Interceptor based Constraint Violation Detection

Qianxiang Wang  
School of Electronics Engineering and Computer Science, Peking University, Beijing, China  
wqx@pku.edu.cn

Aditya Mathur  
Department of Computer Science, Purdue University, West Lafayette, IN, USA  
apm@purdue.edu

Abstract

Monitoring critical events such as constraints violations is one of the key issues of Autonomic Systems. This paper presents an interceptor based approach of constraint violation detection. In our approach, the monitor code is independent of functional code, and the monitor code can be generated automatically from XML-based constraint specifications. The experiment shows that our approach is feasible and is especially suitable for interface level constraints.

1. Introduction

Kephart et al. proposed four aspects of self-management [1]. While self-configuration and self-optimization are mainly about performance, self-healing and self-protection are mainly about survival, which is a critical property of Autonomic Systems. Several researchers believe that the first step towards self-healing systems is to monitor critical events [2] [3] [4], which are usually caused by internal bugs, external attacks, or human miss-operations, etc. If the software can detect these critical events automatically as early as possible, then it can take some remedy action and avoid potential disaster. One of such critical-event detection approach is to specify sufficient constraints in advance and then verify the system behavior at runtime against the specified constraints [5].

1.1. Problem

Constraints can be applied to different entities in systems: from an entity that is small in size such as a variable, parameter, to an entity that is large size such as a procedure, class, and even the entire system. In this work, we consider four main categories of interface level constraints.

(1) Constraints of value region, such as value region of method parameters, value region of class attributes, and value region of return value.

(2) Constraints of time region, such as the response time of requests, interval between two responses.

(3) Constraints of spatial value relationship, such as the relationships between attributes, relationships between parameters, and even relationship between attributes and parameters.

(4) Constraints of temporal value relationship, such as the temporal value relationship between attributes or methods. For example, partial order of method invocation is one critical factor for the client to obtain correct responses from the server.

Monitoring code to verify system properties at runtime is implemented according to constraints. For Autonomic Systems, the monitoring code should have the following features. (1) The monitoring code should be easy to maintain, especially at runtime. Most current runtime monitoring approaches embed monitoring code into functional code [5] [6]. These approaches are intuitive and easy to implement. But embedded monitoring code makes functional code difficult to understand, and makes constraints difficult to update: if we want to adjust the constraints, we must reedit and recompile related source code. (2) The monitoring code should be easy to generate from a model. It is preferred that monitoring code be generated from models automatically. Unfortunately, till now, constraints of systems have not received sufficient attention, compared to functions of systems. Most developers seem reluctant to describe constraints of software [7] [8]. Also there is little work explicitly aimed at the formal expression of constraints, which is necessary for generating monitoring code automatically.
1.2. Overview of the Proposed Approach

The proposed approach extracts scattered constraint code from different procedures. It is inspired by the well known idea of “separation of concerns” [9], which has led to the birth of many novel technologies such as aspect-oriented programming [10]. “Design by contract” of Eiffel [23] also recommends separation of constraints. Our approach extracts monitoring code from different procedures and uses an independent interceptor to detect potential constraint violations. The interceptor can intercept both request messages from a client to a server and response messages from a server to a client, check interesting values, and generating warnings when a constraint violation is detected.

Our approach has four key benefits. (1) the monitoring code is independent of the functional code, which makes it easy to adjust constraints at runtime. (2) The monitoring code can be generated automatically from XML-based formal constraint specifications, which makes it easy to specify constraints. (3) The functional code needs not be preprocessed, such as rewriting, which makes development easier. (4) Both the monitoring code and the functional code are less clumsy and hence easier to understand and maintain.

1.3. Paper Overview

The remainder of this paper is organized as follows. Section 2 presents an example of interface level constraints. Section 3 illustrates our approach and runtime the structure of the software. Section 4 presents experimental results. Section 5 relates our work to existing research. We conclude in Section 6 with some discussion.

2. Example of Interface Level Constraints

In this section, we use a class ShoppingCart to illustrate the interface level constraints. ShoppingCart, is a Java class, which is responsible for clients’ shopping processes. Figure 1 lists the interface of ShoppingCart. For simplicity, we list only those methods that are related with constraints of this class. Those methods are “addItem()”, “setClientInfo()” and “checkout()”.

```
interface public class ShoppingCart {
    public Integer addItem(Integer n, String p);
    public void setClientInfo(String info);
    public Double checkout();
}
```

Figure 1. The interface of ShoppingCart

The constraints of class ShoppingCart include:

(1) Parameter Value

One of the input parameters in the method “addItem()” is “Integer n”. This parameter indicates the number of added item, which must be larger than 0.

(2) Return Value

The method “addItem()” returns the value of total count, which must be larger than 0.

The method “checkout()” returns the value of the total price, which must be larger than 0.

(3) Method invocation order

When the client checks out, he/she must have added some product in the shopping cart, and must have filled in the necessary information such as credit card number and mail address. So, only both of the methods “addItem()” and “setClientInfo()” have been executed at least once, can the method “checkout()” be executed. Meanwhile, the method “checkout” cannot be executed twice in sequence, which should be considered as a miss-operation of client.

If the monitoring code is embedded in functional procedures, the code will be scattered in different procedures. Figure 2 lists embedded monitoring code of the ShoppingCart class. The three variables defined at the beginning of class are used to record the method invocation history: If “a_invoked” is true, that indicates “addItem()” has been invoked. If “s_invoked” is true, that indicates “setClientInfo()” has been invoked. If “c_invoked” is true, that indicates “checkout()” has just been invoked. To make the code easy to read, we omit all the functional code and constraints violation handling code.
public class ShoppingCart {
    // states for method order verification
    Boolean a_invoked = false;
    Boolean s_invoked = false;
    Boolean c_just_invoked = false;
    ...;

    public Integer addItem(Integer n, String p){
        Integer m = null;
        // verify input value
        if (n <= 0) {...};
        // function code
        ...;
        if (m <= 0) {...}; // verify output value
        a_invoked = true;
        c_just_invoked = false;
        return m;
    }

    public void setClientInfo(String Addr){
        // function code
        ...
        s_invoked = true;
        c_just_invoked = false;
    }

    public Double checkout() {
        Double total = null;
        // verify method invocation order
        if ((a_invoked == false) || (s_invoked == false) || (c_just_invoked == true)) {...}
        // function code
        ...;
        // verify output value
        if (total <= 0){...};
        p_just_invoked = true;
        return total;
    }
}

Figure 2. Scattered monitoring code

3. Approach

In this section we describe the interceptor based constraints violation detection approach. In our approach, the ShoppingCart class is purely functional, and all constraints are verified by an independent interceptor ConsInterceptor. Section 3.1 introduces the principle of our approach by illustrating the runtime structure of software system. Section 3.2 introduces the implementation of interceptor ConsInterceptor. Section 3.3 introduces how to get interceptor automatically from XML-based constraint specification.

3.1. Runtime structure

Interceptor is a widely adopted design pattern, similar to the Chain of Responsibility pattern from the Gang of Four (GoF) [11]. Software systems (especially loosely coupled software) use interceptor pattern to enhance their flexibility and extensibility. For example, most ORB products use it to enhance security services; some web servers (such as Apache) use it to allow modules to register special handlers with the core server; some application servers (such as Jboss [14],) uses it to provide customized common services such as transactions and security.

An interceptor is located between the client and the server. All request messages from the client to the server and all response messages from the server to the client are intercepted by the interceptor. The interceptor can unmarshal the messages and then analyze the content of messages. If necessary, it can even modify the content of messages. After having analyzed (and probably modified) the message, the interceptor forwards the message to the original target. In addition, if we have multiple concerns with the message, we can set multiple interceptors between the client and the server. These interceptors can be grouped as one interceptor chain, and managed by an interceptor manager (Figure 3). The potential conflicts between different interceptors should be avoided by a policy checking mechanism before the interceptors are deployed.

Figure 3. Runtime structure of interceptor based software

In our approach, we use an interceptor to verify interface level constraints. The interceptor is used to extract constraint-related information from
request/response messages, and then check them against the predefined constraints. When some constraint violation is found, interceptor will report the violation to other entities, so that software can do some remedy action automatically [12].

Compared with the embedded approach, the interceptor based approach makes constraint configuration easier. The interceptor covers all of the monitoring code, so the monitoring code is separated totally from the functional code. When we want to change constraints on the server, we need not change the server class. All we need to do is just update the new interceptor to the interceptor chain. Neither the server class nor current interceptor needs to be touched.

Meanwhile, the monitoring scope of the interceptor based approach is less than that of the embedded approach. In the embedded approach, the monitoring code can be embedded in any place of functional code, so it can monitor the class freely. But in the interceptor based approach, the monitoring code can monitor by only checking messages to and from the server. For those constraints that have no direct relation with the interface, such as an attribute of the class, which can only be accessed by internal methods, the interceptor cannot verify the constraints related to that attribute. But interface level constraints can usually cover most constraints of class.

3.2. Interceptor Implementation

We present the implementation details of interceptors using the same example described in Section 2. Constraints about ShoppingCart are implemented by an interceptor “ConsInterceptor”. These constraints can be implemented by multiple interceptors too, each of which is responsible for a part of the constraints. Figure 4 lists the main code of “ConsInterceptor”. This interceptor verifies the invocation order first, followed by verifying the input parameter constraints. Then the interceptor invokes the real object or next interceptor (Object r = invocation.invokeNext();). When the interceptor gets a response, it verifies the output parameters, and then returns the result. Similar to Figure 2, Figure 4 omits all constraint violation handling code.

```java
public class ConsInterceptor implements Interceptor{
    private int Class_State = 0;
    public Object invoke(Invocation invocation)   {
        MethodInvocation mi = (MethodInvocation)invocation;
        String mn = mi.getMethod().toString();
        // verify invocation order
        switch(Class_State) {
            case 0: Class_State = (mn.equals("checkout")? 1: 2); break;
            case 1: Class_State = (mn.equals("checkout")? 1: 3); break;
            case 2: Class_State = (mn.equals("checkout")? 3: 2); break;
            case 3: Class_State = (mn.equals("checkout")? 4: 3); break;
            case 4: Class_State = (mn.equals("checkout")? 4: 0); break;
        }
        if (Class_State == -1) {…} } // verify input parameters
        if (mn.equals("addItem"))   {
            Object[] args = mi.getArguments();
            Double d = double args[0];
            if ( (args[0] <= 0)) {…};
        } // Invoke the real object or next interceptor
        Object r = invocation.invokeNext();
        // verify return value:
        Double dd = Double r;
        if (mn.equals("addItem") && (dd.doubleValue() < 0)) {
            …
        } if (mn.equals("checkout") && (dd.doubleValue() < 0)) {
            …
        } return r;
    }
```

Figure 4. Code of ConsInterceptor

To help understand the code for verifying invocation order, we show the state transitions of a Finite State Machine (FSM) in Figure 5. In fact, the code of verifying invocation order is generated from this FSM. To make the diagram simple, we use “a” to denote the method “addItem()”, “s” to denote the method “setClientInfo()”, and “c” to denote the method “checkout()”. It is easy to find that in this diagram, different states represent different invocation states: “0” indicates that no method was invoked; “1” indicates that only “addItem()” has been invoked; “2” indicates that only “setClientInfo()” has been invoked; “3” indicates that both “addItem()” and “setClientInfo()” have been invoked and “checkout()” is not the method that has just been invoked; “4” indicates that “checkout()” has just been invoked; “-1” indicates that invocation order violation occurred.
3.3. How to get interceptor from constraints?

Intuitively, we can write the above interceptor manually according to the constraints expressed in natural language (see the example in Section 2). But if the constraints can be expressed in some formal language, then the monitoring code can be generated automatically from the formal expression [5]. In this section, we introduce our initial work on how to express constraints in XML-based language. Then the interceptor can be automatically generated from this XML-based constraint specification.

In J2EE-based application, each Java components (EJB) jar package includes one special file: deployment descriptor, together with other java classes and interface [13]. The deployment descriptor describes component attributes in XML. When the component is deployed, the attributes described in the deployment descriptor are parsed, and the different attributes are weaved into runtime software.

Constraints can be considered as one kind of important attributes of a component. The current J2EE specification supports constraints that are related with transaction, security, etc. Our approach enhances the current deployment descriptor with description of constraints. Figure 6 shows one XML-based description of constraints in Section 3. When class ShoppingCart is deployed, the related XML file is parsed, and the class ConsInterceptor shown in Figure 4 is generated.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<constraint component="ShoppingCart">
  <invocationOrder>
    <happened>
      <method name="c">a</method>
    </happened>
    <happened>
      <method name="c">s</method>
    </happened>
    <last_is>
      <method name="c">a</method>
    </last_is>
    <last_is>
      <method name="c">s</method>
    </last_is>
  </invocationOrder>
  <inputValue>
    <method name="a" param="1" expression=">0">
  </inputValue>
  <outputValue>
    <method name="a" expression=">0"/>
  </outputValue>
  <outputValue>
    <method name="c" expression=">0"/>
  </outputValue>
</constraint>
```

Figure 6. XML-based constraints description

Our current algorithm supports limit constraints, such as “happened”, “last_is”. Section 6 will show that it is still one central issue of our future work.

4. Experiment

We have implemented our approach on the JBOSS-AOP [14] platform, which provides an excellent interceptor infrastructure. In a Java platform, a class loader is responsible for dynamically loading and assembling classes at runtime. JBOSS-AOP developed one special class loader: SystemClassLoader, which is subclass of the native JVM class loader. When SystemClassLoader was set as the default class loader, it loads classes, parses related XML file, generates corresponding interceptors and assembles runtime objects such as server object, interceptors into the runtime structure shown in Figure 3.

The experimental result shows that our approach is feasible. Meanwhile, just like other technologies that can enhance flexibility, the runtime structure in Figure 3 suffers from time cost. To get the detailed data about time cost, we run the system in different contexts to observe the different response time.

The basic platform we used is one PC with Pentium 4 CPU (1.48GHZ), 768MB of RAM. The version of JBOSS-AOP we used is 1.0RC2, which is a standalone
version, independent of JBOSS-AS. To compare the pure time delays under different situations, all functional code is set to empty. We did five groups of experiments. Figure 7 lists the experimental result.

<table>
<thead>
<tr>
<th>Group</th>
<th>With interceptor (10^5 second)</th>
<th>No interceptor (10^5 second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (No error)</td>
<td>20.8</td>
<td>0.136</td>
</tr>
<tr>
<td>2 (Input value error)</td>
<td>18.9</td>
<td>0.158</td>
</tr>
<tr>
<td>3 (Output value error)</td>
<td>20.9</td>
<td>0.156</td>
</tr>
<tr>
<td>4 (no invocation order error)</td>
<td>47.3</td>
<td>0.242</td>
</tr>
<tr>
<td>5 (has invocation order error)</td>
<td>45.4</td>
<td>0.435</td>
</tr>
</tbody>
</table>

Figure 7. Time cost of interceptor

The first group is about invocations without error (no constraint violation). The second group is about invocations with input value error. The third group is about invocations with output value error. To make these three groups comparable, all these three groups invoke the same method “addItem”. To get rid of the environmental noise, we construct one loop to call the method 10,000,000 times once, and call the loop multiple times to get the average response time. The data shown in Figure 7 for groups 1-3 are the mean request response time for one invocation (addItem).

From these data we can find that the response time of invocation with input value error is less than that of invocation without input value error. The reason is that when error is found, the interceptor needs not forward request message to the server, but just replies one “input value error” exception directly to the client. Response time of invocation with output value error is a little more than that of invocation without error, because the interceptor must construct one “output value error” exception to return to the client.

The forth group is about invocation sequence without order violation. We select “a, a, s, c” as the invocation sequence, which can invoke all related methods and enter all right states shown in Figure 5. The fifth group is about series invocation sequence with order error. We select “a, a, a, c” as the invocation sequence, which violates the constraint of “Only both of the methods ’addItem’ and ’setClientInfo’ have been executed at least once, can the method ’checkout’ be executed”, as mentioned in Section 2. Because the interceptor replies one “invocation order error” exception directly to client when it receives invocation to method “c”, the response time of the fifth group is a little less than that of the forth group.

We used the same test suite for the software system without any interceptor. Of course, we changed the server class to the one shown in Figure 2. The corresponding data are listed on the right column of Figure 7.

From the experimental data that we have obtained, we can draw the following two conclusions. (1) Because the time cost of interceptor based approach is at microsecond level, the time cost is affordable. (2) Because time cost of interceptor based approach is hundreds times of that of embedded approach and invocation frequency of fine-grained entities are much higher than that of coarse-grained entities, the interceptor based approach should not be applied to entities with fine granularity. In other words, our approach is more suitable for medium size entities, such as component (EJB, COM, etc.), Web Service.

5. Related work

In this section we introduce some work that has inspired our work.

5.1. Self-management system

Having recognized that the problem detection is the first step towards a self-healing system, IBM [2] proposed a solution based on technologies such as log adapters and log analyzers. In their solution, Common Base Event is used as a common format for log information to create consistency across similar fields and improves the ability to correlate across multiple logs. Yi-Min Wang et al. focus on configuration data stored in the Windows registry [3] [15], and try to detect faulty configurations that can cause a variety of failures. Both of these approaches analyze vast of data to detect the problems. These methods are suitable for problem detections in a large scale.

5.2. Runtime Verification

Runtime verification is a complement to the traditional methods for proving program correctness. The goal of Runtime Verification is to investigate the use of lightweight formal methods applied during the execution of programs. Feng Chen et al. [5] proposed Monitoring Oriented Programming (MOP) approach, which inserts constraints at critical places, and automatically synthesizes and embeds monitoring code in the functional procedures. Kim et al. [16] designed Java-MaC, which inserts probe “filter” into the target program, keeps track of changes of monitored objects, and sends pertinent state information to the event recognizer. Compared with interceptor based approach,
the embedded approach is more efficient. But when facing runtime reconfiguration, which is important to autonomic systems, the interceptor based approach is more flexible.

5.3. Specification of constraints

Traditional interface description languages such as IDL of OMG, MIDL of Microsoft, WSDL of Web Services, focus mostly on functional properties of software. In recent years, many organizations have begun to explore the issue of constraint specifications. For example, OMG has developed Object Constraint Language (OCL) as part of Unified Modeling Language (UML) [17]. However, little is reported on how to generate monitoring code automatically from OCL. The Java Modeling Language (JML) is a behavioral interface specification language to specify the behavior of Java modules [18]. It combines the design by contract approach of Eiffel [23] and the model-based specification approach of the Larch [24] family of interface specification languages, with some elements of the refinement calculus. In JML, specifications are written using special annotation comments, which are embedded in Java class source code. Meanwhile, many software companies such as SUN showed strong interests in the constraints and capabilities for Web Services [19]. But till now, few detailed research is reported in this area.

6. Conclusion and discussion

This paper introduced the approach of interceptor based runtime constraint violation detection, and related XML-based constraint specifications. An experimental reported here suggests that our approach is feasible and suitable for entities such as components and web services. There remain some open issues to be explored:

1. Where do the constraints come from? Zave categorized software requirements into “functions of” and “constraints on” software systems [20], in other words, functional requirements and constraint requirements. Requirement specifications can contribute some constraints. But more detailed interface level constraints should perhaps come from the designers, who are responsible for building the trustworthy software. Then, during the test phase, testers can also find some implicit constraints that should be satisfied. In recent years, inferring constraints from software has also been explored [7] [8].

2. How to generate interceptors automatically from complex constraints? A complex temporal constraint is easy to express with natural languages, but is difficult to express using formal expressions. Although there are some formal expressions such as temporal logic [21], and the corresponding transformation algorithm [22], it remains difficult to express complex temporal relations formally, especially with XML.

3. What’s next? Finding constraint violations is the first step for self-managing software. The flexible software structure provides the ability to adjust constraints at runtime. Then, how does one build the relationship between the condition and action in the assumed rule form of “IF condition THEN action”? Our Rule Model [12] presents an initial effort in this direction.

Acknowledgments

This work is supported in part by the National High-Tech Research and Development Plan of China No. 2003AA115430, the National Natural Science Foundation of China No. 60103001, and the National Grand Fundamental Research Program of China No. 2002CB3120003. The authors would like to thank the Department of Computer Science, Purdue University, for hosting one of the authors.

References


