Towards a Virtual Block Approach to Tame Asynchronous Programming

Takeshi Azegami
Shibaura Institute of Technology
3-7-5 Toyosu
Koto, Tokyo, Japan
ma12003@shibaura-it.ac.jp

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

Paul Leger
Universidad Católica del Norte
Chile
pleger@ucn.cl

ABSTRACT

Asynchronous programming has been widely adopted in domains such as Web development. This programming style usually uses callback methods, a non-blocking operation, allowing high responsible user interactions even if the application works without multi-threading. However this style requires to uncouple a module into two sub-modules at least, which are not intuitively connected by a callback method. The separation of the module brings the birth of other problems: *callback spaghetti* and *callback hell*. This paper proposes a *virtual block* approach to address previous problems. This approach enables a programmer to virtually block a program execution and restart it at arbitrary points in the program. As a result, programmers do not need to uncouple a module even if non-blocking operations are adopted; therefore, callback dependences disappear. Using aspect-oriented programming, this approach uses aspects to control the execution of a program in an oblivious manner. As a consequence, programmers do not need to care whether pieces of code use blocking or non-blocking operations. We implement a proof-of-concept of this approach, named SyncAS, for ActionScript3.

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

Asynchronous programming, aspect-oriented programming, modularity, virtual block

1. INTRODUCTION

Modularity [8] is crucial to implement large-scaled software because it allows developers to decompose a software into several modules, enhancing maintainability, adaptability, and reusability of software. For example in object-oriented programming, a class (*i.e.,* module) gives interfaces that allow other classes to use it while hiding implementation details.

Asynchronous programming has been adopted because of a variety of reasons: hiding latency of the network in distributed applications, maintaining the responsiveness of single-threaded applications, avoiding resource costs of creating many threads. Developers of interactive applications, like found on the Web, widely use this programming style. The basic idea of asynchronous programming is to decompose a blocking operation that waits for its completion into a non-blocking operation that immediately returns control. Whenever the non-blocking operation execution ends, a piece of code, known a callback method, is invoked. Introducing a callback method requires to divide a sequential set of operations into executions of methods that must wait the completion of non-blocking operations.

The use of callbacks in asynchronous programming brings problems that affect the modular development of software. The most notable problems are *callback spaghetti* [6] and *callback hell* [7]. Callback spaghetti refers to the concern implementation that has a complex and tangled control structure because of the continuations over callback methods. Callback hell means deeply-nested callbacks that have dependencies on data that have been returned from previous asynchronous invocations.

This paper proposes a novel approach called *virtual block* to modularly use asynchronous programming. This approach provides programmers by means of a virtual block the execution when a program invokes a non-blocking operation, then restarts it after the completion. We implement this approach in SyncAS, a prototype library for ActionScript3, an object-oriented language with first-class functions. With SyncAS, a programmer can use a module that uses non-blocking operations in a synchronous fashion, meaning that the programmer does not care about asynchronous executions. Modularly, a programmer specifies when the executions should be blocked and/or restarted by Aspect-Oriented Programming (AOP) [4]. As SyncAS is implemented for ActionScript3, we think the virtual block approach can be used on any language that follows the ECMAScript standard.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to asynchronous programming and its problems. Section 3 describes our approach, and Section 4 explains its impact on application developments with asynchronous techniques. Sec-

.

BICT 2014, December 01-03, Boston, United States

DOI 10.4108/icst.bict.2014.257939

tion 5 presents conclusions and some future work.

Availability. An initial implementation of SyncAS is available online on http://www.pragmatic.jp/research/syncas/ as an Adobe Air application.

```
1  function getImageFromUrl(urlStr:String):void {
2    var url:URL = new URL(urlStr);
3    var dl = new Downloader();
4    dl.addEventListener(Downloader.Complete, completeCallback);
5    dl.download(url);
6  }
7  function completeCallback(e:Event):void {
8    Object data = e.data;
9    setData(data);
10 }
```

Listing 1: Asynchronous programming in ActionScript.

2. ASYNCHRONOUS PROGRAMMING PROBLEMS

Asynchronous programming is now widely-adopted between mainstream programmers [1]. This section briefly describes asynchronous programming and its two main difficulties: *Callback Spaghetti* [6] and *Callback Hell* [7].

2.1 Asynchronous Programming

The basic principle behind asynchronous programming is to decompose a synchronous operation that blocks the software execution until its completion into a non-blocking operation and a callback method. The non-blocking operation immediately returns, and the callback method is invoked when this operation is finished [1]. Callback methods are widely used for asynchronous programming. Listing 1 shows an example of the use of a callback method in ActionScript3. The method getImageFromUrl includes a non-blocking operation (Line 5), meaning that a certain callback method needs to be associated to get the result of download (Line 4). To this end, a method called addEventLister is used. The first argument of this method is an event type (Downloader.Complete) and the second is the callback method reference (completeCallback). In this way, a synchronous operation is decomposed into two methods (lines 4 and 5). Moreover asynchronous programming sometimes requires global flags to manage a state inside of an application.

2.2 Callback Spaghetti

The difficulty with using callback methods is fundamentally 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 associated callback methods.

Listing 2 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 url and the data downloaded. The second method, analyze, verifies and converts the received data to an image. The Request class has the send method that actually downloads some data by using the download method of the Downloader class. For this example, we assume download is a blocking operation that takes a significant period of time.

```
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 2: 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(this.url);
    //show image
  }
  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 3: An program rewritten by asynchronous.

Because of several advantages such as reusability, maintainability and adaptability, dividing a system into a composition of modules is natural [8]. Therefore, in Listing 2, Request encapsulates how to get data from sources. Now, suppose that the send method changes from blocking to non-blocking to carry out the data downloaded: downloadAsync. Listing 3 shows the rewritten program of Listing 2 for this change. Two major changes are found in Listing 3. First, invoking analyze is removed from show because send, that invokes downloadAsync, returns immediately without any data. Instead, the reference of the 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, a module that uses an asynchronous method requires to represent the continuation as a callback. As a consequence, if the location of the continuation is far from the call-site, understanding control flow is difficult, leading callback spaghetti.

2.3 Callback Hell

The use of nested anonymous functions is a solution to prevent callback spaghetti when non-blocking operations are adopted in asynchronous programming. Listing 4 uses an anonymous function instead of analyze in Listing 3.

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

Listing 4: Asynchronous program with an anonymous function.

A programmer can write a callback method directly inside a call-site, leading the ease to understand a control flow. However, the definition and immediate execution of an anonymous function does not allow developers to reuse it. Moreover, nested callback methods in large-scale applications leads to a mix of collocated code fragments and creates complex control flows [2].

3. SyncAS

This section presents the description of our approach, reviewing its basic idea: the separation of base concerns of a program and its control flow management. From this section, we define *asynchronous method* as a method that at least contains a non-blocking operation, and *synchronous method* as a method that contains only blocking operations. For example, in Listing 3, the send method is an asynchronous method because downloadAsync is a non-blocking operation.

3.1 Introducing Virtual Block

As mentioned in Section 2, modular programming is adequate to develop large-scale applications because of several advantages. In addition, synchronous programming style makes it easy to understand control flows. However, introducing asynchronous 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, our *virtual block* approach enables a programmer to write a program in a synchronous style while the program is executed asynchronously.

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

We now rewrite listings 2 and 3 with our virtual block approach in Listing 5. The program execution is virtually blocked in Line 4, and the reaming executions are stored as continuations (in this piece of code, Line 5). These continuations are executed after completeCallback is invoked in Line 15. As shown in the class ImageViewer in Listing 5, each method looks synchronous, however send behaves asynchronously. Note that a block in our approach does not mean a real blocking operation, meaning that the runtime engine of ActionScript does not block any operation. As a result, a user can continue using an application without waiting for non-blocking operation executions.

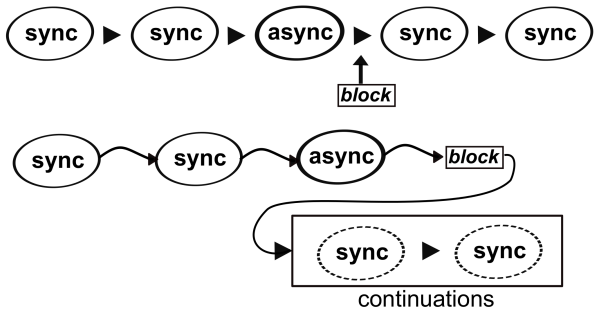

Figure 1: Virtual block approach.

```
1  class ImageViewer {
2    function show(url:URL):void {
3      var data = new Request().send(url);
4      //virtually blocked
5      Image img = convertToImage(url, data);
6    }
7    function analyze(url:URL, data:Byte):Image {/* omitted */}
8  }
9
10 class Request {
11   function send(url:URL):Byte {
12     return new Downloader.download(url);
13   }
14   function completeCallback(e:Event) {
15     //code of completeCallback
16     //restarting the execution
17 } }
```

Listing 5: An example with SyncAS.

3.2 Blocking and Restarting an Execution

Our approach requires a programmer to specify when the execution of a program is blocked and restarted. To modularly allow developers to specify these two points, we use Aspect-Oriented Programming (AOP) [4]. Suppose two programmers P1 and P2 to independently implement ImageViewer and Request of Listing 2. From a modular programming viewpoint, P1 should only know interface of P2 (*e.g.,* method name, number of argument, return type) when P1 uses send of Request. In other words, P1 does not need to know how send is implemented. However, 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 fix this problem, it is necessary to specify a block and its restart at a certain place in each class. Therefore, the control of a program execution crosscuts these classes. To modularize the implementation of the control of an execution, AOP allows programmers to use aspects, as embodied in *e.g.,* AspectJ [3]. In the pointcut-advice model of AOP [5], an aspect specifies program execution points, named *join points*, through predicates called *pointcuts*. When an aspect matches a join point, it takes an action, called *advice*. Therefore, we can use an aspect to specify where a program execution is blocked and restarted in a modular manner.

```
1 // pointcut and advice to block an execution
2 var pcBlock = function(jp, env) {
3   return jp.className == "Request" && jp.methodName == "send";
4 }
5
```

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

Listing 6: An aspect to control a program execution.

Listing 6 shows an aspect that controls the execution in Listing 5. In SyncAS, an aspect is composed of a list of 3-tuples: a pointcut, an advice, and an advice kind. For example, in Listing 6, pcBlock/pcRestart and advBlock/advRestart correspond to pointcuts and advices respectively. The jp object refers to a join point. To expose context information, a pointcut and an advice can take an extra parameter as environment (env) that is shared inside of an aspect. 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 6 in detail. The piece of code expresses two pairs of pointcuts and advices: one is in lines 1–9 for blocking and another is in lines 12–18 for restarting the execution. For 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, which is stored in the environment. As a consequence, SyncAS keeps the statement at Line 8 in Listing 5 as a continuation instead of executing. For the second pointcut-advice pair, the advRestart advice is invoked after the execution of Request.complete. As a result, blocked join points are restarted by using SyncAS.restartJP. In Line 20, an aspect is created with two pairs of pointcuts and advices, which is deployed in Line 22. In this way, an aspect enables a programmer to encapsulate pieces of code that control a flow in one place.

4. IMPACT ON TRADITIONAL ASYNCHRONOUS PROGRAMMING

Module-based development is fundamental for large-scaled applications. In this development, interfaces should make loosely-coupled relations among modules, allowing programmers to concentrate on their tasks. However, as described in Section 2, introducing non-blocking operation breaks this advantage because of callbacks. That is, a programmer has to modify its implementation to follow the specification of an application when the implementation of another module changes. Virtual block approach enables developers to encapsulate pieces of code that handle a control flow in one module (*i.e.,* an aspect), meaning that a programmer does not have to modify the implementation with asynchronous code. Moreover, the independent implementation of control flow and base implementation brings two extra benefits. First, developers can implement previous concerns in an isolated manner. Second, a programmer do not need to know whether methods are synchronous or asynchronous. Thereby, by introducing virtual block approach, dividing tasks into programmers can be easier:

- *experienced programmers* provide asynchronous methods by using non-blocking operations and create aspects to control the program execution,
- *non-experienced programmers* just use provided methods without caring if these methods are synchronous and asynchronous.

5. CONCLUSION

Nowadays, asynchronous programming has been widely adopted because of various reasons. The use of callback methods is a typical solution that enables asynchronous programming. However, this solution generates other drawbacks such as *callback spaghetti* and *callback hell*. In addition, asynchronous programming requires to divide a method into call-sites and its continuations, making complex the understanding of control flows. To solve this problem, we propose a novel approach called *virtual block* that enables programmers to virtually block the execution of a program at arbitrary points and restart it by using aspect-oriented techniques. This paper describes the main idea behind virtual block and discusses its impact on asynchronous programming. With our virtual block approach, a programmer can write asynchronous programming as synchronous style, meaning that the programmer does not have to care if an invocation is asynchronous or synchronous.

Finally, we point out future issues to be tackled. First, the implementation of virtual block approach is inspired by the implementation of AspectScript [9], which is an aspect extension implementation for JavaScript. Although JavaScript and ActionScript follow the ECMAScript specification, there are differences between both languages (*e.g.,* classes and gradual typing). Second, we need to verify the expressiveness of our approach by applying it to some applications and comparing it with existing solutions.

6. 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] K. Kambona, E. G. Boix, and W. De Meuter. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, DYLA '13, pages 3:1–3:9, New York, NY, USA, 2013. ACM.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[4] 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.

[5] 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.

[6] 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.

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

[8] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[9] R. Toledo, P. Leger, and E. Tanter. Aspectscript: Expressive aspects for the web. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 13–24, New York, NY, USA, 2010. ACM.