Monitoring of Quality of Service in Dynamically Adaptive Systems

Sihem Loukil[1], Slim Kallel[1], Ismael Bouassida Rodriguez[2,3], and Mohamed Jmaiel[1]

[1] ReDCAD Laboratory, University of Sfax, Tunisia
[2] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[3] Univ de Toulouse, LAAS, F-31400 Toulouse, France
sihem.loukil@redcad.org

Abstract. Dynamic reconfiguration has been widely recognized as an effective approach to deal with the increasing complexity of dynamically adaptive systems. One of the main challenges in such systems is to provide guarantees about the required runtime quality of service (QoS) attributes, such as performance, reliability, etc. Therefore, it is of paramount importance to make these systems able to monitor the QoS parameters that allow to evaluate such QoS attributes, analyze these parameters in order to detect QoS changes and therefore trigger reconfiguration actions. In this paper, we propose an approach that allows monitoring the QoS parameters of a dynamically adaptive system in order to detect QoS degradation. The proposed approach is based on the Aspect-Oriented Software Development (AOSD) paradigm which allows to keep the monitoring code separated from the business logic code.

1 Introduction

Dynamically Adaptive Systems that can automatically adapt to changes in their environments are increasingly requested [1]. To further increase its usability and reliability, such system needs to keep a certain Quality of Service (QoS) level during its execution. Therefore, it should be able to monitor some quantifiable parameters that allow to evaluate its QoS attributes. Such monitoring allows detecting degradation in the QoS of the system and therefore adapt its structure and/or behavior autonomously in response to this degradation.

As dynamically adaptive systems change behavior and structure at runtime, the monitoring of the QoS of such system poses some challenges. First, these systems need to continuously monitor QoS parameters in order to detect QoS degradation. These parameters are collected from different interacting entities that may be distributed which may require several QoS monitors displayed at different levels. Second, the level of the QoS specification is relatively low which requires access to source code to specify or modify QoS parameters such as data rate, error rate, etc. Thus, it is desirable to specify QoS parameters at a higher level of abstraction and then automatically map these specifications to source

code while providing sufficient flexibility. Third, the QoS concern need to be considered in several parts of the system. Mixing such concerns with business logic concerns increases the complexity of the system and makes both its development and maintenance more difficult [2].

In our previous work [3, 4], we presented an approach which mainly consists in conceiving a whole development process of dynamically adaptive systems ranging from modeling to code generation. This approach allows to specify component-based systems and ensure their reconfiguration at runtime. It allows to handle both the anticipated and unanticipated reconfigurations at design time. This approach is based on the combination of the Architecture Analysis and Design Language (AADL) [5] and its aspect oriented extension AO4AADL [6, 7].

In this paper, we present how our approach can be applied to monitor the QoS of a dynamically adaptive system. For this purpose, we first define and classify QoS parameters that can be monitored at runtime. The proposed classification is based on the architectural vision of distributed component-based systems. More specifically, this classification is tied to distributed component-based systems specified using AADL concepts. Second, we detail how our approach allows specifying QoS parameters at a high level of abstraction using the AOSD paradigm. In fact, several aspects are defined in our work in order to monitor QoS parameters at runtime. These aspects are specified at architectural level using the AO4AADL language. These AO4AADL aspects will be the input of an AspectJ generator, developed in our previous work [6, 7], in order to generate aspects in AspectJ [8]. These aspects will be automatically weaved with the functional code of the system to obtain the final code to be executed. Third, we present how these high level QoS specifications are automatically transformed to executable code that allows the system to continuously monitor QoS parameters at runtime in order to detect QoS degradation.

The remainder of this paper is structured as follows: In Section 2, we present our proposed QoS classification for distributed component-based systems. Section 3 presents the proposed approach to ensure the monitoring of QoS parameters. An illustrative example is introduced in Section 4. Section 5 presents some research studies related to our work. Finally, Section 6 concludes the paper and gives some directions for future work.

2 Quality of Service : parameters and classification

We propose a classification of the QoS parameters in order to make easier the evaluation of the QoS attributes of a software system. This classification is inspired from the one proposed by the standard ISO/IEC 9126 [9]. This standard proposes to classify the QoS attributes into two categories : (1) internal quality attributes which are properties of subsystems and components, and (2) external ones that are visible on the system level. Inspired from this classification, we propose to classify the QoS parameters based on the architectural vision of a distributed component-based system specified using AADL concepts.

From an architectural perspective, the software architecture of a distributed component-based system (*system* in AADL) is composed of a set of communicating composite components (*processes* in AADL). Similarly, each composite component is composed of a set of interconnected indivisible components (*threads* in AADL). The communication between components is ensured via a set of connections (*connection* in AADL). Our idea consists in classifying QoS parameters according to the level at which such parameter can be monitored. An indivisible component and a composite component in AADL have the same architectural structure. For this reason, we consider that QoS parameters that can be monitored at indivisible component level and at composite component level are the same. We distinguish three levels of QoS monitoring according to the structure of distributed component-based systems : **indivisible component/composite component level**, **architecture level** and **communication level**.

The adopted generic QoS attributes that can be measured and evaluated at runtime for a distributed component based system are :

- *Performance* is an indication of the responsiveness of a system. It can be measured in terms of throughput, latency and processing time.
- *Reliability* is the ability of a system to remain operational over time. It can be measured through the loss rate of transmitted messages.
- *Load Balancing* is the distribution of workloads across system components. It can be measured as the distribution of the components on the system.

3 Overview of the proposed approach

In this section, we present a general overview of our proposed approach towards specifying and monitoring QoS parameters in order to detect QoS degradations.

Our approach involves a main component namely a QoS manager. This later is defined at two levels : at the composite component level (process level) to be able to manage QoS parameters of its interacting AADL subcomponents (threads), and at the architecture level to manage the QoS parameters of the interacting AADL composite components (processes) that form the whole system architecture. At both levels, the QoS manager is in change of monitoring QoS parameters, analyzing collected data and perform reconfiguration actions when QoS degradations are detected. Therefore, the activity of the QoS manager is divided into three main modules.

- A QoS monitor module : This module is composed of a set of *monitoring aspects* responsible for monitoring and collecting QoS parameters. Gathered QoS parameters are stored into a QoS database for further retrieval.
- A QoS analyzer module : Towards inspecting the collected information and track down QoS degradation. This module sends notifications to the reconfiguration module when QoS degradation is detected in order to trigger appropriate reconfiguration actions.
- A reconfiguration module : Defines a list of reconfiguration actions that can be applied to the system when QoS degradations are detected. These reconfigurations are encapsulated into a set of *reconfiguration aspects*.

The proposed QoS manager is then able to handle a closed feedback loop (MAPE) with four phases : Monitoring, Analysis, Planning and Execution. In this paper, we focus on the first phase of the MAPE loop. For this purpose, we detail in the following how the QoS parameters are specified at architectural level using AO4AADL aspects. We present also some examples of the generated AspectJ code from such AO4AADL specifications. These AspectJ aspects will ensure the monitoring of the QoS parameters at runtime.

3.1 QoS monitoring

Monitoring the QoS of a dynamically adaptive system aims to observe its constituting components to collect data about QoS parameters. This monitoring is performed through a set of AspectJ aspects that are automatically generated from a high level QoS specification using the AO4AADL language. In fact, our proposed QoS monitor module is composed of a set of architectural monitoring aspects specified in AO4AADL. These aspects are intended to intercept the architectural elements through which QoS parameters can be captured.

As mentioned previously, the QoS parameters can be monitored at three levels. For each level, we define the considered QoS parameters and we give some examples of the structure of the AO4AADL aspects used to specify such QoS parameters. Such aspect is composed of two parts : pointcut and advice. The pointcut defines the architectural element to intercept in order to get the value of the corresponding QoS parameter. The advice defines the information value to save in the QoS database. We present also some examples of the generated AspectJ code from AO4AADL specifications to ensure the monitoring of QoS parameters at runtime.

QoS parameters monitored at indivisible/composite component level

Throughput is the number of sent messages through a set of output ports of a given indivisible or composite component within a time interval. Formula 1 is used to calculate this information.

$$Throughput_c = \sum Throughput_{outport_i} \tag{1}$$

- i={1..p}, p ≤ n and n = the number of output ports of the component.
- $Throughput_{outport_i}$ is the number of sent messages through an output port of the component within a given time interval.

The monitoring of the throughput is achieved through one AO4AADL aspect whose structure is given in Listing 1.1. The pointcut (lines 2–3) of such an aspect intercepts the set of output ports of the given component to be monitored. The advice (lines 7–13) captures the number of sent messages through this set of output ports. Once the time interval of monitoring is elapsed, this advice sends the captured number of sent messages to the QoS database.

```
1  aspect Monitoring_Throughput_<Identifier> {
2    pointcut Throughput_<Identifier>(): execution(outport(<Out_port_identifier_1>(..)))
3                      || ... ||execution (outport(<Out_port_identifier_n>(..)));
4      variables{counter : Integer_Type; t : Time_Type;}
5      initially{counter = 0; t=System.currentTimeMillis()+period;}
6
7    advice after(): Throughput_<Identifier> (){
8         if(System.currentTimeMillis()<t){counter:=counter+1;}
9         else if(System.currentTimeMillis()=t){
10            counter:=counter+1;
11            send_Value!(counter);
12            counter=0;
13            t=System.currentTimeMillis()+period;}}}
```

Listing 1.1: Monitoring the throughput of a component

Listing 1.2 presents the generated AspectJ code from Listing 1.1. The interception of the execution of the output port is transformed to the interception of the method *sendOutput()* of the *PortsRouter* class (lines 3–4). This class carries out the correct routing of messages through ports. Moreover, the subprogram *send_Value* is transformed to a simple execution of the method *send_ValueImpl()* of a generated *SubPrograms* class (line 14).

```
1  aspect Monitoring_Throughput_<Identifier> {
2    pointcut Throughput_<Identifier>():
3         execution(* PortsRouter.sendOutput(<Out_port_identifier_1>(..)))
4         || ... ||execution (* PortsRouter.sendOutput(<Out_port_identifier_n>(..)));
5
6      public static final GeneratedTypes.IntegerType COUNTER = new GeneratedTypes.IntegerType(0);
7      public static final GeneratedTypes.TimeType T =
8                               new GeneratedTypes.TimeType(System.currentTimeMillis()+period);
9
10   void after(): Throughput_<Identifier> (){
11        if(System.currentTimeMillis()<T){COUNTER = COUNTER+1;}
12        else if(System.currentTimeMillis()==T){
13            COUNTER = COUNTER+1;
14            SubPrograms.send_ValueImpl(COUNTER);
15            COUNTER=0;
16            T=System.currentTimeMillis()+period;}}}
```

Listing 1.2: Generated AspectJ code from Listing 1.1

Loss rate of messages defines, within a time interval, the percentage of unprocessed messages compared with the number of received messages by a given component. This information can be measured only for components characterized as follows: To each input port that receives a message corresponds an output port that sends a result. To compute this information, we use formula 2.

$$Loss_c = 1 - \frac{\sum Throughput_{outport_i}}{\sum Input_{inport_i}} \tag{2}$$

- i={1..p}, p ≤ n and n = the number of output ports of the component.
- $Input_{inport_i}$ is the number of received messages through an input port of the component within a given time interval.

To monitor the loss rate of messages within an indivisible/composite component, two architectural aspects are needed. The first aspect is intended to capture the throughput of the considered component. The second aspect captures the number of received messages through the set of input ports of the component.

Processing time measures the elapsed time to execute an operation within a given component. It is measured using the following formula:

$$Time_{Proc_{op}} = Time_{EndExec_{op}} - Time_{StartExec_{op}} \tag{3}$$

- $Time_{EndExec_{op}}$ is the end time of execution of the operation.
- $Time_{StartExec_{op}}$ is the start time of execution of the operation.

One architectural aspect is needed to monitor the processing time. This aspect intercepts the execution of the subprogram that corresponds to the operation to be monitored. Two types of advices are defined in this aspect : a before advice which will capture the start time of the execution and an after advice that will capture the end time of execution.

QoS parameters monitored at architecture level

Distribution measures the dispersion rate of a specified component type C_i on the set of composite components of the system. This information is related to the architectural style of the system. It is monitored at architecture level since its computation requires knowledge of the total number of components deployed throughout the system. It is measured for each composite component using formula 4:

$$Distribution_{node} = \frac{Node_{components(C_i)}}{System_{components(C_i)}} \tag{4}$$

- $Node_{components(C_i)}$ is the number of components of type C_i deployed in a given composite component.
- $System_{components(C_i)}$ is the number of components of the same type C_i deployed in the whole system.

The monitoring of the distribution parameter is performed through a set of aspects. One aspect is attached to every composite component that may contain components of type C_i. The structure of the corresponding AO4AADL aspect is given in Listing 1.3. The number of components of type C_i deployed on one composite component can change due to reconfiguration actions that can affect the system during its execution. To be able to capture the new number of components of type C_i within one composite component, the aspect should intercept the reconfiguration actions related to the addition, removal and migration of components (lines 2–4). Therefore, after each invocation of one of these reconfiguration actions, the aspect transmits the new number of components of type C_i to the QoS database module (lines 8–12).

```
aspect Monitoring_Distribution {
  pointcut Distribution(): execution(subprogram(addThread(ComponentType,..)))
     || execution(subprogram(removeThread(ComponentType,..)))
     || execution(subprogram(migrateThread(ComponentType,..)));
    variables {counter:Integer_Type;}
    initially {counter=0};

  advice after(): Distribution (){
       if (this="addThread") {counter:=counter+1;}
       else if((this="removeThread") or (this="migrateThread")) {counter:=counter-1;}
       send_Value!(counter);
  }
}
```

Listing 1.3: Monitoring the distribution on a composite component

Distribution can refer also to the dispersion rate of a specified composite component type N_i on the system. This information is monitored on a cluster of composite components. The formula used to monitor such parameter is similar to formula 4 except that here we are interested in composite components instead of indivisible components. To monitor such paramter, an aspect is attached to each composite component of the cluster. Such aspect intercepts the reconfiguration action related to the connection and disconnection of composite components of type N_i and transmits the new number of these components to the QoS database.

QoS parameters monitored at communication level

Loss rate of messages measures, within a time interval, the percentage of lost messages through a connection. This parameter is computed using formula 5.

$$Loss_{conn} = \frac{Received_{msg}}{Sent_{msg}} \tag{5}$$

- $Received\_msg$ is the number of received messages through the destination port of the connection.
- $Sent\_msg$ is the number of sent messages through the source port of the connection.

Two architectural aspects should be specified to monitor such QoS parameter. One aspect is used to monitor the throughput of the source output port of the connection. The other aspect is intended to monitor the number of received messages through the destination input port of the connection.

Latency represents the elapsed time to transfer a message through a given connection. Formula 6 is used to compute this parameter.

$$Latency_{conn} = Time_{Receipt_{msg}} - Time_{Sending_{msg}} \tag{6}$$

- $T_{Receipt_{msg}}$ is the receipt time of the message on the destination port.
- $T_{Sending_{msg}}$ is the sending time of the message through the source port.

The monitoring of this QoS parameter needs the specification of two architectural aspects as mentioned in the previous QoS parameter. The only difference here lies in the advice part. In fact, the first aspect in this case is intended to capture the sending time of the message through the source output port of the connection and the second one is responsible of capturing the receipt time of this message on the destination input port of the connection.

All gathered QoS parameters from the monitoring module are stored in a QoS database for later use by the analysis module in order to be able to detect QoS degradations. This QoS database allows keeping a trace of all the captured QoS parameters for further retrieval when needed.

4 Illustrative example

To demonstrate the benefits of our approach, we introduce the Flood Prediction System (FPS) [10] as an illustrative example. This system presents a set of nodes that communicate and cooperate to carry out flood predictions. Three types of components are considered : sensor nodes, computation nodes and office nodes.

Sensor nodes sense and collect the data relevant for calculations such as pressure, rainfall, and temperature. Sensed data are periodically transmitted to the corresponding computational node. Computation nodes connect the sensor nodes, examine the data correctness and maintains a record of all draw values. Some calibrations are performed on the draw data. Later, a prediction operation is invoked to execute some static measurements in order to provide prediction on river flow. This prediction is transmitted to the office node. The Office Node verifies the results with the available online information, predicts for the entire region, issues alerts and initiates evacuation procedures.

In the following, we will detail the usefulness of monitoring the considered QoS parameters using aspects.

Processing time To ensure more system reliability, the processing time of data within computation nodes should not be out of a certain time interval. Results provided so quickly or so late are not reliable for prediction measurements performed at the office node level. Therefore, the processing time of subprograms responsible for data checking, calibration and prediction should be monitored. Listing 1.4 presents the AO4AADL aspect to monitor such QoS parameter.

```
1 aspect Monitoring_ProcTime_CompNode {
2   pointcut ProcTime_CompNode(): execution(subprogram(Data_Checking(..)))
3                           || execution(subprogram(Calibration(..)))
4                           ||execution(subprogram(Prediction(..)));
5
6   advice before(): ProcTime_CompNode(){send_Time!(System.currentTimeMillis());}
7   advice after(): ProcTime_CompNode(){send_Time!(System.currentTimeMillis());}}
```

Listing 1.4: Monitoring the processing time

Distribution Due to several reconfiguration actions or to nodes failure, the distribution of sensor nodes may be unfair. For example, let's suppose that we

dispose of three computation nodes. The first one is connected to six sensor nodes (2 sensor nodes of each type), the second one is connected to three sensor nodes (one sensor node of each type), however, no sensor node is connected to the third one. To ensure a fair distribution of sensor nodes along the river, three sensor nodes (one of each type) which are connected to the first computation node should be disconnected from this later and connected to the third computation node. Listing 1.5 presents the structure of the AO4AADL aspect responsible for monitoring the distribution. This aspect is attached to every computation node of the system and intercepts the addition and removal of connections between a sensor node and a computation node.

```
aspect Monitoring_Distribution_Cluster {
  pointcut Distribution_Cluster():
  execution(subprogram(ConnectNodes(SensorNode, ComputationNode)))
   || execution(subprogram(ConnectNodes(SensorNode, ComputationNode)))
   || execution(subprogram(DisconnectNodes(SensorNode, ComputationNode)))
   || execution(subprogram(DisconnectNodes(SensorNode, ComputationNode)));

   variables {counter:Integer_Type;}
   initially {counter=0};

  advice after(): Distribution_Cluster (){
       if (this="ConnectNodes") {counter:=counter+1;}
       else if(this="DisconnectNodes"){counter:=counter-1;}
       send_Value!(counter);}}
```

Listing 1.5: Monitoring the distribution on a cluster of nodes

5 Related work

Authors in [11] propose an approach for monitoring the QoS of web services. This approach relies on monitoring tools such as Jpcap for latency measurement. It is based on aspect-oriented programming and requires implementation details. Therefore, this approach stills implementation dependent, as the language of coding aspect is dependent on the selected programming language. Our approach is different since it allows the monitoring of QoS parameters through specifying architectural aspects at a high level of abstraction (architectural level) independently from the programming language.

Authors in [12] present an approach to Cloud service monitoring based on Aspect-Oriented Programming. This approach monitors QoS parameters using AspectJ aspects. Similarly to [11], aspects are hard coded and the approach is dependent on the selected programming language.

The work presented in [13] proposes an aspect-oriented QoS specification method based on the combination of UML and RTL. The main objective of this approach is to specify the QoS parameters separately from the system concerns and facilitate their monitoring. Similarly to our approach, this work allows to specify the QoS parameters at a high level of abstraction. However, no details are mentioned about the code generation of these aspects.

6 Conclusion and Future work

In this paper, we have proposed an approach for monitoring QoS attributes in dynamically adaptive systems. For this purpose, we proposed a QoS parameters classification taking into account the level at which a QoS parameter can be monitored in the case of a distributed component-based system specified using AADL concepts. We presented later how our approach ensures the monitoring of the QoS parameters using the AOSD paradigm. First, these QoS parameters are specified into AO4AADL. Then, they are automatically translated into AspectJ code to ensure the monitoring of these parameters at runtime.

As future work, we aim to focus on the analysis and reconfiguration phases. We plan to apply our approach to specific domains such as event-based systems.

References

1. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems **4** (2009) 1–42
2. Dowling, J., Cahill, V.: The k-component architecture meta-model for self-adaptive software. In: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns. (2001) 81–88
3. Loukil, S., Kallel, S., Jmaiel, M.: Managing architectural reconfiguration at runtime. International Journal of Web Portals **5** (2013) 55–71
4. Loukil, S., Kallel, S., Jmaiel, M.: Verifying runtime architectural reconfiguration of dynamically adaptive systems. (2013) 169–176
5. SAE: Architecture Analysis & Design Language. (2004)
6. Loukil, S., Kallel, S., Zalila, B., Jmaiel, M.: Toward an Aspect Oriented ADL for Embedded Systems. In: Proceedings of the 4th European Conference on Software Architecture. (2010)
7. Loukil, S., Kallel, S., Zalila, B., Jmaiel, M.: Ao4aadl: Aspect oriented extension for aadl. Central European Journal of Computer Science **3** (2013) 43–68
8. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: Proceedings of the 15th European Conference on Object-Oriented Programming. (2001) 327–353
9. Organization, I.: ISO/IEC 9126: Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for Their Use. (1991)
10. Hughes, D., Greenwood, P., Coulson, G., Blair, G.: Gridstix: Supporting flood prediction using embedded hardware and next generation grid middleware. In: Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks. (2006) 621–626
11. Rosenberg, F., Platzer, C., Dustdar, S.: Bootstrapping performance and dependability attributes of web services. In: Proceedings of the IEEE International Conference on Web Services (ICWS06. (2006) 205–212
12. Mdhaffar, A., Halima, R.B., Juhnke, E., Jmaiel, M., Freisleben, B.: Aop4csm: An aspect-oriented programming approach for cloud service monitoring. In: Proceedings of the 11th International Conference on Computer and Information Technology. (2011) 363–370
13. Zhang, L.: Aspect-oriented qos modeling for cyber-physical systems. Journal of Software **7** (2012) 1083–1093