2008

Integrating Statechart Assertions into Java Components Using Aspect

Drusinsky, Doron; Michael, James Bret; Otani, Thomas W.; Shing, Man-Tak

Integrating Statechart Assertions into Java Components Using AspectJ[1]

Doron Drusinsky[2], James Bret Michael, Thomas W. Otani and Man-Tak Shing
Department of Computer Science
Naval Postgraduate School
1411 Cunningham Road, Monterey, CA 93943, USA
{ddrusins, bmichael, twotani, shing}@nps.edu

Abstract

This paper addresses the need for rapid and robust integration of external statechart assertions with the software components of a system of systems (SoSes) for the purpose of runtime verification of the complex SoS behaviors. We describe a framework for connecting assertions to statechart models or to plain Java code using AspectJ. The framework manages connections using a single reusable AspectJ file; designers only need to modify a few lines of source code at the top of the file to link the reusable statechart assertions to a new component. We demonstrate the framework with an example involving a traffic light control system.

Keywords: Aspect-oriented software development, formal specification, Statechart assertions, run-time execution monitoring, validation and verification.

1 Introduction

We showed in our previous work [1-3]:

(1) Executable specifications allow system designers to have a better understanding of their intended behavior;

(2) It is easier for system modelers to specify system behaviors using statechart assertions than text-based specifications, because statechart assertions are visual, intuitive, and resemble statechart design models;

(3) The ability to test statechart assertions independent of the system design ensures that system designers truly understand the required system behavior without being tainted by any pre-conceived solutions.

(4) The fact that individual software components all satisfy their formal behavior specifications is not enough to guarantee that the integrated SoS will behave correctly as specified. The availability of StateRover's automatic white box tester is essential in checking the correctness of the integrated system, and the availability of the executable statechart assertions via run-time execution monitoring makes the automatic checking of test results possible and cost-effective.

We also advocate the use of pre-tested generic statechart assertions to lessen the development time and improve the quality of the statechart assertions in the validation and verification of the SoSes. To that end, we need: (1) an effective way for system modelers to identify the applicable assertion in the assertion repository, and (2) a robust way to integrate the executable code of the reusable assertions with the software components of the SoS.

This paper addresses the need for robust yet flexible integration of external statechart assertions into the software components. The rest of the paper is organized as follows. Sections 2 and 3 provide brief overviews of the statechart assertions and the AspectJ. Sections 4 and 5 present the framework for connecting the statechart assertion implementation code to the Java components using AspectJ, and Section 6 draws the conclusion.

2 The Statechart Assertions

In this section, we give an overview of the statechart assertions using a simple statechart model of a traffic light control (TLC) system.

[1] The research reported in this article was funded in part by a grant from the National Aeronautics and Space Administration. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

[2] Also with Time-Rover Software Inc. www.time-rover.com

978-1-4244-2173-2/08/$25.00 ©2008 IEEE

2.1 The Traffic Light Control System

The traffic light controller is a real-time reactive system that monitors and coordinates the converging traffic at the junction shown in Figure 1. It controls the lights, a camera and a counter in response to the events from its environment (e.g., user events – *start*, *reset;* sensor events – *newCar*, *newTruck,* and timer events – *timeout*).

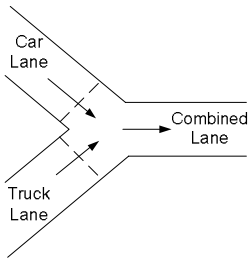

Figure 1. A traffic joint

Figure 2 shows the top level view of the portion of the TLC system for controlling truck lane traffic. It consists of three main states: *CoarseState_Red*, *Green*, and *Idle*. The TLC statechart transitions from the *Idle* state to the *State_1* state when it receives the *start* event. While in the *State_1* state, the TLC statechart toggles between the *CoarseState_Red* state and the *Green* state in response to the *timeout* event from an external timer.

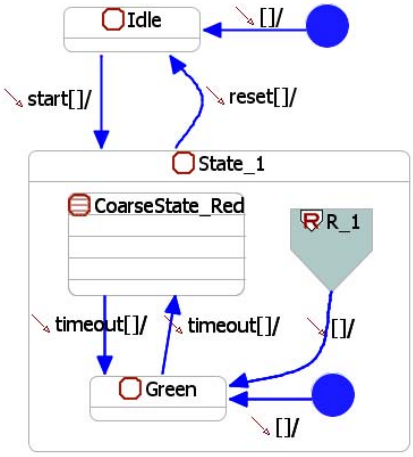

Figure 2. Top level page

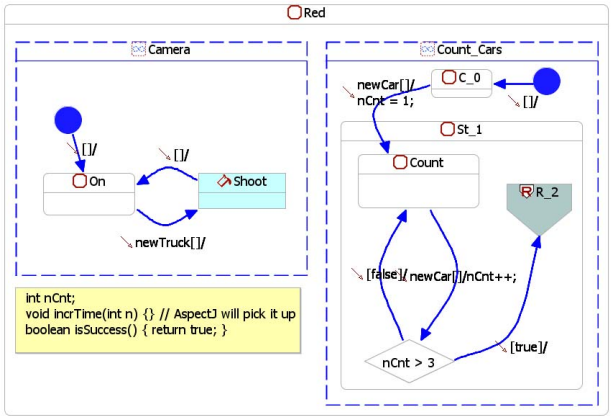

Figure 3. Refined contents of the *CoarseState_Red* state

When in the *CoarseState_Red* state, the TLC forks into two orthogonal activities: a *camera* activity – which takes pictures of any truck entering the junction while the light is red, and a *counter* activity responsible for counting cars passing the junction and causing the TLC to switch to the *Green* state via the page connector *R2* when three cars has passed the junction.

The StateRover's code generator generates one Java controller class for each statechart file. In our case study, we have one statechart diagram file consisting of two pages shown in Figures 2 and 3. The StateRover's code generator automatically connects the two statecharts into a single statechart and generates a single *PrimaryTLC* class for the executable code.

2.2 Statechart Assertions as Embedded Sub-statecharts

Consider the following correctness behavior of the TLC system:

Requirement R1: *At most three newCar events can immediately follow any newTruck or timeout event.*

The statechart assertion for requirement R1 is illustrated in Figures 4.

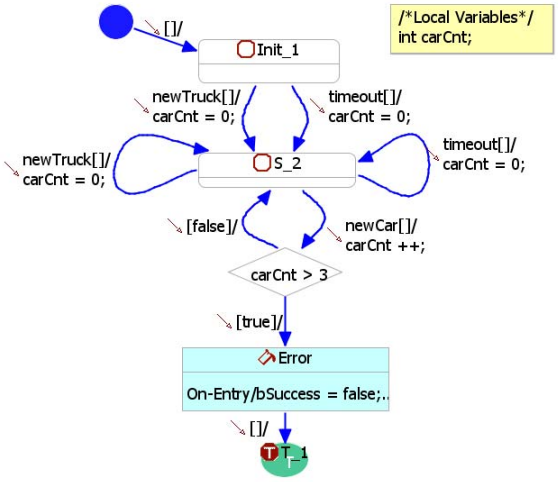

Figure 4. Statechart assertion for requirement R1.

A hardcoded approach for associating the statechart assertion with the TLC statechart of Figure 2 uses sub-statecharts, as illustrated in Figure 5, where the sub-statechart node labeled *assertion* refers to the statechart assertion of Figure 4. This approach is supported by the StateRover toolset and is discussed extensively in [4].

The major drawback of the hard-coded approach is that it only works within the StateRover environment. The software components must be re-entered as StateRover statechart models in order to associate them with the statechart assertions. Hence, we propose a more flexible way to connect the assertions to the software components at the Java source code level.

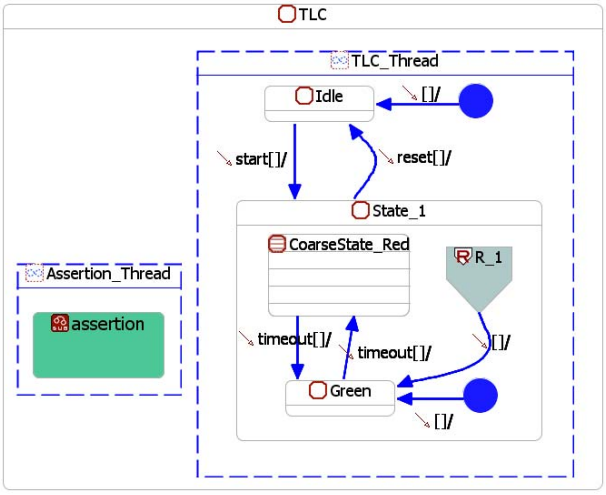

Figure 5. Figure 2 with embedded assertion sub-statechart

3 AspectJ

AspectJ is an extension of the Java programming language to support the increasingly popular aspect-oriented software development (AOSD) paradigm [5]. The AOSD addresses the need to better support the separation of crosscutting concerns, i.e. concerns that cut across several classes in an object-oriented system, by providing new constructs and tools to modularize these concerns into single units called *aspects*. (Refer to the October 2001 issue of Communications of the ACM for a good overview of AOSD and aspect-oriented programming.) AspectJ introduces several new language constructs (e.g., *pointcuts* and *advices*) to define where and how a crosscutting concern is addressed [6]. AOSD with AspectJ is supported by the Eclipse AspectJ Development Tools (AJDT) [7].

3.1 Aspect Declaration

An aspect is a modular unit of crosscutting implementation defined by aspect declaration, like the *assertionAspect* shown below. (Refer to Listing 1 at the end of the paper for the complete listing.)

```
public aspect assertionAspect {

   // private attributes
   Assertion assertion;

   // a set of pointcuts
   pointcut construct(): ...;

   pointcut execMethod(): ...;

   pointcut execIncrTime(): ...;

   pointcut execIsSuccess(): ...;

   // a set of advices
   boolean around(): execIsSuccess() {...}

   after() returning: construct() {...}

   after() returning: execIncrTime() {...}

   after() returning: execMethod() {...}

   // a set of utility methods
   private void fireAssertionIncrTime(...) {...}

   private void fireAssertion(...) {...}

   private void checkAssertionLegality() {...}
}
```

An aspect declaration has a form similar to that of class declaration; it may include pointcut declarations, advice declarations, and all other kinds of declarations allowed in Java class declarations.

3.2 Pointcut Designator

A pointcut defines a set of well-defined points (called join points) in the execution of the program where the crosscutting concerns should be addressed. AspectJ provides several primitive pointcut designators. For example, the *construct* pointcut designator in Listing 1

```
pointcut construct(): target(PrimaryTLC)
   && call(void execTRConstructor(boolean));
```

defines the set of join points at which the method of the target *PrimaryTLC* object matching the Java signature

```
void execTRConstructor(boolean))
```

is called during the execution of the program, and the *execMethod* pointcut designator

```
pointcut execMethod(): target(PrimaryTLC)
   && execution(int *(..))
   && !execution(int getInstanceID(..))
   && !execution(int getPS(..))
   && !execution(
            int execTReventDispatcher(..))
   && !execution(int execTRminorCycle(..))
   && !execution(
            int execTRFireSubstatecharts(..));
```

defines the set of join points at which any method of the *PrimaryTLC* object that return an *int* is invoked, except the five methods (*getInstanceID*, *getPS*, *execTReventDispatcher*, *execTRminorCycle* and *execTRFireSubstatecharts*) specified.

3.3 Advice

Advice defines the body of the code and the time relative to each join point in the pointcut the code should be executed. AspectJ supports 3 kinds of advices (*before*, *after*, and *around*). As the name suggested, the code for the *before* and *after* advices are executed before and after the join point. The *around* advice runs in place of the join point it operates over. For example, the advice

```
boolean around(): execIsSuccess(){
   ...
}
```

replaces the PrimaryTLC's *isSuccess* method and returns the result of the assertion's *isSuccess* method instead. In addition, the *after* advice has two special cases, denoted by *after returning* and *after throwing*. The *after returning* advices are run if the preceding computations at the join point terminate normally, otherwise, the *after throwing* advices matching the exception are run. For example, the advice

```
after() returning: construct(){
   ...
}
```

is run after the PrimaryTLC's constructor method is successfully executed; it instantiates a new instance of the Assertion object and calls its *checkAssertionLegality* routine to inspect the assertion code for unsupported features.

4 Connecting Statechart Assertions to Java Components using AspectJ

The benefit of AspectJ in our context is that it provides a dynamic mechanism of associating assertions with the body of code (or UML artifacts) they are asserting about. Hence, for the TLC example, rather than using a hard-wired linkage between the TLC of Figure 2 and the assertion of Figure 4 (hardcoded using a sub-statechart, a-la, Figure 5), the association between the two is done using the intermediate AspectJ file shown in Listing 1.

This aspect manages a two way association between the TLC and the assertion as shown in Figure 6.

1. It detects events of interest within the *PrimaryTLC* and propagates them to the assertion. This is done in *execMethod* pointcut. In particular, the TLC events of interest are all *PrimaryTLC* methods that return an *int*, i.e., all *PrimaryTLC* event-handlers, except those specified. AspectJ automatically detects every invocation of a *PrimaryTLC* method of interest, then checks to make sure that there is a valid assertion object, and propagates the event to the assertion via its *fireAssertion* routine.
2. It detects the success or failure of the assertion and propagates that boolean value back to the *PrimaryTLC*, so the *PrimaryTLC* can report that value back to the JUnit test driver if needed. This is done in *execIsSuccess* pointcut. AspectJ will substitute the execution of the *PrimaryTLC*'s *isSuccess* method with the assertion's results whenever the *isSuccess* method of the *PrimaryTLC* is called.

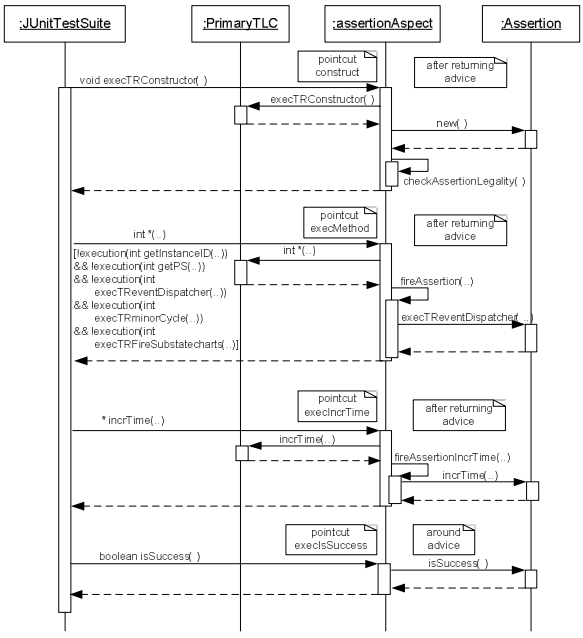

Figure 6. Connecting Statechart Assertions via AspectJ

5 A Framework for Integrating Statechart Assertions Code to Java Components

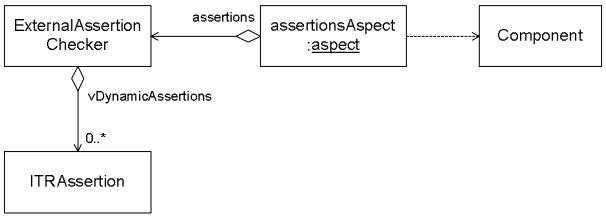

Figure 7. The framework class model

Figure 7 shows the class diagram of the framework for connecting zero or more assertions to a component. The framework uses the *vDynamicAssertions* vector in its *ExternalAssertionChecker* class to keep track of zero or more associated assertions in the assertion repository. Each time when AspectJ detects the invocation of any event-handler of the component, it will propagate the event to the assertions by using the *ExternalAssertionChecker* centralized event dispatcher shown below:

```
public int execTReventDispatcher(
             int nEvent, Object obj0,
             Object obj1, Object obj2)  {
  int nRet = 0;
  Iterator<ITReventDispatcher> it =
             vDynamicAssertions.iterator();
  while (it.hasNext()) {
    ITReventDispatcher assertion = it.next();
    nRet += assertion.execTReventDispatcher(
             nEvent, obj0, obj1, obj2);
  }
  return nRet;
} /* execTReventDispatcher */
```

When AspectJ detects the invocation of the isSuccess method of the component, it will substitute the execution of the component's *isSuccess* method with the *ExternalAssertionChecker's isSuccess* method of the shown below:

```
public boolean isSuccess() {
   Iterator it =
             vDynamicAssertions.iterator();
   while (it.hasNext()) {
      ITRAssertion assertion =
                  (ITRAssertion)it.next();
      if (!assertion.isSuccess()) return false;
   }
   return true;
}
```

6 Conclusions

This paper presents a new framework for connecting assertions to statechart models or to plain Java code using AspectJ. The framework eliminates the need for the designers to reengineer (or re-specify) the software components of a SoS as StateRover statechart models in order to associate them with the statechart assertions.

The proposed framework manages the connections using a single reusable AspectJ file. Designers only need to modify a few lines of source code in the 4 pointcuts at the top of the file (e.g. replacing PrimaryTLC with a new component's class name and updating the signature patterns of the methods of interest in the new component) to link the reusable statechart assertions to a new component.

AspectJ uses the standard Java run-time reflection to extract information about the method calls to the component at run-time and make corresponding calls to the statechart assertion. Such approach allows flexible integration of external statechart assertions into the software components of a SoS and enables the designers to modify either the software components or the assertions in isolation. However, the use of Java run-time reflection may add a considerable overhead; a 30-40 fold slowdown is reported in [8].

References

[1] D. Drusinsky and M. Shing, "TLCharts: Armor-plating Harel Statecharts with Temporal Logic Conditions", *Proc. 15th IEEE International Workshop in Rapid Systems Prototyping*, Geneva, Switzerland, 28-30 June 2004, 29-36.

[2] D. Drusinsky, M. Shing and K. Demir, "Creation and Validation of Embedded Assertions Statecharts", *Proc. 17th IEEE International Workshop on Rapid Systems Prototyping*, Chania, Greece, 14-16 June 2006, 17-23.

[3] M. Shing, D. Drusinsky and T. Cook, "Quality assurance of the timing properties of real-time, reactive system-of-systems," Proc. 2006 IEEE/SMC International Conference on System of Systems Engineering, Los Angeles, CA, April 24-26, 2006, 6 pages.

[4] D. Drusinsky, *Modeling and Verification Using UML Statecharts - A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*, Elsevier, 2006, ISBN 0-7506-7949-2.

[5] Aspect-Oriented Software Development Web Site (AOSD), http://www.aosd.net

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, "An Overview of AspectJ," *Proc. 15th Euro. Conf. Object-Oriented Programming (ECOOP 2001)*, Budapest, Hungary, June 18-22, 2001, *LNCS* 2072, Springer, 2001, 327-353.

[7] AspectJ Web Site, http://www.eclipse.org/aspectj/

[8] M. Oriol, "Techniques of Java Programming: Reflection," Techniques of Java Programming Lecture notes, Department of Computer Science, ETH Zurich, Switzerland, May 15, 2006. http://se.ethz.ch/teaching/ss2006/0284/book/Lect5.pdf

Listing 1.

```
import java.lang.reflect.*;
import java.lang.NoSuchFieldException;

public aspect assertionAspect {
   // this aspect connects a primary (
   // PrimaryTLC type) and the assertion(s)
   Assertion assertion;

   pointcut construct(): target(PrimaryTLC)
      && call(void execTRConstructor(boolean));

   pointcut execMethod(): target(PrimaryTLC)
      && execution(int *(..))
      && !execution(int getInstanceID(..))
      && !execution(int getPS(..))
      && !execution(
               int execTReventDispatcher(..))
      && !execution(int execTRminorCycle(..))
      && !execution(
               int execTRFireSubstatecharts(..))
      ;

   pointcut execIncrTime(): target(PrimaryTLC)
      && execution(* incrTime(..));

   pointcut execIsSuccess(): target(PrimaryTLC)
      && execution(boolean isSuccess(..));

   // Instead of PrimaryTLC's isSuccess() use
   // the assertion's isSuccess()
   boolean around(): execIsSuccess() {
      if (assertion == null) return false;
      try {
         Method method = assertion.getClass().
            getMethod(
               "isSuccess", new Class[] {});
         Boolean retVal = (Boolean)method.
            invoke(assertion, new Object[] {});
         System.out.println(
            "assertion.isSuccess =" + retVal);
         return retVal.booleanValue();
      } catch (NoSuchMethodException nsfe) {
         System.err.println(
            "incrTime() not found in Assertion; "
            + nsfe);
      } catch (Exception e) {
         System.err.println(
            "Exception in fireAssertionIncrTime:"
            + e);
      }
      return false;
   }

   // After returning from constructor do ...
   after() returning: construct() {
      assertion = new Assertion();
      checkAssertionLegality();
      // some features supported by embedded
      // assertions are not supported here
      System.out.println(
         "Assertion constructed by aspect");
   }

   // After returning from calls to PrimaryTLC's
   // incrTime() do ...
   after() returning: execIncrTime() {
      if (assertion == null) return;
      Object[] _args = thisJoinPoint.getArgs();
      fireAssertionIncrTime(_args);
   }

   // After returning from calls specified in
   // execMethod() do ...
   after() returning: execMethod() {
      if (assertion == null) return;

      String sMethodName =
         thisJoinPoint.getSignature().getName();
      Object[] _args = thisJoinPoint.getArgs();
      fireAssertion(sMethodName, _args);
   }

   private void fireAssertionIncrTime(
                              Object[] _args) {
      if (_args.length == 0) {
         System.err.println(
            "incrTime must have an int argument");
         return;
      }
      try {
         if (_args[0] instanceof Integer) {
            Integer nI = (Integer)_args[0];
            assertion.incrTime(nI.intValue());
            System.out.println(
               "assertion.incrTime(" +
               _args[0] + ") -- fired");
         }
      } catch (Exception e) {
         System.err.println(
            "Exception in fireAssertionIncrTime:"
            + e);
      }
   }

   private void fireAssertion(
           String sMethodName, Object[] args) {
      try {

         // Rather than firing the method directly
         // we fire execTReventDispatcher()
         // because it takes care of side-effects
         // such as "primaryEntered" etc.
         Field field = assertion.getClass().
            getField("TREVENT_" + sMethodName);
         int n = field.getInt(assertion);
         switch (_args.length) {
         case 0:
            assertion.execTReventDispatcher(n);
            System.out.println("assertion."
               + sMethodName + " - fired via "
               + "execTReventDispatcher");
            break;
         case 1:
            assertion.execTReventDispatcher(
                                    n,_args[0]);
            System.out.println("assertion."
               + sMethodName + "(Object) "
               + "- fired via "
               + "execTReventDispatcher");
            break;
         default:
            System.err.println(
               "Unsupported number of args in "
               + "fireAssertion ("
               + _args.length + ")");
         }
```

```
      } catch (NoSuchFieldException nsfe) {
        // Not every event the PrimaryTLC
        // receives is in the assertion.
        // For those that are not, we reach this
        // spot --> do nothing
        System.out.println("ignoring "
          + sMethodName + " (are assertion "
          + "statecharts generated with "
          + "isPublic checked?)");
      } catch (Exception e) {
        System.err.println(
          "Exception in fireAssertion:" + e);
      }
    }

    // some features supported by embedded
    // assertions are not supported here
    private void checkAssertionLegality() {
      if (assertion == null) return;
      String sUnallowedMethod = "";
      try {
        Method method = assertion.getClass().
          getMethod("primaryEntered",
            new Class[] {String.class});
        if (sUnallowedMethod != null)
          sUnallowedMethod += " primaryEntered";
        method = assertion.getClass().
          getMethod("primaryExited",
            new Class[] {String.class});
        if (sUnallowedMethod != null)
          sUnallowedMethod += " primaryExited";
        method = assertion.getClass().
          getMethod("primaryFlowchartEntered",
            new Class[] {String.class});
        if (sUnallowedMethod != null)
          sUnallowedMethod +=
            " primaryFlowchartEntered";
        method = assertion.getClass().
          getMethod("primaryFlowchartExited",
            new Class[] {String.class});
        if (sUnallowedMethod != null)
          sUnallowedMethod +=
            " primaryFlowchartExited";
      } catch (NoSuchMethodException e) {
        // ignore: no such method is ok
      } catch (Exception ex) {
        System.err.println("Exception in "
          + "checkAssertionLegality: " + ex);
      }
      if (!sUnallowedMethod.equals("")) {
        System.err.println("Assertion event "
          + "unsupported by AspectJ assertion "
          + "weaving: " + sUnallowedMethod);
      }
    }
}
```