In event-driven programming we can react to an event by binding methods to it as handlers, but such a handler binding in current event systems is explicit and requires explicit reason about the graph of event propagation even for straightforward cases. On the other hand, the handler binding in reactive programming is implicit and constructed through signals. Recent approaches to support either event-driven programming or reactive programming show the need of using both the two styles in a program. We propose an extension to expand event systems to support reactive programming by enabling the automation of handler bindings. With such an extension programmers can use events to cover both the implicit style in reactive programming and the explicit style in event-driven programming. We first describe the essentials of reactive programming, signals and signal assignments, in terms of events, handlers, and bindings, then point out the lack of automation in existing event systems. Unlike most research activities we expand event systems to support signals rather than port signals to event systems. In this paper we also show a prototype implementation and translation examples to evaluate the concept of automation. Furthermore, the comparison with the predicate pointcuts in aspect-oriented programming and the details of the experimental compiler are discussed.

Keywords: event-driven programming, reactive programming, signal, behavior

1. Introduction

Recently reactive programming attracts a lot of interest since the need for reactive programs is steeply increasing, for example applications for mobile devices and web browsers. FRP (Functional-Reactive Programming) [7], [12], [25], [34], [35], [36] successfully introduces signals into functional programming. The concept of signals might come from data-flow languages [3], [4], [6], [16], [33], where variables are described as continuous data streams rather than states at a specific time. Such variables are signals, which can participate in the calculation to generate other signals or be given to functions for triggering the reactions; they are very declarative. This programming style is widely used in hardware design and spreadsheet programs, where the expressions are usually described without explicitly specifying the time. When FRP introduces signals into functional programming for reactive programs such as GUI programs, a time signal and event streams are also used in order to properly describe the temporal states in the program. An event denotes something happening, for example a mouse click occurs, and an event stream is a series of events on the timeline. Programmers can use event streams with the time signal to get a constant value (snapshot) of a signal at a specific time. They help to describe the states on the timeline more specifically while keeping the description declarative. This style is followed by several research activities and inspires the contributions towards writing GUI libraries for FRP [10], [27]. In these systems signals are given to describe the propagation of value change and then the compiler can translate them to some kind of events underneath.

The OO (Object-Oriented) community has been developing events, which look similar to signals but actually different. In OO languages that directly support event-driven programming, events are first-class objects, and an event handler bound to a specific event is implicitly invoked when that event happens. Although the OO community notices the convenience of using signals, OO languages have not been directly integrated with signals. Signals are brought into existing event systems, and existing GUI libraries are wrapped in FRP style [18], [22], [30]. Although events have been used with objects for a long time and well integrated into OO languages, propagating values by events is not implicit enough as propagating by signals. All handler bindings, in other words the statements for setting up the methods that react to specific events, must be explicitly specified according to the graph of event propagation. Even for a straightforward use case of events, programmers have to prepare all events and handlers, and bind them together one by one. For a complex event scenario programmers need to prepare too many events and handlers, and might lead to redundant event propagation. Thus, the research activities devoted to OO integration borrow signals from reactive programming and focus on how to integrate signals with events and objects. Most research results come up with a conclusion that the existence of events is still necessary, and signals are used to implicitly propagate parts of the change of values [31]. As a result, events and signals appear in both the FRP solution and the OO integration.

This research is targeted at allowing programmers to use both the implicit style and the explicit style in a program since the re-

---

1 The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
a) yungyu@acm.org
b) chiba@acm.org
search activities mentioned above show the need of using both the
two styles together. The contributions of this paper can be sum-
marized as follows. First, we compare event-driven programming
with reactive programming to point out the need of implicit bind-
ing in event systems. Second, we show how event systems can be
expanded to cover the implicit style in reactive programming
by enabling the automation of handler bindings. In order to show
the feasibility of our idea, we give a prototype implementation of the
automation on an event system, and discuss the advantages
over ones without the automation. The issues that might happen
when implementing this concept on OO languages are discussed
as well. Moreover, the analysis on the essentials of reactive pro-
gramming might give a better understanding of events and sig-
als. Note that this paper is an extended journal version of the
one we presented at COMPSAC2015*. The contents we added
to this extended version include:

- A more complicated event example in Sections 4.4 and 4.5
to show how the automation works among different objects.
We first introduce a kind of graph that makes it easier to
check the dependencies among events and handlers on differ-
ent objects. Then how our compiler infers through object
references is explained.

- The details of our compiler implementation in Section 5.1.
After briefly explaining the implementation of the event sys-
tem we based this proposal on, how the idea of automation
can be concretely implemented is demonstrated with sample
code.

- Qualitative analysis of our reactive extension in comparison
with others. In Section 5.4 we compare with the original
event system and in Section 5.5 we compare with the pred-
icate pointcuts in aspect-oriented programming. At the end
of Section 6, we also compare the the meanings of reactive
used in FRP and other event enhancements.

2. Motivating Example

Programs written in reactive programming can also be imple-
mentated by event-driven programming, though the code might
look quite different. Here we take the example of spreadsheet
programs to discuss the equivalence between what reactive pro-
gramming can do and what event-driven programming can do.
Although spreadsheet programs are developed for accounting,
they can be regarded as an interactive programming environment.
Cells are fields (or variables), and sheets are some sort of objects
that hold a large number of fields. We can give a cell a constant
value or an expression, where formulas can be used to perform
complex calculations. Figure 1 shows a sheet in a spreadsheet
program, where B1 and C1 are given constant values: 2 and 1 re-
spectively, and A1 is given an expression “B1 + C1”. As a result,
the value in the cell A1 will be always equal to the sum of the
values in the cells B1 and C1 even if you arbitrarily change the
value of B1 or C1.

Such a sheet defined in spreadsheet programs can be easily imple-
mented by FRP languages. For example, Fig. 2 shows an

Fig. 1 A sheet in a spreadsheet program.

```
1 <body onload="loader()">
2   <table width="800">
3     <tr><th>A</th><th>B</th><th>C</th></tr>
4     <tr><td>1</td><td>3</td><td>2</td></tr>
5     <tr><td>2</td><td>3</td><td>1</td></tr>
6   </table></tr>
7 </body>
```

Fig. 2 Using Flapjax to implement the sheet example.

```
1 class Sheet {
2   var a: Int = _
3   var b: Int = _
4   var c: Int = _
5   def setB(lub: Int) { b = lub; }
6   def setCInc: Int() { c = c+1 }
7   def ha() { a = b + c; }
8   def ec[Unit] = afterExec(setC)
9   def eb[Unit] = afterExec(setB)
10  def ec[Unit] = afterExec(setC)
11  eb = ha;
12  ec = ec
13 }
```

Fig. 3 Using EScala to implement the sheet example.

implementation in Flapjax [22], which is a JavaScript-based lan-
guage supporting FRP. This program uses HTML elements to
draw the sheet and cells, then gets and sets signals from/to the
cells. Here we do not explain the syntax of Flapjax in detail but
focus on the assignment in Line 18. We can consider the assign-
ment without the declaration of a:

```
 a = b + c;
```

Note that what the three variables hold are signals rather than
constant values. The assignment looks not much different from
the one in imperative programming languages such as Java, but
the meaning is quite different. Whenever b or c is changed, a
is updated automatically. The assignment is always effective and
looks like an equation (although it is not bidirectional: only the
change of the right-hand side can trigger the update of the left-
hand side). Signals are very similar to the cells in spreadsheet
programs, and thus can easily describe the expression in the sheet
example. In reactive programming all updates are automatic and
implicit.

On the other hand, such an assignment in imperative program-
mong languages is only effective just after the assignment is exe-
cuted, and the value at the left-hand side might not be the same
as the one at the right-hand side later until the assignment is exe-
cuted again. Nevertheless, it is still possible to implement such a
program by imperative programming languages with events; we
can use events to denote the value change and ask an event han-
dler to execute the assignment again. Here we use EScala [15], an

---

*1 Enabling the Automation of Handler Bindings in Event-Driven Program-
ming. YungYu Zhuang and Shigeru Chiba. Year: 2015, Volume: 2,
Pages: 137–146, DOI: 10.1109/COMPSAC.2015.48, Publisher: IEEE.
event system that provides events, handlers, and bindings based on Scala [26], to write the sheet example. EScala allows declaring a special kind of field named event to denote something happening. The event can be either implicitly triggered before/after a method call or imperatively triggered through a method-call syntax. As shown in Fig. 3, we have two methods setB and setC, which set the fields $b$ and $c$ respectively (Lines 5–6). Then we can declare two events $e_b$ and $e_c$ that denote the happening of value change of $b$ and $c$ through setB and setC using the primitive afterExec given by EScala, respectively. Line 8 says that $e_b$ is the event occurring after the method setB is executed—here we assume that setB only changes the value of $b$ and leave the discussion about join point model in Section 3. Line 9 is interpreted similarly for $e_c$. The next step is preparing a handler that reacts to the two events properly. As in most event systems, methods play the role of handler in EScala. As shown in Line 10, in this example we need a handler $h_a$ that executes the assignment for updating the value of $a$ according to the values of $b$ and $c$. Finally we have to connect the events and the handler, or else the latter is unrelated to the two events (Lines 11–12). The two statements mean that $h_a$ should be executed after $e_b$ and $e_c$. Such statements are handler bindings, which bind the handler to events. The calculation in the event version is the same as the signal version, but it is manual and explicit. All events must be manually declared, and the handler bindings for them must be manually stated as well. These drawbacks not only make the code longer but also increase the risk of bugs. When the body of $h_a$ (Line 10) is modified, we must carefully update the handler bindings in Lines 11–12 to ensure consistency between the bindings and the handler body.

The observation that both the paradigms can implement reactive programs motivates us to analyze the essentials of reactive programming from the viewpoint of event-driven programming. By comparing the essentials of reactive programming with event-driven programming we can know how to expand event systems to cover the implicit style in reactive programming.

3. An Expanded Event System Supporting Reactive Programming

In this section we propose an expanded event system that can cover both event-driven programming and reactive programming. To know what the extension should be, we clarify what the essentials of reactive programming are and point out what is necessary to support them in event-driven programming. We first describe how these essentials work in reactive programming, and then describe them in terms of events, handlers, and bindings. The comparison between the two descriptions reveals an insufficiency in existing event systems, and led us to propose the expanded event system.

*2 Note that EScala supports event composition to improve the abstraction, but here we just enumerate events and bind the handler to them individually to simplify the explanation. For example, we can declare a composed event instead of $e_b$ and $e_c$ as shown below:

\[
\text{evt ebc[Unit]} = \text{afterExec(setB)} || \text{afterExec(setC)}
\]
The extension must be able to automatically infer the involved events and implicitly bind the handler to them.

3.3 The Expanded Event System

By comparing the two descriptions we can know that it is possible to translate a signal assignment to a handler. However, in existing event systems programmers have to manually check the body of the handler and infer the involved events by themselves. There is no mechanism to bind a handler to multiple events at once without event composition, either. Programmers have to enumerate the involved events or specify certain rules to filter the involved events manually. Once the body of the handler is modified, the bindings might be no longer consistent with the handler body. In other words, existing event systems lack a kind of automation, which means automatically inferring the involved events and implicitly binding the handler to them.

As shown in Fig. 5, an extension to event systems for supporting reactive programming must provide such automation in handlers, which means that a handler can be implicitly bound to all the involved events that are automatically found inside itself. If an extension can enable the automation in handlers, all existing reactive mechanisms can be simply translated to events and handlers.

3.3.1 Automatic Inference

The extension must be able to automatically infer all the involved events in a handler. As shown in Fig. 5, the involved events, \( e_b \) and \( e_c \), are the events for the value change (or more broadly, the setting) of the fields (or variables) that are read in the handler, \( b \) and \( c \). In the terminology of AOP (Aspect-Oriented Programming) these involved events are the join points matched by the set pointcuts for the fields that are read in the handler. Furthermore, we can consider the handler in a more generic way since the body of a handler in event systems might have method calls and conditional branches. In that case, the fields (or variables) that are read in the methods called by the handler must be recursively inferred since they might also be involved in the reevaluation. The fields (or variables) in all conditional branches should also be taken into account since any of them might be involved in the reevaluation at runtime.

3.3.2 Implicit Binding

The extension must also be able to implicitly bind a handler to all the involved events inside itself. When any involved event inside the handler occurs, the handler will be executed to update the values of fields (or variables). Such a binding must be implicit enough to bind a handler to multiple events without specifying the events individually. Event composition is not satisfying either since we still need to explicitly compose individual events into a higher-level event. A similar concept is the filter in event systems or the predicate in AOP, but they are usually used to filter events selected by other event detectors and compose the result into a higher-level one.

4. A Prototype Implementation

In order to show the feasibility of automation, we expand DominoJ [37] (DJ) to ReactiveDominoJ (RDJ) by enabling the automation in method slots as an example of such an expanded event system. DJ is a language that supports event-driven programming, which motivates us to propose RDJ. Note that RDJ is a prototype implementation and has several limitations, but the extension we proposed in Section 3 is more generic and can be implemented in any event system.

4.1 Method Slots and DominoJ

DJ is a Java-based language developed for introducing method slots, a generic construct supporting multiple paradigms. A method slot is an object’s property, which can hold more than one closure at the same time. DJ replaces the methods in Java with method slots. All method-like declarations in DJ are method slot declarations. For example, public void \( \text{setX} \) (int \( nx \)) { this.x = nx; } the method-like declaration for \( \text{setX} \) is a method slot declaration. It is some sort of field that holds an array of closures. When the method slot is called, all closures in it are executed in order with the same given arguments and the value returned by the last one will be regarded as the return value of this method slot (if its return type is not \( \text{void} \)). The body of a method slot declaration is the default closure; it is optional. If the default closure is declared, it will be created and inserted into the array when the owner class is instantiated. At runtime the closures in a method slot can be added or removed using assignment operators, for example using += operator as shown below:

\[
\text{setX} += \text{.update};
\]

means that creating a closure calling the method slot \( \text{.update} \) and appending it to the end of array in \( \text{setX} \).

We can use DJ as an event system. A method slot declaration is equivalent to an event declaration. When the method slot is called, the event (i.e., the join point in the terminology of AOP) occurs and the handlers (closures) in it are executed. The default closure of a method slot can be regarded as the default handler for this event. Besides imperatively triggering by calls, a method slot can also be implicitly triggered after or before other method slot calls by using the assignment operators:

\[
(\text{event}) (\text{assignment_operator}) (\text{handler});
\]

The statement using += operator we showed above is used to let the event (method slot) \( \text{update} \) on an object \( \text{o} \) be triggered after \( \text{setX} \) is triggered. Note that in DJ a method slot can be not only an event but also a handler for other events. Thus, there is no difference between event-event binding and event-handler binding, and all the bindings are dynamically set at runtime by the assignment operators.
public class Sheet {
  private int a, b, c;
  public void setB(int nb) { b = nb; }
  public void setC(int nc) { c = nc; }
  public void updateA() { a = b + c; }
}

4.2 A New Syntax for Enabling the Automation

RDJ allows using the braces operator to enable the automation in method slots (i.e., handlers) in DJ. For example, if we have a method slot updateA in a class Sheet as shown in Fig. 6, we can use the following statement to enable the automation in updateA:

```java
{this.updateA} += this.updateA();
```

Since the fields read in the called method slots are also involved Any method slot that is called in any closure of the method slot inference in the braces operator is described by a piece of pseudo-code as shown in Fig. 7. The braces operator takes one of them fields. In other words, the statement means that binding the handler this.updateA() to all the involved events inside itself as we described in Section 3. The braces operator makes it possible to bind a handler to a set of events that are involved in its body without explicitly specifying them. The braces operator selects the involved events inside a method slot by checking all closures in it at runtime. The semantics of the inference in the braces operator is described by a piece of pseudo-code as shown in Fig. 7. We can consider the relation between all the method slots that belong to the objects held in local variables and parameters. The design decision and the limitation of RDJ will be discussed in a later subsection.

Some readers might notice that in Fig. 7 we only consider the fields and the method slots on the same object (Lines 5 and 7). It is a simplified inference in RDJ based on OO design rather than a limitation of the extension we proposed in Section 3. We observed the convention of OO design and made the assumption: usually fields are only directly used inside the owner object and other objects must access them through getters and setters. We can consider the relation between all the method slots reading a field and all the method slots writing the field as an extended getter-setter relation. We name it reader-writer relation since there might be more than one getter/setter and a getter/setter might get/set more than one field. The reader-writer relation extends getter-setter relation to N-to-N and is not limited to the naming scheme. The inference of the braces operator (Fig. 7) is a process of finding all the writers by a given reader through the fields read in the called method slot, which are the fields read during executing the closures in this method slot, are inferred (Lines 5 and 7). To simplify the design, we simply ignore the method slots that belong to the objects held in local variables and parameters. The design decision and the limitation of RDJ will be discussed in a later subsection.

To make our code clear the underscore symbol _ can be used within the braces operator to refer to the method slot at the right-hand side of the assignment operator. For example, enabling the automation in updateA can be simplified:
4.3 Rewriting the Motivating Example

To show how the automation can be used we first use DJ to write the sheet example in Section 2 and then rewrite it by RDJ to compare with Flapjax. In order to make it easier to compare we assume that there was a Flapjax-like Java-based language, which could be used to rewrite Fig. 2 to Fig. 10 (a). The class IntCell (Fig. 11) is used to emulate the HTML element and follows the getter-setter manner in OO design rather than accessing a field directly; it might be used to wrap a GUI component in existing libraries by extending a class such as the JTextfield in Swing. Behavior is a class representing signals, the usage of which, such as insertValueB and extractValueB, is the same as in Fig. 2. Line 14 shows the merit of reactive programming: the propagation among signals can be simply described.

Figure 10(b) is the DJ version, where only events are used. Note that no additional event declaration is needed since the getter/setter declarations in DJ can also be considered as the events declared for getting/setting the field. Line 14 of Fig. 10(a) is now described in terms of getters and setters as shown in Line 18 of Fig. 10(b). However, the handler ha(updateA) will not be automatically executed to update the value of a. We need to check the body of ha and find e\textsubscript{b} and e\textsubscript{c} by ourselves as mentioned in Section 2. Here the events e\textsubscript{b} and e\textsubscript{c} are b.setValue and c.setValue respectively since we know the relation between getter and setter. Then we can bind the handler this.updateA to them individually, or prepare a higher-level event this.changed as shown in Line 20 for event composition (Lines 16–17) and bind this.updateA to this.changed (Line 18). This program works well as the Flapjax version in Fig. 2, but the inference is not automatic and the binding is explicit. Once the body of updateA is modified, we have to carefully check the bindings in order to make them consistent with the events inside the body.

Figure 10(c) shows the RDJ version. Most lines of code are the same as the DJ version, but the bindings and the higher-level event lines 16–18, 20 of Fig. 10(b) are eliminated. Note that we still need a binding to enable the automation in the handler (Line 16 of Fig. 10(c), otherwise the compiler cannot know whether it is the implicit style used in reactive programming or the explicit style used in event-driven programming. We use the new syntax given by RDJ to automatically infer all the involved events in the handler and implicitly bind the handler itself to them. By comparing Fig. 10(a) with Fig. 10(c), it is also easy to see how a signal can be translated to a field as we discussed in Section 3. The signal assignment for a in Line 14 of Fig. 10(a) is moved to a method slot updateA (Lines 18–20 of Fig. 10(c)), and an additional line for enabling its automation must be stated outside of it (Line 16 of Fig. 10(c)). On the other hand, programmers do not have to explicitly specify which ones are signals, and transformations to/from constant values such as insertValueB and extractValueB are not necessary. RDJ expands the events in DJ to make a step towards reactive programming.

4.4 Dependency Graph

In order to compare the handler bindings in DJ with RDJ, here we use a graph to describe the dependencies among events and handlers: a node is an event or a handler, and an edge is a dependency relationship. As to the relationship, we use a solid line to note a handler binding and use a dashed line with words for the event name.
other types of relationships. Note that here we do not distinguish events from handlers since a method slot can be regarded as both of them. The dependency graph of this.changed in Fig. 10 (b) is shown in Fig. 12, which means that a higher-level event changed depends on two events b.setValue and c.setValue, and the handler updateA is bound to changed. In the DJ version we know which events the higher-level event changed should depend on since we have the knowledge about the default closure of the handler updateA and the getter-setter relation between getValue and setValue. We have to check the implementation of the default closure of updateA, manually enumerate all the involved events, and then explicitly bind them to this.changed. It is not easy to ensure the correctness of the dependency since the dependency behind the default closure of a method slot might be changed later for further extension. On the other hand, in Fig. 10 (c) we simply enable the automation in updateA, in other words using the braces operator on it to select all the involved events, and let the RDJ compiler do the rest. The RDJ compiler recursively infers and selects all the involved events in the body of updateA based on the clues as shown in Fig. 13 (a) and Fig. 13 (b), which can be found in the default closure of updateA and the declaration of IntCell, respectively. The inference performed by the RDJ compiler is shown in Fig. 14, where (this.updateA) is a kind of higher-level event as the this.changed in Fig. 12. Note that the dashed lines are completed by the RDJ compiler and only the one solid line needs to be constructed by programmers.

4.5 Inferring Involved Events through Object References

Figure 18 (a) is an example in DJ for demonstrating how the braces operator infers all the involved events through object references. This example implements a class Meter, which automatically calculates and prints out the distance between two given Shape objects in the Cartesian plane when either the name or the coordinates of the two objects are changed. Note that a Shape object can be placed on another Shape object by giving the parent object as the second argument to its constructor, and its coordinates will be relative to its parent object rather than the origin. In Shape class and Meter class we define several events and bind handlers to construct the dependency graphs as shown in Fig. 15 and Fig. 16, respectively. Now we give two Shape objects s1 and s2, to the Meter object m, where the parent of s2 is s3. Whenever the name or the coordinates of s1, s2, and s3 are modified, m will recalculate the distance between s1 and s2 and print out a string. The RDJ version of this example is shown in Fig. 18 (b), where all events and explicit bindings for event composition are replaced with the line of enabling the automation in this.print (Line 39). In DJ version twelve bindings (with += operator) are used, while in RDJ version only one binding is needed. Figure 17 shows how the RDJ compiler infers all the involved events through the references to these objects. This example shows that the braces operator can not only infer the involved events located on different objects, but can also greatly reduce the number of bindings for event composition.
4.6 The Limitations of RDJ

In this subsection we discuss the limitations of RDJ, which are not the limitations of the extension we proposed in Section 3 but might occur when implementing it on an OO language such as DJ, especially when we are faced with making design decisions.

Only fields can be translated to signals. In Section 3, we described a signal as a field or a variable that is written in a handler with the automation, but in RDJ we ignore variables (i.e., local variables and parameters). There is a limitation that a local variable in RDJ cannot work as signals. As shown in Fig. 19 (a), the assignment of the local variable \( a \) (Line 5) cannot be translated to a handler with the automation and passed to other method slots. The reason is that the braces operator is used for method slots, while a method slot in RDJ is an object’s property and cannot be declared inside a method slot body like an inner method or closure. Only fields are used to infer involved events. In RDJ, local variables and parameters are not involved in the inference. For example, using the braces operator on the method slot \( \text{updateA} \) in Fig. 19 (b) cannot select involved events inside \( b \) and \( c \). Since RDJ is a Java-based language, where the arguments are evaluated with pass-by-value strategy, a special class [18], [30] or a first-class method slot might be necessary to wrap the expressions passed to \( \text{updateA} \).

The usage of the braces operator is not declarative. In RDJ, the automation of a handler is dynamically enabled by a statement in a method slot body rather than statically declared in its owner class. This means the concern of enabling the automation might be tangled with other concerns. This issue can be considered as a consequence of allowing dynamically inferring events at runtime. The braces operator can be regarded as a kind of event detector, but it is not separated from other code [5]. For example, if the automation of \( \text{updateA} \) in Fig. 6 must be enabled for all object instances of \( \text{Sheet} \), programmers have to add the binding state-ment to the constructor of \( \text{Sheet} \) object. However, this statement is used to enable the automation in \( \text{updateA} \) but not directly related to the construction of a \( \text{Sheet} \) object. A new modifier for method

---

```java
public class Shape {
    private Shape parent; private String name;

    public Shape(Shape parent) {
        this.parent = parent;
    }

    public String getName() {
        return parent.getName() + " & " + name;
    }

    public void setName(String s) {
        name = s;
    }

    public void moved(int dx) {
        x += dx;
    }

    public int get() {
        if (parent == null) return x;
        return parent.get() + x;
    }

    public void moved(int y) {
        y += y;
    }

    public int getY() {
        if (parent == null) return y;
        return parent.getY() + y;
    }

    public class Meter {
        public void print() {
            System.out.println("The distance between " + from.getName() + " and " + to.getName() + " is " + distance().toString());
        }
    }
}
```

(a)

```java
public class Sheet {
    private int b, c, d;

    public Sheet() {
        int c = this.d = this.c = this.a = b = c;
    }

    public void updateA() {
        int c = this.a + this.c + this.d;
    }

    public void notify() {
        ... updateA(this.c = this.a); updateA();
    }
}
```

(b)

---

The usage of the braces operator is not declarative. In RDJ, the automation of a handler is dynamically enabled by a statement in a method slot body rather than statically declared in its owner class. This means the concern of enabling the automation might be tangled with other concerns. This issue can be considered as a consequence of allowing dynamically inferring events at runtime. The braces operator can be regarded as a kind of event detector, but it is not separated from other code [5]. For example, if the automation of `updateA` in Fig. 6 must be enabled for all object instances of `Sheet`, programmers have to add the binding statement to the constructor of `Sheet`. However, this statement is used to enable the automation in `updateA` but not directly related to the construction of a `Sheet` object. A new modifier for method
slots, for example reactive, could be introduced as syntax sugar
to make it declarative. However, this design limits the way to give
arguments and only the involved events in the default closure can
be inferred.

Difficult to filter only the events for value change. As we men-
tioned in Section 3 an expanded system should be able to infer
the events for value change or more broadly write access since the
former is a subset of the latter. However, in RDJ there is no easy
way to filter out the events occurring when writing fields with the
same values; it is hard to get only the events for value change. A
possible solution is to check the values in the fields at the begin-
nning of the handler or insert another handler to check whether the
handler should be called or not, but programmers have to manage
the history of the values for fields. The history-based language
features for AOP such as Ref. [2] or CEP (Complex Event Pro-
cessing) [9], [13], [17] might be better solutions to resolve this
issue in such an expanded event system.

Propagation loop cannot be totally avoided. When a field is not
only read but also written in a handler, enabling the automation
in this handler might cause a propagation loop. In current design
of RDJ, the compiler can avoid such a case by excluding a han-
der from the involved events for itself. For example, even if the
updateA in Fig. 6 is modified to:
public void updateA() { a += b + c; }

enabling the automation does not bind updateA to itself. In this
case the intention is clear since an update is not expected to trig-
ger itself again. However, if two handlers are set to trigger each
other, it is hard to know the programmers’ intention. For exam-
ple, suppose that we have one more method slot named updateB
in the class Sheet in Fig. 6:
public void updateB() { b = a + c; }

and the automation in updateA and updateB are both enabled.
When updateA is called, an endless loop will happen. In modern
spreadsheet programs such circular reference can be detected and
a warning will be shown. However, in programming it is hard to
detect such propagation loops until runtime. A static
analysis of the dependencies between fields should help detect
endless loops, but a loop might only appear depending on cer-
tain conditions at runtime. This issue also happens in FRP lan-
guages, and might remind readers of the advice loops in AspectJ.

What RDJ currently supports is similar to applying the concept of
lflow(adviceexecution) in AspectJ to handlers. A more generic
solution to this issue might be introducing execution levels [32]
to reactive programming.

5. Evaluation

Since in Section 4.3 the usability of such an expanded event
system has been shown by rewriting the motivating example, in
this section we further translate complex use cases of signals to
evaluate its capability. A qualitative analysis of comparing it with
the original event system and the predicate pointcuts in AOP is
included as well. Before that, we go through the RDJ compiler
implementation to explain how it works, and measure the impact
of introducing the automation to DJ.

5.1 The Implementation of RDJ

The RDJ compiler transforms RDJ code into plain Java code and
then compiles into Java bytecode as what the DJ compiler
does. In the DJ compiler, a method slot is implemented by an
array of closures with the methods for calling the closures in the
array, adding a closure to the array, and removing closures from
the array. For example, the following binding:
this.updateA += o.update;
is transformed into the following statement in Java:
this.updateA$after(c);

where c is a closure calling o.update. It means appending the
closure c to the end of the array for the method slot this.updateA.
The RDJ compiler implementation is based on the DJ compiler im-
plementation and the semantics shown in Fig. 7, but the selection
of method slots is split into compile-time and runtime in order to
improve the runtime performance. At compile-time for every
method slot the compiler collects the writer method slots accord-
ing to its default closure and generates a helper method that binds
a given closure to all the writer method slots. Furthermore, using
the braces operator on a method slot is transformed to calling the
helper method for that method slot. Then at runtime the corre-
sponding helper method is called according to the actual type of
the owner object to collect method slots and bind the given clos-
cure to them. Note that a method slot is owned by every instance.
It is not shared among the instances of the same class.

Taking the example of Fig. 10 (c), the following helper method
is generated for using the braces operator with the += operator on
the method slot updateA:
public void updateA$helper$after(Closure closure) {
  if(a != null) a.setValue$after(closure);
  if(b != null) b.getValue$after(closure);
  if(c != null) c.getValue$after(closure);
}

No method slots are directly selected in this helper method since
the fields a, b, and c, which are read by updateA, are not written
by any method slot in the owner object. However, the method
slots a.setValue, b.getValue, and c.getValue are called in up-
dateA, so that the given closure is passed to the helper meth-
ods for them recursively. The body of the helper method set-
Value$after is empty since no fields are read in setValue
of IntCell as shown in Fig.11. On the other hand, the helper
method getValue$after looks like this:
public void getValue$after(Closure c) {
  setValue$after(c);
}

The method setValue$after is a method for appending a closure
to the array of the method slot setValue, which has been gen-
erated in the DJ compiler. Note that the helper methods set-
Value$after and getValue$after are declared in IntCell. After generating the helper methods the compiler transforms
the following binding:
{._} += this.updateA();

into the call to the helper method:
this.updateA$helper$after(c);

The prototype compiler of RDJ is built on top of DJ and JastAddJ [11],
and available from https://github.com/cog-tokyo/rdj

© 2016 Information Processing Society of Japan
A more powerful implementation is possible. As we explained above the braces operator in the current version of the RDJ compiler only takes the default closure into account due to the performance concern. In this implementation, only the default closure is inferred for a method slot given to the braces operator. It is possible to infer all the closures added to the given method slot at runtime if we implement reflection API for method slots. However, in such an implementation all the selection must be done at runtime and the performance will not be good. Another solution is to generate helper methods for all the method slots that might be added to the given method slot. For example, if we use the braces operator on the method slot this.updateA as follows:

```java
this.updateA += logger.debug;
```

and then append a closure calling another method slot logger.debug to this.updateA somewhere in the program:

```java
this.updateA += logger.debug;
```

It might be possible to find out all the method slots like logger.debug, generate helper methods for them, and prepare a switch for calling every helper method in the event selection for {this.updateA}. If the statement for adding logger.debug is executed, switching on the call to the helper method for logger.debug. Then the events selected by logger.debug can also be selected when selecting the events for this.updateA. In other words, not only the default closure but also the closures calling other method slots in the given method slot are inferred for selecting the events. As a trade-off the whole program must be compiled together to find out all the method slots that might be added to the method slot given to the braces operator; separate compilation is not available. Such an implementation is more powerful since after using the assignment to add the closure calling a method slot, we do not have to use the braces operator for that method slot again as follows:

```java
this.updateA += logger.debug;
```

In the latter one we do not have to apply the braces operator again for this.refresh. Some readers might notice that investigating all closures in a method slot can further improve the modularity when using the braces operators along with event-driven programming or AOP, though it might be a little slower and disable the separate compilation. In this paper we only implement the simple one that infers the default closure for showing such an extension to method slots is possible. Implementing such a powerful one that infers all the closures in the method slot is included in our future work.

### 5.2 Preliminary Microbenchmarks

To measure the impact of enabling the automation in method slots we can consider only the inference overheads since a binding using the braces operator is eventually boiled down to the bindings that are manually enumerated in DJ. For example, the binding in Line 16 of Fig. 10(c) is boiled down to the following

```java
{logger.debug} += this.refresh();
```

When the object held by this.updateA is changed to another object the handler is still bound to the given object rather than the new one. Usage is not recommended since it is hard to trace. For example, suppose that there is a setter method slot setB for b in Fig. 10(c) and at first the object held by the field b is Ob. Then the method slots selected by the braces operator in Line 16 of Fig. 10(c) include the setB on p and the setValue on Ob. When the object held by b is changed to another object Ob, by calling the setB on p, the handler updateA() will be executed and thus has a chance to check if the object held by b is the same. However, the handler is still bound to the setB on Ob rather than the one on Ob. A possible work-around is to prepare two additional handlers: one for unbinding the handler updateA() from Ob before setB is called, and one for re-binding the handler updateA() to Ob after setB is called, either by the compiler or by programmers themselves.

### Fig. 20 How the compiler transforms a binding with the braces operator.

where c is an anonymous class which extends the Closure interface of DJ and calls this.updateA(). Therefore, at runtime the helper method updateAHelperLater is called for collecting all method slots and binding the closure c to them. Note that for all subclasses of PlusSheet that override setValue and setValue, the compiler generates helper methods for them individually, so that the corresponding helper method can be called on the object according to the actual type of the object; the dynamic method dispatch in Java is preserved. Note that in order to simplify the implementation the RDJ compiler generates helper methods for every subclass of the apparent type of the given object. The procedure generate_helper adds (suppose that += operator is used) the given closure to collected method slots immediately rather than returns the set of collected method slots. The proper helper method can be selected by the dynamic method dispatch in OOP. As a consequence, if a method slot is selected in both different helper methods, it is difficult to avoid binding to the same method slot twice.

A limitation of this implementation is the changes of objects. The object referred to by a field might be changed after the binding is performed. In that case, the handler is still bound to the previous object rather than the new one. Such usage is not recommended since it is hard to trace. For example, suppose that there is a setter method slot setB for b in Fig. 10(c) and at first the object held by the field b is Ob. Then the method slots selected by the braces operator in Line 16 of Fig. 10(c) include the setB on p and the setValue on Ob. When the object held by b is changed to another object Ob, by calling the setB on p, the handler updateA() will be executed and thus has a chance to check if the object held by b is the same. However, the handler is still bound to the setB on Ob rather than the one on Ob. A possible work-around is to prepare two additional handlers: one for unbinding the handler updateA() from Ob before setB is called, and one for re-binding the handler updateA() to Ob after setB is called, either by the compiler or by programmers themselves.

### Fig. 20 How the compiler transforms a binding with the braces operator.

where c is an anonymous class which extends the Closure interface of DJ and calls this.updateA(). Therefore, at runtime the helper method updateAHelperLater is called for collecting all method slots and binding the closure c to them. Note that for all subclasses of PlusSheet that override setValue and setValue, the compiler generates helper methods for them individually, so that the corresponding helper method can be called on the object according to the actual type of the object; the dynamic method dispatch in Java is preserved. Note that in order to simplify the implementation the RDJ compiler generates helper methods for every subclass of the apparent type of the given object. The procedure generate_helper adds (suppose that += operator is used) the given closure to collected method slots immediately rather than returns the set of collected method slots. The proper helper method can be selected by the dynamic method dispatch in OOP. As a consequence, if a method slot is selected in both different helper methods, it is difficult to avoid binding to the same method slot twice.
bindings by inference:

```
b.setValue += this.updateA();
c.setValue += this.updateA();
```

The cost of these bindings in RDJ is the same as in DJ except that a closure must be dynamically created to hold the arguments passed to the handler. Since the arguments passed to the handler might be non-literal and thus need to be dynamically evaluated, it is not able to create the closure in advance. A way to avoid the additional overheads is to prohibit programmers from giving non-literal arguments to the handler. In the following preliminary microbenchmarks we compare the two implementations to separate the overheads of creating the handler closure from the inference overheads: RDJ1 always creates the handler closure while RDJ2 does not, though in both cases no arguments are given. The three metrics we use to measure the impact of the inference are shown in Fig. 21. We bind and then unbind a method slot to the involved events of another method slot one million times to get the average. The results of running the microbenchmarks on OpenJDK 1.7.0.65 with Intel Core i7 2.67GHz 4 cores and 8GB memory are shown in Fig. 22. Note that the DJ compiler was version 0.2 taken from DominoJ project web site, and the result shown in each graph includes the time of performing a binding and an unbinding. Figure 22 (a) shows that binding and then unbinding the same number of method slots in RDJ1 always takes three times as long as in DJ. On the other hand, the performance of RDJ2 is very close to DJ; the inference overheads are negligible and do not grow with the breadth. The difference can be considered as the overheads of creating the handler closure. In Fig. 22 (b) we inserted a number of closures that call a method slot whose default closure carries no reference to measure the overheads of iterating a closure. The result is linear and shows that the average of iterating a closure carrying nothing is about 30ns in both RDJ1 and RDJ2. In Fig. 22 (c) the only method slot is selected through a number of method slot calls. Similarly, the result is also linear in both implementations, and shows the overheads of traversing an object are about 217ns. To benefit from both the implementations the current version of RDJ compiler allows arbitrarily giving arguments, but automatically optimizes when no arguments need to be evaluated; the impact is not significant. Furthermore, the inference overheads only appear in binding or unbinding a handler.

5.3 Translation Examples

We have shown typical translation examples of signals, but it might be interesting to check if it is possible to translate complex use cases such as the examples given in the REScala paper [30]. REScala is a hybrid event system that supports both events and signals, and provides conversion API as primitives for complex usage of signals. Translating such complex use cases in REScala to RDJ not only evaluates the feasibility of the extension we proposed in Section 3 but also gives a good understanding of such kinds of primitives for signals.

The signals and primitives given in REScala can be lowered to fields, method slots, and bindings in RDJ according to the following translation rules. A `Var` is translated to a field with the same type. A `Signal` is translated to a field and a method slot (handler), and the automation for that method slot should be enabled if any `Var` or `Signal` is used in the signal assignment. An `evt` (event) is translated to a method slot that is bound to all the events given in the event declaration by using operator `+=`. In general, one line for declaring a `Signal` in REScala will be translated to three lines in RDJ, and one line for declaring an `evt` in REScala will be translated to at least two lines in DJ and RDJ. For a `Signal`, RDJ needs one line for declaring a field, one line for declaring a method slot for updating the value in the field, and one line for enabling the automation in the method slot. For an `evt` DJ and RDJ need one line for declaring a method slot and at least one line for
binding; if there are more than one event given at the right-hand side of the event declaration, more bindings are needed for event composition.

The first example we are going to check is the direct conversion from a signal to an event as shown in Fig. 23 (a). Note that the signal canLive in Lines 3–5 of (a) is translated to the field canLive, the method slot updateCanLive, and the binding for enabling the automation in updateCanLive as shown in Lines 3–7,10 of (b); the event shouldDie in Line 6 of (a) is translated to the method slot shouldDie and the bindings for it as shown in Lines 8,11–12 of (b). The function changed used in Line 6 of (a) is a primitive used to convert the signal canLive to an event, which is exactly the call to the handler updateCanLive in (b) since changed means the event when the given signal is updated. Such a REScala code can be translated to RDJ code step by step, though the lexical representation tends to be a little longer. The next example is shown in Fig. 24 (a), where the function hold is a primitive for converting an event to a signal. Here how to translate click, circle, and lastClickOnCircle (Lines 1–3) is the same as the steps in the previous example. However, click(), circle() and lastClickOnCircle (Lines 1–3) are different (Lines 4–12) is the same as the steps in the previous example. We need an extra field holdClick and an extra method slot updateHoldClick for updating holdClick as shown in Lines 3–4 of Fig. 24 (b). Furthermore, we need an extra statement that binds updateHoldClick to the event click (Line 11) since function hold means updating the value of the anonymous signal when the given event occurs. The third example is function fold as shown in Fig. 25 (a), which is a primitive used to perform stateful conversion of events to signals. The function fold in Line 2 takes an initial value 0 and a function that is used to evaluate the value of the signal nClick when the event click occurs. In this case, the function taken by fold is exactly the handler for updating the field nClick—the method slot updateNClick in Line 3 of Fig. 25 (b). Furthermore, the event at the left-hand side of fold, click, is the event that updateNClick must be bound to as shown in Line 6 of (b). The last example we want to discuss here is function snapshot, which is a primitive introduced to integrate signals into event-driven computations. In Line 3 of Fig. 26(a) function snapshot returns a signal to lastClick whose value is updated according to the value of another signal position when the given event click occurs. In RDJ, it means assigning the value of position to lastClick when the event clicked occurs. Thus, what we need to do is to set the value of position to lastClick in the body of updateLastClick as shown in Line 5 of (b) and bind updateLastClick to the event clicked (Line 9 of (b)).

To conclude, these complex use cases of signals in REScala can be translated to RDJ code step by step according to the translation rules we explained above. It might give a clear understanding of such primitives for signals since the RDJ code shows how to describe them in an event system where the signal notation is not given. It also shows the automation we proposed is sufficient even in these complex use cases and that the primitives for signals can be simply translated to handler bindings. Although the number of lines of code in RDJ is increased, it can be expected and does not explode; it is generally two or three times as long as in REScala. Note that the drawback that the binding in RDJ is not declarative is not a limitation of the extension proposed in Section 3 but the limitation of DJ. As a consequence of allowing the addition or removal of closures in a method slot at runtime, the binding must be a statement within the body of a method slot rather than a declaration.

5.4 Comparing with the Original Event System

Using the braces operator to select events can avoid binding to all involved events explicitly. The merits include writing the dependency among events only once, improving the encapsulation, and simplifying the event propagation. On the other hand, if only parts of events need to be selected, it is not proper to use the braces operator. In such a case, binding a handler to the events selected by the braces operator will cause unnecessary event propagation. In this subsection we compare with DJ since it is the original event system that RDJ is based on. Comparing RDJ with other event systems such as EScala might be possible, but it is not easy to see the consequence of using the brace operator due to the difference between them and DJ.
Write the dependency once. The dependency among events in a typical event mechanism is usually described twice. As shown in Fig. 10 (b), once for calculating values (Line 22) and once for making it automatic (Line 16–17). In other words, for each dependency programmers have to maintain it in two places. In RDJ the bindings for all dependencies in a method slot are reduced to a binding to the higher-level event, so that the number of lines of source code can be smaller. The maintainability can also be improved since we do not have to ensure the consistency between the binding and the calculation. The binding to a method slot with the braces operator is always the same no matter how the source code for the calculation is modified.

Improve the encapsulation. The inference mechanism in RDJ improves the encapsulation of a class. Some readers might notice that in Fig. 10 (c) it is also possible to write the following statement in the body of main rather than the binding in Line 16:  
```java
{ } += p.updateA();
```

The braces operator improves the encapsulation of a class since using the braces operator can bind a handler to the events inside the object without directly touching the fields in that object. The fields in the class can be observed through getter methods and the observers can be unaware of the implementation of the class. In this example the fields of PlusSheet, i.e. a, b and c, and the field in IntCell, i.e. value, do not have to be exposed by declaring as public while they still can be observed outside. In other words, the fields do not have to be public and the clients can be oblivious to the implementation details of PlusSheet.

Simplify the event propagation. It is also possible to simplify the event propagation since the handler is directly bound to the selected method slots through the fields behind the higher-level event. **Figure 27** shows that parts of the dependency graph of Fig. 18 (a). When from.setX or from.moveX is executed, from.move and this.changed will be triggered in order and then cause the execution of this.print. However, with the automation the propagation is simpler. **Figure 27** shows the bindings eventually performed by the RDJ compiler. In other words, the event propagation in a typical event mechanism is reduced to one step.

Unnecessary event propagation. In some special cases, using the braces operator to select events for binding might cause unnecessary event propagation. For example, suppose that in the sheet example we only want to update b when b is changed:

```java
a = b + c;
```

However, using the braces operator on updateA in Line 18–20 of Fig. 1 (c) will select the events for both b and c. On the other hand, in the DJ version it is obvious that programmers can configure them since the bindings are manually enumerated (Line 16–18 of Fig. 1 (b)). Similarly, in the shape and meter example (Fig. 16), it is not able to select only events for x but not y, or to select event moveX but not event setX. This consequence is the same in FRP languages since all variables are signals. As we discussed in the previous subsection, REScala can satisfy this need by providing both Var and Signal: Var is for normal variables and Signal is for signals. In RDJ, using the braces operator can be considered as converting all fields read in a method slot to signals. Therefore, if this is not desirable, programmers have to properly separate those field accesses from the method slot taken by the braces operator or manually enumerating the bindings as in DJ. That is why we propose expanding the original event system rather than a new system; both the implicit style in FRP and the explicit style in the original event system are supported.

5.5 Comparing with the Predicate Pointcuts in AOP

The inference of the braces operator in RDJ can be regarded as some sort of predicate. However, it is different from the predicate pointcuts in AOP since the braces operator takes a handler body as a hint rather than describing which events should be selected. We can regard event composition as merging multiple sources to a sink; the sources are lower-level events and the sink is the higher-level event. In that sense, the predicate pointcuts in AOP are used to describe the sources while the braces operator in RDJ takes a sink as the hint to infer the sources. For example, a statement in reactive programming that prints the values of two signals x and y (we simply assume that they are variables in a Point object due to the difference between functional-reactive languages and OO languages) can be rewritten using AspectJ as shown in **Fig. 28**.

The write access to the fields x and y are selected by set pointcut. Then predicate pointcuts such as target and within can be used to further select the join points. The RDJ version is shown in **Fig. 29**. We can move advice body to a method slot and enable its automation by using the braces operator. The RDJ version is very implicit while needs a method slot body for inferring which events to select.

As an extension to AspectJ, there are also research activities devoted to pointcuts for doing similar things as the braces operator. Dataflow pointcuts [20] are proposed to trap join points according to dataflow. It can be used with other kinds of pointcuts to address the origins of values between join points. However, the implementation for Java-like languages is not trivial. A successive work [1] showed a formal and practical framework that stati-
cally propagates dataflow tags to trace data dependencies. On the other hand, RDJ traces method call chains rather than low-level dataflows, so the implementation is simpler and feasible in Java.

6. Related Work

In the world of events, more and more techniques are introduced to make events more powerful and expressive. For example, Ptolemy [28] supports quantification and type for events. EventJava [14] considers event correlation, and EScala [15] discusses implicit events found in AOP. However, such advanced event systems still lack the implicit style in reactive programming. Other research activities such as Frappé [8] and SuperGlue [21] can be regarded as examples of using events and signals together since they use signals in specific components. The signals are considered as objects’ properties. This approach allows using signals in a limited scope at language level for a specific usage.

Other examples of using events along with signals include the library approach such as Flapjax [22]. This approach makes it easy to use signals in existing languages since signals are represented by existing elements in the languages. There are also several libraries developed for the reactive support in collections. The incremental list in Scala.React [18] is a functional-reactive data structure for Scala [26], which can automatically propagate incremental change. Although in such libraries signals might be implemented through events underneath, the involved events cannot be automatically inferred. Programmers need to manually specify which fields or variables are signals in order to ask the underneath to create handler bindings properly.

Unlike the research activities mentioned above that introduce the signals in FRP to make event propagation implicit, ReactiveX [29] enhances the Observer pattern with regard to concurrency and error handling by proposing an API for asynchronous programming and is implemented as a library in several platforms and languages: Rx.NET [23] is a library on .NET for writing asynchronous and event-based programs with LINQ, and RxJava [24] is the implementation for Java VM. Although those extensions to event systems are named with the term reactive, what they provide is asynchronous handling rather than implicit propagation. ReactiveX helps programmers to react to the events for data items in a collection without blocking, and prevent from writing tangled callbacks. On the other hand, with our proposal programmers can avoid manually enumerating events and bindings, and thus make event propagation as implicit as using signals in FRP languages. In a certain sense, ReactiveX is a broad enhancement on event handling, while our proposal is a deep enhancement with regard to event propagation.

7. Conclusion

We analyzed the essentials of reactive programming from the viewpoint of event-driven programming, and pointed out the need of implicit binding in existing event systems. To satisfy this need we proposed an extension that enables the automation of handler bindings to support the implicit style in reactive programming. Then we gave an implementation to show the feasibility of such an extension. Although the implementation is a prototype that has several limitations, it showed the advantages over the event system without the automation. The design decisions and limitations happening when implementing the extension on OO languages are discussed as well.

References

[1] Alhadi, D., Bouzkoutta, A., Belbidia, N., Debbabi, M. and Bhattacharyya, P.: The Dataflow Pointcut: A Formal and Prac-

ical Framework, Proc. 8th ACM International Conference on Aspect-

oriented Software Development, AOSD ’09, pp.15–26, ACM (online),

DOI: 10.1145/1509239.1509244 (2009).

[2] Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. and Tibble, J.: Adding Trace Matching with Free Variables to AspectJ, OOPSLA ’05, pp.345–364, ACM (2005).

[3] Benveniste, A., Guermic, P.L. and Jacquemot, C.: Synchronous pro-
gerating with events and relations: The SIGNAL language and its semantics, Science of Computer Programming, Vol.16, No.2, pp.103–149 (1991).

[4] Berzy, G., Gonthier, G., Gonthier, A.B.G. and Lalitte, P.S.: The Esterel Synchronous Programming Language: Design, Semantics, Implementation (1992).

[5] Bockisch, C., Malakuti, S., Aksit, M. and Katz, S.: Making Aspects Modular event-driven object interactions in Scala, AOSD’11, pp.25–30, ACM (2011).

[6] Costi, P., Pilaud, D., Halbwachs, N. and Plaice, J.A.: LUSTRE: A declarative language for real-time programming, POPL’87, pp.178–188, ACM (1987).

[7] Cooper, G.H. and Krishnamurthi, S.: Embedding dynamic dataflow in a call-by-value language, ESOP’06, pp.294–308 (2006).

[8] Courtenay, A.: Frappé: Functional Reactive Programming in Java, PADL’01, Springer-Verlag, pp.29–44 (2001).

[9] Cupola, G. and Margara, A.: Processing flows of information: From data stream to complex event processing, ACM Comput. Surv., Vol.44, No.3, pp.15:1–15:62 (2012).

[10] Czaplicki, E. and Chong, S.: Asynchronous functional reactive pro-
gerating for GUIs, PLDI’13, pp.411–422, ACM (2013).

[11] Ekman, T. and Hedín, G.: The JastAdd Extensible Java Compiler, OOPSLA’07, pp.1–18, ACM (2007).

[12] Elliott, C. and Hadak, P.: Functional reactive animation, ICFP’97, pp.263–273, ACM (1997).

[13] Elzio, O. and Niblett, P.: Event Processing in Action, Manning Pub-
lcations Co., 1st edition (2010).

[14] Eugster, P.T. and Jayaram, K.R.: EventJava: An Extension of Java for Event Correlation, ECOOP’09, pp.570–594 (2009).

[15] Gasum, V., Satahni, L., Mezini, M., Núñez, A. and Noyé, J.: EScala: Modular event-driven object interactions in Scala, AOSD’11, pp.227–

240, ACM (2011).

[16] LabVIEW System Design Software, available from (http://www.ni.com/labview/).

[17] Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems, Addison-Wesley Longman Publishing Co., Inc. (2001).

[18] Maier, I. and Odersky, M.: Higher-Order Reactive Programming with Incremental Lists, ECOOP’13, pp.707–731, Springer-Verlag (2013).

[19] Masuhrara, H., Endoh, Y. and Yonezawa, A.: A fine-grained join point model for more reusable aspects, APLAS’06, pp.131–147 (2006).

[20] Masuhrara, H. and Kawauchi, K.: Dataflow Pointcut in Aspect-

Oriented Programming, APLAS, Ohtori, A. (Ed.), Lecture Notes in Computer Science, Vol 2995, pp.105–121, Springer (2003).

[21] McDirmid, S. and Hsieh, W.C.: SuperGlue: Component program-
ing with object-oriented signals, ECOOP’06, pp.206–229, Springer-

Verlag (2006).

[22] Meyrovich, L.A., Gaha, A., Biskin, I., Cooper, G.H., Greenberg, M., Bromfield, A. and Krishnamurthi, S.: Flapjax: A programming language for Ajax applications, OOPSLA’09, pp.1–20, ACM (2009).

[23] Microsoft Corporation: Reactive Extensions for .NET, available from (http://msdn.microsoft.com/en-us/data/577609).

[24] Netflix, Inc.: Reactive Extensions for the JVM, available from (https://github.com/ReactiveX/RxJava).

[25] Nilsson, H., Courtenay, A. and Peterson, J.: Functional reactive pro-
gerating, continued, Haskell’92, pp.51–64, ACM (2002).

[26] Odersky, M., Spoon, L. and Venners, B.: Programming in Scala: A Comprehensive Step-by-step Guide, Artima Incorporation, USA, 1st edition (2008).

[27] Ohshima, Y., Munzner, A., Frauenberg, B. and Kaehler, T.: KScript and KSWorld: A Time-aware and Mostly Declarative Language and Interactive GUI Framework, Onward’13, pp.117–134, ACM (2013).
[28] Rajan, H. and Leavens, G.T.: Ptolemy: A Language with Quantified, Typed Events, *ECOOP*’08, pp.155–179 (2008).

[29] ReactiveX: An API for asynchronous programming with observable streams, available from ⟨http://reactivex.io⟩.

[30] Salvaneschi, G., Hintz, G. and Mezini, M.: REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications, *Modularity*’14, ACM Press (2014).

[31] Salvaneschi, G. and Mezini, M.: Reactive behavior in object-oriented applications: An analysis and a research roadmap, *AOSD*’13, pp.37–48, ACM (2013).

[32] Tanter, É., Figueroa, I. and Tabareau, N.: Execution Levels for Aspect-oriented Programming: Design, Semantics, Implementations and Applications, *Sci. Comput. Program.*, Vol.80, pp.311–342 (2014).

[33] Wadge, W.W. and Ashcroft, E.A.: *LUCID, The dataflow programming language*, Academic Press Professional, Inc. (1985).

[34] Wan, Z. and Hudak, P.: Functional reactive programming from first principles, *PLDI’00*, pp.242–252, ACM (2000).

[35] Wan, Z., Taha, W. and Hudak, P.: Real-time FRP, *ICFP’01*, pp.146–156, ACM (2001).

[36] Wan, Z., Taha, W. and Hudak, P.: Event-Driven FRP, *PADL’02*, pp.155–172, Springer-Verlag (2002).

[37] Zhuang, Y. and Chiba, S.: Method slots: Supporting methods, events, and advices by a single language construct, *AOSD*’13, pp.197–208, ACM (2013).

**YungYu Zhuang** received his Ph.D. degree in information science and technology from the University of Tokyo in 2014. Currently he serves as Project Research Associate in the University of Tokyo, and will be Assistant Professor at National Central University from August 2016. His research interests include programming language design and software engineering.

**Shigeru Chiba** is Professor at the University of Tokyo. Before starting his current position, he worked at Tokyo Institute of Technology and University of Tsukuba. He received his Ph.D. in science in 1996 from the University of Tokyo. His research interests are in programming language design and implementation.