular context. The resulting designs characterized by a statically built-in adaptation module addressing limited and immutable adaptation concerns. This feature makes them very hard to be tailored to other execution environments. The requirements for adaptation with multiple concerns further exacerbate the problem. As different adaptation concerns might have radically different properties of interests, it is possible for adaptation logics for different concerns to generate conflicting adaptation behaviors. For instance, for a mobile phone, one adaptation concern for minimizing power consumption may choose to reduce the CPU execution speed while another adaptation for optimizing UI response time could choose to set CPU in full speed. A systematic mechanism is needed to explicitly resolve possible conflicts between different adaptation concerns. However, three main streams of approaches identified during our research, exhibit major limitations in supporting adaptation with multiple concerns.

One major stream of approaches is based on Aspect Oriented Programming (AOP) [12, 13]. AOP allows adaptation logics to be modularized as individually deployable units (aspects) that can be directly weaved in the application business logics off-line or during run-time [14, 13]. In order to reduce interference between different aspects, AOP aims for so-called “obliviousness,” which requires different aspects to be mutually orthogonal and thus not interfering with each other. However, in many situations this requirement is very hard to achieve. Partially due to this limitation, existing AOP-based adaptation approaches normally focus on one particular adaptation concern at a time [15].

Other adaptation solutions provide algorithms to guide the system to adapt with multiple concerns [11, 5]. The resolution process for possible conflicts is only implicitly discussed and entangled with other adaptation processes. For instance, in [5], the main concern is application self-healing while the concern for software structure maintenance is only implicitly considered and integrated in their designed adaptation algorithms. In these approaches, no clear separation between adaptation concerns and no clear integration process are provided to support different adaptation concerns. This tight coupling makes adaptation modules hard to develop and almost impossible to be reused.

Some software adaptation platforms identify the importance in detecting and resolving possible conflicts in adaptation with multiple concerns. In these approaches, utility function based solutions are used to detect and solve conflicting goals [16, 17]. The semantics of those adaptation actions is simplified / represented to a manually assigned “utility gain” set by a “system administrator” [17] or voted by a sealed bidding mechanism [16]. However, in a changing context, the utility values tagged to each adaptation action should also be altered to reflect new context concerns. Due to their static nature, those utility function based approaches, it is argued, cannot provide effective conflict resolution support in the changing context.

This paper aims to provide a more systematic way to support adaptation behavior evolution and provide supports for adaptation with multiple concerns. It provides a comprehensive description of our approach, describing the design, implementation, and performance of our adaptation framework — Transformer — and its supporting service-oriented middleware implementation. Rather than
Figure 1. Context-specific fusion of adaptation behaviors. Depending on the context, a different set of candidate strategies can be selected and composed into a custom-generated global adaptation module matching the current environmental conditions. Here, for TV watching adaptation scenario, three adaptation modules can be selected for different adaptation concerns: TV optimization strategy, Structural maintenance strategy, and Self-healing strategy. When system context changes to the MobileTV context, a different set of basic adaptation modules shall be selected.

using a monolithic module for adaptation, Transformer constructs system global adaptation module by composing multiple basic adaptation modules during run-time, according to context to date. This work is based on our previous work on declarative component design [18] and the ACCADA framework support for integrating context-specific adaptation knowledge [19, 20]. Compared to the ACCADA framework — which supposes only one adaptation module at a time to steer dynamic compositional adaptation — Transformer provides more systematic support for adaptation composition and conflict resolution.

Major contributions presented in this paper are as follows: Firstly, we identified the key principles in designing and implementing composable adaptation modules and extended the paradigm of separation of concerns to the design of adaptation modules. Following these design principles and enforcing them during implementation, our framework allows multiple adaptation modules to be composed to create more complex global adaptation behavior, just like components are assembled to build applications. Adaptation developers then do not need to re-invent the wheel for each new adaptation requirement. Secondly, our framework allows system global adaptation module to evolve and re-optimize by dynamically composing reusable adaptation modules in response to context changes. System administrators are relieved from the burden to manually construct adaptation behaviors for all different contexts; rather, Transformer automatically selects and reuses preexisting adaptation blocks each dealing with domain-specific adaptation concerns. Thirdly, our approach explicitly isolates the conflict resolution process from the other adaptation processes, which allows a clearer view to the detection and resolution of such conflicts. A prototypical solution is proposed based on the semantics of adaptation actions. The effectiveness of our architecture is demonstrated both from a qualitative and a quantitative point of view. Simulation results show the soundness of our implementation in terms of lines of code, adaptation capabilities and reusability of adaptation modules.

The rest of the paper is organized as follows. Section 2 describes a motivational example and analyzes related challenges. Section 3 presents the key components in our adaptation process and provides the design of the Transformer framework. Section 4 describes the service-oriented middleware implementation for our framework. In Sect. 5, multiple adaptation processes are described by using scenario-based simulation. Simulation results are shown to prove the effectiveness and flexibility of our approach in Sect. 6. We discuss related work in Sect. 7, and provide our conclusions in Sect. 8.
In this section, we firstly describe a family of case studies to motivate the need for adaptation in a changing context with multiple co-existing adaptation concerns. Then, the challenges in realizing the motivational example are identified.

2.1. Motivational Example

Let us suppose that John has a set-top box he uses to watch TV programs from a cable TV provider. John likes to experience his TV at top quality, but occasionally wants to view the programs with his iPhone while he cannot stay in front of his TV. The set-top box is expected to adapt its behavior so as to maximize the overall Quality of Experience (QoE)\(^\text{[21]}\) as perceived by John. This very simple scenario already calls for complex treatments, which can be more easily described by a few cases:

- **TV watching**: John is watching the news; his current top interest is TV QoE. The current context requirement is to optimize the TV application. One basic adaptation logic is to always use components with best video quality to construct TV application and allocate enough resources for those components to work effectively. Let us assume the adaptation module for TV application is implemented with such strategy.

- **Mobile TV**: John is roaming in his house to do some house work so he cannot stay in front of the TV. However, he still wants to enjoy the TV program anyway. So he starts the MobileTV program in the set-top box. This application retrieves the video stream from TV program, transcodes it, and transfers the compressed video stream over wireless to John’s iPhone. As the MobileTV application transcodes the original TV video stream into a much smaller and comparably low quality video stream, it does not need high quality video stream source. In this scenario, the optimization concern is to use normal TV source quality and make a balance between the quality of the compressed mobile video stream and wireless transmission speed.

- In all the above cases, John would like to have self-healing capabilities for the enabled applications so as to deal with possible application crashes. Of course, he might want to disable such feature if there has been no application crashes for some time.

What above are just three cases out of many possible situations that build up John’s perceived quality of experience. For each new scenario, as it represents a new set of adaptation requirements, a new adaptation module is needed. For instance, in the MobileTV scenario, an adaptation module needs to handle constraints from at least three different aspects, namely: (1) Application structural maintenance for maintaining the correctness of adaptive software architecture. This feature is needed for all architecture-based adaptation solutions (2) Adaptation logics for MobileTV application; and (3) Self-healing adaptation logics to tolerate possible application crashes. Traditional approaches use a monolithic adaptation module to statically integrate these adaptation logics and optimized for certain context. When context changes, for instance, from TV watching shift to MobileTV, the adaptation module will not work correctly despite these two contexts share many adaptation requirements in common. As we can see from Fig. 1, in both contexts, the adaptation logics for structural maintenance and self-healing adaptation are required. However, the monolithic solution makes it very hard to reuse adaptation modules across multiple contexts. Our conclusion is that a better solution to address such problem is to divide the adaptation modules into small adaptation building blocks representing particular adaptation goals matching well-defined “scenarios”. Each of those blocks assumes that some contextual hypotheses are valid and focuses on one domain-specific adaptation concern. These adaptation blocks can be used to construct more complex adaptation modules and could (possibly) be reused across multiple contexts. Rather than providing a static adaptation module for all possible contexts, in this approach, a system global adaptation module is built by selecting a set of such “blocks” and “fusing” them into a new contextual adaptation module. Compared to the one-size-fits-all (monolithic) approach, this solution can enhance the reusability of the adaptation modules. For instance, as depicted in Fig. 1, the self-healing module can be used in all three above scenarios. As shown in Fig. 1, compared to the traditional approaches, this solution is more flexible in dealing with changing environments while
enhancing the reusability of adaptation modules. However, as different adaptation modules target different adaptation concerns, they might generate conflicting adaptation plans.

As the TV optimization and MobileTV adaptation modules have totally different goals, if employed together, they might generate conflicting adaptation behaviors.

Listing 1 shows sample adaptation logics for both adaptation modules. For simplicity reasons, we only list adaptation logics that might generate conflicts. From this listings, we can see that the TV optimization logic tries to increase TV quality whenever CPU usage is low. In comparison, as the source TV quality has little impact towards toward MobileTV quality, the adaptation logics for MobileTV will try to set TV quality to 50% (normal quality) to save enough CPU for Mobile video transcoding process. When, for instance, current TV quality is 60% and CPU usage is 65%, for TV optimization module, TV quality ought to be left unchanged. For the MobileTV optimization logic, this setting should be 50%. In this case, conflicts arise. Those conflicts must be systematically resolved when adaptation modules are composed to build more complex adaptation behaviors.

```
% TV optimization logic
...
if CPU < 50% --> TV_quality = TV_quality * 1.1
if CPU > 75% --> TV_quality = TV_quality * 0.9;
if (TV_quality<LowerBound && (decode_overrun_increase > Max_threshold) -->
   disable other CPU_intensive components.
...
% MobileTV optimization logic
...
if CPU > 60% and TV_quality > 50 --> TV_quality=50;
if CPU < 50% and QueueLength < Low_threshold --> Transcode_quality =
   Transcode_quality * 1.1
...
```

Listing 1: Sample adaptation logic

2.2. Challenges

In order to realize the vision proposed in the motivational example, the following three challenges are identified during our research.

1. The first challenge is to enhance the reusability and composability of adaptation logics. In most adaptation framework, as the system adaptation modules are not designed to be composable, no clear design principles are identified to facilitate the composition of adaptation modules. In implementation, the adaptation logics and adaptation actuation mechanisms are mixed together. This coupling makes adaptation modules very hard, sometimes impossible to be composed with each other.

2. The second challenge is to support the evolution of the system adaptation behavior such that it matches the new requirements of the fast changing context. For instance, in the motivational example, when the context migrates from TV to MobileTv, our framework should guarantee that the most appropriate set of adaptation components are selected. Many existing adaptive systems allow their adaptation modules to evolve only through certain AI based learning process — for instance, using history data to build adaptation logics with the decision tree algorithm in [22]. However, it is argued, such algorithms are not capable to deal with the fast changing and unexpected context.

3. The third challenge is the systematic conflict detection and resolution support in composing multiple adaptation modules. In order to make coherent and pointed adaptation, an effective adaptation framework needs to provide explicit support for the conflict detection and resolution. However, as described in the introduction, existing solutions either provide no clear conflict resolution process or provide some static integration that is tailored for particular application domains. The conflict resolution logics normally are mixed with adaptation logics in the form of binary code. This strong coupling makes those resolution logics hard to be managed or revised. In order to effectively deal with conflicts from multiple adaptation concerns, it is argued, the conflict resolution mechanism has to be elicited from the other
parts of adaptation process. This separation, it is argued, provides programmers more clear view of conflict resolution process and allows more flexible solutions for this process.

3. TRANSFORMER ADAPTATION FRAMEWORK

In order to deal with the three challenges described in previous section, Transformer is introduced. Firstly, the framework architecture is introduced. Then, in the following sections, general descriptions for key adaptation modules of this framework are provided.

3.1. Multiple Domain-specific Modelers Support

From the motivational example we can see that the adoption of the separation of concerns paradigm in the design of the adaptation modules can effectively reduce the complexity of the adaptation modelers, at the same time enhancing their reusability. Multiple basic adaptation modelers are used in our framework to satisfy multiple domain adaptation goals (e.g. minimization of consumption of CPU consumption, maximization of video quality, etc.) and maintain integrity constraints (such as functional dependence). Here, we firstly introduce the basic adaptation elements — Domain-Specific Modelers (DSM).

3.1.1. Domain-specific Modeler with Context-constraints.

In software adaptation, system evolution can be driven by different viewpoints — for instance, security and performance [23]. Each concern expresses a set of adaptation strategies and constraints for a particular adaptation goal and is modularized as a reusable adaptation modeler — DSM. Here the notion of DSM focusing on grouping and encapsulation of adaptation concerns into separate, independent entities, is a concept close to the “aspect” in AOP. Normally, each DSM has its specific effective domain — applicable context. Only within this “applicable context” it can effectively conclude the right adaptation actions. To reflect this fact, we denote it as Domain-Specific Modeler with Context-constraints (DSMC). Such context requirements can be expressed as a list of Context Matching Constraints (CMC). CMC consists of a DSMC’s possible range of acceptable resource values. For instance, a CMC list of constraints such as \{ (cpu\_speed > 70MHz), (Preference = “TV Quality”) \} shows that this DSMC best fits for a context where the CPU clock is faster than 70MHz and the current preference of the user goes to optimizing the quality of the TV experience. The CMC influences a DSMC’s reusability. The stricter the CMC of a DSMC, the narrower the “applicable context” will be. As shown in the motivation example, the DSMC for Self-healing can be reused in many contexts where the user requires fault-tolerant features. In contrast, other DSMC, such as the DSMC for TV optimization, will only be used in particular contexts and for particular applications. DSMC are represented as an adaptation function $F$ with such constraints. Let us call $SC$ as the set of all possible configurations and $Z = \mathcal{P} (\text{Resource} \times \text{Value})$ as the set of all possible execution contexts. Let us call $M$ the set of all available DSMC, $A$ the set of all possible actions. The $\mathcal{P}$ function represent any subset of $A$. Then for any DSMC $d \in M$ we represent as $F_d$ a function such that, given $sc_{\text{current}}$ and $z_{\text{current}}$, the corresponding adaptation actions are selected for $d$. To express this we write:

$$F_d : (SC \times Z) \xrightarrow{(sc_{\text{current}}, z_{\text{current}})} \mathcal{P}(A) \text{ if } z_{\text{current}} \in \text{CMC}, d \in M$$  \hspace{1cm} (1)

In other words, whenever there is a significant change requiring the system to adapt, each DSMC $d \in M$ computes the set of adaptation actions $A_d$ — denoted as adaptation plan — according to system current configuration $sc_{\text{current}} \in SC$ and current context $z_{\text{current}} \in Z$. To represent this process we shall write:

$$A_d = F_d(sc_{\text{current}}, z_{\text{current}}) \quad d \in M.$$  

The actual nature of the elements in $A$ highly depends on the underlying component implementation. Only those actions supported by the underlying component model can be successfully executed. For instance, in our previous work in declarative real-time component model [18], a management interface was designed which permitted to start, stop, and change
properties of components. In such case, the possible adaptation actions are \{activate, stop, setProperty\}.

3.1.2. Context-specific Modeler Selection. As mentioned already, the fact that each DSMC \(d\) has its own particular CMC constraints means that it can only effectively conclude its adaptation actions when the context matches its requirements. One familiar example is a notebook’s power management — a modeler optimized for battery normally would not exhibit satisfactory performance for the user while the notebook is plugged in. Due to this reason, for a particular context, only the part of the installed modelers that matches this environment should be used for the adaptation process. Given the CMC from each DSMC, it is possible to design a dynamic selection scheme to select the “right” set of modelers to be used in the ensuing adaptation process, that is, the set best matching the current context conditions. We call this process as DSMC selection and denote it as function

\[
S : \mathcal{M} \xrightarrow{\sim} \mathcal{P}(\mathcal{M}) \tag{2}
\]

Expression (2) states that, given \(\mathcal{M}\) — the set of all installed modelers — the semantic function \(S\) selects the set of applicable DSMC \(\mathcal{P}(\mathcal{M})\) from the set of all modelers by checking a similarity degree with current context. In Sect. 4.2.4 we describe our current design in the context matching degree calculation.

3.1.3. Model Fusion. After the system chooses the right set of modelers, each selected modeler will be inquired to compute its adaptation plan according to its domain-specific adaptation goals. In order to achieve a coherent architectural adaptation, a process is needed to identify and resolve possible conflicts between adaptation plans. In our framework, a Model Fusion module is designed to provide explicit conflict resolving support. By fusing adaptation plans generated from selected DSMC into a conflict-free adaptation plan, this module, together with selected DSMC, practically generates a global adaptation model.

This process is denoted as function

\[
\mathcal{H} : \{A_1, A_2, \ldots, A_i, \ldots, A_n\} \xrightarrow{z_{current}} \mathcal{P}(\mathcal{M}) \xrightarrow{} \mathcal{P}(A) \tag{3}
\]

\(\mathcal{H}\) is a function that maps multiple modelers’ computed action sets into one conflict-free adaptation plan. A common adaptation plan is computed based on many factors: the characteristics of all selected DSMC \(\mathcal{P}(\mathcal{M})\), involved adaptation actions and system current context \(z_{current}\). By taking all these factors into account, the Model Fusion module comes out with a final agreed adaptation policy \(A^\ast\):

\[
A^\ast = H[A_1, \ldots, A_n] \mathcal{P}(\mathcal{M}), z_{current}]
\]

As we can see, in order to design an effective Model Fusion function, the fusing logics in this module needs count all these factors. Due to this complexity, designing a Model Fusion module for any arbitrary set of DSMC, it is argued, remains a very challenging task. In Sect. 4.2.5, a fusion algorithm based on the semantics of adaptation actions and context matching degree is provided as a proof of concept in merging the adaptation plans from multiple DSMC. Discussions on tailored version of Model Fusions for limited context are also provided in Sect. 4.4.

3.2. Framework Architecture

In this section, we introduce the design of Transformer adaptation framework which provides adaptation behavior evolution via the DSMC composition. Key modules discussed in the previous section are explicitly integrated into this framework.

The Transformer adaptation framework is designed under the MAPE-K reference model [3]. As can be clearly seen from Fig. 2, this adaptation framework makes use of an external control loop, consisting of six basic modules — Event Monitor, Adaptation Actuator, System Basic Run-time
Figure 2. Transformer adaptation framework with multiple DSMC support

(SBR), Model Fusion, Validation, DSMC and DSMC Selector. Some of the modules can be directly mapped to the elements in the reference model. For instance, the Event Monitor corresponds to the Analyze module in MAPE model and SBR corresponds to the Knowledge plane of MAPE-K, as it keeps tracks of installed component states and their configurations. It also works as the central coordination for the whole adaptation process.

In this framework, considerable extensions are made in the part of Planner element with respect to the MAPE-K reference model. Instead of using one single Planner, in our framework, three types of modules — DSMC, DSMC Selector and Model Fusion — collaboratively build the Plan element. The Validation module — another extension to the MAPE-K model — is designed to check and maintain system consistency and system integrity.

The Event Monitor module observes and measures various system states. It sends notifications to trigger a new round of the adaptation process. Possible sources of adaptation may include, for example, a new component being installed or the CPU utility reaching a status that may have a significant effect on the existing system configuration. It could also be a simple Timer that triggers periodically. In our current middleware implementation, basic component lifecycle events — install, uninstall, update — and service events — registering, unregistering, updating — are natively supported by the SBR. User custom events are also supported via predefined Event Monitor interface. The Adaptation Actuator carries out the actual system modification. Its 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. Available types of Adaptation Actuators determine the adaptation action set $A$. Performing one adaptation action might also trigger a set of adaptation actions until the system reaches its new preferred state. The Event Monitor and Adaptation Actuator provide standard ways to monitor and manage the installed component instances.

In order to make a correct and pointed adaptation, a system adaptation module requires handling constraints from different aspects — this includes handling application functional aspects such as interface compatibility, adaptation aspects such as security, fault-tolerance, user’s preference, performance etc. For each adaptation concern, its corresponding adaptation logics can be modularized as a DSMC. DSMC is designed to modularize adaptation strategies for certain adaptation concern and can be individually deployed and un-deployed. From this perspective, the
DSMC has similar features as the aspects in AOP. However, rather than assuming these concerns will not interfere with each other as AOP, in this framework, the Model Fusion module is designed to provide explicit conflict detection and resolution. By fusing those computed adaptation plans into a conflict-free one, together with selected DSMC, this module effectively creates a global adaptation modeler with multiple adaptation concerns. As more than one DSMC can be installed simultaneously, each provides a particular adaptation feature. Not all these adaptation features are needed in every context. A modeler selection module — DSMC Selector — is supported in this framework. From the definition in the previous subsection, it determines the subset of the DSMC that are appropriate for the current context.

As the global adaptation modeler is constructed during run-time, in the Transformer adaptation framework, it is possible that the fused adaptation model is not thoughtfully tested. How to evaluate the correctness and effectiveness of the fused global modeler is one of the major issues in our system design. In Transformer, the Validation service is provided to prevent the system to take unintended adaptation actions, which may lead to catastrophic consequences for the system.

3.3. Key DSMC Design Principles

As illustrated in the Fig. 2, our framework follows an external approach which supports separation of the adaptation mechanism from the application logic. According to the Salehie’s recent survey [24], all 16 surveyed projects on self-adaptive systems use such external approach to a certain degree. However, separation of adaptation mechanism from application logic itself cannot guarantee the adaptation module to be reusable in other environments. In many surveyed implementations, the adaptation modules are hard-coupled with other adaptation executing modules and are not designed to be reusable or composable. In this section, key design principles for DSMC are identified to enhance the reusability and composability of the adaptation modules.

- Orthogonality, which is meant to lower the chance of interference of concerns and reduce the complexity this interference may bring to the later conflict detection and resolution process. It is quite similar to the design principle of AOP. However, in our framework, it is not mandatory requirements due to the existence of the Model Fusion module.
- Clear separation of planning process from actuation process. This principle aims at enhancing the reusability of adaptation modules. Adaptation modules should be separated from underlying system implementation details. We addressed this principle by forbidding a DSMC to perform any adaptation action inside its reasoning logic. DSMC can only expose their plans to the system run-time through their pre-defined DSMC interface.
- Limitation of adaptation actions. A DSMC can only use adaptation actions with well-defined semantics. The ontology among supported adaptation actions should also be defined. This requirement is to promote composability DSMC and to simplify the conflict detection and resolution performed in the Model Fusion process.

Based on the above principles, the DSMC is designed and implemented as reusable and composable components (see Sect. 4.2.3). As the Transformer adaptation framework only provides a general guideline on the functional properties of system key modules, in the following section, a middleware-based implementation is introduced to demonstrate the feasibility of contextual DSMC selection and composition.

4. MIDDLEWARE-BASED IMPLEMENTATION

In the previous section, basic functionalities of the above mentioned key modules are identified. However, last section only provided a general guideline on the design issues of these modules. In this section, we show how we made use of those guidelines to design a service-oriented middleware architecture which, it is argued, provides concrete support to our Transformer framework.
4.1. Pluggable and Modular Middleware

In this middleware, several key modules are designed corresponding to the process described in Sect. 3, with detailed software system design. However, in order to strike a balance in implementation complexity while keeping system flexibility, some revisions are made. This software architecture is an enhancement of our previous work for context specific adaptation [19], in which only two modelers (the context-specific modeler and functional dependence modeler) are used to deal with system application architecture maintenance and context-specific adaptation requirements.

Figure 3 shows the key middleware architecture and current implementation of middleware modules. In this middleware, many key modules, such as DSMC, are directly corresponding to the modules in the framework design — for instance, the IEventMonitor interface is used to support DSMC specific events and IDSMCResolver is designed for the DSMC modeler. Other service interfaces are designed to provide important auxiliary functions. For instance, IContextAccess provides an updated list of context factors and ICMDCalculation provides context matching degree calculation service.

In compositional adaptation, in order to achieve run-time compositional adaptation, maintenance of software functional architecture is one of the key requirements. Accordingly, one new module — Structural Modeler — is specifically provided to handle functional dependence between components. It is actually a special DSMC. As an application can only be formed when all functional requirements are fulfilled, this characteristic gives the Structural Modeler the uppermost priority in all the software adaptation aspects. At the same time, it is rather stable; in particular, it is invariant to context. That is why, in the software design, this Structural Modeler is separated from the modeler selection process. In fact, it is always included in the adaptation process whatever the context. For one system run-time, only one instance of this modeler will be installed.

In order to check the correctness and effectiveness of the fused global adaptation behavior, the Validation service was designed so as to provide reflective adaptation service. By using this service, system administrator or third party programmers can get internal adaptation related information and reason about possible adaptation errors.

In this middleware, each module is implemented based on our declarative and reflective component model — a service-oriented component model extended from component model in [8] with context-specific meta-data support. This component model is used to construct not only this middleware itself but also the business components of the manageable applications. Details for this component model can be found in [18, 19].

4.2. Key Modules

In the following section, the current implementation of key elements of this middleware is introduced. Implementation of several modules is comparably mature and stable, as it is the case, e.g. SBR, Structural Modeler. These modules have been intensively studied in other researches and are also major topics of our previous work [19] Other implementations, such as Model Fusion, are still in active research stage. The design of Event Monitor and Adaptation Actuator is not the focus of this paper and will not be introduced here. You may find detailed design for these two modules in [20].

4.2.1. System Basic Run-time. In this middleware architecture, the SBR is the module which implements basic management functionalities. It (1) installs, discovers, executes and uninstalls components, as well as manages the component instance registry; (2) manages the service registry and monitors the service changes to provide run-time service component supports; currently, basic events are internally supported by the system run-time. Users can also implement their custom Event Monitor and let SBR to have improved knowledge about system changes (3) parses the meta-data attached with installed component (4) provides adaptation actuation supports. In this SBR, basic Adaptation Actuators (active, stop, change properties) are natively provided. It will also manage/inject the references between components. By invoking specific methods declared in components’ meta-data, the system can effectively enforce the bindings between the components;
(5) manages an instance — SystemContext. The instance keeps an updated global context image. It contains all installed components’ meta-data, their states, instance references and the dependence between them. When the system context changes — e.g. new component deployed or adaptation actions are taken to enable a component — the SBR will automatically update the SystemContext and the corresponding component configurations.

This SBR is implemented by using the component configuration, component descriptions and the reference management defined in the declarative service specification [25]. However, this SBR is enhanced with: 1) clear separation between the sensor, actuator and planning parts. This separation makes adaptation composition become possible 2) global context management which is mandatory for DSMC to make accurate adaptation. The SBR as well as the Structural Model are originally developed in our previous work [19]. Detailed discussion of this module is out of the scope of this paper.

4.2.2. Structural Modeler. As the application is constructed, configured and reconstructed during system run-time, how to derive the functional and structural dependency among components is one of the key requirements in compositional adaptation.

The Structural Modeler consists of several processes, the most important of which is functional dependence compatibility check. Based on this information, the system can determine whether a component is “structure-satisfied” — that is, all its required interfaces have corresponding provided interfaces. A component can only be activated when it is “structure-satisfied”. This guarantees that a component can only be initialized when all its functional dependences are satisfied.

The Structural Modeler provides the following functional interface:

```java
public interface IStructuralModeler {
    public List<ComponentConfiguration> resolveSatisfied(SystemContext sc);
}
```

The above interface takes a list containing all managed component configurations. Each time the Structural Modeler is inquired, it will check — for each “enabled” component — whether all the required interfaces from one managed component have their corresponding service provided. The Structural Modeler gets a list of all enabled components from current system context sc.
After this process, it returns a list of all the components that satisfied the functional constraints. All these components will be tagged by Structural Modeler as “structure-satisfied”. Then, the “structure-unsatisfied” component configurations will be given by the complement set of the satisfied component list with respect to all available components. As a component can only be activated when it is “structure-satisfied”, possible adaptation actions of Structural Modeler are “activate” and “stop” and manage component functional dependence by changing component lifecycle state.

As one of the core modules in all run-time adaptation systems, such as Servicebinder [11] or Perimorph [9], the design of this module has been intensively studied. However, unlike these approaches, in our middleware, Structural Modeler is defined following DSMC design principle and separated from the other modules. This separation makes the Structural Modeler very concise and easy to be updated.

### 4.2.3. DSMC

As the Structural Modeler deals with the functional related constraints in building and maintaining the software architecture, the DSMC deals with domain-specific adaptation goals. However, in order to facilitate the DSMC Selector to select the “best” candidate DSMC, each DSMC needs to provide ways for the system to access its capabilities and resource constraints.

**DSMC functional interface.** As different DSMC might have totally different adaptation behaviors and considerations, in order to effectively reuse DSMC across different domains, they must implement the same adaptation interface to allow interoperability. Each DSMC calculates its adaptation plan and exposes its plan to adaptation modules through the following interface:

```java
public interface IDSMCResolver {
    String getDSMID();
    IDSMCMetadata getDSMCMetadata();
    AdaptationPlan resolveAdaptationPlan(SystemContext sc, ChangeEvent cev);
}
```

**Listing 2: IDSMCResolver interface**

In this interface, the getDSMCMetadata method is to get this DSMC’s meta-data. DSMC Selector can access those meta-data which includes its functional dependences, provided service interfaces, and CMC. The resolveAdaptationPlan method is the key method for the DSMC to generate adaptation plan. It has two major parameters: the SystemContext allows DSMC to get the current snapshot of system status, for instance, the list of current enabled components, the list of disabled components, etc. This information is necessary for DSMC to make correct and accurate adaptation plans. The second parameter — ChangeEvent — denotes the just happened change. This parameter informs the DSMC with updated change information. The returned reference (with type AdaptationPlan) keeps the DSMC resolver’ calculated strategy. In current prototype, the AdaptationPlan class contains two lists. The DSMCSatisfiedComponents contains the list of all DSMC satisfied components which means it should be activated from this DSMC perspective. All installed components not in this list are tagged with “DSMC_unsatisfied”. The list of changeProperties returns a list of Properties and their new intended value. System run-time inquires each enabled DSMC to get adaptation plans and sends results to the Model Fusion module.

As can be seen from the interface design, compared to the Structural Modeler, DSMC can make more fine-grained adaptation actions (setProperty) as Structural modeler can only perform component lifecycle management by activate and stop adaptation actions. As described in the previous section, each DSMC has its “applicable context”. The following section introduces a meta-data based approach to expose a DSMC’s context requirements.

**Meta-data for DSMC.** As this middleware is constructed via our declarative & reflective component model, it provides a Java interface based functional description format as well as users customable description language support. A DSMC’s functional requirements — name, required/provided interfaces, and implementation — are directly supported. In order to express
context constraints of a DSMC, which are the non-functional requirements, a simple constraints
description format — ua.mw.language.NCMD is designed.

Listing 3 describes the meta-data of a hypothetical self-healing DSMC. It illustrates how the
context constraints can be described in the component meta-data. It is based on our previous work
on declarative and reflective component model [26], which allows new description languages to
be used to describe non-functional requirements of components by using a special element —
context-specific — to specify context requirements in possible different format.

```
<?xml version="1.0" encoding="UTF-8"?>
<drcr:component xmlns:drcr="http://www.past.ua.ac.be/~ninggui/drcr/v1.0.0" name="DSMC for Self-healing">
  <reference bind="setTimerSensor" cardinality="1..1" interface="ua.mw.
eventreasoer.selfhealing.TimeEvent"
  name="batterysensor" policy="static" unbind="unsetTimerSensor"/>
  <service>
    <provide interface="ua.mw.DSMCResolver.IDSMCResolver" />
  </service>

  <context-specific language="ua.mw.language.NCMD">
    <![CDATA[
      option.selfhealing, boolean, ==, true, 0.5;
      resource.cpu.usage, double, <, 50.5, 0.3;
      option.time, hour, >, 21, 0.2
    ]]>}
  </context-specific>
</drcr:component>
```

Listing 3: Sample meta-data for DSMC

Similar to the approach [22], we express a DSMC’s context requirements via context requirement
conditions. However, rather than treating each sub-condition equally, we identify that different
constraints normally have different impacts towards the applicability of a DSMC. A new argument
— the impact factor — is introduced to express this difference. As can be seen from Listing 3, the
format of context matching conditions is organized as follows:

**CMC with Impact Factor:** Each DSMC’s context constraints are written in the following format:

```
Context Factor, Type, Operator, Value, and Impact Factor;
```

separated by commas. Context Factor denotes the name of context factors. As an example,
battery.percentage denotes how much battery is required, Time preference denotes when it is
appropriate to use this DSMC, etc. Here, Type specifies the data type of context values. Currently,
basic java types, such as string, integer, double, float, Boolean, are supported. Operator here is used
to specify the compare operators, three types of operators are currently supported: “>” denotes
greater than, “<” denotes less than and “==” means equal. The Impact Factor is a normalized weight
that expresses the influence of these context sub-conditions. If Impact Factor of sub-condition \( i \)
is expressed as \( IF_i \), and the total number of sub-conditions is \( N \), then

\[
\sum_{i=1}^{N} IF_i = 1.
\]

The DSMC shown in Listing 3 has three context sub-constraints: user’s preference, CPU usage
and time. This DSMC has three sub-conditions: “preference.selfhealing” with the biggest impact
(0.5), “resource.cpu.usage” with medium impact (0.3) and “option.time” has the minimum impact
(0.2). These constraints determine the applicable domain of this DSMC. The weaker the restriction,
the bigger is the range of the application domain. The DSMC Selector will examine this information
during run-time to determine which DSMC are appropriate to be used for the current context.
4.2.4. DSMC Selector. DSMC Selector locates “best-fitting” domain modelers for current context. System context knowledge comes from three different sources: through system basic metrics, through a context-aware discovery, or through inference. In our current implementation, DSMC Selector gets system context knowledge by periodically reading current context information from Context Manager through its IContextAccess service interface.

The selection process is based on calculation results of the normalized context matching degree (NCMD) by the auxiliary module NCMD Calculation. A NCMD is calculated as follows: For a DSMC with \( N \) sub-conditions, for any sub-condition with context resource \( cr^i \), operator for evaluation \( o^i \) and the value of required context resource value \( v^i \) and its impact factor \( ip^i \). \( eval \) is the function that returns 1 if the value of resource \( cr^i \) in the execution context \( e \) is among the domain specified by the operator \( o^i \) to the targeted value \( v^i \). Otherwise, it returns 0. The context matching degree can be calculated via the following formula:

\[
NCMD = \sum_{i=1}^{N} eval((cr^i, o^i, v^i), e) \times ip^i
\]  

(5)

For instance, if user John’s current context is “resource.cpu.usage = 41%”, “option.time = 7:00”, “option.selfhealing = true”, NCMD of the DSMC described in Listing 3 is calculated as \( 1 \times 0.3 + 1 \times 0.5 + 0 \times 0.2 = 0.8 \). Based on the context matching degree, the DSMC Selector chooses the DSMC whose NCMD are above certain threshold with current context values. In current implementation, value 0.3 is arbitrarily defined. If ratings of all candidate DSMC are below that threshold, only the Structural Modeler will be used to guide adaptation. In this case, only application architecture will be maintained.

The NCMD based selection is by no means the only rating strategy that our framework allows. Other existing context-aware technologies, e.g., Context toolkit [27] or SOCAM [28], can be utilized to develop more advanced context retrieval systems by making use of semantic reasoning logic to infer additional context knowledge. For example, if the moving sensors report no movement for a period of time and the door sensor reports that the sleep room is closed, it can derive a new context “the user is sleeping” from current context knowledge. Obviously such enhanced context knowledge can increase the accuracy on how the DSMC are selected. Secondly, other existing semantic graph matching schemes [29] can be used to calculate how semantically similar between a DSMC context requirements and current context. Detailed discussion on this is out of the scope of this paper.

4.2.5. NCMD-based Model Fusion Modules. As described in Sect. 3, the process of combining these models and resolving possible conflicts becomes vital in guarantying system correctness. Our framework explicitly separates it from other adaptation process. The clear separation allows alternative realizations to be provided to optimize the functional implementations.

As identified in Sect. 3, many factors need to be considered to have an effective conflict resolution solution. Among these factors, the Adaptation Actuator ontology — namely the semantics of adaptation actions and the relationships between those actions — plays an important role for conflict identification and resolution. Due to the complexity to describe relationships between user defined actuators, in current prototypical implementation only three basic adaptation actions are supported — [activate a component], [stop a component] and [change property]. In this implementation, the ontology of these adaptation actions is hard-coded in the Model Fusion module.

For the current prototype, the Model Fusion is designed with two major basic principles/assumptions. 1) A component can only be enabled when it satisfies all modelers’ constraints, otherwise, it should be disabled. This principle is used to determine lifecycle related adaptation actions and it is mainly represented in Lines 1–5. 2) we assume that the adaptation action proposed by DSMC with higher NCMD is more accurate than the one proposed by DSMC with lower NCMD. This assumption is used to resolve conflicts related to the property settings. Thus, If two or more DSMC try to set the same property of one component, only the value from DSMC with highest NCMD will be used (Lines 5–12). The final adaptation plan will be finalized according to system current state. For instance, if a component is already enabled, the “enable” action will not be executed (Lines 13–15).
The just defined algorithm aims at resolving the conflicts while maintaining software structure for an arbitrary set of DSMC. Most of its rules are natural requirements, such as a component can only be enabled when it satisfies all modelers’ constraints. However, in the fusion process, this scheme does not take the context knowledge into account, except for the NCMD. Although this set works fine in our scenarios and simulations, it is clear that the more DSMC, the more intertwined their concerns, the more difficult it is to merge those concerns effectively and correctly. This should not come as a surprise, as it is a situation common to other approaches where multiple concerns need to be resolved into one coherent strategy or application.

As an example, in AOP-based programming, the issue of detection and resolving conflicts among aspects is now becoming a hot research topic. Some of these new researches, such as Lagaisse’s approach of Semantic interference [30] can be largely reused in our system. Compared to AOP-based approach in which aspects may be attached virtually to any position of the original source code [31] and each aspect can implement their customized actions, our work clearly defines what DSMC can do (that is, adaptation interface and action sets) and when they can do it (that is, only during adaptation process). Therefore we believe that conflicts could be identified and resolved in a much easier way.

One of our current works is to extend current middleware with IActutatorModel interface to represent Adaptation Actuator’s ontology. At the same time, the OWL language [32] is used to build the ontology of actuators, which allows the system to automatically parse the relationship between user custom actuators and allows user specific Adaptation Actuators to be automatically integrated by using semantic interference logics. A detailed discussion on this can be found in [20].

Algorithm 1: Hardcoded Model Fusion with three types of actuators

Ensure: system produces an adaption plan without conflicts in terms of Actuators semantics

\[ \text{alladaptationplans} \] — set containing all adaptation plan
\[ \text{SystemContext} \] — class instance that keeps system current installed components and their properties and references
\[ \text{[satisfied]} \] — set containing all satisfied components that can be activated
\[ \text{[unsatisfied]} \] — set containing all satisfied components that should be disabled
1. \[ \text{[satisfied]} = \text{[structure satisfied components]}; \]
2. for each adaptation plan \( p \) in \[ \text{[alladaptationplans]} \]
3. \[ \text{[satisified]} = \text{[satisified]} \cup p.getSatisfiedComponents(); \]
4. end for
5. \[ \text{[unsatisfied]} = \text{systemcontext.getEnabledComponents()} - \text{[satisfied]}; \]
6. \[ \text{[orderedadaptationplans]} = \text{NCMD\_decrease\_ordered ([alladaptationplans])}; \]
7. \[ \text{[new property set]} = \{\}; \]
8. for each adaptation plan \( p \) in \[ \text{[orderedadaptationplans]} \]
9. for each prop in the \( \text{[newproperties(p)]} \)
   // if the property has not been set by other DSMC with higher NCMD
10. if prop \( \in \{\text{new property set}\} \)
11. \[ \text{[new property set]} = \text{[new property set]} \cup \text{prop}; \]
12. end if
13. \[ \text{[newly satisfied components]} = \text{[satisfied]} - \text{systemcontext.getEnabledComponents}(); \]
14. \[ \text{[newly unsatisfied components]} = \text{[unsatisfied]} - \text{systemcontext.getDisabledComponents}(); \]
15. \[ \text{fusedplan} = \text{getfusedAdaptationplan([newly satisfied components], \}} \]
\[ \text{[newly unsatisfied components], [new property set]);} \]
16. return fusedplan;

4.2.6. Validation Service. In the Model Fusion module, an algorithm is provided to explicitly fuse multiple adaptation plans into one conflict-free global adaptation plan; however, based on our current implementation, there is still no guarantee that the fused plan will perform as intended. How to evaluate the effectiveness and efficiency of the fused global modeler is one of the major issues in our system design.

As the effectiveness and efficiency can only be evaluated according to current context optimization goal, it is very unlikely to have a general validation policy to deal with so diverse context environments. Rather than providing solution for specific contexts, our middleware system provides a so called Validation service to expose system internal states — for instance, the list of enabled components, currently used DSMC, and the adaptation actions proposed by each modeler.
as well as a trace of the adaptation actions being executed. Third-party analysis programs can be installed to validate adaptation decisions taken by various modelers, on-line or logged for off-line analyses. The service also allows also external modules to “revise” adaptation plans composed by Model Fusion. This information can help system users to validate various modelers including the DSMC Selector. For instance, a “ping-pong” effect among several DSMC in a short time might indicate an ill-designed DSMC Selector.

Currently, a prototype of loop detection and avoidance algorithm is designed and implemented. This prototype uses this service to detect and break possible unlimited adaptation loops. Theoretically, all configuration changes — including lifecycle state changes as well as property configuration changes — need to be tracked. Due to the huge number of states and corresponding high overhead, our current implementation only tracks installed component’s basic lifecycle states for on-line adaptation loop detection and resolution. Detailed introduction of this algorithm is out of the scope of this paper.

4.3. System Adaptation Process

In order to better demonstrate the interactions between these key modules, in this section, a UML sequence diagram is provides to illustrate this process. Here, a general process is described which can be used for custom adaptation process. As described in Fig. 4, the selection process works as follows:

The SBR will monitor all the native OSGi events and user custom events through the IEventMonitor interface. When the SBR reaches a point at which certain adaptation needs to be taken, the SBR asks the DSMC Selector for a set of contextual matching DSMC. The DSMC Selector checks the meta-data from all registered DSMC by invoking each DSMC’s getDSMCMetaData method. The context matching degree will be calculated by the NCMD module through ICMDComputation interface. The NCMD module returns all the DSMC’s normalized matching degrees via the method “List CMDCalculation (List context, List<DSMCMetaData>).”

Figure 4. Sequence diagram for the general adaptation process
Then, DSMC Selector returns the references of those DSMC whose NCMD is greater than the system assigned threshold (in this case, 0.5).

After the basic system run-time gets those DSMC' references, it iterates through all the DSMC references and invokes method “resolveAdaptationPlan(SystemContext sc, ChangeEvent cev)” from each DSMC to get DSMC-specific adaptationPlan. Each adaptationPlan designs a set of actions that should be taken from the DSMC’s perspective. Currently, the adaptationPlan contains a list of DSMCSatisfiedComponents, and newProperties to describe the preferable target state which can be converted to adaptation actions needed to be taken.

After these adaptation plans are returned from individual DSMC, the SBR sends those adaptation plans to the Model Fusion module. This module fuses the list of adaptation plan via Algorithm 2, which prioritizes the adaptation plans according to DSMC’ context matching degree. After collecting the fused adaptation plan from Model Fusion, the SBR sends it to the Validation module, if it exists, to identify possible adaptation problems. Currently, our on-line loop detection and resolution algorithm is used. After Validation process, the fused adaptation plan will be sent back to SBR for execution.

In this process, by separating three kinds of responsibilities: 1) knowing when a modeler is useful, 2) selecting among different modelers, and 3) generating a conflict-free adaptation plan, our software platform integrates different modelers and various knowledge into the system during the run-time, thus extending the system adaptation features while being transparent to system managed applications.

4.4. Architectural Reconfigurability

In addition to the core functions discussed in the previous section, this middleware architecture is also designed to be extendable through the adoption of the service component model. Alternative realizations can be provided to optimize the functional implementations. As those modules are designed under service component model, they can be easily installed and exchanged during run-time.

For instance, as can be seen in the left part of Fig. 5, two variants of the IModelFusion service implementation are provided. We can use a simple Model Fusion module as we described in the previous section or use a more intelligent Model Fusion module which takes user’s feedbacks to readjust the fusion policy. This is one of our ongoing research works. In the right part of Fig. 5, two implementations of the ICMDCalculation are provided, using our NCMD based calculation or applying semantic graph matching scheme [33]. As the semantic-based context matching scheme takes the ontology among context resources into account, it can achieve more accurate context matching degree compared to our existing NCMD solution. Detailed discussion is out of the scope of this paper.

By selecting an appropriate service provider for those services, the middleware realization can better match the requirements of the deployment environments. This approach also facilitates the evolution of middleware structure to provide a better implementation to the targeted system.
As discussed in the previous section, our current implementation focuses on providing a general solution that allows adding arbitrary adaptation capability during run-time. This matches those systems that must run in uncertain environments with adaptation behavior evolutions capabilities. Due to the complexity of involved factors, our current implementation is not capable to guarantee correct adaptation behaviors. In particular, the Model Fusion module is still under active development. However, this reconfiguration feature allows the deployed system to be easily tailored towards certain, for instance safety-critical environments, in which guaranteed adaptation behavior is one of the key requirements.

As an example, this middleware can be easily reconfigured with custom DSMC Selector and Model Fusion modules. The custom DSMC Selector service provider only selects those DSMC that have been fully tested and verified in predefined contexts. A Model Fusion can be custom designed and thoughtfully tested for all predefined combinations of possibly used DSMC. In this way, this system can work with more expected adaptation behaviors, at the cost of the flexibility to add arbitrary adaptation capabilities during run-time. Many existing context-aware adaptation solutions can be integrated into our framework in this form. We are also working on a hybrid approach, which uses pre-calculated system configurations for certain known “context scenarios” to guarantee system correctness while the existing best-effort approach will be used in the other unspecified contexts.

5. CASE STUDIES

In this section, we explain how our framework solves the problem described in Sect. 2. Firstly, we discuss the case study of the contextual DSMC selection.

5.1. Context-specific Selection

In this middleware design, the Context Manager will provide update information of current context factor values. Such values can be accessed by using interface IContextAccess. In this case study, only two different context resources are considered: option.optimization.target (a string) and option.selfhealing (a Boolean).

When the context changes, for instance when the user changes his/her optimization.target from TV application to MobileTV application, different DSMC will have different context matching degree reflecting the new context. Thus, a different set of DSMC will be chosen. The following section will describe those context matching conditions for each DSMC.

5.1.1. Context Matching Conditions for Individual DSMC. Basically, the functional part describes a DSMC’s name, implementation, provided and required service, etc. For the sake of limiting the length of this paper, the functional part of the DSMC meta-data will not be described here. The CMC for these three DSMC is listed as follows:

DSMC for TV optimization:

option.optimization.target, string, ==, TV, 1.0;

DSMC for Self-healing:

option.selfhealing, Boolean, ==, true, 1.0;

DSMC for MobileTV: option.optimization.target, string, ==, MobileTV, 1.0;

In this case study, in order to limit the number of involved context factors, each DSMC is assigned with only one condition. Obviously, compared to the real life case, this is merely an illustrative example to demonstrate the DSMC selection scheme.

5.1.2. NCMD-based Selection. In this case study, we envisage the user’s preference is as follows: a) system with fault-tolerant adaptation capabilities b) the user focus on a single optimization target (which, in this case, can be TV or MobileTV ). Of course, context manager can provide context
information with more context factors. As the CMC in the case studies only involves two context factors, only these two related factors are listed.

When the context changes, the NCMD of specific DSMC also changes, which results in a different set of selected DSMC. For instance, as can be seen in Table I, when the context factor optimization.target is TV application and self-healing is set to true, DSMC self-healing and DSMC TV will be selected. In contrast, when the user’s optimization target is set to MobileTV application, DSMC-MobileTV and DSMC Self-healing will be chosen. If the user does not want to have self-healing and wants to optimize TV application, then only DSMC TV will be selected.

Table I. NCMD based DSMC selection

<table>
<thead>
<tr>
<th>Context factor value</th>
<th>NCMD</th>
<th>Selected DSMC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Self-healing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TV</strong></td>
<td>1.0</td>
<td>{DSMC-selfhealing, DSMC-TV}</td>
</tr>
<tr>
<td><strong>Recording</strong></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>optimization.target=TV</strong></td>
<td>1.0</td>
<td>{DSMC-selfhealing, DSMC-MobileTV}</td>
</tr>
<tr>
<td><strong>option.selfhealing = true</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>optimization.target=MobileTV</strong></td>
<td>1.0</td>
<td>{DSMC-MobileTV}</td>
</tr>
<tr>
<td><strong>option.selfhealing = true</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>optimization.target=MobileTV</strong></td>
<td>0</td>
<td>{null}</td>
</tr>
<tr>
<td><strong>option.selfhealing = false</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>optimization.target=</strong></td>
<td>0</td>
<td>{null}</td>
</tr>
<tr>
<td><strong>option.selfhealing = false</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.2. Adaptation Strategy

As we discussed in Sect. 2.2, applications, such as a TV application and a MobileTV application, will interfere with each other while competing for system resources — in this case, the TV and MobileTV applications could not run simultaneously with maximal quality due to lack of system resources.

5.2.1. DSMC **TV**. In order to maximize the user’s TV watching experience, two main sets of adaptation policies are implemented in this modeler — one is to always use the component with higher quality to build TV application; another one is to always allocate enough resources to the TV application by disabling other components.

As we can see, the adaptation logic of this DSMC is implemented as Alg. 2. The first part of it is to select components with highest quality to build TV application: another one is to always allocate enough resources to the TV application by disabling other components.

5.2.2. DSMC **Self-healing**. In order to implement the self-healing capability, the system needs to acquire the updated state of targeted components. The management interface of our component interface is used to get all properties values to date. This can also help to check whether a component works correctly. If it raised exceptions, we shall deduce that component might have problems. In such a case, the repairing process begins. Such process is now described.

As in our current prototype only three types of actions are supported (enable, disable and change property), no single actuator can directly repair a faulty component instance. However, this action can be built up by orchestrating three actions in the following 3 steps:

1. Detect whether there is an error; if yes, add faulty component (denote as faulty) to faulty-ComponentList. At the same time, remove faulty component from SatisfiedComponentList. This step will disable the faulty component and remove its resources including its registered service interface in service registry.
Algorithm 2: Use components providing best quality

Requires: Installed components CPU usage information

Ensure: always use the component with highest TV Quality and Allocate enough CPU time to the TV application

1. for all cmp in satisfiedComponents
2. visited =false;
3. for all cmp in satisfiedComponents
4. if cmp is for TV application and (not visited)
5. similarset = all components with same functional contract ;
6. visited=true for (similarset)
7. select one with highest quality attributes cp_quality_high;
8. remove all components except cp_quality_high
   from enabled component list(ecl);
9. end if; end for ;
10. // check for performance violation
11. if ((Decoder.getProperty(overrun_increase) > Max_threshold)
    and (Decoder.getProperty(Quality)<LowerBound))
12. for all cmp in satisfiedComponents
13. if component not from TV application
14. cmp.getProperty(CPU_consumption) from cmp attributes ;
15. record the cmp with highest CPU usage;
16. end if; end for
17. remove cmp from enabled component list(ecl);
18. end if
19. set ecl back to adaptationPlan
   . . . other adaptation logics, omitted . . .
20. return adaptationPlan

2. After a component is disabled, the component’s provided service is disabled. This event will trigger another round of adaptation. If DSMC’s faultyComponent list is not empty, add faultyComponent to the enabledComponent List. After this step, this component — denoted as A — will be re-initialized for reparation.

3. After component A is initialized and registers its provided services, this event will trigger another round of adaptation. DSMC Self-healing will set the properties of repaired components to the newly initialized component instance.

As we can see, the adaptation process is built in a logic order and takes 3 steps to repair a faulty component. The main reason of this design is due to the limited adaptation actions that our system supports. Other implementations (for instance, performing adaptation inside the DSMC’s reasoning logic) might work more efficiently. However, such an approach would violate our design principles to separate the actuator from the resolving process. Bypassing the fusion process might result in unchecked adaptation possibly leading to unexpected consequences. Another solution is to provide more complex actuators, for instance, turning “repairing component” into an atomic action. However, in our experience, the richer and the more complex the basic adaptation action set, the more difficult will be the formulation of effective fusion rules.

5.3. Step-wise Application Construction

As mentioned already, adaptation actions might trigger additional adaptation process. System will stop adaptation when there is no next adaptation action to be performed. Thus, this adaptation process is built step-wise. Here, we will demonstrate how TV application is constructed with multiple modelers co-leading the step-wise adaptation process. As we can see from Fig. 6, two decoder components with different quality metrics are available. This ambiguity cannot be solved by Structural Modeler alone as it does not only depend on structural constraints.

Three modelers are involved in the construction process of the TV application. However, as DSMC Self-Healing will only join the adaptation plan when an error is detected, for simplicity of discussion here we only discuss those modelers which affect the construction process — Structural Modeler and DSMC TV.
1. Stream Receiver will be enabled as it is “structure-satisfied”, and the same applies to the DSMC TV. The enabled component list will then be Stream Receiver. Its service interface will be registered automatically by our system run-time.

2. After the service of Stream Receiver is registered, Decoder_A and Decoder_B will be assessed as structure-satisfied by Structural Modeler; the enabled component list of Structural modeler will then become \{Stream Receiver, Decoder_A, Decoder_B\}. Because of the TV optimization rule, only Decoder_A will be enabled as it provides higher quality. So the fused enabled component list becomes \{Stream Receiver, Decoder_A\}. As Stream Receiver is already enabled, only Decoder_A will be enabled.

3. TV Rendering component will then be enabled, after the service interface of Decoder_A gets registered. TV application will then be successfully constructed.

5.4. Discussions on Conflict Resolution

In the motivational example, we point out that even simple adaptation logics for TV and MobileTV applications (DSMC TV and DSMC MobileTV), due to their totally different adaptation goals, will generate conflicts. Other modules, such as Structural Modeler, can also generate different adaptation plans with other DSMCs.

In Transformer, most conflicts are avoided as the setting of NCMD makes DSMC TV and DSMC MobileTV not eligible for being used simultaneously. It is obviously an simplified setting. For instance, with CMCs as following settings, it is possible that both DSMC can be used for adaptation.

For DSMC for TV optimization,

\[
\text{option.optimization.target, string, ==, TV, 0.6; resource.cpuusage, <, 0.6, 0.4;}
\]

DSMC for MobileTV

\[
\text{option.optimization.target, string, ==, MobileTV, 0.6; resource.cpuusage, <, 0.5, 0.2; resource.bandwidth, <, 2, 0.2}
\]

With this setting, when \{\{target, MobileTV\}, \{cpuusage, 0.5\}, \{bandwidth, 1.5\}\}, the NCMD for DSMC TV is 0.4 while DSMC MobileTV’s is 0.8, both greater than the selection threshold 0.3. As those NCMD reflect how similar the DSMC is with respect to the current context, we assume that the higher the NCMD, the more accurate the adaptation actions that can be calculated by a DSMC will be. In the case at hand, DSMC MobileTV has higher context matching degree. Thus, when DSMC TV sets TV quality to, for instance, 66 while DSMC MobileTV set TV quality to 50, the adaptation actions proposed by DSMC MobileTV will be used after the Model Fusion process.

\[
\{TV, quality=66\} \rightarrow \text{Model Fusion} \rightarrow \{TV, quality=50\}
\]  

(6)

Of course, here, other adaptation conflicts can also occur between structural modelers and DSMC for component lifecycle management. For instance, structural modeler denote a component should
be “enabled” and DSMC TV denotes it should be “disabled”. As discussed in Sect 4.2.5, those conflicts related to lifecycle management can be resolved based on the principle that a component can only be enabled when all modelers specify that it can be enabled. Thus, in this case, this component should be disabled.

6. SIMULATIONS AND COMPARISONS

In this section, we validate our framework design and implementation both from a qualitative and a quantitative point of view including such concerns as implementation complexity, adaptation flexibility, overhead introduced by our framework, etc.

6.1. System Implementation Complexity

We use the OSGi framework as our implementation platform. OSGi [25] 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 its current state, our implementation focuses on providing a lightweight implementation for local application management. We also extended the monitoring and actuation to distributed environments by using Distributed OSGi [34].

<table>
<thead>
<tr>
<th>Modules</th>
<th>Sub-modules</th>
<th>Lines of code</th>
<th>Binary size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitoring</td>
<td>Reflections of code</td>
<td>142</td>
<td>2353</td>
</tr>
<tr>
<td></td>
<td>Monitoring</td>
<td>599</td>
<td>7913</td>
</tr>
<tr>
<td>Parsing</td>
<td>Model classes</td>
<td>1076</td>
<td>26932</td>
</tr>
<tr>
<td></td>
<td>Parser class</td>
<td>1075</td>
<td>45375</td>
</tr>
<tr>
<td>Structural Modeler</td>
<td>Functional constraints</td>
<td>975</td>
<td>20897</td>
</tr>
<tr>
<td></td>
<td>Reference management</td>
<td>201</td>
<td>5988</td>
</tr>
<tr>
<td>Adaptation Actuator</td>
<td>Dispose management</td>
<td>459</td>
<td>11782</td>
</tr>
<tr>
<td></td>
<td>Instance management</td>
<td>359</td>
<td>8789</td>
</tr>
<tr>
<td></td>
<td>Meta-Function Invoking</td>
<td>296</td>
<td>6950</td>
</tr>
<tr>
<td>Model Fusion</td>
<td>Merge two sets of adaptation plan</td>
<td>156</td>
<td>5740</td>
</tr>
<tr>
<td>DSM Selector</td>
<td>Normalized CMD</td>
<td>247</td>
<td>9370</td>
</tr>
<tr>
<td>Auxiliary code</td>
<td></td>
<td>700+</td>
<td></td>
</tr>
</tbody>
</table>

Currently the DRCom component modelgui08a,gui08b developed in our previous researches is supported in Transformer. The current version of DRCom enhances the OSGi declarative Service component model with supports for context requirement descriptions. The figures listed in Table 2 shows the complexity & footprint of each modules in terms of line of code and compiled binary file size.

As discussed in Sect. 3, this system is implemented via 6 key modules. The lines of code of each implemented module are shown in Table II. 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. The size of those compiled modules are also very small which , to some extend, enables Transformer to be used in resource-constrained environments.

One of the modules that are not mentioned in Sect. 4 is the Meta-data Parsing module. 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. Clearly the implementation complexity of DSMC is highly implementation specific, thus the lines of code of this module is omitted.
6.2. Adaptation in Different Contexts

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. This includes, to name but a few, modules for detection, modules for component management, adaptation logic as well as the execution modules. Changing system adaptation logic actually means that almost the whole part of system run-time needs to be redesigned and redeployed. This hard coupling makes changing a system adaptation behavior very complex and almost impossible during run-time.

**Development complexity in supporting multiple contexts:** In this paper, the adaptation logics are run-time formed by composing several DSMCs’ adaptation plans. Each DSMC only deal with domain-specific adaptation knowledge. Compared to the traditional “one-modeler-for-all” solutions such as the one in [5, 36, 9], this feature allows the DSMC to be implemented in a much simpler way.

Without the burden to implement software maintenance tasks, a DSMC adaptor can be implemented very concisely. For instance, one adaptation for TV optimization adaptation (described in Alg. 2 can be implemented in less than 120 lines of codes. On the other hand, an ad-hoc approach would need re-implementing a new version of a basic component management run-time (in our case, about 2000 lines). Thus, programmers can focus on one domain-specific adaptation concern rather than having to take care of those low level details.

<table>
<thead>
<tr>
<th>Context change</th>
<th>Approaches</th>
<th>Binary reusable module</th>
<th>Non reusable modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>TV → MobileTV</td>
<td>Standalone approach</td>
<td>None</td>
<td>Structural + Self-healing + TV</td>
</tr>
<tr>
<td></td>
<td>ACCADA</td>
<td>Structural Modeler</td>
<td>Self-healing + TV</td>
</tr>
<tr>
<td></td>
<td>Transformer</td>
<td>Structural Modeler + Self-healing</td>
<td>TV</td>
</tr>
</tbody>
</table>

**Difference in reusability:** Three different approaches are compared in terms of reusability: (a) the stand-alone approach, combining system run-time with all adaptation strategies (stand-alone), (b) the one separating the functional maintenance requirements and Contextual requirements of our previous approach (ACCADA) [19] and (c) the Transformer framework reported in this paper.

In (a), adaptation modules are tightly coupled with system run-time, thus it is impossible to reuse the adaptation module in other environments. In (b), Structural Modeler is separated from other adaptation modules and can be reused across multiple contexts while the contextual adaptation logics cannot be reused. In contrast, (c) provides the most reusability towards those adaptation logics. Each DSMC can be possibly reused to composite more complex adaptation logics, just as multiple components are used in composing applications.

As we can see from Table III, from “TV Optimization” context switch to “MobileTV Optimization” context, the traditional standalone approach (a) cannot reuse the binary code of the adaptation modules. In (b), Structural modeler is separated from other adaptation logic and can be binary reused across multiple contexts during run-time. In (c), most of the modules can be binary reused during run-time. In particular, Table III shows how Structural modeler and DSMC for Self-healing adaptation logic can be directly reused (about 2000+230 lines of code).

In our current framework, each DSMC can be independently deployed and (possibly) reused across multiple contexts. For instance, the DSMC for Self-Healing are used in both TV and MobileTV contexts. During context changes, only the adaptation strategy related to that domain should be altered and all the remaining modules can be kept largely unchanged. For instance, we can simply add self-healing feature to the system by adding/removing the corresponding DSMC to/from the registry. All these adaptations happen during the run-time. This lets the system deal...
more effectively with context transitions. Table IV provides a more exhaustive comparison between application specific adaptation, ACCADA and our current framework.

6.3. Architecture Performance

To evaluate the performance of the system adaptation, 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 provided interface, one required interface, and one attribute (except the first component which only has one provided service interface). The newly installed component is structure-satisfied when installed. The size of each component is the same — 20.6 KB. In order to avoid the impact of component execution towards simulation results, all these component execution parts are disabled during the experiment. Here, as we are only interested in the framework performance, when installing a new component, the component initialization time is not counted as it may vary according to different implementations.

In this test, only one DSMC will be used. Its major function is performing a simple admission control algorithm: the arrival component will only be enabled when constraint (7) holds true.

$$\sum \frac{\text{execution time}}{\text{period}} < 1$$  \hspace{1cm} \text{for all enabled components.}  \hspace{0.5cm} (7)

As hardware platform, 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 SUN JAVA 1.6.0.2 SDK on Windows Vista.

Installing a new component normally consists of five main steps: component loading, meta-data processing, structural modeling, context modeling, and Model Fusion for merged adaptation plan. The actuation time is not shown here as it depends on each component implementation. Figure 7 shows the absolute times spent in each step of the process. Each value is the arithmetic mean of 250 runs of the experiment. In order to better illustrate the trend of different steps, we use two Y direction axes in expressing the 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).

As we can see from Fig. 7, component installation time grows slowly 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 account for about 60%–80% of total time — keep comparably stable when \( n \) grows. In contrast, the other three key elements will grow with the number of managed components. The structure modeling process mainly deals with matching composability between installed components which has computing complexity \( O(n) \).
DSMC here will check whether the new component can satisfy the resource requirements which also have complexity $O(n)$ (stateless implementation, no optimization). Here, the Model Fusion processing time is for post-processing the modeling results from two modeling processes. The fusion process uses Alg. 1 to fuse adaptation plans from multiple adaptation modules. The complexity is mainly from the set functions and sort functions whose complexity grows with the number of proposed adaptation actions.

Simulation results show that our framework scales well when the number of managed components grows. However, the DSMC processing time confines to the simple algorithm described here in which only one DSMC is used. In the next section, we evaluate the performance when multiple DSMC are involved in the adaptation process.

6.4. Multiple DSMC Performance Evaluation

In this section we test the overhead of managing multiple adaptation logics into separate modules as compared to standalone approaches. Each domain-specific adaptation modules is tested for execution time. As this execution time is affected by the execution route, we also compared the execution time in two different conditions, that is “TV application Structure Check” and “faulty Component detected”. Each simulation is performed 1000 times and the average values are calculated to soften the impact introduced by e.g. Java’s garbage collection.

<table>
<thead>
<tr>
<th>Table V. The performance of adaptation modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Standalone</td>
</tr>
<tr>
<td>Transformer</td>
</tr>
</tbody>
</table>

**Execution Time:** As we can see from Table V, in different adaptation processes, there is no significant overhead difference in terms of execution time. For instance, for TV application construction (recomposition process), about 26.28$\mu$s are needed for standalone solution while DSMC based solution needs about 27.418$\mu$s. The separation of adaptation logics into discrete adaptation modules only introduces little overhead mainly from service calls. Less than 2$\mu$s overhead are needed for the separation of adaptation modules. If an error is identified, about 30$\mu$s are needed for adaptation, in which self-healing modules take about 22$\mu$s and TV optimization process needs about 8$\mu$s. We can also see that the execution times for two different adaptation modules are
Model Fusion Time: Another key overhead introduced in our approach is due to the Model Fusion process. In order to analyze the performance of such process, we introduced one simple DSMC, which only iterates through all installed components and gets their properties. This DSMC tags all installed components as “enabled” and performs one setProperty action. This DSMC is deployed multiple times (from 1 to 9) to simulate multiple DSMC adaptations. Each instance computes actually exactly the same adaptation plan. This design actually creates the worst case scenario for Model Fusion (maximum interference between DSMC). Each set fusion operation needs about $O(n \times m)$ time for the fusion complexity. In this worst case scenario $m$ is equal to $n$.

At this point, the plan generated by different DSMC will be fused into one adaptation plan according to the fusion rule set specified in Sect. 4.2.5. From Fig. 8, we can see that the Model Fusion time depends on two basic factors: one is the number of managed components, and the other factor is the number of utilized DSMC. The more DSMC are used, the more Model Fusion time is required. However, such time grows about linearly with the number of utilized DSMC. According to our programming practice, for any given context, normally less than 5 DSMC will be used. This means that for each round of adaptation, less than 30$\mu$s fusion time can be expected.

Another performance penalty in our prototypic design stems from the fact that for each adaptation action, all the enabled DSMC are inquired, even though the current event might not pertain all the enabled DSMC. For instance, when Event Monitor notifies event “Decoder frame dropping rate > 2%”, it raises an adaptation process which involves Structural modeler, DSMC TV and DSMC Self-healing. However, such a change does not pertain the Structural modeler and the DSMC Self-healing. Our on-going work is to design a more fine-designed interface that enables each DSMC to declaratively register their interested system events, ranging from general application adaptation events to application specific events. During system evolution, only those DSMC involved with the changes will be invoked. From our current experience, this approach can greatly decrease the overhead. However, it also needs more flexible modeler fusion rules which are still one of our key future research topics.

6.5. Self-Healing Experiments

Figure 9 compares the processed video frames of the TV application, in the presence of node failures, when the overall infrastructure is managed by a human administrator versus our multiple DSMC architecture. This TV application is composed as in Fig. 6. In order to avoid its impact towards system adaptation, the processing part is set to be empty. Our framework is triggered by periodic task checking for faults every 10 seconds. To compare the approach in the same level, we...
Figure 9. Processing video frames with human/DSMC based Healing

assume a human being will also check system correctness every 10 seconds (which is rather frequent for human beings). Error is introduced at 15, 205, 445 seconds.

Under human administration, when a failure occurs in our system (simulated by setting the Decoder component to unavailable), the MobileTV application becomes unavailable and all the video data gotten from remote site could not be handled which results in Video data loss. In order to handle this failure, the system operator must react as follows: (1) first, he/she must detect the failure; (2) the operator must know how the application is composited; (3) based on application construction knowledge, he/she should understand the failures and decide what reconfiguration adaptation is needed. (4) The operator has to unload the failed component instance and install a new component instance. (5) The newly initialized component’s property are restored to the values just before error happens. (Whether this requirement can be fulfilled depends on whether the system has a logging service). In our simulation, it takes at least 50 seconds for the whole process. During this time, video data (at least 1500 frames of video data until the end of the recovery) will be permanently lost.

By way of contrast, the frame loss is much less than in the human-based adaptation when DSMC for Self-healing is used. In our simulation, less than 10ms is needed to repair the errors (for the whole adaptation process, 3 rounds of adaptation, although for each round of adaptation, about $22\mu s$ is needed for reasoning), system recovery-time is mainly attributed by the 10 second interval of system error check, which contributes to an average delay of about 5 seconds.

7. RELATED WORK

There is a substantial body of literature on reconfigurable middleware systems. First, our framework builds on our earlier work on our context-specific composition framework [19]. Compared to this earlier work, our framework exhibits a richer and more coherent set of features to support multiple domain-specific knowledge and a more complete design and implementation considerations towards the requirements of context-specific run-time component composition.

JMX [37] 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; the dependencies between the managed objects are not exposed and managed. Thus an application can only be constructed via its own property methods.

SmartFrog [6] 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, 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.

Sicard et al. [5] identify novel requirements on reflective component models for architecture-based management systems. The construct layer is designed for the meta-data checkpoint and replication. The interfaces and processes for self-repairing are defined, such as lifecycle management, setter/getter interface as well as the 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, their approach does not have a clear definition of and separation between system services. Such hard-wired architecture makes it very difficult to reuse their framework across different contexts.

Garlan et al. [2] 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 [8] 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, the static adaptation policy and the resolving process limit its usage in the changing environments.

Kasten et al. [9] propose the Perimorph framework to achieve run-time composition and state management for adaptive system. Perimorph 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 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 the complex dependence between components, Kon et al. [4] propose an integrated architecture for managing dependencies in distributed component based systems. The architecture supports automatic configuration and dynamic resource management in distributed heterogeneous environments. Their approach provides the explicit context-knowledge support in deal with component dependence. However, how to support the context changes is not specified in their approach.

Fujii and Suda [22] propose the SeGSeC framework to compose an application by synthesizing its workflow based on the semantics of components and contexts of users. Their framework uses the semantics similarity and learning based algorithm in their synthesizing process. Their approach supports context-specific work flow synthesizing. However, their framework does not consider how to systematic integrate the structural constraints with context-specific modeler. Their system computes the preferred flow plan but no guarantee is provided that that plan will be reached. It is also not clear how new adaptation policy can be supported in this system.

For the conflicts resolving among different models, in the context of Aspect-oriented programming, in [31] the authors identify the problems and propose a formal way, based on off-line model analysis, to detect semantic conflicts between aspects. By providing an extended UML class diagram for Aspects-Classes Connections Modeling they identify those relationships between Aspects and Classes and possible conflicts. As Aspect-oriented programming mainly works at source code level, it is very hard to accurately know the semantics of those weaved codes. In contrast, our framework provides a well-defined adaptation interface with well-defined semantics of available adaptation actions and their relationships. Those adaptation plans are collected and reasoned at global level. Compared to Aspect-based programming, our approach provides reduced complexity in implementation and more ease to identify the conflicts.
8. CONCLUSION AND FUTURE WORK

In this paper we have described our approach to adaptation evolution with reusable adaptation modules. We have shown in particular how to integrate multiple domain-specific adaptation logics with run-time adaptation composition by our Transformer adaptation framework. As components are used in building applications, in our framework, different DSMC are used to build up more complex adaptation logics. Service-oriented middleware is implemented followed the design of the adaptation framework to demonstrate the feasibility of this framework. As our framework makes strict definition of adaptation actions and adaptation interface, we are able to detect conflicts in a systematic way. We also provide an explicit Model Fusion module to detect and resolve possible conflicts. Compared to other frameworks, our system can greatly enhance the reusability of adaptation modelers across multiple contexts. This approach has been compared with other application-based adaptation frameworks e.g. in terms of code lines and memory usage. 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.

We introduced a general approach to provide adaptation evolution support as well as multiple adaptation concerns. However, several limitations of our current implementation hinder its application in certain mission-critical areas. We identify the following areas as still open for further research:

Firstly, in our system, those fusion rules are defined with generic guidelines that provide basic conflict resolving mechanisms. In our motivating scenarios these rules appear to be effective. However, the one described here is by no means the only possible fusion rule set. How to effectively create, validate and evaluate a set of Model Fusion rules for arbitrary combinations of the DSMC’s and context remains an open question. Currently, a semantic actuator model is being designed based on OWL [32] to support user custom actuators so that their relationships can be reasoned upon via existing OWL tools.

Secondly, for each DSMC, meta-data is used to express its context-matching conditions; however, the format of our meta-data is currently not powerful enough to specify DSMC supported events, adaptation actions and possible conflicts. Lack of this information makes the fusion rules hard to be defined and verified. Currently, we are adding meta-data to describe a DSMC’s events of interest and invoke the modelers only when one of those events occurs. Preliminary measurements already show significant improvement of performance with much less fusion actions and conflicts.

Thirdly, deeper analysis of the effectiveness and correctness of the fused model is needed. As the system adaptation behavior is determined by the DSMC, DSMC selecting algorithm and the fusing algorithm, un-intended adaptation behavior might be generated and possibly lead the system into unsound configurations. In this paper, a reflective service is designed and an on-line detection and verification algorithm was proposed to detect and break adaptation loops. Such algorithm cannot handle more complex errors e.g. inappropriate adaptation actions. This limitation makes our system currently unfit to be used in mission-critical systems. However, this uncertainty can be largely reduced by limiting the number of DSMC and possible contexts the system can experience.

ACKNOWLEDGEMENTS

This work is partially supported by the Zhejiang Provincial Natural Science foundation of China under Grant No.Y1100289,Y1100726,Y12F020194, and Science Technology Department of Zhejiang Province Grant No. 2011C33G2060108. This work is partially supported by IBBT (the Interdisciplinary Institute for Broadband Technology), an independent research institute founded by the Flemish government.

REFERENCES


32. W3C Foundation. Web ontology language (owl), W3C OWL working group, v 2.0. [http://www.w3.org/TR/owl2-overview/](http://www.w3.org/TR/owl2-overview/).


