A Library to Modularly Control Asynchronous Executions

Hiroaki Fukuda
Shibaura Institute of Technology
3-7-5 Toyosu
Koto, Tokyo, Japan
hiroaki@shibaura-it.ac.jp

Paul Leger
Escuela de Ciencias Empresariales
Universidad Católica del Norte
Chile
pleger@ucn.cl

ABSTRACT

Asynchronous programming style has been widely adopted for a variety of reasons one such being the rise of Web applications. Using non-blocking operations is a major approach to enforcing asynchronous programming. A non-blocking operation requires decomposing a module that consists of a set of blocking operations into more than two modules, in turn, leading to a variety of challenges such as *callback spaghetti* and *callback hell*. This paper presents SyncAS, a library specification to address these two problems. This library specification follows the ECMAScript standard; indeed, we provide a prototype implementation for ActionScript3. SyncAS proposes a novel approach that enables programmers to *virtually* block a program execution and restart it at arbitrary points of the program, instead of waiting for a callback execution. As a result, programmers do not need to decompose a module even if non-blocking operations are adopted. SyncAS uses aspect-oriented programming to control the program execution in an oblivious manner.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Language**]: Language Constructs and Features

General Terms

Design, Language

Keywords

Virtual block, asynchronous programming, aspect-oriented programming, modularity

1. INTRODUCTION

This paper proposes a design and prototype implementation of a library that allows for modular control asynchronous executions on Web development by using Aspect-Oriented Programming (AOP) [2]. Recently asynchronous programming has been widely adopted for a variety of reasons. The main idea of asynchronous programming is to decompose a blocking operation that waits for its completion into a non-blocking operation, which immediately returns control. Whenever the non-blocking operation execution ends, a piece of code known as callback, is invoked. The use of callbacks is commonly used in event-driven programming, which is a common approach on Web development. For example, JavaScript and ActionScript provide support for user interactions employing this style of programming. However, event-driven programming is difficult to maintain because programmers have to divide sequential executions into several callback methods, resulting in tangled control flow over callback methods. This problem is well-known as *callback spaghetti* [4]. This problem can be addressed with nested callbacks, which have dependencies on data that have been returned from previous asynchronous invocations. Unlike callback spaghetti, this approach can keep sequential pieces of code at one place (*i.e.,* not scattered). However, this sequential code presents an unfamiliar program structure, known as *callback hell* [5]. As a consequence, the piece of sequential code generated are not easily reused.

This paper proposes a library to modularly control asynchronous executions. This library, named SyncAS, *virtually* blocks the execution when a program invokes a non-blocking operation, then restarts it after the completion. With SyncAS, a programmer, who implements a module that uses non-blocking operations, specifies when the executions should be blocked and/or restarted by Aspect-Oriented Programming, meaning that the administration of the blocking and restarting of operations can also be controlled. We propose a prototype implementation of SyncAS for ActionScript3.

2. ASYNCHRONOUS PROGRAMMING PROBLEMS

Asynchronous programming is now widely-adopted among mainstream programmers [1]. This section briefly describes asynchronous programming and some of difficulties by showing an example of callback spaghetti because of the space limitation.

Callback Spaghetti. The difficulty with using callback methods is the fact that converting a particular synchronous call-site into an asynchronous call-site. This is because the transformation requires a programmer to represent the continuation of the original site as a callback method. Using a viewer of images downloaded from a server, we illustrate the problems concerned with callback methods.

Listing 1 shows two classes: ImageViewer and Request. The ImageViewer class contains two methods. The show method receives a certain url and invokes send, provided by Request, with url. Then, show invokes analyze method with two arguments such as a url and the data downloaded. The second method, analyze, verifies and converts the received data to an image. The Request

SAC'15 April 13-17, 2015, Salamanca, Spain.
Copyright 2015 ACM 978-1-4503-3196-8/15/04...$15.00.
http://dx.doi.org/10.1145/2695664.2696034

class has the send method that actually downloads some data by using download method of the Downloader class. For this example, We assume download is a blocking operation that takes a significant period of time for obtaining the result.

```
class ImageViewer {
  function show(url:URL):void {
    var data = new Request().send(url);
    Image img = analyze(url, data);
    // show image
  }
  function analyze(url:URL, data:Byte):Image {
    if (verifyData(data))
      return convertToImage(url, data);
    else
      // throw an exception
  }}
class Request {
  function send(url:URL):Byte {
    return new Downloader().download(url);
  }}
```

Listing 1: An example of synchronous program

```
class ImageViewer {
  var url:URL;
  function show(url:URL):void {
    this.url = url;
    var req = new Request().setNext(analyze);
    req.send(url);
  }
  function analyze(data:Byte):Image {
    if (verifyData(data))
      return convertToImage(this.url, data);
    else
      // throw an exception
}}

class Request {
  var next:Function;
  function setNext(f:Function):void {
    next = f;
  }
  function send(url:URL):void {
    var dl = new Downloader();
    dl.addEventListener(Downloader.Complete, completeCallback);
    dl.downloadAsync(url);
  }
  function completeCallback(e:Event):void {
    next(e.data as Byte);
  } }
```

Listing 2: Previous program rewritten to support asynchronous style

Because of several advantages such as reusability, maintainability, dividing a system into a composition of modules is natural. Therefore, in Listing 1, Request encapsulates how to get data from sources. Now, suppose that send method changes from blocking to non-blocking to carry out the data downloaded: downloadAsync. Listing 2 shows the rewritten program of Listing 1 for this change. Two major changes are found in Listing 2. First, invoking analyze is removed from show because send, that invokes downloadasync, returns immediately without any data. Instead, the reference of analyze method is passed as a callback by using setNext, which is defined in Request for this purpose. Second, an attribute called url is defined in ImageViewer to keep an URL object created in show because this object is also used in analyze. As described previously in this section, using a module that uses an asynchronous method requires representing the continuation as a callback. Consequently, if the location of the continuation is far from the call-site, understanding control flow is difficult, leading to callback spaghetti.

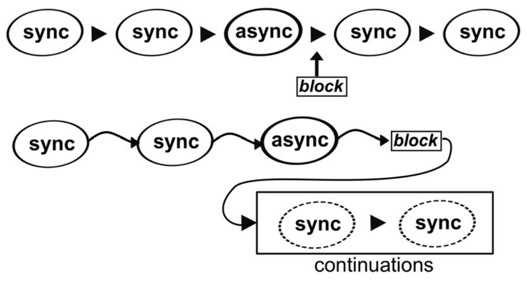

Figure 1: Virtual block approach

3. SYNCAS

This section presents our approach, reviewing its basic idea, followed by separation of concerns in a program and the management of control flow. From this section, we define the word *asynchronous method* as a method that contains non-blocking operations, and *synchronous method* as a method that contains only blocking operations. For example, in Listing 2, send is an asynchronous method while downloadasync is a non-blocking operation.

3.1 SyncAS in a Nutshell

We propose a lightweight library called SyncAS that enables programmers to *virtually* block the execution of a program at arbitrary points, and then restart it by releasing the block. As mentioned in Section 2, modular programming is adequate to develop a large-scale applications. In addition, synchronous programming 1makes it easy to understand control flows. However introducing asynchronous programming into modular programming requires several changes such as dividing one method into call-site and its continuations, then passing a continuation to another module as a callback, making complex control flows. Therefore, SyncAS enables a programmer to write a program in a synchronous style while the program is executed asynchronously.

Figure 1 shows how an application with SyncAS behaves. In Figure 1, *sync* represents a synchronous method invocation and *async* represents an asynchronous method invocation. With SyncAS, after an async invocation, the rest of invocations are not executed but kept as continuations. These continuations are executed when the async invocation ends.

```
class ImageViewer {
  function show(url:URL):void {
    var data = new Request().send(url);
    // virtually blocked
    Image img = convertToImage(url, data);
  }
  function convertToImage(url:URL, data:Byte):Image {..}
}
class Request {
  function send(url:URL):Byte {
    return new Downloader().download(url);
  }
  function completeCallback(e:Event) {
     // code of completeCallback
     // After this execution, virtual block is restarted
}}
```

Listing 3: Rewriting the image viewer example with SyncAS

We now rewrite Listing 1 and 2 with SyncAS in Listing 3. The program execution is virtually blocked in Line 4, and the remaining executions are stored as continuations (in this pieces of code, Line 5). These continuations are executed after completeCallback is invoked in Line 15. As shown in the class ImageViewer in Listing 3, each method looks synchronous, however send asynchronously behaves. Note that a block in SyncAS is not real blocking, meaning that runtime engines such as JavaScript engines do not block any operation.

3.2 Blocking and Restarting Executions

SyncAS requires a programmer to specify when the execution of a program is blocked and restarted. Suppose two programmers P1 and P2 to independently implement ImageViewer and Request in Listing 3. From a modular programming viewpoint, P1 should only know the interfaces of Request (*e.g.,* method name, the number of arguments, and return type) when P1 uses methods in Request (*e.g.,* send). Suppose P2 now changes the implementation of send from using download to downloadAsync to enhance responsiveness. In this case, the modified application does not work correctly.

To address aforementioned modularity issue, programmers P1 and P2 need to modify show to add a block in Line 4 of Listing 3. This change cannot be acceptable because the effect of modification in a module is disseminated to every call-site. In other words, this approach breaks modular programming advantages. To make an application in Listing 3 works correctly, a block should be written in ImageViewer class while the corresponding restart should be done in Request class, and a block and its corresponding release should be used as a pair, resulting in crosscutting concerns. For this reason, SyncAS adopts AOP [2] to encapsulate the pieces of code concerned with a set of block and restarts in a module (*i.e.,* aspect). SyncAS provides a simplified API for this purpose as follows.

```
SyncAS.addAsycOperation(block, restart);
```

In this API, block represents the point in which the program execution should be virtually blocked while restart refers to when this execution restarts. With the example of Listing 3, this API can be used as follows.

```
SyncAS.addAsycOperation("Request.send","Request.completeCallback");
```

Note that this API can be used in an implicit manner, meaning that P2 can write this code behind P1.

3.3 An Aspect to Control Flow

The key technique behind the previous API is AOP, which allows programmers to use aspects to modularize crosscutting concerns like present in callback spaghetti.

```
1  // pointcut and advice to block an execution
2  var pcBlock = function(jp, env) {
3    return jp.className == "Request" && jp.methodName == "send";
4  }
5  var advBlock = function(jp, env) {
6    SyncAS.blockJP(jp);
7    env.add("blockJp", jp);
8  }
9  // pointcut and advice to restart an execution
10 var pcRestart = function(jp, env) {
11   return jp.className == "Request" &&
12          jp.methodName == "completeCallback";
13 }
14 var advRestart = function(jp, env) {
15   SyncAS.restartJP(env.get("blockJp"));
16 }
17 var virtualBlock = new Aspect([pcBlock,   advBlock,   AFTER],
18                               [pcRestart, advRestart, AFTER]);
19 SyncAS.deploy(virtualBlock);
```

Listing 4: A pseudo implementation of API by an aspect

In the pointcut-advice model of AOP [3], an aspect specifies program execution points of interest, named *join points*, through predicates called *pointcuts*. When an aspect matches a join point, it takes an action, called *advice*.

Listing 4 shows a pseudo implementation of the API applied to Listing 3. In SyncAS, an aspect is composed of a list of 3-tuples: a pointcut, an advice, and an advice kind. For example, in Listing 4, pcBlock/pcRestart and advBlock/advRestart correspond to pointcuts and advices respectively. The jp and env refer to a join point and an environment respectively. For example, the virtualBlock aspect stores the join points blocked. Finally, SyncAS provides a deploy method to activate an aspect.

We now review the Listing 4 in detail. The piece of code expresses two pairs of pointcuts and advices: one is in lines 1–8 for blocking and another is in lines 10–16 for restarting the execution. For the first pointcut-advice pair, the advice is invoked after the execution Request.send. In this advice, SyncAS.blockJP blocks the execution of the matched join point. As a consequence, SyncAS keeps the statement in Line 5 of Listing 3 as a continuation. For the second pointcut-advice pair, the advRestart advice is invoked after the execution of Request.complete. As a result, the blocked join point is restarted by using SyncAS.restartJP. In Line 17, an aspect is created with two pairs of pointcuts and advices, which is deployed in Line 19. In this way, an aspect enables a programmer to encapsulate pieces of code that control a flow in one place.

4. CONCLUSION

Asynchronous programming has been widely adopted nowadays and callback is a typical solution that enables asynchronous programming. This programming style, however, produced other drawbacks called *callback spaghetti* and *callback hell*. In addition, introducing asynchronous programming into a modular programming requires to divide a method into call-site and its continuations, making complex control flows. To this end, this paper proposes a modular library called SyncAS that enables programmers to *virtually* block the execution of a program at arbitrary points and restart it by using aspect-oriented techniques. In this paper we describe the basic idea of SyncAS. With SyncAS, a programmer can write asynchronous executions using a synchronous style, meaning that the programmer does not need to concern whether an invocation is asynchronous or synchronous.

As future work, we plan to apply SyncAS to existing practical applications that use complex asynchronous executions.

Acknowledgement. This work was partially supported by JSPS KAKENHI Grant Numbers 26330089.

5. REFERENCES

[1] G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen. Pause 'n' play: Formalizing asynchronous c#. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 233–257, Berlin, Heidelberg, 2012. Springer-Verlag.

[2] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda, and A. Mendhekar. Aspect oriented programming. In *Special Issues in Object-Oriented Programming*. Max Muehlhaeuser (general editor) et al., 1996.

[3] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622, pages 46–60, 2003.

[4] T. Mikkonen and A. Taivalsaari. Web applications - spaghetti code for the 21st century. In *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications*, SERA '08, pages 319–328, Washington, DC, USA, 2008. IEEE Computer Society.

[5] M. Ogden. Callback hell. `http://callbackhell.com/`.