2015 IEEE 39th Annual International Computers, Software & Applications Conference

Enabling the Automation of Handler Bindings in Event-Driven Programming

YungYu Zhuang and Shigeru Chiba
The University of Tokyo
Tokyo, Japan
{yungyu,chiba}@acm.org

***Abstract*—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*.**

Keywords*—*event-driven programming; reactive programming; signal; behavior;

I. 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) [11], [30], [31], [22], [32], [6] successfully introduces signals into functional programming. The concept of signals might come from data-flow languages [29], [5], [3], [2], [15], where variables are described as continuous data streams rather than states at a specific time. Such variables are signals, which can participate in 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 [24], [9]. 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 are similar to signals but different concepts. 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 [20], [17], [26]. 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 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 can 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 result 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 [27]. 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 research activities mentioned above show the need of using both the two style together. The contributions of this paper can be summarized as follows. First, we compare event-driven programming with reactive programming to point out the need of implicit binding 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 the one 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 programming might give a better understanding of events and signals.

0730-3157/15 $31.00 © 2015 IEEE
DOI 10.1109/COMPSAC.2015.48

IEEE computer society

A1	=B1+C1		
	A	B	C
1	3	2	1
2			

Fig. 1. A sheet in a spreadsheet program

II. MOTIVATING EXAMPLE

Programs written in reactive programming can also be implemented 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 programming 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. Fig. 1 shows a sheet in a spreadsheet program, where B1 and C1 are given constant values: 2 and 1 respectively, 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 implemented by FRP [11] languages. For example, Fig. 2 shows an implementation in Flapjax [20], which is a JavaScript-based language 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 assignment 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 programming languages is only effective just after the assignment is executed, and the value at the left-hand side might not be the same as the one at right-hand side later until the assignment is executed 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 handler to execute the assignment again. Here we use EScala [14], an event system that provides events, handlers, and bindings based on Scala [23], to write the sheet example. EScala allows to declare a special kind of field named event to

```
1  <body onload="loader()">
2    <table width=200>
3      <tr><th>A1</th>
4          <th>B1</th>
5          <th>C1</th>
6      </tr>
7      <tr><th><input id="A1" size=2 value="0" /></th>
8          <th><input id="B1" size=2 value="2" /></th>
9          <th><input id="C1" size=2 value="1" /></th>
10     </tr>
11   </table>
12 </body>
13
14 <script type="text/flapjax">
15 function loader() {
16   var b = extractValueB("B1");
17   var c = extractValueB("C1");
18   var a = b + c;
19   insertValueB(a, "A1", "value");
20 }
21 </script>
```

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(nb: Int) { b = nb; }
6    def setC(nc: Int) { c = nc; }
7
8    evt eb[Unit] = afterExec(setB)
9    evt ec[Unit] = afterExec(setC)
10   def ha() { a = b + c; }
11   eb += ha;
12   ec += ha;
13 }
```

Fig. 3. Using EScala to implement the sheet example

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 Sec. III. 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[1]. 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 the consistency between the bindings and the handler body.

[1] 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:

```
evt ebc[Unit] = afterExec(setB) || afterExec(setC)
```

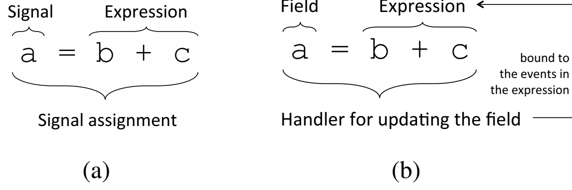

Fig. 4. The signal assignment in the sheet example

The observation that both the two 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.

III. 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.

A. The essentials of reactive programming

The essentials of reactive programming are *signals* and *signal assignments*. Fig. 4(a) shows the *signal assignment* in the sheet example mentioned in Sec. II, where a is a *signal* and b + c is the expression assigned to the *signal*. We can give the description of *signals* and *signal assignments* as follows:

- A *signal* (i.e. behavior) is a time-varying field or variable, the value of which is implicitly reevaluated when any of the *signals* involved in its reevaluation varies. Then its value change also implicitly causes all the reevaluation that it is involved in.
- A *signal assignment* is composed of a *signal* and the expression assigned to the *signal*. The *signal* expression describes how to reevaluate the value of the *signal*. It also implies that which *signals* are involved in this reevaluation and the expression has to be reevaluated for setting the value of this *signal* when any of the involved *signals* varies. Here the involved *signals* are the *signals* that are read in the expression.

In the sheet example a is the *signal*, the value of which will be implicitly reevaluated when any of b and c varies according to the expression in the *signal assignment* "a = b + c".

B. In terms of events, handlers, and bindings

Although event-driven programming and reactive programming are different paradigms, what they can do are very similar. They both can be used to implement reactive programs such as the sheet example we mentioned in Sec. II. We can also translate the essentials of reactive programming, *signals* and *signal assignments*, into a description in terms of events, handlers, and bindings as listed below:

- A *signal* is translated to a field or variable whose value will be set when any of the events involved in its reevaluation occurs.
- A *signal assignment* is translated to a handler for setting the value of the field (or variable) at the left-hand side by reevaluating the expression at the right-hand side. Furthermore, this handler is bound to all events involved in the expression at the right-hand side. Here the involved events are the value change of the fields (or variables) read in the expression. Whenever such an involved event occurs, the handler is executed to reevaluate the expression and then set the value of this field (or variable).

Fig. 4(b) shows how the *signal assignment* in the sheet example is translated in an event system. This *signal assignment* is described as a handler for executing "a = b + c" to update the value of a. Note that we can simply say the handler is bound to all the involved fields (or variables) inside itself since the field (or variable) at the left-hand side is written rather than read.

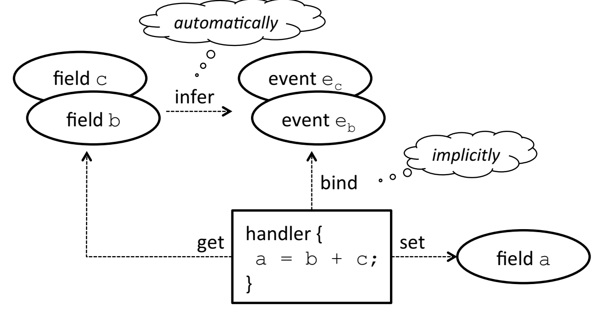

Fig. 5. The extension must be able to automatically infer the involved events and implicitly bind the handler to them

C. 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.

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.

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.

IV. A PROTOTYPE IMPLEMENTATION

In order to show the feasibility of *automation*, we expand DominoJ [33] (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 Sec. III is more generic and can be implemented in any event system.

A. 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 setX(int nx) { this.x = nx; }
```

the method-like declaration for 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 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:

```
s.setX += o.update;
```

means that creating a closure calling the method slot o.update and appending it to the end of array in s.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:

$\langle$*event*$\rangle$ $\langle$*assignment_operator*$\rangle$ $\langle$*handler*$\rangle$;

The statement using += operator we showed above is used to let the event (method slot) update on an object o be triggered after s.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.

```
1 public class Sheet {
2   private int a, b, c;
3   public void setB(int nb) { b = nb; }
4   public void setC(int nc) { c = nc; }
5   public void updateA() { a = b + c; }
6 }
```

Fig. 6. A simplified sheet example using fields

B. 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:

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

When the value of b or c is set by setB/setC, this.updateA() will be executed to update the value of a. In other words, the statement means that binding the handler this.updateA() to all the involved events inside itself as we described in Sec. III. 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 pseudocode as shown in Fig. 7, where *M* is a method slot and O_M is the owner object of *M*, *getClosuresIn* returns all the closures in a specified method slot, *findFieldsReadIn* returns all the fields read in

```
1 procedure braces_operator(O_M, M) {
2   S = new Set();
3   foreach c in getClosuresIn(M):
4     foreach f in findFieldsReadIn(c):
5       if f isOwnedBy O_M:
6         foreach m in findMethodSlotsThatWrite(f):
7           if m isOwnedBy O_M:
8             S.add(m);
9     foreach (O_N, N) in findMethodSlotsCalledIn(c):
10      if O_N isOwnedBy O_M:
11        S.add(braces_operator(O_N, N));
12  return S;
13 }
```

Fig. 7. The inference in the braces operator

```
1 public class Debug {
2   private boolean e;
3   private String f;
4   private Object g;
5   public void setE(boolean ne) { e = ne; }
6   public void setF(String nf) { f = nf; }
7   public void setG(Object ng) { g = ng; }
8   public void resetG() { g = null; }
9   public boolean getG() { return g; }
10  public void print() {
11    if(e)  System.out.println(f);
12    else  System.out.println(g);
13  }
14 }
```

Fig. 8. Another class Debug

a specified closure, *findMethodSlotsThatWrite* returns all the method slots that write the specified field in the default closure, and *findMethodSlotsCalledIn* returns all method slots that are called in a specified closure. First, all the involved fields in the method slot, which are the fields read during executing the closures in this method slot, are inferred (Lines 3–4). Then all the method slots that write any of these involved fields are regarded as the involved events and selected (Lines 6–8). Take Fig. 6 as an example. {this.updateA} infers a set of method slots that write any of the involved fields, this.b and this.c, and thus the method slots that write this.b or this.c, this.setB and this.setC, are selected. Any method slot that is called in any closure of the method slot given to the braces operator are recursively inferred (Lines 9–11) since the fields read in the called method slots are also involved in the execution of this method slot. In this example no method slots are called in updateA, so that this step is skipped. If another method slot d.print is called in updateA, where d is a field added to Sheet to hold an object instance of another class Debug as shown in Fig. 8. Then the set of involved events inferred by {updateA} can be considered as shown below:

$$\{\text{this.updateA}\}$$
$$\rightarrow (\text{this.setB}, \text{this.setC}) \cup \{\text{d.print}\}$$
$$\rightarrow (\text{this.setB}, \text{this.setC}) \cup (\text{d.setE}, \text{d.setF}, \text{d.setG}, \text{d.resetG})$$
$$\rightarrow (\text{this.setB}, \text{this.setC}, \text{d.setE}, \text{d.setF}, \text{d.setG}, \text{d.resetG})$$

Note that only the method slots that belong to the objects held in the fields in this owner object are recursively inferred (Line 10 of Fig. 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.

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 Sec. III. 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 a set of fields on the owner object as shown in Fig. 9. For example, in Fig. 8 using {this.getG} will select both this.setG and this.resetG. Note that the events selected by the braces operator are the calls to the writers. In the terminology of AOP they are the join points matched by method pointcuts but not field pointcuts. As a result, in RDJ the execution of a closure is atomic and a handler cannot be bound for being executed just before/after the field is written inside a closure. The join point model is consistent with the one adopted by DJ, which is a region-in-time model [18]. Matching an arbitrary join point inside a closure is not supported.

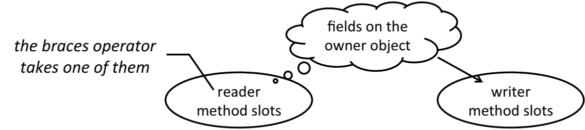

Fig. 9. The braces operator infers the writers

To make 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:

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

It is automatically translated by the RDJ compiler. In RDJ, the method slot that we want to infer the involved events (the method slot given to the braces operator) and the handler for reevaluation (the method slot call at the right-hand side) are not necessarily the same. The syntax of the braces operator allows observers to register a handler through a public method slot for getting a notification when private fields in the subject object are written.

C. Rewriting the motivating example

To show how the automation can be used we first use DJ to write the sheet example in Sec. II and then rewrite it by RDJ to compare with Flapjax. In order to make it easier to compare we assume that there were 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 extractValueB and insertValueB,

```
1 public class PlusSheet
2 extends Sheet {
3  private IntCell a1 = null;
4  private IntCell b1 = null;
5  private IntCell c1 = null;
6  public PlusSheet() {
7   super(2, 3);
8   setHeaders("A1", "B1", "C1");
9   a1 = new IntCell(0);
10  b1 = new IntCell(0);
11  c1 = new IntCell(0);
12  Behavior b = b1.extractValueB();
13  Behavior c = c1.extractValueB();
14  Behavior a = b + c;
15  a1.insertValueB(a);
16  add(a1);
17  add(b1);
18  add(c1);
19  pack();
20 }
21 public static
22 void main(String[] args) {
23  PlusSheet p = new PlusSheet();
24  p.show();
25 }
26 }
```

(a)

```
1 public class PlusSheet
2 extends Sheet {
3  private IntCell a = null;
4  private IntCell b = null;
5  private IntCell c = null;
6  public PlusSheet() {
7   super(2, 3);
8   setHeaders("A1", "B1", "C1");
9   a = new IntCell(0);
10  b = new IntCell(0);
11  c = new IntCell(0);
12  add(a);
13  add(b);
14  add(c);
15  pack();
16  b.setValue += this.changed;
17  c.setValue += this.changed;
18  this.changed += this.updateA;
19 }
20 public void changed(int v);
21 public void updateA(int v) {
22  a.setValue(b.getValue() + c.getValue());
23 }
24 public static
25 void main(String[] args) {
26  PlusSheet p = new PlusSheet();
27  p.show();
28 }
29 }
```

(b)

```
1 public class PlusSheet
2 extends Sheet {
3  private IntCell a = null;
4  private IntCell b = null;
5  private IntCell c = null;
6  public PlusSheet() {
7   super(2, 3);
8   setHeaders("A1", "B1", "C1");
9   a = new IntCell(0);
10  b = new IntCell(0);
11  c = new IntCell(0);
12  add(a);
13  add(b);
14  add(c);
15  pack();
16  {_} += this.updateA();
17 }
18 public void updateA() {
19  a.setValue(b.getValue() + c.getValue());
20 }
21 public static
22 void main(String[] args) {
23  PlusSheet p = new PlusSheet();
24  p.show();
25 }
26 }
```

(c)

Fig. 10. Using Flapjax-like pseudocode (a), DJ (b), and RDJ (c) to implement the sheet example

```
1 public class IntCell {
2   private int value = 0;
3   public void setValue(int v) { this.value = v; }
4   public int getValue() { return this.value; }
5   public Behavior extractValueB() { ... }
6   public void insertValueB(Behavior b) { ... }
7     :
8 }
```

Fig. 11. The source code of IntCell

```
1 public class Sheet {
2  private int b, c, d;
3  public void updateA() {
4   int a;
5   a = this.b + this.c;
6   this.d = a;
7  }
8  public void count(int s) {
9   ...
10 }
11  :
12 }
```

(a)

```
1 public class Sheet {
2  private int a, d, e;
3  public void updateA(int b) {
4   int c;
5    :
6   this.a = b + c;
7  }
8  public void notify() {
9   updateA(this.d + this.e);
10 }
11  :
12 }
```

(b)

Fig. 12. Only fields are taken into account

is the same as in Fig. 2. Line 14 shows the merit of reactive programming: the propagation among signals can be simply described.

Fig. 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 22 of Fig. 10(b). However, the handler h_a (updateA) will not be automatically executed to update the value of a. We need to check the body of h_a and find e_b and e_c by ourselves as mentioned in Sec. II. Here the events e_b and e_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.

Fig. 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 in 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 this.updateA 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 Sec. III. 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/extractValueB are not necessary. RDJ expands the events in DJ to make a step towards reactive programming.

D. The limitations of RDJ

In this subsection we discuss the limitations of RDJ, which are not the limitations of the extension we proposed in Sec. III but might occur when implementing it on an OO language such as DJ.

Only fields can be translated to signals. In Sec. III, 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. 12(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. If the assignment in Line 5 of Fig. 12(a) can be wrapped in a local method slot, the braces operator could enable the automation in the local method slot to let a be used as a signal later.

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 updateA in Fig. 12(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 [26], [17] or a first-class method slot might be necessary to wrap the expressions passed to 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 [4]. 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 mentioned in Sec. III 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 occurred 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 beginning 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 [1] or CEP (Complex Event Processing) [16], [12], [8] 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 handler 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 trigger itself again. However, if two handlers are set to trigger each other, it is hard to know programmers' intention. For example, 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 loop until runtime. A static analysis of the dependencies between fields should help detect endless loops, but a loop might only appear depending on certain conditions at runtime. This issue also happens in FRP languages, and might remind readers of the advice loops in AspectJ. What currently RDJ supports is similar to applying the concept of !cflow(adviceexecution()) in AspectJ to handlers. A more generic solution to this issue might be introducing execution levels [28] to reactive programming.

V. EVALUATION

Since in Sec. IV-C 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. Before that, we briefly go through the RDJ compiler implementation to explain how it works, and measure the impact of introducing the automation to DJ.

A. Preliminary microbenchmarks

In the current RDJ implementation[2], the inference is divided into two parts. At compile-time for every default closure the reference to all its writer method slots are prepared. At runtime the braces operator collects the reference carried by the closures in the given method slot as described in Fig. 7. Note that only the default closure is used to determine whether a method slot is a writer or not. For example, the compiler prepares the reference to b.setValue and c.setValue for the default closure of updateA in Fig. 10(c). Then at runtime all the reference carried by the closures in this.updateA are collected. If we add a closure calling another method slot o.debug, the reference carried by the closures in o.debug will be recursively collected.

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 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

[2]The prototype compiler of RDJ is built on top of DJ and JastAddJ [10], and available from the project webpage.

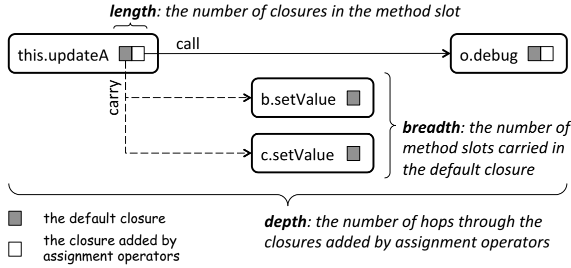

Fig. 13. The metrics for measuring the impact

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. 13. 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. 14. 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. Fig. 14(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. 14(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. 14(c) the only one 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.

B. 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 [26]. 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 Sec. III but also gives a good understanding of such kinds of primitives for signals.

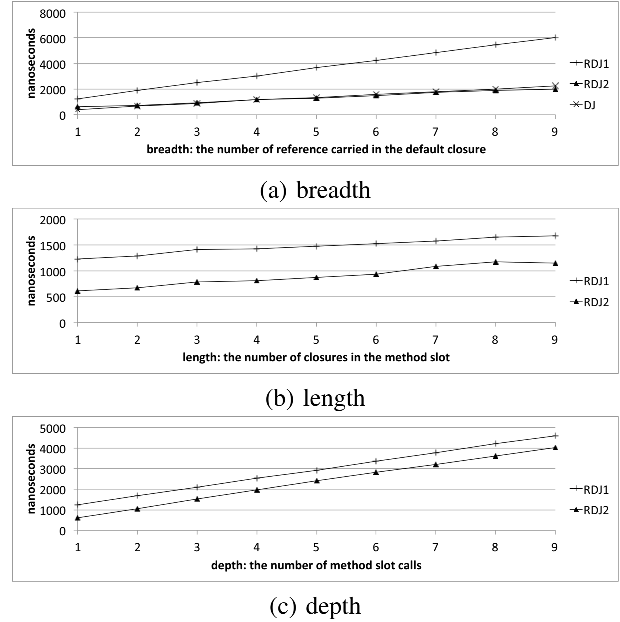

Fig. 14. The three microbenchmarks

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. 15(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[3] 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,

[3]The keyword $retval in DJ is used to get the value returned by the previous closure in the same method slot.

```
1 val age = new Var(0)
2 val size = new Var(1)
3 val canLive: Signal[Boolean] = Signal {
4   (age()<=maxAge) && (size()<=3000) && (size()>=1)
5 }
6 evt shouldDie = canLive.changed && !canLive() || killed
```

(a)

```
1  int age = 0;
2  int size = 1;
3  boolean canLive = true;
4  boolean updateCanLive() {
5    canLive = (age<=maxAge) && (size<=3000) && (size>=1);
6    return !canLive;
7  }
8  boolean shouldDie() { return $retval; };
9  // then within the body of a method slot
10 {_} += this.updateCanLive();
11 this.updateCanLive += this.shouldDie;
12 this.killed += this.shouldDie;
```

(b)

Fig. 15. Rewriting the changed example (a) to (b)

```
1 evt click: Event[(Int, Int)] = mouse.click
2 val circle: Var[(Int, Int),Int] = Var((1,1),10)
3 val lastClickOnCircle: Signal[Boolean] = Signal{
4   over(click.hold(), circle())
5 }
```

(a)

```
1  void click(Point p);
2  Circle circle = new Circle(new Point(1,1), 10);
3  Point holdClick = new Point(0,0);
4  void updateHoldClick() { holdClick = p; }
5  boolean lastClickOnCircle = false;
6  void updateLastClickOnCircle() {
7    lastClickOnCircle = over(holdClick, circle);
8  }
9  // then within the body of a method slot
10 mouse.click += this.click;
11 this.click += this.updateHoldClick;
12 {_} += this.updateLastClickOnCircle();
```

(b)

Fig. 16. Rewriting the hold example (a) to (b)

though the lexical representation tends to be a little longer. The next example is shown in Fig. 16(a), where the function hold is a primitive for converting an event to a signal. Here how to translate of click, circle, and lastClickOnCircle (Lines 1–3) is the same as the steps in the previous example. However, click.hold() in Line 4 is an anonymous signal converted from the event click, so that we need an extra field holdClick and an extra method slot updateHoldClick for updating holdClick as shown in Lines 3–4 of Fig. 16(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. 17(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. 17(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. 18(a) function snapshot returns a signal to lastClick whose value is updated according to the value of another signal

```
1 evt click: Event[(Int, Int)] = mouse.click
2 var nClick: Signal[Int] = click.fold(0)( (x, ) => x+1 )
```

(a)

```
1 void click(Point p);
2 int nClick = 0;
3 void updateNClick(Point p) { nClick++; }
4 // then within the body of a method slot
5 mouse.click += this.click;
6 this.click += this.updateNClick;
```

(b)

Fig. 17. Rewriting the fold example (a) to (b)

```
1 evt clicked: Event[Unit] = mouse.clicked
2 val position: Signal[(Int,Int)] = mouse.position
3 val lastClick: Signal[(Int,Int)] = position snapshot clicked
```

(a)

```
1 void clicked();
2 Point position = new Point(0,0);
3 void updatePosition() { position = mouse.position; }
4 Point lastClick = new Point(0,0);
5 void updateLastClick() { lastClick = position; }
6 // then within the body of a method slot
7 mouse.clicked += this.clicked;
8 {_} += this.updatePosition();
9 this.clicked += this.updateLastClick();
```

(b)

Fig. 18. Rewriting the snapshot example (a) to (b)

position when the given event clicked 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 then 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 Sec. III 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.

VI. RELATED WORK

In the world of events, more and more techniques are introduced to make events more powerful and expressive. For example, Ptolemy [25] supports quantification and type for events, EventJava [13] considers event correlation, and EScala [14] 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 [7] and SuperGlue [19] 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 [20]. 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 [17] is a functional-reactive data structure for Scala [23], which can automatically propagate incremental change. The Reactive Extensions for .NET [21] is a library for writing asynchronous and event-based programs with LINQ. Although in these 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.

VII. 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] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. OOPSLA '05, pages 345–364. ACM, 2005.

[2] A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103 – 149, 1991.

[3] G. Berry, G. Gonthier, A. B. G. Gonthier, and P. S. Laltte. The Esterel synchronous programming language: Design, semantics, implementation, 1992.

[4] C. Bockisch, S. Malakuti, M. Akşit, and S. Katz. Making aspects natural: Events and composition. AOSD '11, pages 285–300. ACM, 2011.

[5] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. POPL'87, pages 178–188. ACM, 1987.

[6] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. ESOP'06, pages 294–308, 2006.

[7] A. Courtney. Frappé: Functional reactive programming in Java. PADL'01, pages 29–44. Springer-Verlag, 2001.

[8] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[9] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. PLDI'13, pages 411–422. ACM, 2013.

[10] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. OOPSLA'07, pages 1–18. ACM, 2007.

[11] C. Elliott and P. Hudak. Functional reactive animation. ICFP'97, pages 263–273. ACM, 1997.

[12] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Co., 1st edition, 2010.

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

[14] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. EScala: modular event-driven object interactions in Scala. AOSD'11, pages 227–240. ACM, 2011.

[15] LabVIEW System Design Software.
http://www.ni.com/labview/.

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

[17] I. Maier and M. Odersky. Higher-order reactive programming with incremental lists. ECOOP'13, pages 707–731. Springer-Verlag, 2013.

[18] H. Masuhara, Y. Endoh, and A. Yonezawa. A fine-grained join point model for more reusable aspects. APLAS'06, pages 131–147, 2006.

[19] S. McDirmid and W. C. Hsieh. SuperGlue: component programming with object-oriented signals. ECOOP'06, pages 206–229. Springer-Verlag, 2006.

[20] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. OOPSLA'09, pages 1–20. ACM, 2009.

[21] Microsoft Corporation. Reactive Extensions for .NET.
http://msdn.microsoft.com/en-us/data/gg577609.

[22] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. Haskell'02, pages 51–64. ACM, 2002.

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

[24] Y. Ohshima, A. Lunzer, B. Freudenberg, and T. Kaehler. KScript and KSWorld: A time-aware and mostly declarative language and interactive GUI framework. Onward! '13, pages 117–134. ACM, 2013.

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

[26] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. Modularity'14. ACM Press, Apr. 2014.

[27] G. Salvaneschi and M. Mezini. Reactive behavior in object-oriented applications: an analysis and a research roadmap. AOSD'13, pages 37–48. ACM, 2013.

[28] É. Tanter, I. Figueroa, and N. Tabareau. Execution levels for aspect-oriented programming: Design, semantics, implementations and applications. *Sci. Comput. Program.*, 80:311–342, Feb. 2014.

[29] W. W. Wadge and E. A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., 1985.

[30] Z. Wan and P. Hudak. Functional reactive programming from first principles. PLDI'00, pages 242–252. ACM, 2000.

[31] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. ICFP'01, pages 146–156. ACM, 2001.

[32] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. PADL'02, pages 155–172. Springer-Verlag, 2002.

[33] Y. Zhuang and S. Chiba. Method slots: supporting methods, events, and advices by a single language construct. AOSD'13, pages 197–208. ACM, 2013.