Constraint Violation Detection: A Fundamental Part of Software Cybernetics

Qianxiang Wang

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

Abstract

Monitoring of sensitive events is a key step for controlling the behavior of software. Specifying a sufficient set of constraints prior to software deployment is necessary for detecting the presence of such events during execution. Here we focus on issues related to service level constraints including types of constraints, the runtime structure of constraints and the related entities. An experiment to assess the feasibility of the proposed approach is also reported.

1. Introduction

Software cybernetics views software as a new type of controlled object [1]. Similar with Autonomic Computing [2], monitoring of sensitive events is a key step for controlling the behavior of software. Specifying a sufficient set of constraints prior to software deployment is necessary for detecting the presence of such events during execution. Several researchers have addressed the problem of monitoring sensitive events [3] [4] [5]. The key belief of this paper is that if software can detect sensitive events automatically, and as early as possible, then it can take appropriate actions, thus avoid many potential disasters, and improve performance.

Zave divided software requirements into “functions of” and “constraints on” software systems [6]. This division indicates the importance of constraints. However, for several years, constraint related issues, especially in the post-requirements phases, have not gained sufficient attention. While functional requirements are modeled and implemented as different design and executable modules, constraints is often hidden inside the implementation of functional modules.

Well-controlled software needs support of flexible structure. In recent years, not only in area of software, but also in some other areas such as service computing, grid computing, researchers have attempted to find flexible structure for a new generation of software. Component, middleware, and aspect-oriented programming (AOP) [7] etc. are potential candidates for this requirement. It is well known that software is divided into functional modules, such as procedures, objects, components, etc. Within such a structure, we ask “How can constraints be represented and used at runtime?” and “Can we obtain a flexible software structure by separating constraints from modules that represent functional requirements?” This paper focuses on obtaining answers to these questions.

The basic idea of this work is to separate constraints from functions. We want to do so not only during the requirements phase, but also during design and execution. At runtime, we extract the monitoring code (which is responsible for constraints) from different procedures and use one or more independent interceptors to detect potential constraint violations. An interceptor can intercept the request messages from a client to a server as well as the response messages from a server to a client, and evaluate whether or not any constraint is violated. These interceptors form a flexible interface to the functional modules of the software analogous to the relationship of the knuckles to the bones of a body.

The remainder of the paper is organized as follows. Section 2 introduces different types of service level constraints. Section 3 introduces how constraints are used during the execution of a software application. Section 4 reports an experiment conducted to investigate the feasibility of the proposed approach. Section 5 points to related work. Section 6 concludes this paper with our ideas for further work in this area.

2. What is Service Level Constraints?

There are several types of constraints that are strongly related with sensitive events that need to be monitored. Examples of such constraints are: device, process, memory, communication, etc. [8]. This paper focuses on service level constraints. We believe service level constraints belong to one type of most important constraints of software. And requests from clients trigger most of the sensitive events in application, directly or indirectly. A corrective action is needed when a violation of a constraint of the type mentioned earlier is detected. Different types of events will likely correspond to different corrective actions. For example, some event may be more critical than the others and hence might require urgent action while for the others the action could be delayed.
We classify service level constraints into different types along two orthogonal dimensions. Figure 1 lists the classification and some examples.

<table>
<thead>
<tr>
<th>Parameter related</th>
<th>Time related</th>
<th>Pattern related</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Value Region</td>
<td>Invocation order</td>
<td>Error input ratio</td>
<td>Client number</td>
</tr>
<tr>
<td>Response Value Region</td>
<td>Response time</td>
<td>Error output ratio</td>
<td>Response precision</td>
</tr>
</tbody>
</table>

Figure 1. Different kinds of Service Level Constraints

It is easy to see that violations of any of the constraints of the types listed above are detectable. This paper does not address semantic issues, or other service level constraints that have little relationship with software, for example, the cost of invocation.

During development, we use an independent file to describe constraints and related action rules of the application while the remaining files are used to describe the functions of the software application. Descriptions to constraints are usually written after the code for the functions has been written.

One basic issue arises when constraints are separated from functions: What is the relationship between constraints and functions? Can we separate constraints from functions, just as one separates software from hardware? Is it one high level entity as the control function in Figure 2 shows?

3. Structure of runtime entities

Many current systems are too complex to allow precise prediction of their behaviors. Our solution to this problem is splitting the traditional implementation of a software application into a set of functional entities and a set of constraint entities. This split makes the software easy to understand and adjust. The main runtime structure of a software component is illustrated in Figure 3. From Figure 3 we find that each component has two kinds of interfaces: 1) interfaces (left) by which clients propose their request to that component; and 2) interfaces (upper) for management by which the administrator or the application itself can obtain internal application behavior and send adjustment instructions to the application.

Our work is based on J2EE compliant application server [9]. The application that runs on a J2EE compliant application server is composed of one or more components. In a traditional implementation, a component is an independent module that performs some specific function(s). In the proposed approach, a component performs not only some function(s), but may also check for constraint violation. Functions are managed by a methods manager, while constraints are managed by an interceptor manager.

Figure 2. Vision of constraint and rule

Figure 3. Runtime entities from view of component

Interceptor is a widely adopted design pattern similar to the Chain of Responsibility pattern from the Gang of Four (GoF) [10]. Software systems, especially loosely coupled software, use the 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, PKUAS) use it to provide customized common services such as transactions, security, etc.

We use interceptors to verify service level constraints. Interceptors can extract constraint-related information and check against the predefined constraints. An interceptor manager ensures the independence of constraints during execution. As with the methods manager, an interceptor resides 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 un-marshal the messages and then analyze their content. If necessary, it can modify the content of messages. After having analyzed (and potentially modified) the message, the interceptor forwards it to its intended target. We use multiple interceptors to implement different constraints.
These interceptors can be grouped as one interceptor chain, and managed by the interceptor manager.

A rule engine receives sensitive events and returns the corresponding action instructions. In this work, a rule assumes the form “IF condition THEN action”. A rule engine maintains a table of rules. Conditions are expressions of events, or expressions of events, state-related sub-conditions, and Boolean connectors such as AND, OR, and NOT. Actions are related directly to operations. However, most operations for controlling are used to change parameters of the entire software system; other operations can be wrapped within parameter-based operations. Rule engine is independent of the functional component so that it can resolve rule conflicts across the functional components. The potential conflict between different interceptors should be avoided by some policy checking mechanism before the interceptors are deployed.

The captured event is possibly sent to the rule engine. The rule engine is responsible for searching whether one or more rules are triggered by this event. If some rule is triggered, then the rule engine sends one request to some runtime entity to adjust the software. For example, the adjustment could be to increase instance number, disconnect client, redeploy (reboot) some component, etc. As mentioned before, some constraints are also adjustable, for example, the maximum number of the concurrent clients.

4. Experiment

As an example, let’s consider we have a Java class ShoppingCart which is responsible for a web-based shopping process. Figure 4 lists the interface of ShoppingCart. For the sake of clarity, we list only the methods related the constraints of this class. These 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 4. The interface of ShoppingCart

The constraints of class ShoppingCart include:

- C1: In the method “addItem()”, the first parameter represents the number of added items, which must be larger than 0.
- C2: The method “addItem()” returns the value of total count, which must be larger than 0.
- C3: The method “checkout()” returns the value of the total price which must be larger than 0.
- C4: When the client checkout, he/she must have added some product in the shopping cart and must have filled in the necessary information such as credit card number, mailing address, etc. Hence, only when the methods “addItem()” and “setClientInfo()” have been executed at least once, can the method “checkout()” be executed. Furthermore, “checkout” can not be executed twice in a sequence, which is a miss-operation by the client.

We implemented our approach on the JBOSS-AOP [20] platform that provides the basic interceptor infrastructure. We used version 1.0RC2 of JBOSS-AOP which is a standalone version and is independent of JBOSS-AS.

Although our system directly uses most of the JBOSS-AOP’s infrastructure, it extends JBOSS-AS as follows. (1) We import one initial implementation of the rule engine which is responsible for constraint violation handling; (2) Additional exceptions are defined. Because our current focus is on constraints, the implementation of the rule engine is relatively simple. Our primary interest is in investigating the organization of the interceptor chain, so as to give some guidance to organizing interceptors for different constraints.

The basic platform we used is a PC with Pentium 4 CPU (1.48GHz), 768MB of RAM. We use the above shoppingCart example to observe the time needed to use the interceptors.

To compare time delays under different situations, all functional code is set to empty. We observe relationships between time and the number of interceptors. We also observe the time taken by an interceptor depends on constraint violation. The experimental data is listed in Figure 5.

<table>
<thead>
<tr>
<th>Interceptor Number</th>
<th>No violation</th>
<th>C2 Violated</th>
<th>C4 Violated</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>94.8</td>
<td>34.4</td>
<td>73.9</td>
</tr>
<tr>
<td>3</td>
<td>71.2</td>
<td>25.5</td>
<td>57.3</td>
</tr>
<tr>
<td>2</td>
<td>49.4</td>
<td>18.2</td>
<td>42.5</td>
</tr>
<tr>
<td>1</td>
<td>25.3</td>
<td>8.9</td>
<td>24.0</td>
</tr>
</tbody>
</table>

Figure 5. Time cost of interceptors (10^-6 second)

Four interceptors were generated for each constrains type. The sequence of interceptors is: {C4}, {C1}, {C2}, {C3}. Row 2 corresponds to two interceptors for output parameter constraints merged. The sequence of interceptors is: {C4}, {C1}, {C2, C3}. In row 3, all interceptors for parameter constraints were merged together. The sequence of interceptors is: {C4}, {C1, C2, C3}. In row 4, we use one interceptor (see Figure 3) to check all four constraints.

Data in column 1, we call three methods addItem(), setClientInfo() and checkout() in sequence, and no violation occurred. In column 2, the constraint for output parameter of first method addItem() is violated and hence
only additem() is invoked. In column 3, we call a wrong method in the wrong invocation order: addItem(), addItem() and checkout(). When checkout() is called, only the first constraint C4 is checked, because it is the first constraint in the constraint sequence and is violated.

From the experimental data shown we observe that, merging related interceptors can reduce time overhead. The time decreases almost linearly both in the case of no violation and in the case of some constraint violation. Nevertheless, it is to be noted that merging too many constraints reduces flexibility and makes the automatic generation of interceptors difficult.

5. Related Work

Dynamic AOP, such as JAC [11], AspectWerkz [12] and Prose [13] weave aspects at runtime. This approach provides a powerful solution to the problem application modification by changing the aspects currently woven and weaving new aspects while the application is running. Our current runtime constraint entities are implemented by one AOP platform JBOSS-AOP. Although the concept “aspect” comes from programming and that concept “constraint” comes from requirement, we find that the two are strongly related. Most constraints can be implemented using aspects. However, an accurate relationship between aspect and constraint remains unclear.

Feng Chen et al. [5] proposed Monitoring Oriented Programming (MOP) approach, which insert constraints at critical places, and automatically synthesize and embed monitoring code in the functional procedures. Kim et al. [14] 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, embedded approach is more efficient. But when facing runtime reconfiguration which is important to autonomic systems, interceptor based approach is more flexible.

6. Conclusion

This paper describes our initial effort towards service level constraints violation detection. We attempted to explore service level constraints and keep them relatively independent at runtime and design time, so as to detect sensitive events automatically, and as early as possible.

Acknowledgments

This paper is supported in part by the National High-Tech Research and Development Plan of China No. 2003AA115430 and the National Grand Fundamental Research Program of China No. 2002CB3120003. Thanks Professor Aditya P. Mathur, for hosting the author at Purdue University and for providing so much help in the preparation of this paper.

References

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.