

Making Programs Reliable Using AOP

Shigeta Kuninobu

1 Introduction

We present a method to make reliable application programs using AOP. By using the proposed method, we can automatically make application program more reliable. To make an application program reliable, we usually have to consider reliability at design stage. After that we write an application program with reliability. However, making reliable application program by above mentioned ordinary way may produce malicious program because functional requirements and non-functional requirements are programmed at the same time. Ideally, we should implement functional requirements and non-functional requirements separately. Therefore, we propose a method to install non-functional requirements (e.g. reliability) as an aspect and weave to functional requirements (original application program) by AOP compiler. As an example for reliability, we add recovery functionality to a GUI based application program. That is, we write the recovery functionality as an aspect and generate reliable application by weaving the aspect into the original application source program.

Well known checkpoint recovery scheme makes a backup of memory contents as a checkpoint. And application recovery can be done by restoring the checkpoint information. Generally, checkpoint recovery scheme cannot restore the state just before the unpredictable halt of an application. In section 2, we propose a new recovery scheme which can recover the application just before unpredictable halt. In section 5, we show the result of prototype implementation of recovery scheme proposed in section 2 using AspectJ.

2 Recovery Scheme

This section shows a new recovery scheme for application program which action is determined by a user operation such as pressing the GUI button, using keyboard and/or mouse.

In section 2.1, we describe a conventional checkpoint recovery scheme and its weak points. In section 2.2, we propose a new recovery scheme which overcomes the weak points.

2.1 Checkpoint recovery scheme

Checkpoint recovery scheme makes a backup of volatile memory contents to non-volatile memory periodically. Context recovery after an unpredictable halt can be performed by restoring the non-volatile memory contents back to the volatile memory. According to the scheme, application context can only be recovered up to the recorded checkpoint. The more often we make a checkpoint, the more accurately the recovery can be performed. On the other hand, the more often we make a checkpoint, the slower the main application process becomes.

2.2 Proposed recovery scheme

The recovery scheme proposed in this section have the following features.

- User can restore the program context just before unpredictable halt.
- User may not feel the decrease in application processing speed.

To implement these features, the proposed recovery scheme uses not only checkpoint information but also user operation records. The recovery scheme preserves program context as follows.

- Every user operation is recorded to non-volatile memory. It is important that this process consumes small CPU resource so that user may not feel the decrease in application processing speed.
- CPU utilization monitoring process is started when the user operation record size exceed a certain size (*CPU monitoring threshold*), which user can set arbitrarily (see Figure 1).
- Memory backup process starts when CPU utilization become lower than a certain value (*Memory backup threshold*), which user can set arbitrarily (see Figure 2).
- User operation record is deleted when memory backup has finished.

In order not to make user feel the decrease in processing speed, we add the following condition.

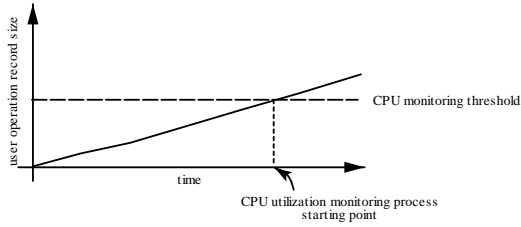


Figure 1: CPU utilization monitoring

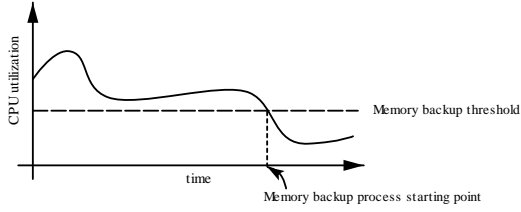


Figure 2: Memory backup process

- When a new user operation occur while the memory backup process is running, the process for user operation takes the priority of memory backup process. That is, the memory backup process is aborted.
 - By this condition, the user operation record size may go up to infinity if the user operation occurs continuously. Therefore we make the memory copy process takes the priority of user operation when a user operation record size reaches the *maximum user operation record size*, which user can set arbitrarily.

According to this scheme, program context recovery after an unpredictable halt is performed by the following two steps.

1. Restore the volatile memory contents by using the checkpoint information.
2. Input the user operations which are recorded in non-volatile memory to the application program.

Our goal is to write the recovery scheme proposed in section 2.2 as an aspect and to make an application program more reliable by weaving the aspect. Figure 3 shows an example application program with recovery functionality. In Figure 3, the parts enclosed by dotted lines (process 1 and process 2) are processes for recovery functionality. Memory copy program which monitors the CPU utilization and starts memory backup process when CPU utilization become lower than threshold value runs in parallel with the main program. Figure 4 shows the memory copy program.

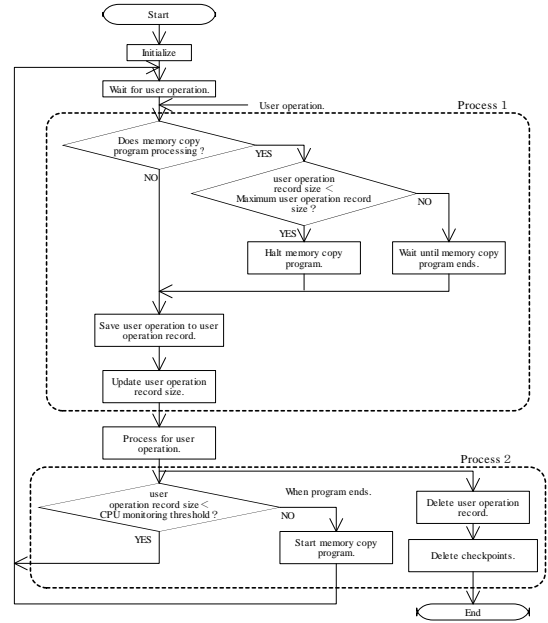


Figure 3: Application program with recovery scheme

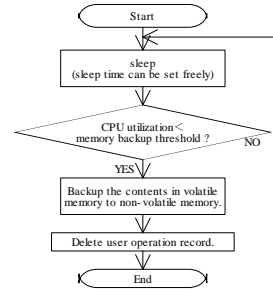


Figure 4: Memory copy program

Section 3 shows the prototype implementation of recovery functionality.

3 Implementation

We use AspectJ to implement the proposed recovery scheme. Following two processes are needed in order to write a recovery aspect.

1. Process for operating user operation record:
 - Records every user operation to non-volatile memory.
 - Inputs user operations which are recorded in non-volatile memory to the application program.
2. Process for controlling memory copy program:

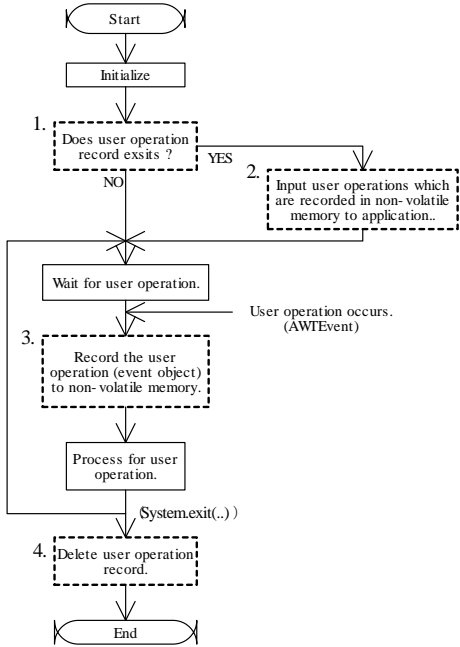


Figure 5: Aspect which operates user operation record

- Starts memory copy program when the user operation record size exceed the CPU monitoring threshold.
- Decides the priority between user operation and memory backup precess (see section 2.2).

In section 3.1 and section 3.2, we explain the detail of above mentioned two processes.

3.1 Process for operating user operation record

In Java environment, event objects (AWTEvent and its inherited objects) are prepared and are generated when the user operates GUI. Also the class to receive those event objects is prepared (EventListener and its inherited classes). Generally, Java GUI application program uses these classes.

Figure 5 shows the program with the processes which operates user operation record. In figure 5, the parts enclosed by dotted lines are the processes which are wove to the original application program.

In the implementation, we recorded every event object in order to record user operation. We deletes the user operation record when System.exit(..) is called. We considered that calling System.exit(..) stands for normal program termination. Context recovery is done by inputting all event objects to the class which receives event objects.

The aspect we made consists of two components, log manager (LogManager.java: 160 lines) and the main aspect

```

1:   pointcut atAllEvents(Object e): args(e) &&
2:     execution(* java.awt.event.*.*(java.awt.AWTEvent+)) &&
3:     !execution(* ReliableAspect.**(..)) &&
4:     !execution(* LogManager.**(..));
5:   before(Object e): atAllEvents(e) {
6:     log.store(e);
7:   }
  
```

Figure 6: Pointcut which takes event objects

```

1:   pointcut targetKey(KeyListener t): target(t) &&
2:     execution(KeyListener+.new(..));
3:   after(KeyListener t): targetKey(t){
4:     key = t;
5:   }

6:   pointcut atFirst(): execution(public static void main(..));
7:   after(): atFirst(){
8:     // Recovery program
9:     for(int i=0;i<log.remain;i++){
10:      AWTEvent obj = (AWTEvent)log.restore();
11:      String s = (obj.getClass()).getName();
12:      // Recovery process for KeyEvent.
13:      if(s.equals(" java.awt.event.KeyEvent")){
14:        String type = ((KeyEvent)obj). paramString();
15:        if( type.indexOf("KEY_PRESSED") != -1 ){
16:          key.keyPressed( ((KeyEvent)obj) );
17:        }
18:        if( type.indexOf("KEY_TYPED") != -1 ){
19:          key.keyTyped( ((KeyEvent)obj) );
20:        }
21:        if( type.indexOf("KEY_RELEASED") != -1 ){
22:          key.keyReleased( ((KeyEvent)obj) );
23:        }
24:      }
25:    }
26:  }
  
```

Figure 7: Program which pass KeyEvent to KeyListener

program (ReliableAspect.java: 155 lines). Figure 6 and figure 7 show the outline of main aspect program.

Log manager is used by the main aspect program and it includes following three methods.

- store(e) : Writes the event object e to log file.
- restore() : Reads the event object from log file.
- deleteLog() : Deletes the log file.

On the other hand, the main aspect program includes the following process.

- Writes user operations (AWTEvent and its inherited objects) to log file:
The atAllEvent pointcut shown in Figure 6 takes every event objects at line #1 and #2, and outputs to log file at line #5, #6 and #7. In Figure 6, line #3 and #4 are written in order not to effect LogManager.java and ReliableAspect.java.
- Recovers the program context:
If the user operation record exists when application program is started, the recovery aspect inputs the event objects which are recorded in the log file to EventListener or its inherited interfaces which are

written in original program. Figure 7 is a program which reads `KeyEvent` objects from log file and inputs the objects to `KeyListener` method. `KeyEvent` object is an object which occurs when user operates a keyboard. `KeyListener` has following three methods.

- `keyPressed(KeyEvent e)` : Method which is called when key is pressed.
- `keyReleased(KeyEvent e)` : Method which is called when key is released.
- `keyTyped(KeyEvent e)` : Method which is called when key is typed.

We can recover the program context by passing the `KeyEvents` to appropriate `KeyListener` method. Line #1 to #5 in Figure 7 takes a pointer to the `KeyListener` object which is declared in original application program. The process which passes the `KeyEvents` to appropriate `KeyListener` method is written in line #13 to #24 in Figure 7. Programs for mouse operation and GUI button operation can be written in the same way.

3.2 Process for controlling memory copy program

As mentioned in Section 2.2, memory copy program which monitors the CPU utilization and starts memory backup process when CPU utilization become lower than threshold value runs in parallel with the main program as a thread. User operation input timing and user operation record size can be taken by an aspect introduced in Section 3.1. The memory copy program is controlled by the main program as follows.

- Memory copy program is started by calling `Thread.start()` method when the user operation record size exceed the CPU monitoring threshold.
- Memory copy program is stopped by calling `Thread.stop()` method when a new user operation occurs while a user operation record size does not reach the *maximum user operation record size*.
- Memory copy program is joined to main program by calling `Thread.join()` method when a new user operation occurs while a user operation record size exceed the *maximum user operation record size*.

4 Results

In this section, we introduce experimental results on the content of Section 3.1 and Section 3.2. We wove an aspect shown in Section 3.1 to two Java GUI application programs, calculator program and draw program (see table 1). As a result of prototype implementation, recovery

Table 1: Application programs

programs	program size	input methods
Calculator	661 lines	GUI button and keyboard
Draw	228 lines	mouse

functionality was added to both calculator program and draw program. We confirmed that the user will not feel the decrease in the processing speed even if we add the recovery functionality. Also we confirmed that memory copy program can be controlled by main program using methods in thread class(`start()`, `stop()` and `join()`).

5 Conclusion

In this paper, we present a method to make reliable application programs using AOP. As an example for reliability, we add recovery functionality to Java GUI application programs. There are advantages of using AOP such as,

- we can implement functional requirements and non-functional requirements separately,
- one reliable aspect (e.g. aspect for recovery functionality) can be used to several programs.

However, proposed methods is currently not used in an industrial project. To use proposed methods in an industrial project, we have to show more advantage for using the proposed methods. Or the risk of changing a current software development method cannot be taken. We think there are many useful additional functionalities other than recovery functionality. AOP should be a useful tool to add such functionalities.