A Visual Constraint Specifying Approach for Adaptive Software

Qianxiang Wang, Min Li, Na Meng

School of Electronics Engineering and Computer Science, Peking University, Beijing, China

wqx@pku.edu.cn, limin05@sei.pku.edu.cn, mengna06@sei.pku.edu.cn

Abstract

Monitoring sensitive events is of great significance for adaptive software. Specifying a sufficient set of constraints prior to software deployment is necessary to detect the presence of such events during execution. Here we propose a visual constraint specifying approach, which can generate monitoring code automatically, so as to make the constraint specification easy, and the adaptation of software easy. The detailed representing approach, generating algorithm, and a typical implementation are introduced also.

1. Introduction

When running in the current dynamic and even chaotic network environment, software should have the ability to monitor different sensitive events, so as to keep the quality of service and protect itself. To detect the presence of such events at runtime, specifying a sufficient set of constraints at developing time is necessary.

Software constraint is firstly introduced in requirement phase. Zave divided software requirements into “functions of” and “constraints on” software systems [1]. Although during the following several years, constraint related issues have not gained sufficient attention, there have been some research works that explore the constraints on software. In design phase, some researchers considered constraint as the third element of software architecture, together with component and connection [2]. OMG proposed OCL (Object Constraint Language) to specify the constraints of object, such as invariants, preconditions and post conditions of operations, methods or state changes [3]. Works on Web Services also consider constraint as one key element to be faced with [4]. In implementation phase, monitor, annotation etc are proposed, to implement software constraints [5] [6]. Although aforementioned work gained good result, research on constraint still seems to be an emerging issue so far.

1.1 Problem

Most of common software development models divide the software process according to the software function: function analyzing, function designing, and function implementing. Considering that constraint always adheres to function, it seems that constraint may be processed similarly: specify the constraint in analyzing phase, refine the constraint in designing phase, and code the constraint in implementing phase, etc. For many years, this mode has been put into practice. But when the main environment of software changes from single computer to network, this model meets some challenges. For example, some constraints can only be confirmed when it is to be deployed. But after the software has been deployed, it is apt to be evolved, so as to be adaptive to the changing network environment.

Dynamic environment requires a flexible implementation of constraint, and an easy way to trace constraint from specification to implementation. Our former paper [6] focused on the former issue, while this paper focuses on the latter.

1.2 Approach Overview

Our approach is based on the following observation about constraint implementation: although constraint is apt to change, constraint implementation framework is stable. Interceptor in middleware [7] and handler in Web Services implementation are all such mature mechanisms. Actually, many current middleware can be considered as different constraint implementation frameworks, e.g. TP (Transaction Processing) Monitor and Authentication Service. Based on those common constraint implementation frameworks, constraint can be changed more simply, compared with the functional part of software.

This paper proposed a visual constraint specifying approach for adaptive software. The benefits of this
approach include: (1) Visual expression reduces the difficulty of constraint specification; (2) The visual specified constraint can generate constraint implementation code automatically, which makes constraint traceable, and makes adaptation of constraint convenient.

The remainder of the paper is organized as follows. Section 2 introduces details about the proposed approach. Section 3 delivers the implementation of the proposed approach. Section 4 points to related work. Section 5 concludes this paper with our ideas for further work in this area.

2. Approach

Constraint is the specification proposed by customers or developers to guarantee the stability of software systems' execution and the credibility of the outputs. Therefore, the same constraint defined by users can be applied to different functions as long as these functions need its support; on the other hand, a function can also make use of several constraints at the same time if only the function requires their warrants in different aspects. In this way, we will bring forward a visual approach to define constraints separately from the functions related to them, as well as a mechanism to segregate the constraints’ specifications from their implementations.

2.1 Visual Representation

People who are used to interacting with computers through GUI (Graphical User Interface) share the opinion that the graphical approach is so intuitive that even users with little knowledge about the field are able to pick up its usage and command the interaction skills after a short period of time. Motivated by the common idea of different people, we expect to employ tables and graphs to simplify the process to acquire customers’ constraints.

Of course, customers may have different kinds of constraints. However, most of the constraints concern the interfaces between different objects since whether the communications on these boundaries succeed or not directly influences the stability and credibility of software systems. Therefore in our approach, we put emphasis upon two kinds of constraints related with interfaces between different objects—parameter related constraints and temporal related constraints.

In order to dwell on our approach, we will introduce an example of the class “ShoppingCart”, which is widely used in web-based shopping system. This class is responsible for handling clients’ shopping processes. Its interface includes: “addItem()”—a method invoked to add products into the shopping cart, “setClientInfo()”—a method used to fill in necessary information of customers, such as credit card number and mail adderss, as well as “checkout()”—a method called when customers need to check out. Detailed information is shown in Figure 1.

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

**Figure 1. Detailed information of ShoppingCart**

From the example of ShoppingCart, we can find out one kind of constraints concerning the parameters. A simplest and most typical constraint is that when the method addItem() is called, the intValue() of the input parameter n must be a positive integer. And if we ignore the constraint; once a user of the shopping system inputs a minus figure by accidence, the system will make out unsatisfactory outputs. Besides, we could also find other constraints with careful observation. For instance, the return values of addItem() and checkout() should be both positive; otherwise, neither method’s output will be passed on to other methods as an input parameter.

![Figure 2. An intuitive way to acquire constraints on parameters](image)

To describe the constraints mentioned above in an intuitive way, we exhibit users tables similar to the one shown in Figure 2, which is companied by some text fields (See Figure 2). The left two columns of the main table are filled in by our tool automatically, according to the parsed results from the target program. In Figure 2, the tab folder “addItem” aims at acquiring parameter
constraints on the method addItem(). When a customer wants to specify constraints on the first parameter n, he (she) needs only to write the constraints into the “constraint” column corresponding to the parameter. However, if one constraint concerns several parameters so that it is difficult for the customer to decide where to write it down; the customer is also encouraged to put it down into the blank after the prompt “Additional Constraint(s)”. Additionally, although the parameter names defined by customers are displayed in the table; when specifying constraints, customers are expected to make use of the appointed form as “P{…}”. And the reason for the regulation is also closely related with our approach’s implementation.

In addition to the constraints about parameters of methods, customers may also worry about the temporal constraints of the method invocation sequence. A most authoritative case to confirm the requirement is mentioned in [13], which goes that in the 914 classes defined in J2SDK 1.3.1, 81 classes are found to have method temporal constraints. And when turning back to our ShoppingCart, we may also find such sequence constraints—C1: before the method checkout() is called, both the method addItem() and the method setClientInfo() must have been invoked so that the operation of checkout can proceed successfully with necessary information; and C2: once the method checkout() is called, it can never be called again unless either of the other two methods is invoked in case that the same operation to checkout is wrongly executed twice. The graphical approach to acquire the temporal constraints mentioned above is still under study. However, we have already made some achievements by supplying customers with tables similar to the one shown in Figure 2 (see Figure 3).

![Figure 3. An intuitive way to acquire constraints on temporal sequence constraints](image)

In Figure 3, each line is associated with a method relevant to the temporal sequence constraints; “START” and “END” in the last two lines are not actual methods of the interface, but the identifier of start point and end point of the invocation sequence. Their introduction is to express the start related constraints and the end related constraints. Each column represents one kind of temporal relationship between methods. Here we focus on four temporal relationships: 1) P: “some event must have happened in the past”; 2) PL: “the last event must be some event”; 3) F: “some event must happen in the future”; 4) FN: “the next event must be some event”. Therefore, in Figure 3, the aforementioned temporal constraint C1 is expressed with “0 AND 1” in Grid (3,1) (“0” represents method “addItem”, and “1” represents method “setClientInfo”. The corresponding relationship between number and method is shown on the head of each line), and C2 is represented with “0 OR 1” in Grid (3,4).

### 2.2 Monitoring Code

After obtaining customers’ constraints, in our approach, the next step is to generate monitoring code from the specified constraints.

For the reason that the constraints are specified separately from functions, the implementation of the constraints—monitoring code will also be generated independently. Therefore, in our approach, we implement dynamical instrumentation of monitoring code into monitored software systems with the technique of AOP (Aspect-Oriented Programming) [9] to verify the software systems’ behavior against constraints at runtime, which will be further explained in Section 3. Once monitoring code detects a violation to the constraints, the monitoring code will be responsible to react by terminating the invocation of the method and reporting errors to customers.

### 3. Constraint Implementation

The implementation of different kinds of constraints is quite different. For instance, for the implementation of parameter constraints, the constraint is only presented as a Boolean expression, whose value will directly influence the subsequent operations of the generated program. On the other hand, temporal constraint will be translated into FSM (Finite State Machine), which can be used to generated monitoring code. This is much more difficult.

This section introduces how to generate FSM from tabular-based temporal constraints. The principle of generating FSM is to list all potential states by combining state of different events, then reduce unnecessary states, and consequently find all transitions between states.
3.1 Algorithm of translating

To make the algorithm easy to understand, we use pseudo code to show the translating algorithm. The translating algorithm includes four main steps: construct initial state table, delete conflicted states, construct translated table, and reduce table. Now, we explain key content of these 4 steps. Of course, there are many detailed contents that cannot be introduced in such a paper.

```java
1 generateFSM(constraint) {
2    string se[] = get_sensitive_events(constraint);
3    int m =
4        getHappenedSensitiveEvents(constraint, se);
5    int n =
6        getJustOccurredSensitiveEvents(constraint, se);
7    int initial_state_num =
8        (n == se.length) ? (1 + 2^m * n) : (1 + 2^m * (n+1));
9    int column_num = (n == 0) ? m+1 : m+2;
10   int coded_states =
11       new int [initial_state_num][column_num];
12   fill_coded_states;
13
14   label_direct_conflict();
15   label_begin();
16   label_end();
17   label_indirect_conflict();
18   delete_error_states();
19
20   int init_fsm = new int[states_num][se.length];
21   for (int i = 0; i < states_num; i++) {
22       for (int j = 0; j < events_number; j++) {
23           check_constraint();
24           calculate_next_state();
25       }
26   }
27   find_non_active_states();
28   find_redundent_state();
29   reduce_states();
30 }
```

Figure 4. Pseudo code for translating algorithm.

Construct initial state table

See line 2-8. The “sensitive events” in the algorithm means those events with temporal constraints. That is, any event appears in the bracket after the label “P”, “F”, “PL” or “FN” is a sensitive event. The completely combination (without considering the order) of these events form the initial state set. Number “m” in the algorithm means the number of “happened” sensitive events. These events can form 2m states. Number “n” in the algorithm means the number of “just occurred” sensitive events. The state number that formed by those events is “1 + 2m * n” (“1” means nothing happened), or “1 + 2m * (n+1)” (if the number of “just occurred” sensitive events is less than the number of “sensitive events”, “1” in (n+1) denotes event that is not “just occurred” sensitive event). Each state represents one case that “which event has just occurred, and what events have happened before”.

Delete conflicted states

See line 10-14. Many states formed in step one are invalid states. Before generating FSM, we must delete them. There are four main kinds of invalid state: 1) direct conflicts. That means, in this state, the just occurred event was labeled “not happened”. 2) “begin” related conflicts. That means, sequence can only start with some method(s), but in some states, some other events happened; 3) “end” related conflicts. That means, in some state, some event happened after event that must be the last event; 4) indirect conflicts. That means, current event occurs, but precondition event(s) did not happen.

Construct initial translated table

See line 16-22. This is the main part of the algorithm. We construct initial FSM first. The width of the table is the length of sensitive event mentioned above. The length of the table is the number of survived states. We process each state according to the current state and the temporal constraint (“which event has just occurred, and what events have happened before”). For each event, we check whether it is valid to occur as next event, following each constraint. If it is valid, then we calculate: when the current event occurred, which is the next state that the FSM will enter.

Reduce table

See line 23-25. As the last step, we check the initial translated table and reduce the table. There are two parts
of work. The first is to delete states that cannot be arrived from the first state (no event happened). The second is to merge states that have the same transitions with others. For two states that have “same transitions”, we mean in these two states, for each incoming event, they will enter the same new state. Merge these states can reduce FSM, but does not affect the graph semantics. After having reduced some states, we get one final FSM that is equal to the high level representation in semantics.

Figure 5 illustrates the FSM generated from figure 3. For each state, all incoming methods (we use “a” to denote the method “addItem()”, “s” to denote the method “setClientInfo()”, and “c” to denote the method “checkout()”) and target states(S0, S1, S2, S3, S4 and -1 for error state) are listed explicitly.

![Figure 5. FSM generated from figure 3](image)

4. Implementation

Basically, we implement our approach as a plug-in of Eclipse so that it enjoys feasible extension point and easy integration with the platform of Eclipse. And our tool is currently based on aspect implementation with the support of JBoss-AOP[8], which provides the ideal interceptor infrastructure so far. The generated monitoring code takes the form of interceptor, which can be weaved into target code (the code we want to verify) on the JBoss-AOP platform. When a user fills a constraint into the table, our tool will generate proper interceptor to check whether the constraint can be satisfied or not.

Furthermore, in order to supply customers with friendly GUIs to acquire their constraints and generate monitoring code, we have mainly resolved problems in two aspects.

Constraints Obtainment

In our approach, we provide graphic user interfaces to users. And for each method to be restricted with users’ limitations, we create a separate table, each item of which is a parameter of the method (See Figure 2). In this way, users can easily write down their constraints on a specific parameter into the blank corresponding to the parameter in the table, of course, with a predefined manner of writing constraints. And once a user submits his (her) constraints, we will store them into an array for later use.

As a matter of fact, four notations are imported to express four temporal relationships. Besides, there are four blanks for each method to collect four kinds of propositions respectively. And the default logical relationship between the four blank is “AND”, while the default logical relationship between blanks belonging to different methods is “AND”, too.

Monitoring Code Generation

Once the aforementioned constraints are provided by users, our approach can get monitoring java code, with the help of some templates predefined by our tools. When a method is invoked, its name and parameters will be passed to the monitoring code for decision about whether or not the invocation meets the constraints settled by customers in advance. And once the invocation is found to be invalid, the monitoring code will prevent the invocation and send messages to customers.

Moreover, since the monitoring code to be instrumented often follows certain formats and contains similar contents, we generate all java files in the same process: (1) Read common information of the final files from some initial files and write them into a new file; (2) Replace the parts not specified in the initial files with constraints given by users, and then write them into the new file so that the constraint will be thoroughly transformed to code.

Additionally, to deploy newly generated java files into projects already created by users, we modify the “jboss-aop.xml” file, which is generated automatically once users create their projects as JBoss AOP projects. Thus, we have to mention here that only if users create JBoss AOP projects, can our mechanism explained in the paper be realized so far.

5. Related work

Our work on the graphical model of interface was
mainly motivated by work of Whaley et al. [13], while the work on the generating FSM is mainly motivated by work of rewriting algorithms which are implemented in Maude [14].

Parnas et al have focused on precise and readable requirement documentation for a long time [10]. They insisted that when specifying the requirement of the software, tabular mathematical representation has advantages of clarity, consistency, unambiguousness, integrity and suitability over the textual representation. Compared with Parnas’s work, our work emphasizes how to generate code from those visual specified constraints automatically.

Visual OCL (Object Constraint Language) introduces a visualization of OCL based on the OCL meta-model [11]. As an alternative solution to textual OCL, it has the same expressive ability with textual OCL, and considers little about how to generate code. Excitingly, although constraints in our approach have overlap with constraints of OCL, it generate constraint code completely, not only the code framework.

EJB is a popular java component model adopted by software developers at server side [12]. According to EJB specification, EJB developers or deployers can specify constraints for methods of component, such as transaction and security. Most J2EE compliant application server products provide visual specification interfaces, which make constraint specification easy. Nevertheless, current EJB specification only cares about a few kinds of constraints.

6. Conclusion

This paper proposed a visual constraint specifying approach. The proposed approach reduced the difficulty of constraint specification, and make adaptation of constraint convenient. In future, we plan to express more familiar constraints, enhance the expression by graphical notations, and combine our approach with OCL, which has a widely influence on industry.

This paper is supported by the National High-Tech Research and Development Plan of China No. 2003AA115430 and the National Grand Fundamental Research Program of China No. 2002CB31200003.

References


