Supporting Methods and Events by An Integrated Abstraction

YungYu Zhuang
University of Tokyo
http://www.csg.ci.i.u-tokyo.ac.jp/

Shigeru Chiba
University of Tokyo
http://www.csg.ci.i.u-tokyo.ac.jp/

ABSTRACT

Events have been introduced into a number of programming languages since they are known as a useful programming abstraction. Although those languages provide a language construct directly supporting events, they also provide a similar construct in parallel, which is a method. This paper proposes a new language construct named *method slot* and a new language *DominoJ*, which is a Java-based extension supporting method slots. A method slot is a single language construct supporting both methods and events by an integrated abstraction. This paper shows how method slots work as methods and events.

1. INTRODUCTION

Event-driven programming has been recognized as a useful mechanism in a number of domains such as user interface, embedded systems, databases [11], and distributed programming [5]. The basic idea of events is to register an action that is automatically executed when something happens. A number of techniques and libraries have been developed for providing event support [3, 9]. For example, the Observer pattern [3] is a well-known technique for event-driven programming in a language without event support. A built-in language support for events has been also proposed in a number of languages. Implicit invocation languages and aspect-oriented languages might be classified into this category.

Although the usefulness of events are well recognized, previous languages have been dealing with events as an additional programming abstraction and they have been separately providing other established abstractions such as functions and methods. For example, EScala [4], which is one of the latest programming languages with event support, provides events and methods in parallel. Programmers choose which language construct they use, either events or methods, according to programming contexts. However, events and methods are similar abstractions. Both of them are used to execute a code block although they are differently triggered; events are reactive but methods are proactive.

This observation led us to develop a new language construct named *method slot*. It is an integrated construct of events and methods. We have then developed the *DominoJ* language, which is an extension of Java and supports method slots.

The rest of the paper is organized as follows. Section 2 briefly introduces EScala and a typical language construct for event-driven programming. Section 3 presents the design of method slots and DominoJ. Section 4 mentions related work and Section 5 concludes this paper.

2. MOTIVATION

Events directly supported by language constructs provide two significant properties, which are not provided by programming conventions such as the Observer pattern. The two properties are *event composition* and *implicit invocation*.

Listing 1 shows a program in EScala[1], which is an extension of Scala. In this example, two events moved and changed are declared by the evt keyword. afterExec makes a primitive event, which means method invocation. The moved event occurs just after the execution of the setPosition method. The changed event is declared as the composition of the moved event and the primitive event afterExec(setSize). The ability to compose a higher-level event from lower-level ones is significant for modular programming [4].

An event handler, which is a method executed when an event occurs, is bound to the event by the += operator. At line 6 and 7, the redraw method is bound to the changed events on the object itself and its parent. An event handler bound to an event is implicitly invoked when the event occurs. The event handler does not have to be explicitly invoked at places where the execution contexts match the condition triggering the event. If a direct language support for events is not available and thus the Observer pattern is used, then the event handler must be explicitly invoked at multiple places and scattered over the program. Such a program shows less locality with respect to event handling and thus it is difficult to understand. Maintaining the scattered fragments of the code for invoking an event handler at appropriate places is also error-prone [4].

Although language constructs for events remarkably improve the design of programs, existing languages with such constructs have two similar constructs in parallel; one for events and the other is for methods. In EScala, events are

[1]The syntax follows the example in EScala 0.3 distribution.

RAM-SE'12, June 13, 2012, Beijing, China

Listing 1: The component example in EScala

```
1  class Component(name: String, parent: Component) {
2    var left = 0; var top = 0;
3    var width = 0; var height = 0;
4    evt moved[Unit] = afterExec(setPosition)
5    evt changed[Unit] = moved || afterExec(setSize)
6    changed += redraw
7    if (parent != null) { parent.changed += redraw }
8    def setPosition(x: Int, y: Int) {
9      left = x; top = y
10   }
11   def setSize(w: Int, h: Int) {
12     width = w; height = h
13   }
14   def redraw () {
15     System.out.println(name + ": redraw.");
16     // redraw itself
17   }
18 }
```

explicitly declared by the evt keyword while methods are by the def keyword. Although methods are also used as event handlers, the event itself must be declared separately from methods. This separation is beneficial to a certain extent since the design intentions become explicit; what are events is clearly visible.

However, events and methods have notable similarity. Both of them lead the execution of a code block although events are reactive but methods are proactive. Integrating similar mechanisms into one can be always an option of programming language design. It will simplify a language and help us understand essential abstraction as Self [10] provided prototype-based objects and revealed essence of object-oriented programming. Suppose that an event is a method call (strictly speaking, an event does not correspond to a method but a method call. We usually say an event is a method in this paper for simplicity). When that event happens, the body of the method with the name specified by that method call is executed. At the same time, if that event is bound to an event handler, then the body of that event handler, which is also a method, is also executed.

The only difference is whether or not a code block must be explicitly associated with the event. A method does not need explicit association but it is implicitly associated by default. Thus we could recognize that calling a method is regarded as causing an event that the method is implicitly associated with. It is typical understanding but not essentially right that we strongly discriminate between method calls and events (or methods and event handlers) by the idea that methods are proactively called but event handlers are reactively invoked.

3. DOMINOJ

We propose a new language construct named *method slot*, which integrates both methods and events while preserving the two useful properties of event mechanisms such as event composition and implicit invocation. We also show our prototype language named *DominoJ* supporting method slots. It is an extension of Java.

3.1 Method Slot

In DominoJ and languages supporting method slots, it is not possible to directly call a method. Suppose that an expression component.move(), which would call a method move on an object component if the language were plain Java. This expression is interpreted in DomoinoJ as one causing an event on a method slot with the name move on the object component.

A method slot is an array of function closures and is an object's property like a field. In DominoJ, a method declaration declares not only a method but also a method slot with the same signature (name and parameter types). Whenever a new object is created, new method slots are created and attached to that object since method slots are per-instance attributes. A method slot initially contains only a function closure that invokes the method with the same signature as that method slot. The target object for that method invocation is the same object that the method slot belongs to. The elements in a method slot can be manipulated during runtime by the operators mentioned below.

When an event occurs on a method slot of an object, all the function closures in the method slot are executed in order. For example, when an event occurs on a method slot move of an object component, the function closures in the move method slot are executed. If it contains only the initial element, the move method is invoked on the object component (Fig. 1).

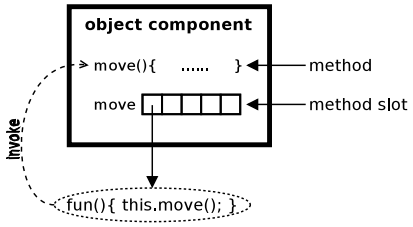

Figure 1: An object component which has a method slot move

3.2 Operators for method slots

DominoJ provides four operators for manipulating an array element in a method slot: =, ^=, +=, and -=, as shown in Table 1. These operators are the only syntax extensions in DominoJ. It is possible to add and remove a function closure to/from a method slot at runtime.

Their operands at both sides are method slots. A method cannot be their operand. Those operators except -= create a new function closure and add it to the method slot at left-hand side. When the function closure is executed, it causes an event on the method slot at the right-hand side. For example, the following expression appends a new function closure:

```
component.move += observer.updated
```

where move and updated are method slots and component and observer are objects. The function closure is appended to the method slot move of the object component. When the function closure is executed, it causes an event on the method slot updated of the object observer. The objects component and/or observer are optional; if they are not explicitly specified, the this object is used instead.

The -= operator removes function closures from the method slot at left-hand side. It removes function closures that will cause an event on the method slot of the object specified by

Table 1: The four operators for method slots

operators	description
`=`	add a new function closure and remove the other elements from the method slot.
`^=`	insert a new function closure at the beginning of the array.
`+=`	append a new function closure to the end of the array.
`-=`	remove function closures that will cause an event on the method slot at the right-hand side.

the operand at the right-hand side. A function closure is also automatically removed when the target object is garbage-collected. For example, if a function closure will cause an event on the method slot move of the object component, then it is removed when component is garbage-collected. When an event occurs on a method slot, nothing is executed if the method slot does not contain any function closures. No exception is thrown.

A method slot is typed; its parameter types and return type are statically specified when it is declared. The function closures in a method slot must have the same parameter types and return type as the method slot. Thus, the two method slots given as the left and right operands of the operators such as `+=` must share the same type.

3.3 A method slot as an event mechanism

A method slot also can be used as an event in EScala. By default, it is an event implicitly triggered when a program calls a method with the same name as the method slot's. An *event handler* is also a method slot and it can be bound by the `+=` operator to an event, which is another method slot. The event handler is executed when the event is triggered. Note that a program cannot directly call a method in the semantics of DominoJ. It can only invoke a method via an event on the method slot with the same name. However, since the syntax for triggering an event on a method slot is the same syntax as one for calling a method in Java, programmers can be unaware of method slots and see that a method is directly called as in Java and then an event is implicitly triggered.

It is also possible to remove the default function closure (*i.e.* an empty method body) added when a method slot is declared. For example,

```
component.move -= component.move
```

this expression removes the default function closure in a method slot move. Note that the right operand above is interpreted as a method move. This is an exception since the right operand is otherwise a method slot. A program can trigger an event on a method slot that contains no function closures. Triggering such an event does not cause anything.

Event composition is also possible in DominoJ. Listing 2 shows a DominoJ version of the component example mentioned in Section 2. In this program, changed is a higher-level event composed from other events (line 14). Since it is bound to setPosition and setSize (line 9 and 10), it is triggered after setPosition or setSize are executed. If changed were bound by not the `+=` operator but the `^=` operator, it would be triggered before setPosition or setSize.

Listing 2: The component example in DominoJ

```
1  public class Component {
2    private String name = "";
3    private Component parent = null;
4    private int left = 0; private int top = 0;
5    private int width = 0; private int height = 0;
6
7    public Component(String n, Component p) {
8      name = n; parent = p;
9      setPosition += changed;
10     setSize += changed;
11     changed += redraw;
12     if (parent != null) { parent.changed += redraw; }
13   }
14   public void changed(Object[] args) {}
15   public void setPosition(int x, int y) {
16     left = x; top = y;
17   }
18   public void setSize(int w, int h) {
19     width = w; height = h;
20   }
21   public void redraw (Object[] args) {
22     System.out.println(name + ": redraw.");
23     // redraw itself
24   }
25 }
```

Note that method slots setPosition and setSize are events and changed is an event composed from the two events. Unlike in EScala, events are composed by the operators such as `+=` in DominoJ since there is no clear distinction between events and event handlers. Rewriting the program not to use the higher-level event changed is also possible. The following statements:

```
setPosition += redraw;
setSize += redraw;
```

directly bind an event handler redraw to setPosition and setSize. In this case, the higher-level event changed is not used.

4. RELATED WORK

We have already presented EScala and the differences from DominoJ. EScala has a typical event mechanism supporting both event composition and implicit invocation well, but events are supported by a separate construct from method calls. DominoJ integrates methods and events by a single construct while event composition and implicit invocation are supported.

An aspect-oriented language such as AspectJ can define implicit events by pointcut and advice. Pointcuts are event definition and advice is an event handler. Event composition can also be declared in pointcuts. Although AspectJ provides a powerful event mechanism supporting implicit events and event composition, it has some shortcomings as discussed in [4]. Events in AspectJ are global and class-based—while DominoJ allows events to be set per object. Furthermore, it breaks modular reasoning since aspect-oriented languages require global knowledge for compiling and loading classes [1, 6]. On the other hand, DominoJ introduces events by being an extension to object-oriented languages, so it does not have such an issue.

With C# [7] we can declare an event, define its delegate type, and bind the corresponding action to the event. Event composition can be done by adding a delegate to two or more events. Although the delegate interface hides the executor

from the caller, implicit invocation is not supported. The event must be triggered manually when the change happens. In other words, we cannot let an event happen after specified method calls implicitly and must use an explicit statement to trigger it. This forces us to put the same invocation code in all places where we need it, as what we have to do in Java. Another disadvantage is that we always have to check if the event equals `null` before triggering it. The purpose of the check is ensuring that there is at least one delegate for the event. Otherwise it will raise an exception. This is not reasonable from the viewpoint of event mechanism since it just means no one handles the event. In DominoJ no handlers for an event does not raise an exception, and the one triggers the event on a method slot is unaware of handlers.

Ptolemy [8] is a language with quantified, typed events. Ptolemy allows a class to register handlers for events, and also allows a handler to be registered for a set of events declaratively. It has the ability to treat the execution of any expression as an event. The event model in Ptolemy solves the problems in implicit invocation languages and aspect-oriented languages. There are two differences between Ptolemy and DominoJ. First, Ptolemy does not support implicit invocation, which is one of the most significant properties as an event mechanism, whereas DominoJ supports it. Second, all events in Ptolemy's model are global and cannot be set for a specified object though the events can be filtered in the handlers by some mechanisms. The binding in DominoJ is object-based, so it can describe the interaction between objects more properly.

EventJava [2] extends Java to support event-based distributed programming. It combines events and methods by introducing event methods, which are a special kind of asynchronous method. Event methods can specify constraints and define the reaction in themselves. They can be invoked by a unicast or broadcast way. Events satisfying the predicate in event method headers are consumed by a reaction. Context-aware applications can be accommodated easily by the mechanism. The main difference between EventJava and DominoJ is that events in EventJava are also global and application-level. Although EventJava can define complex events clearly, it cannot select the events from an individual object. Moreover, events must be invoked explicitly. Another important difference is that EventJava only supports static one-to-one relations between events and event handlers while DominoJ can maintain one-to-many relations and change the relation at runtime.

5. CONCLUSIONS

We discussed similarity between events and methods and proposed an integrated construct of events and methods. We then presented a new language DominoJ, which supports that construct, named method slot. A method slot is a closure array and is an object's property like a field.

Our future work is to define the semantics to clarify the behaviors of method slots. We also have to evaluate the benefit of method slots. By comparing method slots with typical event mechanisms, the advantages and disadvantages can be analyzed in detail.

6. REFERENCES

[1] S. Chiba, A. Igarashi, and S. Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 539–554, New York, NY, USA, 2010. ACM.

[2] P. T. Eugster and K. R. Jayaram. EventJava: An extension of Java for event correlation. In *ECOOP'09*, pages 570–594, 2009.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[4] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. EScala: modular event-driven object interactions in Scala. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 227–240, New York, NY, USA, 2011. ACM.

[5] A. Holzer, L. Ziarek, K. Jayaram, and P. Eugster. Putting events in context: aspects for event-based distributed programming. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 241–252, New York, NY, USA, 2011. ACM.

[6] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 49–58, New York, NY, USA, 2005. ACM.

[7] Microsoft Corporation. C# language specification.

[8] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP'08*, pages 155–179, 2008.

[9] The Qt Project. Signals & Slots. `http://qt-project.org/doc/signalsandslots`.

[10] The Self project. `http://selflanguage.org/`.

[11] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proc. of the Int'l Conf. on Management of Data (SIGMOD '90)*, pages 259–270. ACM Press, 1990.