Declarative Events for Object-Oriented Programming

Technical Report TUD-CS-2010-0122 *

Vaidas Gasiunas Lucas Satabin
Mira Mezini
Technische Universität Darmstadt
{gasiunas,satabin,mezini}@informatik.tu-darmstadt.de

Angel Núñez Jacques Noyé
École des Mines de Nantes
ASCOLA Research Team (EMN-INRIA, LINA)
{angel.nunez, jacques.noye}@mines-nantes.fr

Abstract

In object-oriented designs inversion of control is achieved by an event-driven programming style based on imperatively triggered events. An alternative approach can be found in aspect-oriented programming, which defines events as declarative queries over implicitly available events. This helps to localize definition of events and avoid preplanning, but lacks a clean integration with object-oriented features and principles. The contribution of this paper is a concept of object-oriented events that combines imperative, declarative and implicit events and provides their seamless integration with object-oriented features, preserving encapsulation and modular reasoning. We present an efficient and type-safe implementation of the concept as an extension to the SCALA language.

1. Introduction

Stability and reusability of software are among the major goals of decomposing software into modules. Design principles supporting these goals [24] often lead to dependencies that are opposite to the direction of the control flow. Therefore, the so-called *inversion of control* mechanisms are essential for improving stability and reusability of software systems.

One of the major techniques for inversion of control is *implicit invocation* [15]. Instead of invoking a method directly on an object, we can announce an *event* and so implicitly invoke the methods of the objects that are observing and reacting to the event. This programming style, also known as *event-driven programming*, is natural in domains with reactive behavior such as user interface, control automation, business-to-business integration, etc.

Conceptually an event can be understood as a noteworthy state change. It can be a change in a data model, a button press, or even a change in the physical world modeled by the software, such as a movement detected by a sensor. Since in object-oriented approaches state is organized by objects, it is natural to associate events with objects. In addition to attributes to access the state of the object and methods to modify it, an object can also provide information about its state changes in a controlled way by exposing corresponding events. Not surprisingly, a lot of classes in standard Java and C# libraries expose events as part of their interfaces.

In object-oriented designs, implicit invocation is usually achieved by the Observer pattern [14]. The major problem with the pattern is that it produces a significant amount of glue code for describing observer interfaces, for registration/unregistration and for notification of observers. A lightweight alternative is the event-delegate mechanism of C#. C# events can be declared as special class members and are accessed as attributes of objects. Event handlers are defined as delegates – a form of function closures in C#, often defined simply by references to methods of objects. An event provides methods to register and unregister delegates. An event can have one or more parameters, and type checking ensures that only delegates with the same parameter types are registered. An event is triggered using a syntax analogous to the one of method calls with the effect of calling all delegates registered to the event. C# events provide the same functionality as Observer, but significantly reduce its code overhead.

The semantics of an event can be characterized by the set of its *occurrences* at runtime. The occurrences of Observer and C# events are defined imperatively by triggering them. We use the term *imperative events* to refer to events defined in this way.

A different flavor of event-based programming is realized by aspect-oriented programming (AOP) [23]. The goal of AOP is to separate *crosscutting concerns*, e.g., transaction management and security, from the base functionality of an application and define them in separate modules called *aspects*. The separation is achieved through the *join-point interception* mechanism. A *join point* is an identifiable point in the execution of a program, e.g., a method execution or a field access. An *aspect* can intercept join points and insert its own functionality before or after them, or even completely replace them. The join points to be intercepted are specified

* This report is also available as INRIA Research Report RR-7313.

by *pointcuts* – queries selecting a set of join points and binding their parameters to various values characterizing the join point. A typical pointcut language consists of a set of atomic pointcuts selecting different elements of static and dynamic program structures (e.g. execution stack), and various operators to compose them. The functionality to be inserted at the join points selected by a pointcut is specified by the *piece of advice* associated to the pointcut.

Join-point interception provides an alternative to explicit event notification. Like imperative events, AOP achieves inversion of control, because the functionality of aspects is called from classes without having explicit dependencies of classes on aspects. Instead of registering with explicitly triggered events, an aspect intercepts the required events as join points. Thus, join points can be considered as *implicit events*, i.e., events that implicitly exist in the program. Pointcuts can be seen as declarative definitions of events, the occurrences of which are the selected join points.

The AO flavor of event-driven programming has several advantages over C# and Observer style imperative events. First, it separates and localizes definitions of events, contributing to understandability and stability of the design. Second, events defined as pointcuts can be combined with each other or specialized to define other events. Third, treating identifiable points in program execution as implicit events avoids the need to declare and trigger these events explicitly and so simplifies the code and reduces the need for preplanning.

However, the AOP-style of event-driven programming does not align well with object-oriented designs. Pointcut languages are primarily designed to support the selection of event occurrences based on the static structure of a program across class/object boundaries, while in object-oriented designs, we are primarily interested in events of particular objects or groups of objects. Although AO languages often make it possible to include various dynamic conditions into pointcuts and to localize the scope of aspects to the join points of particular objects, they are still not suitable for modelling events as properties of objects in an efficient and intuitive way. Moreover, AOP promotes a *global perspective* on the functionality on the application, which is not compatible with the typical OO heuristics of defining each piece of functionality from the *local perspective* of a certain object using only its state and the interfaces of explicitly referenced objects. As a result, aspects may create sophisticated and implicit interdependencies that complicate modular reasoning, modular compilation, and loading.

The contribution of this paper is the concept of *declarative object-oriented events* that enhances imperative events in the C# style with AOP elements while preserving the goals and principles of object-oriented design. Like in C#, events are declared as a new kind of class members and can be accessed as attributes of objects. However, unlike C#, the occurrences of an event can be defined not only imperatively by explicit triggering, but also declaratively by expressions involving and composing other events. The declarative event definition is one of the elements that our approach shares with AOP. However, unlike typical pointcut languages, the event expressions are defined as properties of objects using only the state of the object and the events available from the interfaces of the referenced objects. For a full support of object relationships in event expressions, we make access to events late-bound and support quantification over polymorphic collections of objects.

Another common element with AOP is that we make method calls available as implicit events and so reduce the code overhead and preplanning associated with explicit event declaration and triggering. Unlike in AOP, however, an event definition can refer only to implicit events triggered by methods in its lexical scope and by methods explicitly declared as observable in the interfaces of the referenced objects, thus supporting modular reasoning in the object-oriented style. Implicit dependencies between modules and the pointcut fragility problem [1] are avoided, since event definitions can only rely on explicit references to statically visible software elements.

We present an efficient and type-safe integration of declarative events into SCALA [31, 32]. We have chosen SCALA because it supports both object-oriented and functional features, in particular higher-order functions. The powerful type system of SCALA enables type-safe and structure preserving encoding of event expressions. Implicit type conversions and the possibility to overload operators makes it possible to embed much of the proposed features without extending the compiler. The implementation of events is based on fine-grained management of dependencies between events of individual objects and a push-based notification process based on these dependencies. In this way, we achieve that Observer pattern and C# events can be replaced by designs based on declarative events without degrading performance. Moreover, declarative event definitions make the dependencies between events explicit, enabling optimizations that were difficult to achieve with imperative events. In particular, we can detect and switch off unused chains of events, avoiding redundant event notifications. Our approach to integrating AO elements into the language design fully preserves modular compilation and loading of classes.

The paper is structured as follows. In Sec. 2 we analyze the strengths and the limitations of the imperative events found in object-oriented designs and the aspect-oriented approach of defining events as pointcuts. In Sec. 3 we present the language design of ESCALA combining imperative, declarative and implicit events and providing their seamless integration with object-oriented features. In Sec. 4 we analyze the advantages of ESCALA and its design trade-offs. In Sec. 5 we describe the implementation of events and the translation of ESCALA to SCALA. We discuss related work in Sec. 6, and conclude in Sec. 7.

```
1  public abstract class Figure {
2      protected Point position;
3      protected Color color;
4      ...
5      public delegate void NoArgs();
6      public event NoArgs changed();
7      ...
8      public virtual void moveBy(int dx, int dy) {
9          position.move(dx, dy); onChanged();
10     }
11     public virtual void setColor(Color col) {
12         color = col; onChanged();
13     }
14     protected virtual void onChanged() {
15         if (changed != null) { changed(); }
16     }
17     ...
18 }
19
20 public class RectangleFigure : Figure {
21     protected Size size;
22     ...
23     public virtual void resize(Size size) {
24         this.size = size;
25         onChanged();
26     }
27     public virtual void setBounds(int x1, int y1, int x2, int y2) {
28         position.set(x1, y1); size.set(x2 − x1, y2 − y1); onChanged();
29     }
30     ...
31 }
```

Figure 1. Figures in C#

2. Problem Statement

In this section, we outline the deficiencies of the way events are supported in mainstream object-oriented languages. Further, we argue that although these deficiencies could be addressed by the aspect-oriented mechanisms of pointcut and advice, there are certain limitations related to using aspects for event-driven programming.

2.1 Imperative Events

Although C# events make the design intention of the developer more explicit and reduce the code overhead of the Observer pattern, their semantics is in principle identical to the semantics of the pattern. In both cases the occurrences of events are defined imperatively by triggering them at the corresponding locations within the program. This way of defining events has several deficiencies.

Separation of Concerns The definition of events by triggering is not localized. Instead, the code preparing the event parameters and triggering the events is tangled with method implementations. This creates some unnecessary dependencies by making, at the implementation level, the methods dependent on the events, whereas the methods are logically independent of them. Moreover, when the state change denoted by an event may happen at multiple places in the code, the definition of the event by triggering becomes scattered over one or even over multiple classes. Tangling and scattering of event definitions makes it difficult to understand and to change them independently. It also makes it difficult to add new events. Moreover, these changes require diving into details of method implementations.

For example, consider a graphical editor working with various figures. Fig. 1 shows a possible implementation of a class Figure and its subclass RectangleFigure in C#. Class Figure contains fields defining its position and graphical attributes, such as color, and methods changing these fields. Class RectangleFigure additionally defines an attribute size and two methods resize and setBounds. To notify clients about its changes, Figure defines an event changed, triggered at the end of every operation of Figure and its subclasses.[1] The definition of the event changed, i.e., the definition of its occurrences, is scattered and tangled within methods of class Figure and its subclasses. To change the notification protocol, e.g. to introduce more fine-grained events or to provide more information about the changes, we would need to change the methods that trigger the events as part of their implementation.

Preplanning Although C# events reduce the overhead of the Observer pattern, support for each event is still associated with additional overhead. Therefore, the developer still has to anticipate all possible events required by the clients and carefully design the interface of the class to support them. In particular, the developer must anticipate which state changes of the class are of interest to its clients, how specific the events should be, and what data should be provided by the events. If the granularity of the events or the associated data is not sufficient to particular clients, the implementation of the class must be changed.

For example, different clients of Figure may be interested in particular changes of the figure, e.g., resizing or moving the figure, and may need different information about the events. As was explained above, since event definitions are tangled with the implementation of the methods on the figures, changing the observation protocol would require changing the methods of the class and its subclasses.

Composition of Events Defining an event by triggering can be seen as a definition of its occurrences by simply enumerating them, i.e. the developer must write the code that explicitly triggers each occurrence of the event. Conceptually, an event could also be defined declaratively in terms of other more primitive events. An event of an object can be defined as a union or specialization of other events of the object. For example, having events denoting figure movement, resizing and its color change, the figure change event could be defined as a union of these events. An event of an object can also be defined in terms of events of other more primitive objects. For example, a change of a drawing can be defined as a change in any of its figures.

Since imperative events are defined only by triggering, they do not directly support the definition of events in terms of other events and force the developer to use workarounds.

[1] The event is triggered indirectly via the method onChanged. This is because in C# an event can be directly triggered only from the declaring class, not from its subclasses. Also, if no handlers are registered with the event, the value of the event is null and the event cannot be triggered.

```
1  public abstract class Figure {
2      ...
3      public event NoArgs changed();
4      public event NoArgs resized();
5      public event NoArgs moved();
6      ...
7      public virtual void moveBy(int dx, int dy) {
8          position.move(dx, dy); onMoved();
9      }
10     public virtual void setColor(Color col) {
11         color = col; onChanged();
12     }
13     protected virtual void onMoved(Point pt) {
14         if (moved != null) moved(pt); onChanged();
15     }
16     protected virtual void onResized(Size size) {
17         if (resized != null) resized(pt); onChanged();
18     }
19     protected virtual void onChanged() { ... }
20     ...
21 }
22
23 public class RectangleFigure : Figure {
24     ...
25     public virtual void resize(Size size) {
26         this.size = size; onResized();
27     }
28     public virtual void setBounds(int x1, int y1, int x2, int y2) {
29         position.set(x1, y1); size.set(x2 − x1, y2 − y1);
30         onResized(); onMoved();
31     }
32     ...
33 }
```

Figure 2. Defining events of figures in terms of each other

In the case of events of the same class, a method triggering an event can also trigger other events. For example, Fig. 2 shows a refactored version of classes Figure and RectangleFigure with more specific events denoting figure movement and resizing. To define that every resized or moved event is also a changed event, the methods triggering the specific events, i.e., onMoved and onResized, additionally trigger onChanged.

To define an event in terms of events of other objects, an object can register handlers to the events of the other objects and trigger its events in these handlers. The reactions may contain statements transforming the parameters of the event and conditional statements defining dynamic filters on the event occurrences. For example, a drawing can register to the changed event of its figures and in the handler trigger its own changed event.

The problem with a design using the aforementioned workarounds is that it is not declarative and does not directly capture the design intention, making it more difficult to understand. Defining higher-level events by triggering may also impact efficiency because multiple events are then explicitly triggered when certain changes occur, even when these events are not used. The approach based on methods triggering multiple events creates undesired dependencies, because methods triggering lower-level events are also responsible for triggering higher-level events.

Moreover, not all forms of event composition can be modeled in this way. For example, the changed event in Fig. 2 is defined to include the union of moved and resized events, but in the cases when these events overlap changed is triggered twice. This, e.g., happens in setBounds, which moves and resizes the figure, and therefore triggers both moved and resized events.

In general the problem is that instead of abstracting over the same events in different ways, we trigger new ones. For example, to model a change to the bounds of the figure both as a resized and a moved event, we must trigger two independent events. Then by modeling these events also as occurrences of changed, we actually create two more events. There is no way to detect that all four events represent the same logical event. As a result we cannot properly model union of events and other typical set operations such as intersection and difference of sets.

2.2 Aspect-Oriented Programming

The AOP approach to defining events addresses the disadvantages of imperative events.

First, the definition of an event as a pointcut is localized, which achieves a better separation of concerns. Pointcuts are defined independently of the methods triggering the intercepted join points. This makes the design more stable, because existing pointcuts can be changed and new pointcuts can be defined without changing the observed classes.

Second, the availability of join points as implicit events reduces the need for preplanning, because classes are made observable without any special preparations. The most stable and thus the most useful join points are executions (or calls) of public methods. They make it possible to insert functionality of aspects before and after method executions and thus conceptually correspond to the events triggered at the beginning and the end of methods. Indeed, our analysis of Java and C# libraries showed that a vast majority of events are triggered at the beginning or at the end of the methods.

Finally, unlike imperative events, pointcuts can be composed using the operators provided by the pointcut language. For example, the pointcut language of AspectJ [22] supports the conventional logical operators – disjunction, conjunction, and negation – as well as the possibility to restrict join points by conditions defined as Java expressions. Pointcuts can be declared as named members of an aspect and referenced in the definition of other pointcuts, which enables their reuse. Unlike imperative events, pointcuts can share join points, and the operators on pointcuts can be interpreted as proper set operators on their join points. For example, a disjunction of two pointcuts selects their shared join points only once.

Despite these advantages, the pointcut-advice mechanism does not align very well with object-oriented features. In the following, we elaborate on what we mean by this, referring primarily to AspectJ features. In Sec. 6 we discuss in detail to what degree the listed problems are addressed in other AOP approaches.

Events from the Perspective of Objects AO design is focused on separation of crosscutting concerns often involving different parts of an application. Therefore, AO design tends to view the application as a whole and modularize its different concerns into aspects that cut across the object-based design. Consequently, the pointcut language of AspectJ is designed to quantify over the static structure of an application: the members of a class, the classes of the application, inheritance relationships between classes. Although the pointcut language supports dynamic conditions in pointcuts, these conditions are executed in a static context and can use only static variables and methods.

In a typical OO design, each piece of functionality is defined from the perspective of a certain object using the methods, attributes and relationships of the object. An object does not have a complete knowledge of the world, only of its own state and the interfaces of directly referenced objects. In a proper OO design all functionality is defined from a local perspective, while global functionality is avoided as much as possible. In object-oriented designs we are primarily interested in events of certain objects rather than events of all instances of certain classes. For example, a view is interested in the events of a model that it observes rather than all instances of the model class. If the model is a composite structure, the view may be interested in the events of the objects constituting the model. To support event definitions of this kind we need two things. First, pointcuts must be able to refer to events of certain objects and quantify over relationships among objects. Second, pointcuts must be defined in the context of objects in terms of their attributes and relationships.

Event Reactions in Objects An aspect is primarily considered as a module of some crosscutting concern and is either stateless or modeled as a singleton. This means that reactions to events, modeled as pieces of advice, are executed in the context of singleton objects. To model objects reacting to events of other objects, e.g., views observing the models that they display, we need an infrastructure analogous to that of the Observer pattern: an aspect must maintain a registry of objects interested in its events and notify the registered objects in its pieces of advice [20].

Variation of Event Definition The definition of an event may not only depend on the state of the object, but may also vary depending on its type. For example, the definition of the figure change event depends on the type of the figure. Although certain attributes and methods are shared by all figures, other attributes and methods are available only to specific figure types. For example, RectangleFigure of Fig. 1 can be changed by methods setBounds and setSize. Such methods are not applicable to all types of figures, e.g., PolylineFigure would be modified by modifying its points.

A pointcut intercepting changes of any figure would need to refer to methods of Figure and all its subclasses. The problem with such a pointcut is that the objects interested in the changes of a figure would be exposed to all specific types of the figure. Changes to specific figure types would not be isolated, because they may require updating the definition of the pointcut. The design would not be extensible with new types of figure, because the pointcut would need to be changed to include their new methods.

To vary the definition of an event depending on the type of an object and to hide these variations from its clients, we must support events as attributes of objects. Moreover, to support the use of events over polymorphic references to objects, the references to events must be resolved dynamically depending on the dynamic type of the object.

Fine-Grained Events As argued in [35] and [38], join points or the data related to the join points are often insufficient to define all useful events. Since availability of events depends on the granularity of decomposition into methods, an oblivious design cannot guarantee availability of all necessary events as join points. The events that do not naturally correspond to the boundaries of methods must still be modeled as events that are explicitly triggered at the appropriate locations within the methods of the class.

Modularity The availability of join points as implicit events reduces the need for preplanning. Nevertheless, unconstrained quantification over implicit events has also several drawbacks.

If the interface of a class is not specifically designed for observation by aspects, the decomposition of a class into methods can be more or less accidental and can be changed in the next version of the class. The result is *fragility of the pointcuts* intercepting such methods. This problem can be mitigated by limiting join-point interception to public methods only, because then the aspect, like all other clients of a class, would depend only on its public interface. This would, however, reduce the set of available implicit events and would increase the need for explicit event triggering.

Even more problematic are pointcuts based on method name patterns, because then there are no explicit dependencies between the pointcut and the methods to which they refer. Implicit dependencies make the software more difficult to analyze and to maintain. For example, if a method is removed, or its signature is changed, it is difficult to detect the pointcuts that are affected by the change.

The lack of explicit interfaces for observation implies that a class is completely open to aspects and so complicates *modular reasoning* about its properties, because the aspects can directly or indirectly affect the behavior of the class at every join point.

Last but not least, uncontrolled join-point interception also compromises the typical implementation-related properties of object-oriented languages. In object-oriented languages, classes are considered as modules that can be independently checked, compiled, and even loaded. It is difficult to preserve these properties in presence of join-point interception. To compile or to load a class in a modular way,

without knowing all the aspects that can potentially observe it, we must prepare all join points of the class for a potential interception, which creates a significant performance overhead.

3. Object-Oriented Events in ESCALA

In this section, we present ESCALA, an extension of the SCALA programming language [31, 32] with object-oriented events. Basically, we extend the idea of events as object members, realized in C#, with the possibility to define events by declarative expressions. The event expressions are analogous to pointcuts in aspect-oriented languages, but are better integrated into the object-oriented paradigm. From AOP we also borrow the idea of join points as implicit events.

SCALA was chosen as a base language, because it supports a combination of object-oriented and functional features supplied with a powerful type system, which enables structure and type preserving embedding of events and event operators in the language. The presented concepts are not specific to SCALA, however, and can be applied to any object-oriented language.

3.1 Different Kinds of Events

An event is declared as a class member with the keyword event. An event can collect data about the context of its occurrence. The type of the collected data can be inferred from the definition of the event or specified explicitly in square brackets after its name.[2] An event can be either defined by an event expression or declared as imperative, which means that it is defined by explicit triggering. Event expressions make it possible to define events in terms of other events, at the lowest level relying on *primitive events*.

There are two kinds of primitive events: In addition to *imperative events*, which are defined by triggering, ESCALA also supports *implicit events*, which mark language-specific actions during the execution of a program, such as the beginning or the end of the execution of a method. Implicit events in ESCALA correspond to join points in aspect-oriented languages. We adopt the point-in-time join point model of Masuhara *et al.* [25]. Whereas the join point model of AspectJ associates a join point to the entire execution of an action, this model defines a join point as an event occurring either before or after an action. For the sake of simplicity, the only actions we will consider in this paper are method executions. Extending ESCALA to support a richer join-point model is a possible direction of future work.

In contrast to *implicit events*, we will refer to the events explicitly declared as class members as *explicit events*. An explicit event is either *imperative* or *declarative* depending on whether it is defined imperatively by triggering or declaratively by an event expression.

For illustration, Fig. 3 shows the definition of the events of Figure in ESCALA. The events moved, changed, and invalidated are defined declaratively. The event moved is defined by the expression after(moveBy), more precisely after(this.moveBy), thus referring to the implicit events denoting the points in time after the executions of the method moveBy on this. The event resized is declared as abstract, because Figure does not provide any methods resizing the figure. Nevertheless, such an event is useful because it can be refined in subclasses. The event changed is defined as a union of resized, of moved, and the points after the executions of setColor. Finally, the event invalidated is defined by attaching the information about the current figure bounds to the event changed.[3]

```
1  abstract class Figure {
2    ...
3    event moved[Unit] = after(moveBy)
4    abstract event resized[Unit]
5    event changed[Unit] = resized || moved || after(setColor)
6    event invalidated[Rectangle] = changed.map(_ => getBounds())
7    ...
8    def moveBy(dx: Int, dy: Int) {
9      position.move(dx, dy)
10   }
11   def setColor(col: Color) {
12     color = col
13   }
14   def getBounds(): Rectangle
15   ...
16 }
```

Figure 3. Figure in ESCALA

```
1  class Connector(val start: Figure, val end: Figure) {
2    start.changed += updateStart
3    end.changed += updateEnd
4    ...
5    def updateStart() { ... }
6    def updateEnd() { ... }
7    ...
8    def dispose() {
9      start.changed -= updateStart
10     end.changed -= updateEnd
11   }
12 }
```

Figure 4. Figure connector

3.2 Event Reactions

The mechanism of binding reactions to events is analogous to the one of C#. A reaction can be registered to an event by a statement of the form $e \mathrel{+}= f$, where e is an expression referring to the event and f is a function implementing the reaction. Analogously, the statement e -= f unregisters the reaction from the event. A reaction to an event can be any function which can take the data value of the event as argument, i.e., the data type of the event must be a subtype of the argument type of the function. A reaction can be defined by a reference to a method of an object, as well as by an arbitrary function value.

[2] We use SCALA's type Unit to declare events providing no data.

[3] See Sec. 3.3 for the definition of map. The parameter of the map operator is a closure calling the method getBounds on this object.

Fig. 4 shows an implementation of a connector connecting two figures, represented by the fields start and end. To update the respective ends of the connector after changes in the figures, the connector registers the method updateStart as a reaction to the event start.changed, and the method updateEnd as a reaction to the event end.changed. In the example, registration takes place in the constructor of the class.[4]

Class Connector also provides a method dispose, in which the reactions are unregistered from the observed events. Explicit unregistration is necessary for garbage collection, because event-reaction binding creates a link from the object declaring the event to the object implementing the reaction. In general, if an object referencing an event has a shorter life time than the owner of the event, it must unregister from the event in order to be garbage collected. Analogous explicit unregistrations are also necessary when using C# events or the Observer pattern.

3.3 Event Expression Language

The semantics of an event can be described as *a function on event occurrences* that tells whether the event matches a particular event occurrence, and, if so, returns a value containing relevant information about that occurrence. Let Occ be the set of all event occurrences, and $Event(S)$ denote the set of events with data type S. Then $Event(S)$ is the set of functions of type $Occ \to S^{\perp}$, where $S^{\perp} = S \cup \{\perp\}$. If the event matches a given event occurrence, it computes a value from S; otherwise, it returns a special value $\perp$. The same event occurrence can be selected by multiple events; different events can collect different data about the same event occurrence.

Event expressions enable to define new events by means of references to explicit and implicit events of objects, as well as by means of operators to combine and transform other events.

Implicit events are referred by expressions of the form before($expr.M$) and after($expr.M$) denoting the points before and after the executions of method M with the object $expr$ as a target. The semantics of a reference to an implicit event before($expr.M$), respectively after($expr.M$), is a function that selects occurrences denoting the points before, respectively after, the executions of the method M on the object referenced by $expr$. The data of a before event is the tuple of parameters passed to the method, while the data of an after event is a pair consisting of the tuple of parameters and the return value of the method.

Explicit events are referenced by expressions of the form $expr.E$, where $expr$ is a SCALA expression evaluating to an object, and E is the name of an event exposed by the object. The prefix expression $expr$ can be skipped to refer to events of this. The semantics of a reference to an explicit event depends on whether it is defined imperatively or declaratively.

[4] In SCALA, the statements inside the block of the class are executed as a part of its constructor.

$$or_{[S,T]} :: Event(S) \times Event(T) \to Event(S \cup T)$$
$$or_{[S,T]}(e, e') = \lambda x. \begin{cases} e(x), & \text{if } e(x) \neq \perp \\ e'(x), & \text{otherwise} \end{cases}$$

$$and_{[S,T]} :: Event(S) \times Event(T) \to Event(S \times T)$$
$$and_{[S,T]}(e, e') = \lambda x. \begin{cases} \perp, & \text{if } e(x) = \perp \\ \perp, & \text{if } e'(x) = \perp \\ (e(x), e'(x)), & \text{otherwise} \end{cases}$$

$$\mathit{diff}_{[S,T]} :: Event(S) \times Event(T) \to Event(S)$$
$$\mathit{diff}_{[S,T]}(e, e') = \lambda x. \begin{cases} \perp, & \text{if } e'(x) \neq \perp \\ e(x), & \text{otherwise} \end{cases}$$

$$\mathit{filter}_{[S]} :: Event(S) \times (S \to Bool) \to Event(S)$$
$$\mathit{filter}_{[S]}(e, c) = \lambda x. \begin{cases} \perp, & \text{if } e(x) = \perp \\ e(x), & \text{if } c(e(x)) \\ \perp, & \text{otherwise} \end{cases}$$

$$map_{[S,T]} :: Event(S) \times (S \to T) \to Event(T)$$
$$map_{[S,T]}(e, f) = \lambda x. \begin{cases} \perp, & \text{if } e(x) = \perp \\ f(e(x)), & \text{otherwise} \end{cases}$$

$$empty_{[S]} :: Event(S)$$
$$empty_{[S]} = \lambda x. \perp$$

$$any_{[O,S,U]} :: List(O) \times (O \to Event(S)) \to Event(T)$$
$$any_{[O,S,U]}(l, f) = \lambda x.\ foldl\,(l, \perp, join(x))$$
$$\text{where } join(x) = \lambda s.\lambda o. \begin{cases} s, & \text{if } s \neq \perp \\ f\ o\ x, & \text{otherwise} \end{cases}$$

Figure 5. Event operators

The semantics of a reference to an imperative event is a function that selects any triggering of the event and returns the data passed as parameters. The semantics of a reference to a declarative event is the semantics of the expression defining the event (the $this$ reference in the defining expression is replaced by the reference to the object resulting from the evaluation of the prefix expression). Lookup of the definition of a declarative event is explained in Sec. 3.5.

Event operators of ESCALA make it possible (a) to combine the sets of occurrences selected by operand events by typical set operators, (b) to filter them by additional conditions, and (c) to transform the data returned by the operand events. Since events are semantically described by functions, the semantics of operators on events are described by higher-order functions, listed in Fig. 5.

- The expression $e \mid\mid e'$ denotes the union of two events. Its semantics is defined by the function $or(e, e')$: the result is an event that matches an occurrence if it is matched by e or e'. The data type of the resulting event is a common

supertype of the data types of e and e', which we describe as a union of the sets denoting the types.[5] When both e and e' match an occurrence, the occurrence is selected only once with the data collected by e.

- The expression e && e' denotes the intersection of the two events with the semantics defined by the function $and(e, e')$. The resulting event matches an occurrence only when both e and e' match it. The collected data value is a pair of the values collected by e and e'.
- The expression $e \setminus e'$ denotes the set difference between e and e' with the semantics defined by the function $diff(e, e')$. The resulting event selects an occurrence if it is selected by e, but not by e'. In case of a match the data collected by e is returned.
- The expression e && c filters the event e by the condition c. The condition is an arbitrary boolean function taking the data of the event as a parameter. The semantics is defined by the function $filter(e, c)$. The resulting event matches an occurrence if e matches it and if c evaluates to true when applied to the value collected by e.
- The expression e.map(f) applies function f to the value collected by the event e. Its semantics is defined by the function $map(e, f)$. The resulting event matches an occurrence when it is matched by e, and in case of success the data collected by e is transformed by f.
- The expression empty denotes the event that never matches. It is defined as a function that always returns $\perp$.
- The expression l.any(f) aggregates events of a collection of objects, where l is a collection containing objects of type O, and f is a function from an object of type O to an event. The aggregated event matches an event occurrence, if the occurrence is matched by *any* of the events computed by applying f to the elements of l. The semantics of the operator can be compactly defined by left-folding the collection with the initial value $\perp$, and with the combinator $join$. Given an event occurrence x and a function f, this combinator returns a function that takes the last result of matching s and returns it in case of success, otherwise takes the next object o in the collection, retrieves the corresponding event, and matches the event occurrence. Left-folding is used to repeat matching until success occurs or there is no object left, which corresponds to a failure ($\perp$ is returned).

The event expression language is designed to support the object-oriented style of programming. Events are defined from the perspective of an object using its attributes, relationships, and other events. An object can use events of the referenced objects by means of expressions of the form $expr.E$. Also, both the conditions of a filter expression and the function of a map expression are evaluated in the context of the object defining the event; they can use its fields and methods. For example, the definition of the event invalidated at line 6, Fig. 3 uses the map operator with a function that calls getBounds on this.

[5] In the implementation, we rely on the unification of types provided in SCALA, which is more restrictive.

```
1  class Drawing(val figures: List[Figure]) {
2    event invalidated = figures.any(_.invalidated)
3    ...
4  }
5
6  class View(val drawing: Drawing) {
7    drawing.invalidated += repaint
8    def repaint(bounds: Rectangle) { ... }
9    ...
10 }
```

Figure 6. Drawing

The aggregation of events of a collection of objects (by l.any(f)) provides a convenient way to quantify over one-to-many object relationships. For example, Fig. 6 shows the definition of a Drawing that contains an arbitrary number of figures stored as elements of its field figures. The invalidated event of the Drawing aggregates the invalidated events of its figures. Such an event can be used to repaint the invalidated area of the View displaying the Drawing (line 7, Fig. 6).

3.4 Evaluation of Events

Event expressions appear in event-reaction binding/unbinding statements and in the definitions of member events. Event expressions that appear in statements are evaluated when the statement is executed. However, an expression defining a member event of a class is evaluated on demand when the event is accessed for the first time; following accesses to the event return the same value.

For example, the expression start.changed in line 2 of Fig. 4 appears within a statement of the constructor of Connector. This expression is evaluated when the constructor is executed. In line 9 of Fig. 4, the same expression appears within the body of the method dispose and will be evaluated as a part of its execution. However, the event changed of the figure referenced by start will be evaluated only once. Thus, since start is an immutable field, the evaluation of start.changed in the constructor and in the method dispose will return the same value.

This evaluation regime preserves the identity of an event declared as an attribute of an object, which is important for properly unregistering reactions. For example, in order to unregister the reaction associated with start.changed in the method dispose, it is important that the expression refers to the same event as the one with which the reaction was registered in the constructor of the class.

The lazy initialization of events has two advantages. First, since the events of an object can reference each other in their definitions, it is important that the referenced events are initialized before the events using them. With lazy evaluation, the correct initialization order is determined automatically.

```
1  class RectangleFigure extends Figure {
2      ...
3      event resized[Unit] = after(resize) || after(setBounds);
4      override event moved[Unit] = super.moved || after(setBounds);
5      ...
6      def resize(size: Size) {
7          this.size = size;
8      }
9      def setBounds(x1: Int, y1: Int, x2: Int, y2: Int) {
10         position.set(x1, y1); size.set(x2 − x1, y2 − y1);
11     }
12     ...
13 }
```

Figure 7. Rectangle figure in ESCALA

Second, lazy initialization improves efficiency, because an event is initialized only if it is actually used.

The result of evaluating an event expression is a function on event occurrences representing the event. This event function is conceptually evaluated at every event occurrence. Technically, the implementation of events in ESCALA follows a more efficient push-based evaluation strategy to be presented in Sec. 5. Functions defining conditions and transforming event data are evaluated as a part of event matching. Since ESCALA is an imperative language, functions used in event expressions may be closures that depend on mutable state of the program, e.g., instance variables of the object declaring the event. Thus, event matching depends not only on the data captured from the event occurrence, but also on the current values of the variables captured by the closures.

3.5 Inheritance and Late-Binding of Events

Like all other members of a class, events are inherited by subclasses. The semantics of event inheritance is analogous to the inheritance of methods. The inherited events can be overridden by binding them to new event expressions. The definition of the event in the subclass must be compatible with the event of the superclass: the data type of the event must be preserved, visibility restrictions can be weakened. The definitions of the events inherited from the superclass can be accessed through the super reference.

Fig. 7 shows how RectangleFigure defines or redefines the events declared in its superclass Figure. Event resized, which was declared as abstract in Figure, is defined to refer to returning from the methods resize or setBounds. Event moved reuses its super definition, extended with returning from the method setBounds.

In abstract classes, events can be declared as abstract. In concrete classes inherited abstract events must be implemented either by declaring them as imperative or by defining them declaratively. For example, the compiler checks whether the abstract event resized declared in Figure is implemented in its concrete subclass RectangleFigure. Abstract events are especially useful for describing interfaces of objects consisting both of methods and events.

Since the same event can be defined differently by different objects, references to events cannot be resolved statically. For example, the definition of the event start.changed in line 2 of Fig. 4 depends on the type of the figure referenced by the field start. Like methods, events are *late-bound* in ESCALA. The access to an event is prefixed by a reference to an object, and is resolved by a dynamic lookup of the definition of the event in the class of the object. For example, the event expression start.changed in line 2 of Fig. 4 would evaluate to the definition of changed defined in the class of the object referenced by start. If the connector is created with start of type RectangleFigure, the definition of changed from the class RectangleFigure would be taken. Although RectangleFigure does not redefine changed, there is some late binding taking place: its inherited definition of changed uses the new versions of resized and moved.

3.6 Mutable Object Relationships in Events

The event expressions in the examples so far use only immutable relationships between objects. For example, the Connector of Fig. 4 immutably connects the figures referenced by the fields start and end, and the Drawing of Fig. 6 contains an immutable list of figures. However, an object may also have relationships that change during its lifetime: the values of its fields change through assignment; one-to-many relationships are usually represented by collections and are changed by adding and removing elements.

If an object reacts to events of other objects referenced by its relationships and these relationships change, the reactions of the object must be unregistered from its old references and registered with the new references. For example, class Connector of Fig. 4 can be changed to enable reconnecting the connector to other figures. This can be achieved by turning start and end into mutable fields (declared with the keyword var instead of val). The problem is that every time the connector is reconnected to another figure, it must be unregistered from the events of the old figure, and registered with the events of the new figure.

The use of mutable object relationships in the definitions of events is even more problematic. As described in Sec. 3.4, the definition of an event is evaluated only once when it is accessed for the first time. If the definition of the event depends on relationships that change after the event is evaluated, the event may become inconsistent with the state of the object. For example, the event invalidated of Drawing (Fig. 6) is defined as a union of the invalidated events of all child figures. Assume that we change the field figures to a mutable collection. If we add or remove figures after the invalidated event of the drawing is evaluated, the event would still refer to the events of the figures that were part of the drawing at the time of its evaluation.

A straightforward solution to the problem is to reevaluate the events that depend on mutable references at every event occurrence. Such a solution would be very inefficient and would be incompatible with the push-based implementation of events described in Sec. 5. Therefore, we use a different approach. We observe changes to the relationships

```
1  class MutableConnector(
2    val start: Variable[Figure],
3    val end: Variable[Figure]) {
4    ...
5    event startChanged = start.changed || start.event(_.changed)
6    event endChanged = end.changed || end.event(_.changed)
7    event figureChanged = startChanged || endChanged
8    ...
9    startChanged += updateStart
10   endChanged += updateEnd
11   ...
12   def connectStart(fig: Figure) { start := fig }
13   ...
14 }
```

Figure 8. Mutable figure connector

```
1 class MutableDrawing {
2   val figures = new VarList[Figure]
3   ...
4   event invalidated = figures.any(_.invalidated)
5   ...
6   def addFigure(fig: Figure) { figures.add(fig) }
7 }
```

Figure 9. Mutable composite figure

between objects and reevaluate only events that depend on these changes. We define class Variable[T] to model observable mutable references and VarList[T] to model observable one-to-many relationships.

Class Variable[T] defines a wrapper to a reference of type T. The class provides an event changed denoting changes to the reference, and a method event for defining events that depend on the reference. The method is parametrized by a function f that computes an event from a value of the reference. The method produces an event that is equivalent to f applied to the current value of the reference. Technically, the event is recomputed not at each event occurrence, but only when the reference is changed.

For illustration, consider the implementation of a mutable connector in Fig. 8. The connected figures are referenced by wrappers of type Variable[Figure] (lines 2 and 3). The event startChanged at line 5 is defined as a union of the changes of the start reference itself and the changes of the currently referenced figure. The changes of the reference are provided by the event changed of Variable. The changes of the figure are obtained by method event of applied to the function _.changed, which simply accesses the event changed of Figure. The event endChanged is defined analogously.

The result is that the events startChanged and endChanged are always kept consistent with the current relationships of the connector. The reactions of the object to these events (lines 9 and 10) always react to events of the currently connected figures. We can also be sure that other events defined in terms of startChanged and endChanged, e.g., figureChanged at line 7, are always consistent with the relationships of the object.

Mutable one-to-many references are implemented as observable lists, instances of the class VarList. The class defines

```
1 trait IFigure {
2   ...
3   event changed[Unit]
4   ...
5   observable def moveBy(dx: Int, dy: Int)
6   observable def resize(size: Size)
7   observable def setColor(col: Color)
8   ...
9 }
```

Figure 10. Figure interface with observable methods

a method any with semantics analogous to the one of the any operator in Sec. 3.3: It aggregates the events of the elements of the list described by the function given as a parameter to the method. The difference is that the method produces an event which always aggregates the events of the *current* elements of the list. Again, as with Variable, the aggregated event is recomputed not at every occurrence, but only after changes to the list.

Fig. 9 shows an implementation of a drawing with a changing list of figures. The invalidated event is defined as an aggregation of the invalidated events of the figures. The definition is analogous to the definition of the event in the immutable Drawing of Fig. 6. However, the list of children can change (e.g., in line 6), but the event is always kept consistent with the current value of the list.

3.7 Observable Interfaces

The clients of a class can use only its public events. Visibility of the explicit events of a class is controlled by the conventional visibility modifiers. Like any other class members in SCALA, events are considered as public by default, but can be declared as private or protected to constrain their visibility.

By default the implicit events of an object are visible only within the object, i.e., an event expression can without restrictions refer to the implicit events based on the visible methods of this object. To make implicit events of an object observable by other objects, the corresponding methods must be declared as observable. For example, besides the explicit events denoting various types of figure changes, we could also enable direct observation of the calls to the public methods of the figure by declaring these methods as observable as shown in Fig.10.

4. A Review of ESCALA's Language Design

The language design of ESCALA addresses the limitations of imperative events outlined in Sec. 2.1 as well as the problems with using aspect-oriented features in object-oriented designs outlined in Sec. 2.2.

In this section, we discuss how this is achieved by highlighting distinguished features of the ESCALA design: (a) the particular combination of the concept of imperative events found in conventional object-oriented designs with the concept of quantifying over implicit events found in aspect-oriented programming, (b) the seamless integration

```
1  class TemperatureSensor {
2    imperative event tempChanged[Int]
3    ...
4    def run {
5      var currentTemp = measureTemp()
6      while (true) {
7        val newTemp = measureTemp()
8        if (newTemp != currentTemp) {
9          tempChanged(newTemp)
10         currentTemp = newTemp
11       }
12       sleep(100)
13     }
14   }
15   ...
16 }
```

Figure 11. Temperature sensor

of events with object-oriented scoping, inheritance, and subtype polymorphism, and (c) the preservation of the typical object-oriented modularity properties.

4.1 Imperative, Implicit, and Declarative Events

Unlike languages supporting imperative events only, such as C#, ESCALA directly supports definition of high-level events in terms of more primitive events and so *addresses the problem of composing imperative events*. Definitions of high-level events are encoded in a declarative way by expressions rather than indirectly by triggering multiple events in special methods, or by triggering new events in reactions to other events. This makes them better localized and more explicitly encodes the intention of the developer. The developer analyzing and maintaining the code finds the complete definition of the event in one place and does not have to look for all the places where the event could be triggered.

Furthermore, ESCALA's support for implicit events in the style of join points in aspect-oriented programming reduces the need for explicitly triggered events. Often, imperative events are triggered at the beginning or at the end of methods, because methods usually denote logical transactions in the class. Such imperative events can be replaced by references to corresponding implicit events in ESCALA.

By localizing the definition of events and reducing the need for tangling event triggering within methods, support for declarative and implicit events *addresses the problems of separation concerns and preplanning*.

However, unlike aspect-oriented languages, ESCALA does not abandon support for imperative events, because they *address the problem of fine-grained events in AOP*. Events that are not available as identifiable points in control flow (join points in AO terminology) can still be defined by explicit triggering. Such triggering can happen from any SCALA statement using all the local state available in that context.

To illustrate the usefulness of imperative events in situations when we want to trigger events from within a method body or with data that is stored in local variables, Fig. 11 shows the implementation of a driver for a temperature sensor. A loop periodically measures the temperature and notifies about temperature changes. To notify about the changes, the class declares an imperative event tempChanged (line 2) and triggers it every time a change to the temperature is detected (line 9). The event is triggered from inside a loop with a local variable as a parameter. To model such a behavior with declarative events, we would need to replace the triggering of tempChanged by a call to a dummy method with an empty body and define the event by a reference to that method. This would make the design intention less explicit and would create additional complexity.

4.2 Object-Oriented Events and Reactions

Unlike aspect-oriented languages based on static singleton aspects, ESCALA *allows defining events and reactions to these events directly in the context of the objects* that implement the reactive behavior and so addresses the respective limitations of AO outlined in Sec.2.2. Both event expressions and reactions can freely use the state variables, methods, and relationships of the object defining the reactive behaviors. Events can be filtered and their parameters transformed by any SCALA expression valid in the context of that object. Technically, this is achieved by defining the filter predicates and the transformation functions as SCALA closures.

ESCALA is designed to support different kinds of relationships between objects in the definitions of events. References to multiple objects are explicitly supported by the any operator, which makes it possible to aggregate events of a collection of objects. Mutable references between objects in event definitions are supported by the library classes Variable and VarList, which observe changes to the relationships and automatically update the events depending on these changes.

More importantly, access to events is made late-bound to support polymorphic references to objects in combination with the *possibility to vary the definition of an event depending on the object type*. An event declared in an interface or in an abstract class can be defined differently in different subclasses. Clients of an object may use its events without depending on their definition. Such a design makes it possible to introduce new object types and to change the events of the existing object types without affecting clients. Furthermore, existing event definitions are made extensible and composable by supporting super references to events in subclasses integrated with the semantics of mixin composition available in SCALA.

In statically typed aspect-oriented languages, advice-pointcut bindings are established by the pointcut references in the declarations of the pieces of advice. Such bindings are fixed and cannot be changed during the lifetime of the aspect. On the contrary, the event-reaction bindings in ESCALA are flexible. It is possible to bind and unbind reactions dynamically and at arbitrary places in the code. For illustration, consider Fig. 12, which shows pseudo-code for creating a menu of commands. Method createMenu creates multiple menu items and registers methods of the application class

```
1  class MenuItem {
2    event selected = ...
3    ...
4  }
5  class Application {
6    def createMenu : Menu {
7      val menu = new Menu
8      val open = new MenuItem("Open")
9      open.selected += onOpen _
10     menu.add(open)
11     ...
12     menu
13   }
14
15   def onOpen { ... }
16   def onClose { ... }
17   ...
18 }
```

Figure 12. Creating a menu

as reactions with the selection event of these items. In this example, reactions are registered with objects that are referenced by local variables. Thus, it is not possible to create such bindings outside the method body.

4.3 Encapsulation and Modular Reasoning

Whereas aspect-oriented programming breaks encapsulation with its invasive pointcuts, several elements of the ESCALA design *contribute to a better integration of event quantification with object-oriented encapsulation and with the object-oriented style of modular reasoning*, as we elaborate in the following.

Unlike typical pointcut languages, the event expression language of ESCALA does not support name-based quantification over arbitrary elements of the static structure of a program. Instead, event expressions can be defined only in terms of explicit references to the names visible in the context of their definition, such as the members of the enclosing objects and the interfaces of referenced objects. In this way, dependencies between modules are made explicit, avoiding fragile event expressions/pointcuts.

Also, the requirement to explicitly declare methods as observable provides advantages with regard to object-oriented modular reasoning. First, it gives the developer of a class full control over the observable interface of the class, and thus over the points in its control flow, where behaviors of other classes (aspects) can be inserted as reactions. Second, the requirement to explicitly declare methods that can be observed from outside a class is necessary for enabling modular instrumentation of the join points of the class as explained in Sec. 5.4. ESCALA does not support around advice, and thus the behavior of an object cannot be changed by its observers, which further facilitates modular reasoning in the object-oriented sense.

Supporting object-oriented modular reasoning has certain trade-offs, however. Unlike a class in aspect-oriented languages, a class in ESCALA is not completely oblivious of the ways it can be observed. The developer of a class must explicitly expose the events of the class that are available for observation, which obviously involves some preplanning and overhead compared to completely oblivious aspect-oriented designs.

The effort and the level of preplanning for making a class observable are comparable to the effort and the preplanning needed when designing the interface of a class to its subclasses. Analogously to deciding which methods of a class may be overridden by the subclasses, the developer of an ESCALA class needs to decide which methods may be observed by the clients and declare them as observable. Moreover, if the observation protocol provided by a class is not optimal for certain clients, they can adapt it by defining event expressions over the implicit events exposed by the class.

Yet, there are two remarks we would like to make. First, the overhead involved in making a class observable in ESCALA is much lower than achieving the same goal in a language that only supports imperative events. Second, practical AO designs often use annotations in the base classes to make the pointcuts referring to them less dependent on syntactic details. The overhead involved in annotating base classes is comparable to the overhead of declaring methods as observable in ESCALA.

5. Implementation

In this section, we present the implementation of the event model described in the previous section. In subsections 5.1, 5.2, and 5.3, we present an embedding of the model as a library in an unmodified version of SCALA. It allows using the event model with the standard SCALA compiler, but may result in some code overhead and in less intuitive code. In subsection 5.4, an extension of the SCALA compiler that avoids these problems is presented.

5.1 Representation and Typing of Events in SCALA

The interface of an event is described by the SCALA trait shown in Fig. 13. The += and -= operators[6] (lines 3-4) can be used to register reactions with an event. Other methods (lines 6-9) define operators for combining events or for transforming the event data (one method for each operator in Fig. 5). For example, the || operator defines a new event as the disjunction of its receiver and the events passed as parameters. The map function transforms the event data and is especially useful when events of different types need to be combined.

The generic parameter of Event specifies the type of the data provided by the event; in the case of many values, the Tuple*N* classes can be used to parametrize the event. The parameter type of an event is covariant, i.e., if U is a subtype of T then Event[U] is a subtype of Event[T]. Such a subtyping relation is valid, because a reaction to an event expecting a

[6] SCALA makes it possible to use methods with a single parameter as infix operators, where the target is the left-hand side operand and the argument list the right-hand side one.

```
1  trait Event[+T] {
2    // Register and unregister a reaction to the event
3    def +=(react: T => Unit)
4    def -=(react: T => Unit)
5
6    // event combinators
7    // disjunction
8    def |||[S >: T, U <: S](other: Event[U]): Event[S] = ...
9    ...
10   // transform event parameter
11   def map[U, S >: T](mapping: S => U): Event[U] = ...
12   ...
13 }
```

Figure 13. The event trait

data of type T can also work with a data of a more specific type.

Event combinators and transformers are also strongly typed. For example, the event-disjunction method (line 8, Fig. 13) returns an event whose parameter type is the closest common super-type of the combined events as inferred by the SCALA compiler. This implementation relies on the SCALA type system with generics and type inference. Operators map, filter and any rely on the support for function types in SCALA. For example, the map method (line 11, Fig. 13) takes a function transforming the event parameter and produces an event of the return type of the function.[7]

5.2 The Push-Based Model for Implementing Events

The goal of our implementation of events is to provide an efficiency comparable to the object-oriented designs that use the Observer design pattern. A straightforward implementation of the semantics, trying to match every existing event at every event occurrence, is very inefficient. Hence, we opted for a *push-based implementation strategy*, in which a triggered event propagates only to other events depending on it and to reactions registered with it. The remainder of this subsection elaborates on this strategy.

To represent dependencies between events we build a graph, in which each event is represented by a node and each dependency is represented by a directed edge between the corresponding nodes. The hierarchy of event node classes is shown in Fig. 14. All event nodes implement the Event interface and also inherit from EventNode, which contains functionality shared by all nodes. One kind of EventNodes is ImperativeEvents that are imperatively triggered in code. They serve as the source nodes of the graph. Furthermore, there is a concrete subclass of EventNode for each event combinator (EventNodeOr, EventNodeAnd etc.). The latter nodes are created by the corresponding operators defined in Event trait.

In addition to implementing registration of event reactions, EventNode provides methods to register and unregister special reactions, called *sinks*, responsible for notifying dependent nodes. The nodes implementing event combinators register their sink methods with the nodes representing the events being combined.

[7] The parameter function is defined on a supertype of T, because T is a covariant type parameter of Event and cannot be used in contravariant positions.

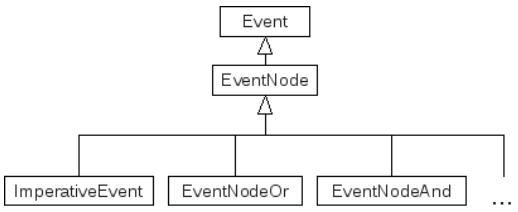

Figure 14. Event hierarchy

Each event occurrence is triggered by calling the apply method on the node representing the corresponding imperative event.[8] When an imperative event node is triggered, the graph is traversed starting from it and the reactions registered with all events that match the triggered event occurrence are collected. The collected reactions are executed only after the matching process is completed; this avoids the execution of event reactions during the event matching process, hence ensuring that the reactions do not influence the matching process.

The reactions of an event node are collected by the method reactions implemented in EventNode. The method collects the reactions directly registered with the event and calls all registered sinks to collect the reactions of the dependent events. The sink methods of the event combinator nodes define their matching semantics by deciding whether to return the reactions to the event or not.

Each imperatively triggered event is associated to a new fresh identifier. The event identifier allows the identification of the notifying event in the graph. Indeed, when a sink is called, it has no other way to identify the originally triggered event than comparing the last seen identifier with the received one.

For illustration, Fig. 15 shows the implementation of the disjunction combinator. The class defines a sink named onEvt (line 4), which is registered with the two events on which the disjunction is applied (lines 12-13). The sink is unregistered when the disjunction event is undeployed (lines 16-17). The sink onEvt saves the event identifier of the received event and uses it to determine whether an event is new. This avoids reacting twice, if both ev1 and ev2 share a particular event occurrence, and thus the disjunction event is notified simultaneously by two different paths in the graph. If a new event is received by the sink, its identifier is saved and reactions method is called to collect the reactions registered with the disjunction (line 8).

The event conjunction is implemented in a similar way. It depends on two events and produces a pair of the two received values by using a mapping function. To be triggered, it must be notified simultaneously by the two depend-

[8] As explained later, implicit events are also mapped to imperative events.

```
1  class EventNodeOr[T](ev1: Event[_ <: T], ev2: Event[_ <: T])
2      extends EventNode[T] {
3    var id = 0
4    lazy val onEvt = (id: Int, v: T,
5      reacts: ListBuffer[() => Unit]) => {
6      if(this.id != id) {
7        this.id = id
8        reactions(id, v, reacts)
9      }
10   }
11   protected override def deploy {
12     ev1 += onEvt
13     ev2 += onEvt
14   }
15   protected override def undeploy {
16     ev1 -= onEvt
17     ev2 -= onEvt
18   }
19 }
```

Figure 15. Event disjunction

ing events. *Simultaneously* means that the sinks are triggered with the same event identifier by two different paths in the graph.

Fig. 16 schematically shows the implementation of the node for quantifying over the events of a varying list of objects. It uses some more sinks and reactions.

```
1  class EventNodeAny[T, U](list: VarList[T], evf: T => Event[U])
2      extends EventNode[U] {
3    lazy val onElementAdded = (target: T) => {
4      evf(target) += onEvt
5    }
6    lazy val onElementRemoved = (target: T) => {
7      evf(target) -= onEvt
8    }
9    lazy val onEvt = (id: Int, v: U, reacts: ListBuffer[() => Unit]) => {
10     reactions(id, v, reacts)
11   }
12   protected override def deploy {
13     list.foreach(target => evf(target) += onEvt)
14     list.elementAdded += onElementAdded
15     list.elementRemoved += onElementRemoved
16   }
17   protected override def undeploy {
18     list.foreach(target => evf(target) -= onEvt)
19     list.elementAdded -= onElementAdded
20     list.elementRemoved -= onElementRemoved
21   }
22 }
```

Figure 16. Quantification over a varying list of objects

This event node computes the events of all elements of the list using the evf function, and registers the sink onEvt with these events (line 13). The sink simply collects the reactions when it is notified by any of the element events. The node registers and unregisters reactions with the events of the observed list triggered when an element is added to the list and removed from it, respectively. The onElementAdded reaction registers the onEvt sink with the event of the newly added element. Similarly, the onElementRemoved reaction unregisters the sink from the event of the removed element.

In order to avoid useless propagations, the deploy and undeploy methods of EventNode are called when the first reaction or sink is registered and when the last sink or reaction is unregistered, respectively. As a result, only events that are actually referenced are represented in the graph; an event node may exist but not be present in any graph, as long as nobody reacts to it or references it.

```
1  class TemperatureSensor {
2    lazy val tempChanged = new ImperativeEvent[Int]
3    ...
4    def run { ... tempChanged(newTemp) ... }
5    ...
6    lazy val logChange = (temp: Int) => { print(temp) }
7    def loggingOn() { tempChanged += logChange }
8    def loggingOff() { tempChanged -= logChange }
9  }
```

Figure 17. Implementation of temperature sensor using events library

5.3 Using ESCALA as a Library

The described implementation of events can be directly used as a SCALA library. Events can be declared as fields of type Event[T]. Imperative events can be created by instantiating ImperativeEvent. Event expressions can be described by calling the combinator operators declared in the Event trait. Reactions to events can be registered and unregistered, using the += and -= operators of the trait, respectively.

Fig. 17 shows the implementation of the temperature sensor introduced in Fig. 11. At line 2 the tempChanged event is declared as a field initialized with an instance of ImperativeEvent, and is triggered at line 4 by calling the apply operator of ImperativeEvent with the event data as parameter. Note that events are declared as lazy SCALA fields to emulate the semantics of the lazy evaluation of events described in Sec. 3.4.

Line 6 demonstrates an implementation of a reaction to an event. The reaction logChange is implemented not as a simple SCALA method, but as a field which is assigned a function value containing the implementation. In this way, we ensure that references to logChange always return the same object, so that the reaction registered at line 7 could be unregistered at line 8. By declaring the field as lazy, we ensure that the function object is created only if it is used.

```
1  abstract class Figure {
2    ...
3    lazy val moved: Event[Unit] = moveBy.after
4    lazy val changed: Event[Unit] = resized || moved || setColor.after
5    ...
6    lazy val setColor = new Observable((col: Color) => {
7      color = col
8    })
9    lazy val moveBy: Observable[Int,Int] = (dx: Int, dy: Int) => {
10     position.move(dx, dy)
11   }
12   ...
13 }
```

Figure 18. Implementation of a figure using events library

Events can also be implicitly triggered before or after method calls. To achieve this, one can use the class Observable provided by the library, which wraps a function and exposes two events named before and after. For example,

Fig. 18 shows the implementation of the Figure class using the library. The method setColor (line 6) is made observable by lifting its implementation into a function value and constructing an instance of Observable with this function value as a parameter. Alternatively, lifting can be performed by implicit SCALA conversions. For example, the moveBy method (line 9) is made observable by declaring it as a field of type Observable[Int,Int] and initialized with a function value containing the implementation of the function. The SCALA compiler will look for an implicit conversion from Function to Observable and apply it.

Events before and after a method execution are made available as members of an Observable object. For example, the moved event of Figure (line 3) can be defined by an event expression that uses the event after of the observable method moveBy. Higher-level events, e.g., the changed event at line 4, can be defined simply by applying the combinator methods defined in the Event trait on references to other events.

Declaring event members, observable and reaction methods as fields complicates their overriding in subclasses. Although SCALA supports overriding of field initializers, super references to field initializers are not possible. The super references must be modeled manually: One can define a protected implementation method and call it from the initializers of the fields. This protected method can be overridden and use its super definition.

5.4 The ESCALA Compiler

As demonstrated in the previous subsection, using the event library directly introduces some code overhead and unnatural encodings of methods and events as fields. These problems are resolved by the ESCALA compiler, which translates events to fields and automatically instruments methods.

Translating event declarations. The ESCALA compiler translates event declarations to the corresponding lazy field declarations. The standard SCALA compiler translates a lazy field to a private field and a getter/initialization method. Overriding value fields is allowed in standard SCALA, which makes it straightforward to implement event overriding. The generated field declarations are internally annotated with a special flag <event> to indicate that unlike normal fields, super references to the field initializers are supported. The call to a super event is translated by the compiler into a call to the corresponding super getter/initialization method.

An imperative event is declared in ESCALA using the imperative modifier and is translated into a field initialized by an expression instantiating the ImperativeEvent class. The field is marked as final and thus cannot be overridden. Overriding an imperative event with a declarative event would break the object interface, because it would prohibit triggering of the event.

Instrumentation of methods. Observable methods are marked in ESCALA source code with the observable keyword. Each observable method is instrumented, and the associated implicit events are generated as illustrated in Fig. 19. The interface of the class remains unchanged, but the implementation of the setColor method is now changed so that each call to it triggers the generated *before* and *after* events. In event expressions, access to implicit events using before or after keywords are replaced by references to the corresponding generated *before* and *after* events. For example, the event expression after(setColor) in line 6 is translated to setColorIntafter in line 24.

```
1  // original source code
2  class Figure {
3    ...
4    observable def setColor(c: Int) { this.color = c }
5    ...
6    event colorChanged = after(setColor)
7  }
8
9  // transformed code
10 class Figure {
11   ...
12   <observable> def setColor(c: Int) = {
13     setColor$Int$before(c)
14     val res = setColor$Color$Impl(c)
15     setColor$Int$after(c, res)
16     res
17   }
18   protected[this] def setColor$Int$impl(c: Int) { this.color = c }
19   final <event> lazy val setColor$Int$before =
20       new ImperativeEvent[Int]
21   final <event> lazy val setColor$Int$after =
22       new ImperativeEvent[(Int, Unit)]
23   ...
24   <event> lazy val colorChanged: Event[(Int, Unit)] = setColor$Int$after
25 }
```

Figure 19. Transformation of observable methods

To enable overriding of observable methods and calling their super implementations, the actual implementation of the instrumented method is moved to a generated implementation method. If a method is already instrumented in the superclass, only the implementation method is overridden in the subclass, as illustrated in Fig. 20. Since setColor is already instrumented in Figure and the corresponding events are generated, the subclass Circle must only override the implementation of the method. The super call to an observable method is transformed to a call to the generated implementation method of the superclass rather than the original observable method. A super call to the original method would create endless recursion, because it would again call the generated implementation method.

With these transformations no superfluous elements are generated and overriding of observable methods is completely supported. The compiler also performs checks ensuring that a method overriding an observable method is also observable, and that referenced *before* and *after* events actually exist (if the associated method is observable).

As was explained in Sec. 3.7, a class may observe any method it defines or one of its parents defines, even if the method is not declared as observable. Such internally observed methods are instrumented on demand by the compiler and marked with the internal flag obsintern to indicate that it

```
1  // original source code
2  class Circle extends Figure {
3    ...
4    observable override def setColor(c: Int) = super.setColor(c)
5  }
6
7  // transformed code
8  class Circle extends Figure {
9    ...
10   protected[this] override def setColor$Int$Impl(c: Int) =
11     super.setColor$Int$impl(c)
12 }
```

Figure 20. Overriding of observable methods

is instrumented but only for internal purposes, i.e., it is not observable from other classes. The generated events are also made available only internally by giving them protected[this] visibility.[9] It is also possible to instrument a method implemented in a parent class by overriding it with an instrumented implementation of the method. This does not break modularity because the compiler can generate the instrumentation for the subclass without recompiling or changing the parent classes.

An internally instrumented method can be explicitly marked as observable in a subclass in order to expose it for observation to other classes. In this case, as far as the instrumentation already exists, the implementation method and the generated events are redeclared with public visibility and implemented by references to their super implementations.

6. Related Work

The design of ESCALA combines elements from event-driven programming, functional reactive programming, and aspect-oriented programming. We discuss the relation to work on each of these areas in dedicated subsections. Other approaches that also combine elements from these areas are discussed in a further subsection.

6.1 Event-Driven Programming

ESCALA supports an event-driven programming style with events modeled as object attributes. This style is widely used in object-oriented software development. It is supported by the Observer pattern [14] and the Implicit Invocation architectural style [15]. In Java, events are modeled by a form of Observer pattern representing events as methods of so-called Listener interfaces. This form of events is also a part of the JavaBeans component model [39]. C# [28] provides language support for declaring and using events as object attributes, which makes events more explicit in the design and reduces the overhead of modeling them by design patterns. This style of declaring events and registering reactions is also adopted in ESCALA. The events of C# are analogous to the imperative events of ESCALA. In addition, ESCALA enables declarative definition of events and interception of implicit events.

[9] It means that they are accessible only from this object.

An alternative to modeling events as object attributes is provided by publish-subscribe systems [12, 17, 33]. They achieve a higher degree of decoupling by making the objects (or components) reacting to events independent of the objects producing the events. Events are published in a global event system, which dispatches the events to their subscribers. This high-level degree of decoupling is, however, not always desired, because it replaces object interaction over explicit interfaces by implicit interaction over the global event system. The implicit dependencies make the software more difficult to understand and maintain. The specification of event subscription in publish-subscribe systems is similar to our declarative definition of events, but is mostly limited to filtering the immediate properties of the event as provided by our filter operator.

A significant body of research in the context of event-driven programming deals with quantification over event sequences or event-based coordination of concurrent processes [3, 9, 13, 16, 18, 19, 30]. This is orthogonal to our work, which focuses on the definition of individual events rather than on their coordination. In particular, join patterns (see, for instance, [3, 19]), although they syntactically look similar to event expressions, are semantically significantly different. Most notably, the and operator of join patterns deal with independent events that need to be synchronized whereas the and operator of event expressions checks that a single event occurrence meets two conditions.

6.2 Functional-Reactive Programming

The idea of defining events by declarative expressions originates in functional reactive programming (FRP) [10, 11, 26, 29]. Events are modeled as lists of event occurrences, and operations on events are modeled as combinations and transformations of such lists. The central concept in FRP is the concept of a behavior. A behavior is a changing value that is modeled as a function of time. Unlike computations based on conventional variables, which produce *fixed values* using the current values of the variables, computations based on behaviors can produce new behaviors, i.e., *changing values*. In FRP, events and behaviors are tightly related. From a behavior, one can obtain an event stream representing the list of its discrete changes. The other way around, a stream of events can be interpreted as a list of changes of a value and used to construct the corresponding behavior.

In spite of a common interest for declarative events, the goals of ESCALA and FRP are very different. While FRP promotes a fully declarative approach to modeling behavior as a function of time or a function of events representing the input, our goal is to provide declarative events for conventional OO designs. In our approach, reactions to events can be dynamically registered and unregistered; they can update the state of the owner object as well as perform any other side effects. ESCALA events are designed to support encapsulation, inheritance and subtype polymorphism. Moreover, we integrate AOP ideas, such as quantifying over implicit

execution events and modeling the semantics of events as queries over event occurrences. Modeling events as queries enables events with shared occurrences and typical set operations on events.

Our implementation of events is analogous to the pure push-based implementation of FRP in FrTime [8] and Flapjax [26]. Events are represented as nodes in a graph with edges representing the dependencies among the events. The semantics of event operations is implemented by update methods of the nodes, which decide whether and with what value an event must be triggered after new occurrences of the input events. Since the semantics of event operations in FrTime and ESCALA are different, their concrete implementation and evaluation is different, too. In FrTime, node updates are performed as soon as the nodes are notified, while in ESCALA, the graph is completely traversed before executing the collected reactions. This two-phase approach is necessary for modeling shared event occurrences, as explained in Sec. 5.1, and for avoiding side effects during event matching. Also, we optimize the graph so that it does not contain edges not leading to a reaction, while in FrTime the nodes are updated independently of their use.

6.3 Aspect-Oriented Programming

ESCALA is not designed as an aspect-oriented (AO) language, but rather as an object-oriented language integrating certain ideas of aspect orientation. In the following, we discuss the relation between ESCALA and AO languages that share some concepts with ESCALA. Unlike AO languages discussed below, quantification over join points in ESCALA is deliberately limited to the dynamic relationships of the object defining the event. The lack of quantification over the program structure or over the control flow of the execution makes ESCALA unsuitable for crosscutting concerns with global effect such as logging or profiling.

Classes in ESCALA are similar to aspects because they can define declarative events (counterparts of pointcuts in AO) as queries over implicit events as well as reactions to these events (counterparts of pieces of advice in AO). CAESARJ [2, 27] and Classpects [36, 37] are aspect-oriented languages that unify aspects and classes by making it possible to declare pointcuts and pieces of advice directly in classes. Unlike AspectJ-style aspects, aspects in these languages can be freely instantiated and their instances can be referenced and used just like normal objects. Furthermore, interception of join points of individual objects is enabled by per-this deployment in CAESARJ and instance-level weaving in Classpects with the effect of limiting the pointcuts of the aspect instance to the join points of the registered objects only. Instantiation of aspects and instance-level deployment is also supported in multiple other approaches, including AspectS [21] and AspectSOUL [7]. More expressive aspect deployment strategies are proposed in [40].

However, despite the mentioned similarities, there are important differences between the above AO languages and ESCALA. Although pointcuts are declared as class members in Classpects and CaesarJ they are still not considered as object attributes. They are evaluated in the static context of a class rather than in the context of an object. On the contrary, ESCALA events are late-bound members of objects.

Instance-level deployment/weaving provides less flexibility than explicit references to events. Instance-level aspects can intercept only the join points that directly occur in the context of the deployed objects. This is often not sufficient to capture all events that occur in the context of the abstraction represented by the object: many other objects may be involved. For example, changes in a drawing include changes to all the components constituting the drawing, but an aspect deployed on a drawing object would be able to intercept only the join points of the drawing object. In general, aspect deployment strategies filter all pointcuts of an aspect, and thus, are unsuitable for relating different pointcuts of an aspect with different objects. Also, explicit deployment makes the definition of the relevant events less explicit, because pointcuts do not contain the complete information about the events to be selected and must be analyzed in combination with the deployment instructions of the aspect.

Technically, the features of ESCALA could be implemented on top of any sufficiently flexible AOP framework. Modeling pointcuts as attributes of objects is possible in AOP approaches supporting pointcuts (also known as joinpoint selectors) as first-class objects [6, 21]. Evaluation of pointcuts in the context of the declaring objects can be modeled by integrating closures into the definition of the dynamic part of the pointcut. An efficient implementation would, however, require a comparable push-based evaluation of pointcuts.

The need for quantification over relationships between objects in pointcuts is addressed by the Alpha language [34]. Pointcuts in Alpha are described as Prolog queries quantifying over various static and dynamic structures of the application, including the heap. The exclusive focus of the work on the Alpha language was to show that a more expressive pointcut language paired with a very rich join-point model enables to write more declarative and stable pointcuts. No attention was paid to the seamless integration of event expressions into an object-oriented setting. Alpha does not support explicit events and does not support pointcuts as attributes of objects. Although Alpha supports much more quantification possibilities, the advantage of ESCALA is that it is limited to a form of quantification that can be efficiently implemented by a push-based approach generating only the events that are actually used.

The problem with the typical implementations of aspect-oriented languages based on static or load-time weaving is that they require knowledge of all available aspects. Support for implicit events in ESCALA is also based on static instrumentation by the compiler, but it is performed in a modular way due to the observable interfaces. The problem of mod-

ular compilation and loading can be mitigated by runtime weaving implemented by reflective techniques [21] or by dynamic bytecode transformation [5, 6]. Reflective techniques are in general less efficient since they rely on an additional level of indirection and name-based lookup of methods. The main problem with the runtime weaving of bytecode is that it creates a significant overhead for deployment and undeployment of aspects or requires dedicated virtual machines with adaptive optimizations [4].

The concept of observable interfaces in ESCALA is similar to Open Modules [1]. An Open Module can expose its internal join points by declaring a pointcut that selects them in its interface; otherwise join points are not visible to the clients. Since ESCALA is an object-oriented language, we apply such encapsulation at the level of objects: the internal event occurrences of an object can be referenced only by its own event definitions. Like ESCALA, Open Modules avoid the problem of pointcut fragility by defining pointcuts only in terms of explicit references to functions and other pointcuts.

6.4 Hybrid Approaches

ESCALA events are related to the hybrid approaches combining ideas of implicit invocation and aspect-oriented programming such as Ptolemy [35] and Implicit Invocation with Implicit Announcement (IIIA) [38]. The three approaches occupy quite different points on the design space. For instance, unlike ESCALA and Ptolemy, IIIA keeps classes and aspects distinct, but, like ESCALA, provides both imperative and implicit events (defined through AspectJ-like pointcuts), whereas Ptolemy does not provide implicit events.

The main point is however that both Ptolemy and IIIA event handlers/pieces of advice react to events characterized by their (event) type. Using event types fully decouple event sources and event sinks while providing robust interfaces between both ends. It also means that events are "application-level" events. Restricting observation to specific sources requires either to set up a filtering mechanism on the observer/sink side, with possible performance issues if too many useless events are emitted, or to fall back on the standard Observer pattern. In line with the philosophy of OOP, we have rather chosen the opposite route by making it easy and efficient to observe selected events from selected objects, handling the Observer pattern behind the scene.

This is facilitated by the possibility of defining events in a very flexible way based on the use of inheritance, composition operators, and late binding. In contrast, IIIA and mainly Ptolemy have more limited ways to compose event abstractions. In IIIA, event types can be organized in an inheritance hierarchy defining their subtyping relationship and implemented by AspectJ-like polymorphic pointcuts (different pointcuts are attached to different classes but produce events of the same type). In Ptolemy a disjunction operator makes it possible to associate a single event handler to a disjunction of event types.

Finally, both Ptolemy and IIIA use a region-in-time event/join-point model: events capture computations in a closure which can be executed by the handler/piece of advice and return a result. In a first step, we have chosen to work with a point-in-time model, which is less powerful but much simpler to reason with. Dealing with a region-in-time model is future work.

7. Conclusions

The hybridization of OOP with event-driven and aspect-oriented genes is a very active area of research. We have developed ESCALA, a new crossbred language which is an interesting point in this design space. ESCALA combines concept of imperative events found in conventional object-oriented designs with the concept of quantifying over implicit events found in aspect-oriented programming; it seamlessly integrates events with object-oriented scoping, inheritance, and subtype polymorphism, while preserving the typical object-oriented modularity properties. Events can be imperatively triggered but also declaratively defined by composition. Declarative events are similar to AOP pointcuts except that, as standard object attributes, they are evaluated in the dynamic context of their enclosing objects and are subject to the standard visibility rules. This avoids pointcut fragility problems and preserves modularity. This also makes it possible to observe specific object instances without the usual boilerplate code associated to the Observer pattern and without performance penalties.

Future work includes completing the current design and implementation with a richer event model and considering an integration of events and pointcuts (that is, the possibility of smoothly combining a point-in-time event model and an AOP-like region-in-time join-point model). It could also be interesting to further investigate the relation of our work with FRP. For example, the classes Variable and Var List introduced in Sec. 3.6 used to enable mutable object relationships in events are conceptually similar to the behaviors in FRP, and thus could be possibly replaced by them. Finally, we have started to consider the combination of our declarative events with approaches describing sequences of events and their upgrading to a concurrent and distributed setting.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *Proceedings of ECOOP '05*, volume 3586 of *LNCS*. Springer-Verlag, 2005.

[2] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *LNCS*. Springer-Verlag, Feb. 2006.

[3] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, 2004.

[4] C. Bockisch, M. Arnold, T. Dinkelaker, and M. Mezini. Adapting virtual machine techniques for seamless aspect support. In *Proceedings of OOPSLA '06*, New York, NY, USA, 2006. ACM.

[5] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *Proceedings of AOSD '04*. ACM Press, 2004.

[6] C.-M. Bockisch. *An Efficient and Flexible Implementation of Aspect-Oriented Languages*. PhD thesis, TU Darmstadt, Feb. 2009.

[7] J. Brichau, A. Kellens, K. Gybels, K. Mens, R. Hirschfeld, and T. D'Hondt. Application-specific models and pointcuts using a logic metalanguage. *Computer Languages, Systems and Structures*, 34(2-3), 2008.

[8] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In P. Sestoft, editor, *ESOP*, volume 3924 of *LNCS*. Springer-Verlag, 2006.

[9] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In K. Lieberherr, editor, *Proceedings of AOSD '04*, Lancaster, UK, Mar. 2004. ACM.

[10] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, 1997. ACM.

[11] C. M. Elliott. Push-pull functional reactive programming. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, New York, NY, USA, 2009. ACM.

[12] P. Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Transactions on Programming Languages and Systems*, 29(1), 2007.

[13] P. Eugster and K. R. Jayaram. EventJava: An extension of Java for event correlation. In S. Drossopoulou, editor, *ECOOP*, volume 5653 of *LNCS*. Springer-Verlag, 2009.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[15] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91: Formal Software Development Methods*, volume 551 of *LNCS*, Noordwijkerhout, The Netherlands, 1991. Springer-Verlag.

[16] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of PLDI '03*, New York, NY, USA, 2003. ACM.

[17] M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul. Filtering and scalability in the eco distributed event model. In *PDSE '00: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, Washington, DC, USA, 2000. IEEE Computer Society.

[18] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3), 2009.

[19] P. Haller and T. Van Cutsem. Implementing Joins using Extensible Pattern Matching. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, volume 5052 of *LNCS*. Springer-Verlag, 2008.

[20] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA 2002, Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 161–173, Seattle, Washington, USA, Oct. 2003. ACM.

[21] R. Hirschfeld. AspectS - Aspect-oriented programming with Squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, London, UK, 2003. Springer-Verlag.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of ECOOP '01*, number 2072 in LNCS, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of ECOOP'97*, volume 1241 of *LNCS*. Springer-Verlag, 1997.

[24] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[25] H. Masuhara, Y. Endoh, and A. Yonezawa. A fine-grained join point model for more reusable aspects. In *Proceedings of APLAS '06*, volume 4279 of *LNCS*, Sydney, Australia, Nov. 2006. Springer-Verlag.

[26] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proceedings of OOPSLA'09*, New York, NY, USA, 2009. ACM.

[27] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings in AOSD '03*. ACM Press, 2003.

[28] Microsoft Corporation. C# language specification. version 3.0. `http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx`, 2007.

[29] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, New York, NY, USA, 2002. ACM.

[30] A. Núñez and J. Noyé. An event-based coordination model for context-aware applications. In D. Lea and G. Zavattaro, editors, *COORDINATION 2008*, volume 5052 of *LNCS*, Oslo, Norway, June 2008. Springer-Verlag.

[31] M. Odersky. The Scala language specification. version 2.7. `http://www.scala-lang.org/docu/files/ScalaReference.pdf`, 2009.

[32] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2008.

[33] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, New York, NY, USA, 1993. ACM.

[34] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In A. P. Black, editor, *Proceedings of ECOOP '05*, volume 3586 of *LNCS*, Glasgow, United Kingdom, July 2005. Springer-Verlag.

[35] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In J. Vitek, editor, *Proceedings of ECOOP '08*, volume 5142 of *LNCS*, Paphos, Cyprus, July 2008. Springer-Verlag.

[36] H. Rajan and K. J. Sullivan. Classpects: Unifying aspect- and object-oriented language design. In *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, USA, May 2005. ACM.

[37] H. Rajan and K. J. Sullivan. Unifying aspect- and object-oriented design. *ACM Transactions on Software Engineering and Methodology*, 19(1), Aug. 2009.

[38] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology*, 20(1), 2010. To appear.

[39] Sun Microsystems. Javabeans(tm) specification. version 1.01. `http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html`, 1997.

[40] E. Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of AOSD '08*, New York, NY, USA, 2008. ACM.