

Programming with Implicit Flows

Guido Salvaneschi¹, Mira Mezini¹, and Patrick Eugster^{2,1}

¹Department of Computer Science, Technische Universität Darmstadt, Germany

{salvaneschi,mezini}@cs.tu-darmstadt.de

²Department of Computer Science, Purdue University, USA

p@cs.purdue.edu

Modern software differs significantly from traditional computer applications that mostly process reasonably small amounts of static input data-sets in batch mode. Modern software increasingly processes massive amounts of data, whereby it is also often the case that new input data is produced and/or existing data is modified on the fly. Consequently, programming models that facilitate the development of such software are emerging. What characterizes them is that data, respectively changes thereof, implicitly flow through computation modules. The software engineer declaratively defines computations as compositions of other computations without explicitly modeling how data should flow along dependency relations between data producer and data consumer modules, letting the runtime to automatically manage and optimize data flows.

Keywords: Reactive programming, event, stream, big data, data-flow

We have come a long way since the early computer systems which were painstakingly fed problem data-sets via punch cards. Computer systems have become much more convenient to interact with and are able to process much larger data-sets, which are kept in large-scale storage systems. However, computer systems are also much more commonly involved in processing of data that is produced or modified in an online fashion, as the program is executing, sometimes in a perpetual manner. This is particularly the case in applications which are specifically developed to react to real-world happenings such as temperature changes or other environmental cues captured through sensors.

The last decade has thus seen the advent of abstractions and paradigms that support the development of *reactive* software. Central to such approaches is the concept of *event* which captures dynamic occurrences that trigger computations. Over the years, several steps have been made in this direction, including language-level support for events, continuous time-changing values (a.k.a. signals or behaviors), constraints, asynchronous execution and futures. The ever-increasing complexity of reactive applications has recently raised new interest around these abstractions. The new paradigm of *reactive programming* focuses on a more holistic view that demands for seamless integration of existing solutions, including constraints resolution

to enforce functional dependencies, automatic updates of dependent values, and interoperability among different reactive abstractions such as signals and event streams. The goal is to raise the abstraction level: Rather than explicitly reifying events in the software, *changes to values of variables* are detected and propagated through programs by re-computing the values of all dependent variables *implicitly*, i.e., by the language runtime.

Interestingly, a similar trend can be observed in recent *big data analysis* software. Not too long ago, such programs were typically perceived as resembling complex queries applied to very large yet static data-sets. A host of programming languages and models have been proposed for such programs. They mostly mix imperative and declarative traits to clearly expose the order of a non-cyclic computation network, and are centered on some form of data-structures conceptualizing the current state of computation. Despite improvements in running time of such analysis programs often due to parallel execution over powerful computation environments, their execution can still take sufficiently long to make repeated complete executions of the same program upon additions or changes to the underlying big data-sets prohibitively expensive. Consequently, recent improvements consist in enabling *incremental computations*, i.e., re-executing only those parts of queries that become invalid or incomplete by changes to analyzed data-sets.

While reactive and big data analysis applications have little in common at first glance, we observe a shared trend in the respective programming models: they strive to capture *what* the computation ought to do, but not *when* (and how) it shall do so, as the data which is subject to the computation changes over time (thus we speak of “data-flows”). It is the execution engines and language runtimes that increasingly carry the burden of determining which parts of computations are affected by which fluctuations in the processed data. As it is unlikely that runtime systems can determine these things entirely on their own — at least in an efficient manner — or that such transparency would even serve the programmer, new abstractions are needed to capture such *implicit flows* in addition to underlying runtime support.

In the following, we first overview the nature and origins of reactive programming and big data analysis and implicit flows

therein. Next, we briefly touch on the state of the art and open challenges towards a unified approach to programming with implicit flows. Unification makes sense not only because of the shared trend towards implicit flows. More importantly it helps coping with the complexity of software that increasingly combines features from both families of applications.

Events and Reactive Programming

Events are a common way for programmers to reason about significant conditions in the environment and in the execution of a program. Dedicated abstractions for events have been supported by some mainstream languages for a long time. For example, in C#, events can be defined as class attributes beside methods and fields and belong to a class' interface. Over the last few years, researchers have proposed increasingly sophisticated event models (cf. Box "Advanced programming with events").

The *integration into the object-oriented (OO) programming model* has been enhanced to extend OO concepts like inheritance to events and event handling. Early approaches like Java_PS [2] implemented events as specific objects. In ES-Scala [11] events are first-class entities. As in C#, they are object attributes just like methods and fields; their definition is subject to polymorphic access and late binding. Our investigations [10] show that this is highly valuable, e.g., enabling programmers to (a) encode the behavior of a class as a state machine and (b) extend it at this high level of abstraction rather than at the level of individual methods.

Events in isolation improve little over the observer design pattern. The difference becomes crucial when *expressive operators for event combination* are available to correlate events to define new (complex) events that capture high-level situations of interest. Advanced systems support operators to combine events with increasing levels of expressiveness. For example, the $e_1 || e_2$ expression in ES-Scala returns an event that fires when either e_1 or e_2 fires. Full-fledged embeddings of complex event processing like EventJava [3], or stream processing languages like SPL [5], support complex queries over event streams including time windows and joins.

In parallel to the development of richer event models, other researchers focused on more *inherent data-flow and change-driven solutions for reactive applications*. These approaches have old roots. For example, the Garnet and Amulet graphical toolkits [12] support automatic constraint resolution to relieve the programmer from manual updates of the view. In functional reactive programming (FRP) [1] developers specify the functional dependencies among time-changing values in a reactive application and the language runtime is responsible for performing the necessary updates (cf. Box "Reactive programming and languages"). FRP has been developed in the strict functional language Haskell and initially applied to graphical animations. The paradigm has been applied to other fields in-

cluding robotics and wireless sensor networks.

The fundamental concept in reactive languages is that programmers *do not directly handle the control flow* but the execution is *driven by the implicit flow of data* and the need to update values. Concretely, programmers specify *constraints* that express *functional dependencies* among values in the application, and the language runtime enforces these constraints without any further effort from the programmer.

More recently, these approaches have inspired many embeddings of DSLs and functional constraints in existing (imperative) programming languages. The advantage of this solution is that programmers specify a functional dependency in an intuitive, declarative way. As a consequence, reactions are directly expressed, do not need to be inferred from the control flow, and can be easily composed.

In practice, (continuous) time-changing values, a.k.a. signals, are not enough. The need for events (i.e., discrete time-changing values) is explained by two observations.

- (a) Events come from external phenomena that are inherently discrete, such as an interrupt or new data from a sensor.
- (b) Events are better suited for modeling certain behaviors: in principle a mouse click can be modeled as a boolean continuous time-changing value that switches to true when the mouse is clicked, but most programmers would rather think of a mouse click as an event. For this reason, existing reactive languages provide both signals and events.

Reactive programming is an emerging trend and identifying the boundaries of this field is hard. However, the following principles seem valid in general.

- *Declarative style.* Reactive behavior is defined in a direct, convenient, declarative style instead of encoding it in design patterns or through imperative updates of program state. Reactions are directly expressed and do not need to be encoded into the control flow of the program.
- *Composition.* Abstractions allow for composition of more complex reactions. Traditional OO applications express reactions in callbacks that are executed when an observable changes. However, callbacks typically perform side effects to modify the state of the application but do not return a value. As a result, they are hard to combine. Instead, events can be combined through combinators, and signals can be combined directly into more complex reactive expressions.
- *Automation.* Programmer effort is reduced by delegating the responsibility of reacting to changes in program state and updating corresponding entities to the language runtime. This solution has several advantages. Reactive code is less error-prone because programmers do not forget to update dependencies (which introduces inconsistencies) and do not update defensively, independently of necessity

Advanced programming with events

Event-based languages include Join Java [1], which captures events by specific asynchronous methods and supports joining of multiple events, and Ptolemy [3] that supports features known from *aspect-oriented programming (AOP)* [2]. In AOP, advices are triggered at points in the execution of the program (e.g., the end of a method call) that are referred to as join points. Join points can be seen as events that occur during the execution and treated uniformly with other events. For example, EScala *before(method)* and *after(method)* events are triggered before and after the execution of methods. Also, in event-based languages that integrate AOP features, programmers can refer to all events of a certain type, a feature that resembles AOP quantification.

As an example of an expressive event system, we show a slice of a drawing application in EScala.

```
1 abstract class Figure { ...
2   protected evt moved[Unit] = after(moveBy)
3   evt resized[Unit]
4   evt changed[Unit] = resized || moved || after(setColor)
5   evt invalidated[Rectangle] = changed.map(() => getBounds())
6   ...
7   def moveBy(dx: Int, dy: Int) { position.move(dx, dy) }
8   def setColor(col: Color) { color = col }
9   def getBounds(): Rectangle ...
10 }
11 class Rectangle extends Figure {
12   evt resized[Unit] = after(resize) || after(setBounds)
13   override evt moved[Unit] = super.moved || after(setBounds)
14   ...
15   def resize(size: Size) { this.size = size }
16   def setBounds(x1: Int, y1: Int, x2: Int, y2: Int) { ... }
17 }
```

Implicit events, like the *after(moveBy)* in the *Figure* class, are automatically triggered at the end of the execution of the associated method (*moveBy* in this case). Events can be defined declaratively by event expressions: the event *changed* is triggered when one of the events *resized*, *moved*, or *after(setColor)* is triggered. EScala events integrate with objects in several ways. Events support visibility modifiers,

abstract events, like *resized*, can be refined in subclasses. Events can be overridden in subclasses (like *moved*) and the inherited definitions can be accessed by *super*. Finally events are late-bound: In the expression *f.changed* the definition of *changed* in *Figure* or in *Rectangle* can be picked up depending on the dynamic type of *f*.

JEScala [4] extends EScala to include asynchronous events and joins like Join Java and EventJava. Join expressions fire an event after two or more events combined by *&* occur in any order. Multiple joins can be combined in *disjunctions* using the *|* operator; when multiple joins fire inside the same disjunction, one is chosen non-deterministically. Joins offer an alternative to thread-based concurrency. In the following Actor example, messages are asynchronous events (Lines 2-3). A disjunction (Line 9) ensures that a single message is processed at a time.

```
1 class Actor {
2   async evt helloMsg[Unit] = ...
3   async evt byeMsg[Unit] = ...
4
5   sync evt threadReady[Unit]
6   async evt start[Unit]
7   start += {while(true){threadReady()}}
8
9   evt (doHelloMsg,doByeMsg) = (threadReady & helloMsg
10                                | (threadReady & byeMsg)
11   doHelloMsg += { println("Hello")}
12   doByeMsg += ...
13 }
```

References

- [1] S. V. Itzstein and D. Kearney. The Expression of Common Concurrency Patterns in Join Java. In *PDPTA*, 2004.
- [2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, 1997.
- [3] H. Rajan and G. T. Leavens. Ptolemy: A Language with Quantified, Typed Events. In *ECOOP*, 2008.
- [4] J. M. Van Ham, G. Salvaneschi, M. Mezini, and J. Noyé. JEScala: Modular Coordination with Declarative Events and Joins. In *MODULARITY*, 2014.

(which wastes computational resources). In addition, automation enables optimization and more automated memory management.

- *Interoperability*. Different reactive abstractions can interoperate. Converting events into signals and back has an important role in practice. Several existing OO applications model state as object fields that are imperatively updated. Conversions allow programmers to take advantage of the design based on signals still preserving compatibility with the existing non-functional code and the event-based design of many applications.

Big Data Analysis

Technologies spearheaded mostly by Google's efforts such as the Google File System (GFS) [4] distributed file system or the distributed implementation of the MapReduce framework originally introduced in the Lisp programming language have ushered in a new area of scalable computing. Through Apache open-source versions of such systems, bundled under the name *Hadoop*¹, these technologies have become widely available; they are currently considered part of the standard toolkit for programming with big data. GFS or Hadoop distributed file system (HDFS) achieve scalability essentially by restricting write operations on files from arbitrary updates to append-only writes. HDFS serves as the default storage medium for data

¹<http://hadoop.apache.org/>

Reactive programming and languages

Reactive programming is based on *constraints* enforced by the language runtime. Consider a functional dependency among the variables a , b and c such that $a = b + c$.


```
1 a = 2           1 a = 2
2 b = 3           2 b = 3
3 c = a + b       3 c := a + b // constraint
4 a = 4 // c is still 5  4 a = 4 // c = 7
5 c = a + b // c = 7    5
```

In imperative programming (left), the functional dependency is satisfied only immediately after the execution of the statement in Line 3. As soon as a change occurs, the functional dependency is no longer valid and must be updated manually (Line 5). Reactive languages (right) automatically enforce constraints (Line 3) recomputing functional dependencies when they are not valid anymore.

As an illustration of more explicit use of constraints consider the following minimal GUI application in the REScala [4] reactive language, which counts mouse clicks on a button and displays the result. In REScala, signals express functional dependencies in a declarative style.

The traditional design, without reactive programming, for such application adopts the observer design pattern. An implementation (simplified for the presentation) using the Scala Swing libraries looks like the following:

```
1 /* Create the graphics */
2 title = "Reactive Swing App"
3 val button = new Button {
4   text = "Click me!"
5 }
6 val label = new Label {
7   text = "No button clicks registered"
8 }
9 contents =
10 new BoxPanel(Orientation.Vertical) {
11   contents += button
12   contents += label
13 }
14 /* The logic */
15 listenTo(button)
16 var nClicks = 0
17 reactions += {
18   case ButtonClicked(b) =>
19     nClicks += 1
20     label.text =
21       "Number of button clicks: " + nClicks
22     if (nClicks > 0)
23       button.text = "Click me again"
24 }
```



The previous code requires inspecting the whole control flow to understand the update logic. For example, the text over the button is initialized in Line 4 and assigned in the statement in Line 23. Line 23 is conditionally executed based on variable $nClicks$, modified in Line 19.

In the reactive programming version using REScala, the whole update logic is captured in Lines 5-11:

```
1 title = "Reactive Swing App"
2 val label = new ReactiveLabel
3 val button = new ReactiveButton
4
```

```
5 val nClicks = button.clicked.count
6 label.text = Signal{
7   (if (nClicks() == 0) "No"
8     else nClicks()) + " button clicks registered" }
9 button.text = Signal{
10  "Click me" + (if (nClicks() == 0) "!"
11               else " again ") }
12 contents = new BoxPanel(Orientation.Vertical) {
13   contents += button
14   contents += label
15 }
```

In reactive languages, conversions between signals and events assume great importance. Conversions allow one to introduce signal-based (declarative) code into OO event-based applications, abstract over state, and concisely express reactive computations.

The following REScala code snippet uses the `snapshot` conversion function to combine a signal that holds the current mouse position and a click event from the mouse. As a result, `snapshot` returns a signal that holds the position of the last mouse click. The other example demonstrates the `last(n)` function, that holds a list of the last n values associated to an event stream. Here, `last(n)` computes the average in a sliding window of five values over a stream of events carrying integers.

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

1 val e = new ImperativeEvent[Double]
2 val window = e.last(5)
3 val mean = Signal { window().sum / window().length }
4 mean.changed += {println(.)}
```

Other reactive languages include FrTime [1], FlapJax [3] and Scala.React [2]. Currently, reactive languages are being extended to support automated propagation of individual elements of non-trivial data-structures (e.g., lists [5]) or to distribution of reactive values over many nodes [6].

References

- [1] G. H. Cooper and S. Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In *ESOP*, 2006.
- [2] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.
- [3] 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 *OOPSLA*, 2009.
- [4] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging between Object-Oriented and Functional Style in Reactive applications. In *MODULARITY*, 2014.
- [5] I. Maier and M. Odersky. Higher-order Reactive Programming with Incremental Lists. In *ECOOP*, 2013.
- [6] G. Salvaneschi, J. Drechsler, and M. Mezini. Towards Distributed Reactive Programming. In *COORDINATION*, 2013.

handled by Hadoop MapReduce, or for results created by the same. With a distributed file system used between MapReduce tasks, many individual local disks used in between map and reduce phases of such tasks, and several mappers and reducers splitting the workload, the MapReduce toolchain is able to scale to very large input files.

To ease the burden on programmers, several high-level scripting and programming languages/language extensions have been introduced, which expose data-flow to enable parallelization. They view programs as directed acyclic graphs (DAGs) with edges representing flow of data and nodes representing (sets of) operations involving data from their incoming edges with results being passed onto outgoing edges. Pig Latin [13] - an untyped scripting language proposed by Yahoo - is a popular example of such a language. Hadoop Pig implements it on top of Hadoop MapReduce. Languages like Pig Latin are used to express data analysis jobs across domains like science and engineering, business and finance, and government and defense. In corresponding programs, intermediate state is typically incarnated by various types of data-structures or collections representing large data-sets, which computations are applied to (cf. Box “Programming with big data”).

In general, languages for big data analysis roughly build upon two abstractions:

1. *Data-structures.* The state of a DAG-based computation at a particular point in the DAG consists in intermediate data, which is conceptualized by a data-structure. Constraints and characteristics of the data (e.g., ordering, indexing) are captured by the specific choice of data-structure (e.g., bag vs. set, set vs. associative map). Pig Latin e.g., leverages bags and maps, while others propose collections and tables (cf. Box “Programming with big data”).
2. *Operations and functions.* Computation itself is expressed via operations more typical of relational query models (e.g., filter, group, join) or functions (e.g., max, min, avg), which are applied to data-structures; results are typically represented again as data-structures.

When data analysis programs or sub-programs are translated to MapReduce jobs, the actual data-structures will never be incarnated as such in a given process’ address space, or even across several such address spaces; these data-structures serve uniquely as conceptual abstractions.

Restricting big data analysis and processing to computations that can be represented as DAGs is a strong limitation. Two major extensions of the computational model promoted by MapReduce and its associated early high-level languages to address this limitation include:

- *Incremental computation.* Support for such computation avoids that upon changes to input data-sets of big data analysis the entire programs have to be re-executed. Incremental computation is particularly sensible in the con-

text of big data – many applications operate on input data-sets such as logs, client activity records, or user records that are constantly extended. Based on the append-only semantics for many such files (by virtue of the distributed file system, e.g., HDFS), extensions to data-sets are naturally captured by stratified appendages.

- *Iterative computation.* Supporting cycles in computations allows for a far more expressive computing model and is similarly relevant in big data processing where often times, due to the sheer size of data, “one-shot” solutions are impossible and computations are iterated until they converge satisfactorily. A popular example is Google’s page rank for determining popularity of web pages used originally as motivation for MapReduce, implemented in that context simply through repeated MapReduce stages. Other examples include many machine learning algorithms such as logistic regression.

Based on these needs, recent programming models (cf. Box “Programming with big data”) aim at supporting either iterative or incremental computing, or both. To that end, data-sets are kept in main memory, partitioned across a number of nodes necessary to accommodate them, thus making cross-accesses for updates much faster than on stored files as promoted by disk-based systems such as MapReduce.

Towards Unified Programming with Implicit Flows

State of the union. The two families of programming languages/language extensions considered in the previous sections share a new paradigm of processing data (changes): *implicit flows* of data (changes) “through” computations. While the two thrusts currently still emphasize different settings and requirements — low-latency in-memory processing on one or few nodes with small data volumes for reactive programming, and high throughput processing of large data-sets distributed across many nodes for big data analysis — confluences are starting to emerge:

1. Approaches in each family are being extended with features characteristic for the other family: implicit propagation of changes in reactive programming is being generalized from simple values to data collections and from local to distributed computations; support for incremental and iterative computations is being added to big data analytics approaches.
2. Approaches with *uniform abstractions for processing heterogeneous stored and online data sources* are emerging: the reactive extensions (Rx) [7] of .NET represent a library-based approach to modeling complex event/stream processing by LINQ [8] operators, which are also used for stored data processing; following DEDUCE [6],

Programming with big data

Several programming languages and models are similar in spirit to Pig Latin. FlumeJava [3] is a library for data-flow processing in Java proposed by Google, and implemented also by Apache Crunch [1]. FlumeJava compiles corresponding tasks to MapReduce jobs at runtime. Like the early Dryad [5] language or Pig Latin, the model comes with standard operators for joining data flows etc., but supports also application-defined functions. The following implements a simple word count in FlumeJava:

```
1 PCollection<String> lines =
2   readTextFileCollection(input_file);
3 PCollection<String> words = lines.parallelDo(
4   new LineToWordFunction<String, String>(),
5   collectionOf(strings()));
6 PTable<String, Long> wordCounts = words.count();
7 wordCounts.write(output_file);
```

First the program reads the `input_file` as a text file, and then, with some degree of parallelization chosen by the runtime, parses lines, generating a collection of strings. Next the program creates a table indexed by words, with the counts for the respective words, before, finally, writing the table to `output_file`.

Early innovators in terms of incremental and iterative computation were the Incoop [2] and iHadoop [4] extensions of Hadoop respectively. Recent examples of data processing models supporting these two features by storing data in main memory include distributed arrays in Presto [6] or resilient distributed datasets in Spark [7]. Incremental computation is

thus far not supported by FlumeJava or Crunch; in the word count example above, incremental computation would consist in augmenting the word counts output to `output_file` following the order of the program, upon extensions to `input_file`. With an in-memory representation of the `wordCounts` table, it would suffice to apply the previous stages to any lines added to `input_file`, and subsequently adding the corresponding new word counts to existing ones in `wordCounts`, or and creating new entries to the table for words which were previously not encountered.

References

- [1] Apache Software Foundation. Incubator Crunch. <http://incubator.apache.org/projects/crunch.html>.
- [2] P. Bhatotia, A. Wieder, R. Rodrigues, U. Acar, and R. Pasquin. Incoop: MapReduce for Incremental Computations. In *SOCC*, 2011.
- [3] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, Efficient Data-parallel Pipelines. In *PLDI*, 2010.
- [4] E. Elnikety, T. Elsayed, and H. Ramadan. iHadoop: Asynchronous Iterations for MapReduce. In *CLOUDCOM*, 2011.
- [5] M. Isard, M. Budi, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007.
- [6] S. Venkataraman, I. Roy, A. AuYoung, and R. Schreiber. Using R for Iterative and Incremental Processing. In *HotClouds*, 2012.
- [7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: a Fault-Tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, 2012.

Shark [14] combines MapReduce, designed for analysis of stored data, with support for processing *online* data.

Outlook. Beside these first steps there is a need for a much stronger confluence. We believe that modern applications would benefit from integration of time changing values a.k.a. signals and big data processing abstractions, making these composable. To enable such compositions we need to conciliate propagation of changes on immutable data in the style of FRP and propagation of changes on mutable data characteristic for big data processing. Fine-grained changes over mutable data structures is an instance of a more general problem: further advances in incrementalization techniques are required. These have been studied for a long time in the database community under the label of *view maintenance*. More recently, incremental solutions have been applied to specific programming domains, e.g., incremental collections. However, attempts to incrementalize a generic program are just at the beginning. Beside incrementalization, language integration of uniform abstractions for implicit data flows may enable optimizations across data-flow graphs and offers opportunities for applying typical compiler optimizations such as inlining, partial evaluation and staging, loop fusion, and deforestation.

Finally, the integration of reactive programming and big data analysis poses a number of challenges concerning the composition of heterogeneous data management and processing strategies. This may require advanced module concepts and related type systems to enable expressing functionality that abstracts over a whole range of processing strategies as well as different data sources/sinks. A key challenge is to reconcile flexibility with static typing to reduce runtime errors. This aspect is especially important in the context of big data where a failure can propagate across dependent computations and invalidate processing already performed.

Acknowledgments

This work has been supported by the German Federal Ministry of Education and Research (BMBF) under grant No. 16BY1206E, by the European Research Council, grant No. 321217, the Alexander von Humboldt foundation, and the US Defense Advanced Research Projects Agency grant No. #N11AP20014.

About the Authors

Guido Salvaneschi is a postdoctoral researcher at TU Darmstadt. He is interested in programming languages, reactive programming, event-based programming and languages for adaptive systems. He holds M.S. and Ph.D. degrees from Politecnico di Milano.

Mira Mezini received the diploma degree in computer science from the University of Tirana, Albania, and the PhD degree in computer science from the University of Siegen, Germany. She is a professor of computer science at the Technische Universität Darmstadt, Germany, where she heads the Software Technology Lab.

Patrick Eugster is an associate professor in computer science at Purdue University on leave at TU Darmstadt, interested in distributed systems and programming languages. He holds M.S. and Ph.D. degrees from EPFL. Patrick is a recipient of a NSF CAREER award (2007) and an ERC Consolidator award (2012), and is a member of DARPA's Computer Science Study Panel (2011).

References

- [1] C. Elliott and P. Hudak. Functional Reactive Animation. In *ICFP*, 1997.
- [2] P. Eugster and R. Guerraoui. Distributed Programming with Typed Events. *IEEE Software*, 21(2): 56–64, 2004.
- [3] P. Eugster and K. Jayaram. EventJava: An Extension of Java for Event Correlation. In *ECOOP*, 2009.
- [4] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, 2003.
- [5] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. P. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu. IBM Streams Processing Language: Analyzing Big Data in Motion. *IBM Journal of Research and Development*, 57(3/4), 2013.
- [6] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu. DEDUCE: at the Intersection of MapReduce and Stream Processing. In *EDBT*, 2010.
- [7] J. Liberty and P. Betts. *Programming Reactive Extensions and LINQ*. Apress, 1st edition, 2011.
- [8] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Object, Relations and XML in the .Net Framework. In *SIGMOD*, 2006.
- [9] E. Meijer and G. Bierman. A Co-relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 54:49–58, Apr. 2011.
- [10] G. Salvaneschi and M. Mezini. Towards reactive programming for object-oriented applications. *Transactions on Aspect-Oriented Software Development XI*, 2014.
- [11] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. EScala: Modular Event-driven Object Interactions in Scala. In *AOSD*, 2011.
- [12] B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferreny, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Trans. Softw. Eng.*, 23(6):347–365, June 1997.
- [13] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *SIGMOD*, 2008.
- [14] R. Xin, J. Rosen, M. Zaharia, M. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *SIGMOD*, 2013.
- [15] T. Uustalu and V. Vene. The Essence of Dataflow Programming. In *APLAS*, 2005.