Towards Reactive Programming for Object-oriented Applications

Guido Salvaneschi, Mira Mezini

Software Technology Group
Technische Universität Darmstadt
$< lastname >$`@informatik.tu-darmstadt.de`

Abstract Reactive applications are difficult to implement. Traditional solutions based on event systems and the Observer pattern have a number of inconveniences, but programmers bear them in return for the benefits of OO design. On the other hand, reactive approaches based on automatic updates of dependencies – like functional reactive programming and dataflow languages – provide undoubted advantages but do not fit well with mutable objects.
In this paper, we provide a research roadmap to overcome the limitations of the current approaches and to support reactive applications in the OO setting. To establish a solid background for our investigation, we propose a conceptual framework to model the design space of reactive applications and we study the flaws of the existing solutions. Then we highlight how reactive languages have the potential to address those issues and we formulate our research plan.

Keywords: Reactive Programming; Functional-reactive Programming; Object-oriented Programming; Incremental Computation

1 Introduction

Most contemporary software systems are reactive: Graphical user interfaces need to respond to the commands of the user, embedded software needs to react to the signals of the hardware and control it, and a distributed system needs to react to the requests coming over the network. While a simple batch application just needs to describe the algorithm for computing outputs from inputs, a reactive system must also react to the changes of the inputs and update the outputs correspondingly. Moreover, there are more tight constraints on computation time, because reactive systems work in real-time and need to react quickly – within seconds or even milliseconds. When the reactive behavior involves non-trivial computations or large amounts of data, various optimization strategies, such as caching and incremental updating, need to be employed.

Object-oriented programming does not provide specific mechanisms for implementing reactive behavior, with two consequences. First, reactive behavior is usually encoded by using the Observer design pattern, whose drawbacks have been extensively highlighted in literature [13,54,52]. For example, the code responsible for update of outputs is usually tangled with the code changing the inputs. As a result, it becomes difficult to understand the computational relations between inputs and outputs and, thus, the intended behavior of the system. Second, the update functionality with the necessary strategies to achieve the desired performances must be implemented manually for each application. Such optimizations, however, introduce a lot of additional complexity, so that it becomes an act of balance between complexity and efficiency.

Various approaches aim to address different aspects of these issues. Event-driven programming (EDP) creates inversion of control to enable modularization of the update code [64,34,28]. Aspect-oriented programming (AOP) enables complete separation of the update concern, by specifying in the aspects the points where the update needs to be triggered [41,67,62,8]. The above approaches fit well with mutable objects, but retain some of the problems related to a programming style based on inversion of control, similar to the well-discussed problems of the Observer design pattern.

Declarative reactive approaches, most notably functional-reactive programming (FRP) [26] and reactive languages, like FrTime [13], Flapjax [54] and Scala.React [52], completely automate the update process. The developer specifies only how a changing value is computed from other values, and the framework ensures that the computed value is automatically updated whenever the inputs are changed. It is, however, not clear whether they can obsolete manual implementation of update code. FRP and reactive languages deal with update of primitive values, and may be too inefficient since they do not provide incremental update of complex structures. Incremental update is provided by other approaches such as LiveLinq [50], but they are limited to specific data structures. Also, declarative approaches based on the functional paradigm do not offer the advantages of typical object-oriented designs, including modularization and component reuse.

In summary, the current state of the affairs is rather disappointing: Developers implement reactive applications in the *comfortable* world of objects, at the cost of relying on programming models whose limitations have been known for a long time. On the other hand, alternatives based on reactive programming offer an appealing solution, but do not succeed because they do not provide the necessary flexibility and do not integrate with the OO design.

In this paper, we propose a research roadmap to fill the gap between OO design and reactive approaches. Our vision is that the concepts developed by FRP and dataflow programming can be integrated with object-orientation to provide dedicate support for reactive applications in mainstream languages. This goal is challenging because reactive abstractions have been explored mainly in the functional setting or in special domains, like reactive data structures. However, the analysis presented in this paper provides a solid background and the first steps in our research plans are already ongoing. Our initial effort is the development of REScala [69], a programming language that integrates events and behaviors *a la* FRP. Thanks to the conversions from events to signals and back, with REScala, reactive abstractions can be easily introduced in OO reactive applications. Our experiments show a significant improvement in the design of reactive software using the functionalities of REScala.

In summary, we provide the following contributions:

– We characterize the design space of reactive applications and discuss the strategies that can be applied to implement reactivity.

- To understand the practical impact of each update strategy, we analyze the implementation of reactive behavior in several real-world OO applications. Our analysis highlights the drawbacks of traditional abstractions.
- We analyze the existing language solutions for reactive systems. We underline their limitations, and the key achievements to take into account in further research.
- We propose a research roadmap which addresses the issues found in the current approaches and has the ultimate goal of combining objects and reactive abstractions in a flexible and efficient language.

The paper is structured as follows. In Section 2 we analyze the design space of reactive software. In Section 3 we present an empirical evaluation of real-world OO reactive applications. Section 4 outlines a possible alternative and discusses other research solutions. Section 5 presents our research roadmap. This paper is an extension of previous work from the same authors [70,68].

2 Design Space in OO Languages

In reactive systems, the outputs of the program need to be updated based on changes of inputs and time. The ways of achieving this goal are however very diverse. In this section, we overview the possible update strategies and discuss the rationale of choosing them.

To make the discussion more clear, in Figure 1 we show a model of a reactive software: The outputs provided to the clients (the objects a and e) must reflect the current state of the inputs of the application (objects b, c, and d) according to certain transformations f and g. Objects can be composed: For example, the object a contains the references a_1 and a_2 to other objects (e.g. by storing them in fields). Dashed arrows model dependencies: Output objects are computed from certain input objects. To clarify how the model applies, consider a weighted graph used to compute a *derived* graph. The derived graph is composed of references to the edges exceeding a `MIN` weight value. In the model of Figure 1 the basic graph can be represented by the b object that contains the references b_1, b_2 and b_3 to the edges. The derived graph can be represented by a and contains the references a_1 and a_2 to the edges. Finally, f is the transformation that produces the derived graph a from the basic graph b by filtering the edges according to the `MIN` value, modeled by the c object. For each update strategy, we show in a Java-like language how the dependent graph is obtained.

2.1 On-demand Recomputation

The most straightforward approach is to recompute the output values each time they are needed. For example, when they are requested by the user to generate a report, or when she refreshes a view. Similarly, in real-time computer games and simulation applications, it is common to recompute outputs automatically at certain intervals of time or simply as often as possible. In OO design, the typical example of this approach are methods returning values computed from the state of the object each time these methods are called.

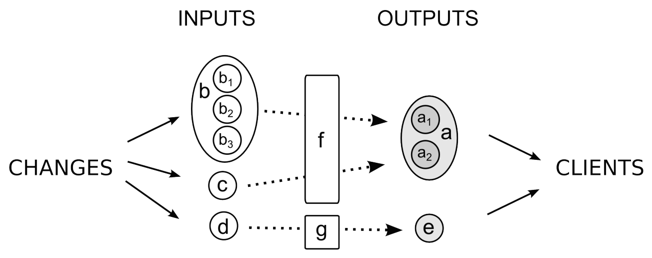


Figure 1. A model of reactive behavior among objects.

The distinguishing aspect of on-demand recomputation is that, after the evaluation, the output is discarded. For example, in Figure 1, every time a client requests a, a is recomputed and b is evaluated to calculate a. Figure 2(a) shows the on-demand recomputation strategy applied to the graph example: Every time the `getDerivedGraph` method is called, the dependent graph is computed from scratch by filtering all the edges (Line 6) and it is returned to the client.

The advantage of this approach is that it is simple to carry out, because the developer just needs to implement procedures computing the outputs from inputs. It also guarantees that the values are always computed from the current state of the program, and thus are always consistent with their inputs. The approach is also memory efficient, because only the inputs need to be stored, but not the outputs or any intermediate computation.

2.2 Caching

Recomputing outputs every time they are requested may be too inefficient, especially in the cases when the computations are expensive or need to deal with large amounts of data. Caching a computed result is a general optimization strategy that avoids repeating the computation. In the model of Figure 1, caching is obtained by saving a and e, letting them available for more than one client access. A typical design is to introduce a field for storing the computation results. The method that computes the dependent value is modified to return the value of the field if it is valid, and to compute and save the result otherwise. Figure 2(b) shows an implementation of the caching strategy: The `derivedGraph` is maintained in a field (Line 3) and returned only if valid, otherwise, the dependent graph is recomputed. When an edge is added to the base graph, the derived graph is invalidated (Line 17).

The cached values are valid only as long as the inputs of the computation do not change. When the inputs change, the cached value must be either recomputed, or invalidated and recomputed at the next request. The latter approach is more efficient when the computed value is used not so frequently, but it is also slightly more complicated. A major issue is to detect changes of the inputs and decide which cached values need to be invalidated. A straightforward approach is to invalidate all cached values after the change of every input. An efficient solution, however, is to analyze the actual dependencies between inputs and outputs, and, after a change to an input, update only the outputs that depend on it – as we explain hereafter.

```
class Graph {
 Edge [] edges;

 getDerivedGraph(){
   Graph g = new Graph();
   for (Edge edge : edges){
     if (edge.weight > MIN)
       g.add(edge);
   }
   return g;
 }
 ...
}
```

(a)

```
class Graph {
 Edge [] edges;
 Graph derivedGraph;
 boolean valid;

 getDerivedGraph(){
  if (!valid){
    derivedGraph=new Graph();
    for (Edge edge : edges){
      if (edge.weight > MIN)
       derivedGraph.add(edge);
  } }
  return derivedGraph;
 }
 addEdge(Edge e){
   edges.add(e);
   valid = false;
 } ...
}
```

(b)

Figure 2. On-demand recomputation (a) and caching with invalidation (b).

2.3 Tracking Dependencies

Instead of updating all outputs after a change to an input, the programmer can rather update only the outputs that actually depend on the changed input. For example, in Figure 1, a change in c requires an update of a, but e is still valid and should not be recomputed. A finer-grained tracking of dependencies can take into account that a only depends on the elements among b_1, b_2 and b_3 that exceed `MIN`. Figure 3(a) shows an implementation of dependency tracking with caching: The dependent graph is maintained in a field, and it is updated only when one of the edges with weight greater than `MIN` is added, i.e. the logic keeps track of the edges on which the derived graph depends. Figure 3(b) shows an implementation of dependency tracking with on-demand recomputation. In this case, the dependent graph is recomputed on every client request. The knowledge about dependencies is maintained by keeping edges in an ordered list (Line 2). In this way, the computation of the dependent graph can be performed by evaluating only a subset of the edges of the base graph. The evaluation is interrupted when an edge not part of the dependencies is encountered (Line 9).

Although tracking dependencies may seem straightforward, implementing this strategy in practice is usually not easy. The programmer needs a precise knowledge of computational relations between outputs and inputs. The dataflow of an application is usually not explicit in imperative code, and a careful code analysis is required to reconstruct it. Moreover, the actual dataflow of an application may depend on dynamic conditions (e.g. dynamic type of a variable in case of subtype polymorphism) and thus may be

```
class Graph {
 Edge [] edges;
 Graph derivedGraph;

 getDerivedGraph(){
   return derivedGraph;
 }
 addEdge(Edge e){
   edges.add(e);
   if (e.weight > MIN)
     derivedGraph.add(edge);
 } ...
}
```

(a)

```
class Graph {
 List<Edge> orderedEdges;

 getDerivedGraph(){
   derivedGraph = new Graph();
   for(Edge edge:orderedEdges){
     if(edge.weight > MIN)
       derivedGraph.add(edge);
     else break;
   }
   return derivedGraph;
 } ...
}
```

(b)

Figure 3. Tracking dependencies with caching (a) and tracking with on-demand recomputation (b).

statically not determinable. Developers must implement the update functionality that corresponds to the detected computational dependencies: After a change of each different input, the update of the corresponding outputs must be called. This may introduce a substantial amount of additional code. The update functionality may also cause modularity problems, because, when implemented in a straightforward way, it may introduce undesired dependencies from inputs to outputs. To avoid such dependencies, the programmer may employ various callback mechanisms (e.g. the Observer pattern), but this further increases the complexity of the implementation.

2.4 Update Incrementalization

Completely recomputing a cached value each time it is invalidated may be too expensive, especially if this value is a complex data structure, such as an array or a graph. A common optimization, in that case, is to update the cached value incrementally depending on the changes to the input. In the model of Figure 1, update incrementalization is an optimization of the functions f and g. This kind of optimization applies in presence of caching: to make updates incremental, the entity to update must be available to receive the changes. Figure 4(a) shows an example of update incrementalization: When a change occurs, the derived graph is not recomputed from scratch but it is modified gradually by adding only the edges that satisfy the condition (Line 13).

The similarity between Figure 3(a) and Figure 4(a), deserves more discussion. In contrast to Figure 3(a), where the dependencies are tracked to notify a change to the dependent graph only when needed, in Figure 4(a) the dependent graph is *always* notified of the change, regardless of the weight of the added edge (Figure 4(a), Line 7). Of course, the edge is added to the dependent graph only if the condition on the weight is satisfied. Figure 3(a) shows that caching with dependency tracking already

```
class Graph {
 Edge [] edges;
 DerivedGraph derivedGraph;

 addEdge(Edge e){
   edges.add(e);
   derivedGraph.add(e);
 } ...
}
class DerivedGraph
    extends Graph {
 addEdge(Edge e){
    if (e.weight > MIN)
      this.add(e);
 } ...
}
```

(a)

```
class Graph {
 Edge [] edges;
 Graph derivedGraph;
 Changes [] changes;

 getDerivedGraph(){
   applyChanges();
   return derivedGraph;
 }
 applyChanges(){
   /* Update derivedGraph
   based on changes.
   Clean added and removed */
 }
 addEdge(Edge e){
   edges.add(e);
   changes.add(new Add(e));
 } ...
}
```

(b)

Figure 4. Update incrementalization (a) and change accumulation (b).

implies some form of incrementality because the dependent graph has to be maintained (caching) and updated selectively (dependency tracking). However, these strategies are conceptually independent, as shown in Figure 4(b), where dependency tracking is demonstrated without caching. The similarity between Figure 3(a) and Figure 4(a) shows that the strategies analyzed in this section are often coupled and isolating them for the sake of the explanation leads to artificial examples: Dependency tracking and incrementalization typically appear together in the applications – the former being necessary to support the latter.

Incremental update requires more fine-grained analysis of the changes to the inputs. It is not sufficient to detect that a certain input has changed, but it is also necessary to get precise information about the change. In addition, the programmer must design algorithms to update the value incrementally after different kinds of changes to the inputs. For example, in case the derived graph is the Minimum Spanning Tree of the original graph, specific domain knowledge in graph theory is required to implement the update algorithm incrementally.

2.5 Accumulating Changes

Accumulating changes is an optimization of the computation of outputs from inputs (i.e. an optimization of f and g in the model of Figure 1). Changes are stored and applied to a cached output, so caching is subsumed by this strategy.

Accumulating changes also implies the incrementalization of the update. Indeed, incrementalization is required to combine the existing object with the incoming changes. Accumulation allows one to arbitrarily choose when to apply the stored changes. One extreme is every time a change occurs (no accumulation), the other is every time the client requests the output. If the update of a value is postponed until the client request, this strategy avoids redundant updates of rarely requested values. However, combining a lot of accumulated changes is more expensive and the response time increases. In some cases, like in databases, the update is postponed until the end of some logical transaction in the inputs. Figure 4(b) shows an example of change accumulation. The changes to the base graph are accumulated in the `changes` array (Line 4). When the client requests the derived graph, the changes are applied and the derived graph is returned (Lines 6–8).

Updating a value after accumulating changes is usually more complicated than updating a value after each primitive change, because it requires more sophisticated data structures to describe the accumulated changes and more sophisticated algorithms to implement the update. As a result, this strategy increases memory consumption. However, accumulating changes also offers opportunities for optimization. For example, some changes can cancel each other. Nevertheless, a complex a domain-specific logic is usually required to take advantage of such cases.

3 Case Studies

To analyze the design issues of OO reactive software, we inspected four reactive Java applications. Our goal is not to develop a systematic empirical study on OO reactive software. Instead, we want to provide a solid background for our research by surveying concrete examples of how reactive features impact OO software design. Due to space reasons, we show only a summary of our analysis. The interested reader can find more details in a technical report [65].

The case studies are of different sizes and cover different kinds of software (two desktop applications, a mobile application, and a library) as well as a variety of external sources of reactive behavior, like network messages, data sampling, values from sensors and user input. Figure 5 summarizes the main metrics of each application.

The **SWT Text Editor** (the `StyledText` widget) implements a text editor in the popular SWT library used by the Eclipse IDE [23]. The application reacts to the insertion of characters and to formatting commands by the user.

The **FreeCol Game** [32] is an open-source turn-based strategy game. The AI of the game controls the opponent players, so the application reacts to the user and the AI. Updates concern the game model and the map in the GUI.

Apache Jmeter [45] supports the performance assessment of several server types (e.g. HTTP). The user specifies a test plan by adding graphical elements to a panel. First, the application must react to changes in the test plan. Additionally, the application is reactive to network events, as the results of the test are visualized in real-time.

The **AccelerometerPlay** Android application is one of the example applications provided by the Android platform [3]. It displays a set of particles rolling on the screen.

Case study	LOC	Types	Cycl. Compl.	LOC/Method	Methods/Type	Fields/Type
SWT Text Editor	9,227	48	4.77	17.39	10.64	4.75
FreeCol Game	170,597	1,175	2.60	10.67	5.77	2.11
Apache JMeter	90,704	1,081	1.84	8.39	7.19	2.13
AccelerometerPlay	460	4	2.00	10.73	4.00	8.00

Figure 5. Main metrics for the case studies.

The inclination of the device is detected by the accelerometer and the particles are updated accordingly.

3.1 Design Choices in the Case Studies

Different design choices concerning reactive behavior are motivated in the case studies by the design, size and kind of software. Our analysis clearly indicates that different applications are better “served” by different points in the design space depicted in Section 2.

The SWT text editor adopts caching and dependency tracking to achieve good performance. It is a typical example of an object with an internal state, which changes in the process of interacting with the user. The editor is implemented in the conventional event-driven style and it is highly optimized to ensure low reaction time. To this end, a lot of fields are used to cache intermediate values, and a complex logic takes care of updating values only when needed.

The FreeCol game lies at the opposite side of the design space and mostly adopts the on-demand recomputation strategy. Since the game behavior is inherently complex, the major issue is managing complexity to keep the development time and the stability of the game within reasonable limits. So, design decisions reducing or at least limiting complexity are favored. A substantial part of the code implements computations of values that are used in the user interface or for AI decision making. Almost all of these values are computed every time they are requested by the user or by the AI.

In the AccelerometerPlay application particle positions are recomputed on-demand every time the screen is refreshed, and incrementalization is applied to efficiently update the positions after conflict resolution. The logic is summarized in Figure 6: Position update comes first, then conflicts with the screen border and conflicts among particles are resolved. Finally, the particles are displayed. Since the application must react quickly, the update logic is based on imperative changes and an iterative algorithm is used for conflict resolution.

JMeter is optimized to make the GUI fast and reactive, changing it in response both to the user and the test events. Due to the different nature of these sources of change, the optimization strategies slightly differ. Caching is used for graphic widgets when the same graphic interface is associated to multiple elements in the test plan and can be reused when the user switches from one element to the other. Incrementalization is applied to optimize updates of graphs and statistics displaying the results of an ongoing test.

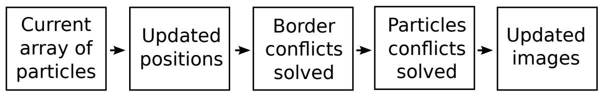


Figure 6. The logic of the AccelerometerPlay application.

3.2 Problem Statement

Our analysis revealed several design issues. We argue that these problems – commented hereafter – are not due to bad choices by the programmers. Instead, as we explain in Section 3.3, they are the consequence of the design limitations and the trade-offs imposed by OO language abstractions.

Code Complexity Manually caching intermediate values requires an accurate logic that is responsible to perform the updates and maintain consistency. Performances increase, but the application becomes more complex. In the SWT text editor, the presence of a lot of fields in each class (70 mutable fields in the worst case) makes reasoning on the behavior of the application really hard, since computations depend on previous state. In addition, the update logic for enabling reactivity pervades a considerable part of the application. For example, the `StyledText` class includes 11 “addListener” methods, 11 “removeListener” methods and 18 “handleEvent” methods. Moreover, the event-handling code includes anonymous classes created on the fly, which also expose callback methods. Finally, since values are separated from their update logic, local reasoning is impossible and understanding the application behavior requires inspecting a lot of code.

On the other hand, on-demand recomputation, like in the FreeCol game, clearly simplifies the logic of the application: As values are generated only when required, the behavior is not hidden by the code that maintains the dependencies.

Hidden Design Intent The AccelerometerPlay application is an example of how reactive functionalities can hide the design intent of the developer. Although it is quite simple (less than 500 LOC), the reactive logic is spread all over the code and a conceptual model like the one in Figure 6 must be harvested from the system of callbacks and events. The origin of this complexity is that the design intent of each update strategy is not explicit in the implementation. For example, certain values are functionally dependent on other values but the design does not express this aspect. Only a careful analysis of the code reveals that a field is never changed directly, but updated after changes of other fields. To reconstruct the intended computational dependencies, the developer must analyze all the update code scattered across the application. Our analysis revealed that computational dependencies are quite common in complex applications. For example, we determined that in the `StyledText` class of the SWT text editor, about half of the 70 mutable fields are not freely changeable, but store values functionally dependent on other fields. Despite that, all fields are declared in the same way and identifying dependent fields requires to reverse engineer the logic of the application, which is lost in the callbacks.

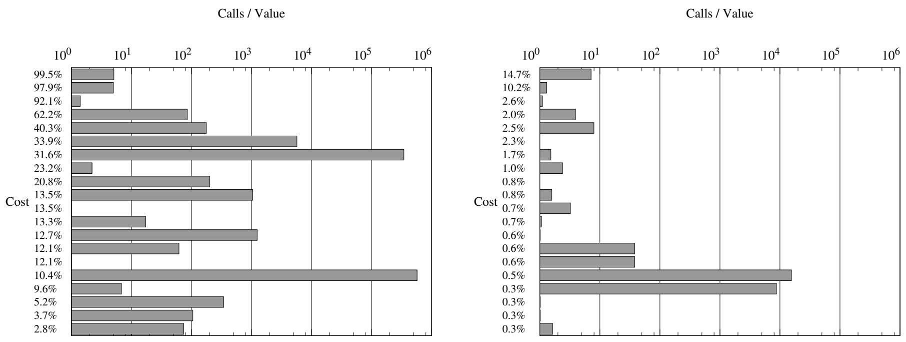


Figure 7. Redundancy of the most expensive methods in the FreeCol game and the SWT text editor.

Redundant Computations The major advantage of on-demand computation is to keep the design simple. However, the overhead that is observed when this design choice is prevalent can be relevant.

We used a profiler and instrumentation via AspectJ to count the potentially redundant calls of the most time-consuming methods in the case studies. The computations after which the returned value does not change (for the same parameters) are potentially redundant. The impact of the design choices on redundancy can be seen in Figure 7. It compares the level of redundancy in the 20 most expensive methods of the FreeCol game and of the SWT text editor, which lie at the opposite positions in the design space. Redundancy is measured as the number of calls per different observed values, i.e., if a method is called a total of 10 times and only 2 different input values and return values are observed, the redundancy is 5. The SWT text editor (right), thanks to its complex logic, shows values of redundancy which are substantially lower. We devoted further investigation to the potential optimizations in the FreeCol game, which largely adopts on-demand recomputation. The results for the methods with higher relative time are shown in Figure 8 (the percentages in the first column do not sum to 100% because methods can be nested). For example, we discovered that the most expensive method (99.57% of the time) has 80% of potentially redundant calls, i.e. the computed value changes only in $\sim$26% of the method calls (Figure 8).

Relative Time	Call / Value	Average Change	Calls	Time
99.57%	5.1	0.2650007264	14,798	76,705
97.98%	5.0	0.2672708364	14,732	75,821
92.13%	1.4	3.8125	29	36,216,475
62.20%	85.0	0.0009699462	8,507,562	83
40.35%	176.5	0.0038796778	5,437,321	84
33.97%	5,697.2	0	28,378,977	13
31.60%	344,846.8	0.1260744986	13,360,833	26
23.23%	2.2	0.0018647649	2,971,606	89
20.82%	201.8	0.0260969848	10,367,994	22
13.56%	1,041.0	0	1,579,745	97
13.56%	1.0	0	5,626	27,471
13.30%	17.3	0.6696185286	6,482	23,392
12.71%	1,241.2	0	1,206,869	120
12.13%	61.9	0	5,451	25,371
12.13%	1.0	0	5,451	25,365
10.49%	571,769.7	0	22,149,669	5

Figure 8. Redundancy analysis for most expensive methods in the Freecol game.

Scattering and Tangling of Update Code When values are intermediately cached, they must be updated in every point of the application where the inputs of the computation are changed. This leads to scattering of the update code.

To evaluate code scattering in the case studies, we considered the places where a field is directly written except the initialization. Figure 9 shows which percentage (y) of fields is updated in x places for each application. The analysis shows that most of the fields are updated only a few times, but the distribution has a *heavy tail*, i.e. there is a consistent number of fields that are updated in many places. Not surprisingly, the SWT text editor, which adopts the caching strategy and is highly complex, has the highest number of fields updated in several places. In the SWT text editor, setters are not used extensively, presumably because they are supposed to be used by the clients of the library but are avoided inside the library for efficiency reasons. However, since it is a best practice to implement setters and getters to encapsulate state, we repeated the analysis for setter methods in Figure 10. With the exception of the AccelerometerPlay application which is too small to suffer from scattering and tangling issues, we found that update scattering is extremely common. For example, in the FreeCol game, in JMeter and in the SWT text editor, respectively, only 38.4%, 46.0% and 30.7% of the fields is updated in just one place. In the worst case, a field was updated in 96 places!

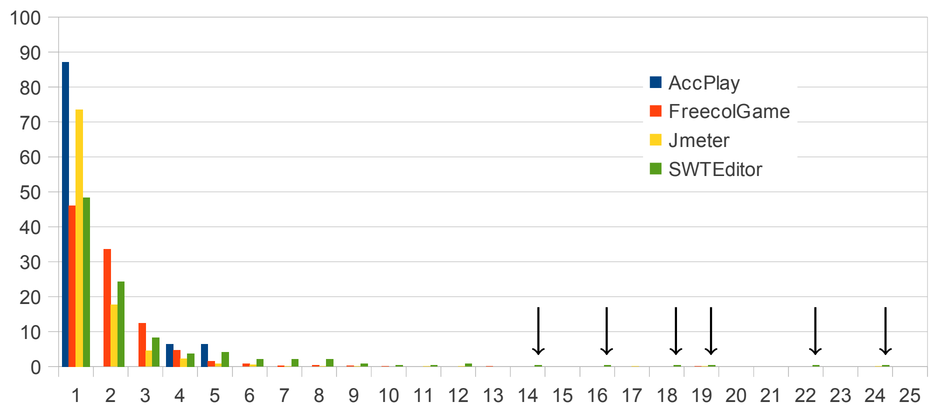


Figure 9. Percentage of fields against number of different update places for that field.

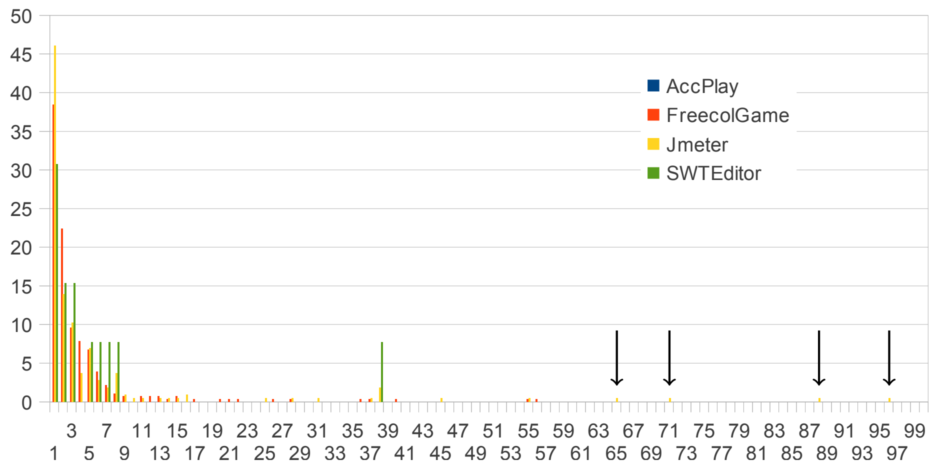


Figure 10. Percentage of setter methods against number of different call places for that setter.

Error-proneness and Code Repetitions When the updates of the dependencies are managed manually, it is often hard for developers to understand when to trigger an update. For this reason, programmers code in a defensive way and introduce an update even when not necessary. For example, in the JMeter application we found cases in which selecting a GUI with the cursor triggers a sequence of four updates of the interface even before the user changes any value. However, update errors are not the only inconvenience that manual updates can cause. When update functionalities are complex, managing consistency by hand can easily lead to code repetitions because the same update pattern is cloned in many places. We evaluated the occurrence of code similarities in the case studies; the results are in Figure 11. The numbers in the table show that for applications of significant size, even when the update functionalities are carefully designed, like in the case of the SWT text editor, it is hard to keep the code clean.

Application	Similarities	Similarities / LOC
SWT Text Editor	29	0.00314
FreeCol Game	281	0.00106
Apache Jmeter	381	0.00420
AccelerometerPlay	0	0

Figure 11. Code repetitions.

3.3 Lesson Learned from the Case Studies

In this section, we summarize the major results from the case studies. A crucial observation is that, in OO applications, reactive entities are separated from the code responsible to keep them updated. This has two bad consequences. First, the dependencies are not explicit, so the design rationale of the application is hard to grasp even for trivial cases. Second, updates are scattered across the application and tangle the rest of the code.

Unfortunately, modularization of update code is hard to achieve in the OO style, because dependencies must be imperatively updated every time an input value is changed – which can occur in several places of the application. Furthermore, manually analyzing dependencies and writing corresponding update code is error-prone; certain dependencies may be overlooked and consequently the programmers can fail to update all functionally dependent values. Therefore, values are often updated defensively without precise knowledge of whether it is actually necessary.

Manually written update code also produces a maintenance problem, because there are no automatic checks ensuring the consistency of the update code with the actual dependencies of the computation. In addition, each time the computational dependencies are changed, the developer must correctly *update the update functionality* to reflect the current state of the dependencies! Errors of such a manual maintenance activity may remain undetected for a long time; forgetting to update a certain value usually does not lead to a crash, and a redundant update may even not cause any visible effects at all, only inefficiency.

Importantly, in our analysis, we observe a clear trade-off between efficiency and complexity. To keep the design simple, programmers accept the cost of on-demand recomputation and potential redundancy. For example, intermediate caching via object fields highly complicates the application because the update logic must be implemented manually. In conclusion, keeping the design simple has a high cost in performance. In some cases, like the FreeCol game, there is a wide space for potential optimization. However, this is not easy to achieve, because the computations involve complicated algorithms and depend on various different inputs.

4 Analysis of Advanced Languages

In this section, we discuss some advanced language concepts to support reactive applications. For each approach, we analyze the problems it addresses and its limitations.

We start the discussion with functional reactive approaches, as they provide interesting insights as how to overcome the problems discussed in the previous section and are the source of inspiration for our planned research, a roadmap of which is presented in the following section.

4.1 Reactive Languages

Reactive languages are based on ideas developed in functional-reactive programming, introduced by Elliott [26] to model time-changing values as dedicated language abstractions. More contemporary incarnations of the concept are integrated in recent reactive languages such as FrTime [13], Flapjax [54], and Scala.React [52]. To make the argumentation more concrete, in Figure 12, we show an example of Flapjax, a dataflow language that overcomes several limitations of reactive design based on inversion of control. Our considerations can be substantially generalized to FrTime and Scala.React. The functionality presented in Figure 12 consists of displaying the elapsed time since the user clicked on a button in a Web page.

Flapjax supports behaviors, i.e. reactive abstractions that model time-changing values (named with a final “`B`” in the code snippet). For example, the value `nowB` is a behavior that represents the current time updated every second. Behaviors can be sampled to obtain “traditional” values via the `valueNow` function: `startTm` is the initial instant of the simulation. In addition, behaviors can be combined with events: In Line 3, the `snapshot` function captures the instant value of the `nowB` behavior every time the `click` event of the `reset` button occurs. As a result, `clickTmsB` always contains the time of the previous click (or `startTm` before any click event). `elapsedsB` keeps the value of the time elapsed from the last click, and `insertValueB` updates the value in the graphic every time the `elapsedsB` value changes (Line 6). The crucial aspect of the reactive semantics is that a declaration like the one in Line 5 expresses a *constraint* rather than a *statement*. The example shows how the language creates implicit dependencies among time-changing values. The general idea is that when the programmer defines a constraint `a=f(b)` and `b` is a time-changing value, the framework automatically detects the dependency of `a` on `b` and is responsible for performing the updates automatically. In Line 5, when either `nowB` or `clickTmsB` changes, the value of `elapsedB` is automatically updated. So, whenever the programmer accesses `elapsedB`, she sees the updated value. Reactive languages provide abstractions to compose time-changing values and combine them with event streams. Eventually, time-changing values are bound to the GUI which automatically reflects the changes. The reader interested in more details can refer to [54].

Automatic dependency tracking addresses several issues highlighted in the case studies. The application is simplified, because the programmer does not shoulder the burden of keeping dependent values consistent (Section 3.2). As a consequence, the errors that can derive from forgetting the updates are automatically avoided (Section 3.2). The update code, which captures the behavior of a program entity, is modularized with the entity, allowing local reasoning and avoiding scattering and tangling with the rest of the application (Section 3.2). In contrast to callbacks, which return void, reactive behaviors can be easily composed. As a result, software is much more readable because the design intention of the programmer is explicit and direct modeling of relations among objects enforces a more declarative style (Section 3.2). Finally, reactive languages automatically derive dependencies and perform only necessary updates (Section 3.2).

In summary, reactive languages are an appealing solution to the issues identified for OO languages. In particular, since updates are performed by the runtime and do not add complexity to the application logic, they have the potential of solving the trade-off between efficiency and simplicity described in Section 3.3. However, there are some crucial issues that prevent their broader adoption.

```
var nowB = timerB(1000);
var startTm = nowB.valueNow();
var clickTmsB = $E("reset", "click").snapshotE(nowB)
                .startsWith(startTm);
var elapsedB = nowB − clickTmsB;
insertValueB(elapsedB, "curTime", "innerHTML");

<body onload="loader()">
  <input id="reset" type="button" value="Reset"/>
  <div id="curTime"> </div>
</body>
```

Figure 12. Automatic dependency tracking in Flapjax.

Functional flavor and immutability Reactive languages impose a functional style, while OO programming features an imperative style. When a lot of code already exists, a functional refactoring of the entire application is in general not acceptable. Some computations are cumbersome to express functionally, while retaining acceptable performance and algorithmic clarity. For example, the conflict resolution algorithm in the AccelerometerPlay application is expressed in an imperative style using for loops, sequences of imperative statements to detect the conflicts, and imperative updates of the particles positions. The AccelerometerPlay application is also an example of performance-critical software. Conflicts among potentially hundred of particles must be solved in a sufficiently short time that the movement appears fluid to the user. Expressing the resolution algorithm in functional style with somewhat acceptable performance, would involve accumulative recursion: Functions in this style are rather hard to understand and, yet, probably not as efficient as encodings based on loops and imperative updates.

A consequence of the functional flavor of reactive languages is that they are effective with primitive values, but do not fit well with mutable objects. Strategies for incremental computation are highly application-specific and a framework can hardly address the problem in a domain-agnostic way. As a result, there is no way to automatically incrementalize object updates. So, reactive languages recompute the dependent object every time the base object changes. To clarify this point, consider the expression `list2 = list1.filter(x>10)`, an instance of the case `a=f(b)` – discussed previously – where `a` and `b` are `list2`, respectively `list1` and `f` is the `filter` function for the given predicate. The expression establishes a dependency between `list1` and `list2` via `filter`. In our case, each time object `list1` changes, the `filter` operator produces a new `list2` object. Instead, imperative approaches update the mutable dependent object in-place, which is more efficient and preserves the object identity. For example, the SWT text editor employs mutable data structures to efficiently store the inserted text.

These observations are symptoms of a more general problem in reactive languages: The update strategy is hardcoded in the reactive framework, so only a point in the design space described in Section 2 is available to the programmer. As a result, efficiency might be an issue even for trivial cases.

Design of complex systems So far, we defended the mutability of OO. Another, even more important reason, why we do not want to abandon the OO style is that it has established itself as the paradigm of choice for complex applications, for reasons related to design clarity and evolvability. A first remark is about modeling: Objects are effective in modeling complex systems because they reproduce the interaction of real-world entities. For example, the hundreds of simulation elements used by the FreeCol game are conveniently represented by objects.

Objects enable the development of large applications by modularizing rather large pieces of functionalities while abstracting over implementation details. For example, all the SWT widgets are ready-to-use components, but at the same time open for future modifications via subtype polymorphism: E.g., the SWT text editor uses a default implementation for the text container, but clients can provide a custom container by implementing a proper interface.

OO also supports reuse by class inheritance. This aspect is crucial in libraries which are developed incrementally, e.g., the `StyledText` class is part of an inheritance hierarchy of depth 7. Another key requirement of complex systems is runtime variability. Objects address this issue via dynamic polymorphism. For example, the simulation elements in JMeter are treated uniformly and late bound depending on the user decisions. In the SWT library, a text editor can be used wherever a generic widget is expected.

4.2 Functional-reactive Programming

As already discussed, functional-reactive programming has been introduced in the seminal paper [26]. Further approaches refined and extended the concept, mostly focusing on the formal semantics of continuous time [59]. This leads to an elegant formalization with a denotational semantics in which behaviors are modeled as functions from time to values [26].

The first FRP implementations were pull-based, i.e. reactive values are pulled periodically by the reactive runtime to update the entities depending on them, e.g. a graphic animation. This approach corresponds to our on-demand recomputation (Sec. 2.1) except that the runtime performs the requests, while in Sec. 2.1 we expect that the client request a result when needed. Note that this model is different from reactive languages like FrTime, Scala.React and Flapjax which are push-based: The dependency graph is updated only when a reactive value changes. Push-pull FRP [27] is an attempt of a mixed model that leverages the efficiency advantages of push-based implementations.

However, exploring the trade-offs between those models and evaluating their impact in practice is still an open problem (Section 5.4).

FRP has been successfully applied to several practical scenarios, including coordinating robots [43], network switches programming [31] and wireless sensor networks [58]. Current work focuses on optimization and safety guarantees enforced by the type system. For example, Krishnaswami *et al.* [49] use linear types to control allocation of new nodes in the dependency graph and avoid memory leaks.

4.3 Observer and Event-Driven Programming

The Observer pattern enables decoupling the code that changes a value from the code that updates the values depending on it. The Observer pattern has a number of drawbacks that have been extensively analyzed by researchers. The main points of criticism are summarized hereafter; the interested reader can refer to [54,52] for a detailed discussion. First, with the Observer pattern, applications are harder to understand because the natural order of dependencies is inverted. A lot of boilerplate code is introduced to correctly implement the pattern. Another problem is that, since callbacks do not return a type but perform imperative updates on the objects state, reactions do not compose. Finally, the notification to the observers is triggered in an imperative fashion and can be easily scattered and tangled with the rest of the application. Interestingly, many of these points of criticism clearly emerged in our case studies.

Event-based languages like C# [14], Ptolemy [64], EScala [34], JEScala [74] and EventJava [28] directly support events and event composition as language constructs. These languages reduce the boilerplate code introduced by the Observer pattern and provide advanced features like quantification over events, event combination and implicit events. Event-based languages integrate well with the imperative paradigm: The callback defines all the operations required for the update of the object state. As a result, this approach preserves object identity – opposed to functional solutions which necessarily compute a different object – and supports efficient fine-grained changes of the updated object.

The main drawback of event-based programming is that a significant amount of reactive dependencies are still modeled by encoding them in a programmatic way, rather than being supported by the language. The update functionality must be designed and implemented explicitly in the callback method. Caching must be managed manually, deciding a proper policy and coding it in the callback as well. Similar considerations apply for any optimization strategy. For example, accumulation of changes must be entirely implemented by the programmer. Another issue is that, since all the update functionality must be coded manually, consistency is not automatically guaranteed. Instead, the developer must take care of correctly notifying all the entities which are functionally dependent. This leads to error-prone code with the risk of notifying *too rarely*, breaking functional dependency, or *too often*, defensively inserting unnecessary updates. Finally, non-functional design choices are hard-coded in the callback implementation: The developer has no option to choose among different non-functional trade-offs discussed in Section 2, such as caching or incremental updates.

4.4 Aspect-oriented Programming

In the context of reactive applications, AOP can be used to intercept objects modifications and keep dependent entities updated. Since AOP supports proper modularization of crosscutting concerns, the update functionalities are separated from the code of the object. For example, the Observer pattern can be implemented in a modular way by using AOP techniques [41]. Other researchers proposed AOP languages to modularize complex relations among objects [67,62]. AOP integrates well with the imperative style and the mutability of object's state. A point of criticism is that AspectJ-like pointcut-advice models, which dominate AOP design, can potentially break OO modularity. However, it has been shown that pointcuts can be integrated with an event system preserving OO-style modular reasoning [34].

Finally, many limitations of event-based programming hold for AOP, too. Most noticeably, updates must be performed explicitly in the aspect code and dependencies are not automatically tracked as in reactive languages. In addition, composition of reactive behaviors is not easy to obtain, since aspects interactions are complex to master compared to expression composition in functional programming.

4.5 Reactive Collections

Reactive collections define functional dependencies among data structures (often expressed via SQL-like queries). The crucial point is that the framework keeps the dependent structures automatically updated when the basic ones change. Efficient incremental updates have been investigated by the database research community for a long time in the context of the view maintenance problem [12]. More recently, researchers introduced these solutions into programming languages, intercepting the updates via AspectJ [76] or using code generation techniques [66] to trigger the updates on the dependencies. Off-the-shelf libraries include LiveLinq [50] and Glazed Lists [37].

Reactive data structures share some design advantages with reactive languages. Computational dependencies are expressed explicitly, so the application is easier to read: *Queries* describe the functional relations between program entities in a declarative manner. Update functionalities are automatically derived – with little or no additional complexity. The framework is in charge of keeping the functional dependencies specified by the query constantly up to dated. Finally, reactive collections implement efficient update of various reusable operators by using incremental changes and caching to avoid redundant computation.

The main limitation of reactive data structures is that the approach is restricted to a specific domain – changing collections. These frameworks provide out-of-the-box reactive data structures, but they do not support automatic update of other types of objects. Another issue is that reactive collections do not automatically detect all computational dependencies, but only exploit a set of predefined ones. In general, it is not possible to specify a generic expression and leave the framework the responsibility of tracking all the dependencies. For example, the predicate in a `filter` operator (Section 4.1) is not part of the dependencies mechanism, so any change to the predicate remains undetected. Apart from providing custom indexes to speedup certain queries, reactive collections rely on hard-coded non-functional design choices depending on the internal implementation. The programmer cannot fine-tune the update strategy or customize the caching behavior. Finally, most of these frameworks come in a very relational flavor. This further limits their integration with OO languages in which they are usually embedded.

4.6 Constraint-based Languages

Constraint-based languages, like Kaleidoscope [33] integrate constraints among variables into OO languages. The focus of constraint-based languages is on expressing relations among time-changing values in the program. However, this can lead to a rather different programming model compared to traditional OO or functional programming. For example, Kaleidoscope supports constraints that the framework attempts to enforce according to a priority ranking. However, prioritizing constraints introduces a semantics that is likely unexpected for programmers, because some constraints can be never satisfied.

In contrast to reactive languages, that focus on composition of time-changing values, constraint-based languages focus on relations among values. As such, constraint-based languages typically do not support combinators for time-changing values nor integrate time-changing values and events. One-way constraints have proved effective in the scope of graphical interfaces: The Lisp-based Garnet [56] and Amulet [57] graphical toolkits support automatic constraint resolution to relieve the programmer from manual updates of the view. The design of one-way constraints is similar to the basic functionalities of reactive languages as supports the same limitation w.r.t. state and the OO model. Interestingly, Garnet and Amulet also support cyclic constraints. A constraint in a cycle is evaluated at most once. If it is asked to evaluate a second time in the same constraint resolution phase, it simply reevaluates to the previous value.

SuperGlue [53] is an OO language that supports behaviors at the component level. Similarly to constraint-based languages, behaviors in SuperGlue are declarative constraints used to connect (*glue*) software components. A prioritization mechanism similar to the one of Kaleidoscope is also available. A peculiar aspect of SuperGlue is that constraints can include quantification over sets of time-changing values. Quantification is expressed using types to refer to a set of variables. It worth noticing that quantification has been independently explored by different approaches to reactive programming as a mean to decouple software components. Examples include AOP, event-based languages like Ptolemy and constraint-based languages like SuperGlue.

4.7 Synchronous Dataflow Languages

In synchronous dataflow languages like Lucid [63], the program defines a network in which a synchronous signal propagates and triggers the computations in the nodes. Lustre [38,11] and SIGNAL [35] are examples of dataflow languages fostering a declarative style in which program and specification coincide. In those languages absence of cycles is enforced by construction to avoid inconsistencies. Esterel [4] is a similar approach but fosters has a more imperative style. In Esterel, cycles are checked by the compiler and rejected statically. Synchronous dataflow languages address the problem of specifying, programming and verifying reactive real-time systems. They enforce a programming model where the reaction time of the systems is virtually zero (synchrony hypothesis). In practice, the reactive system must be proved fast enough compared to the environment. For this reason, synchronous dataflow languages are usually compiled into easily verifiable models, e.g. finite state machines. Guarantees of real-time and memory-bound execution are provided at the cost of limiting the language expressively. A common example of such limitations is relinquishing higher-order behaviors.

A related family of languages are graphical languages like LabVIEW [46] for data acquisition (e.g. from an external board), signal elaboration and industrial control, and Simulink [17] used for system simulation especially in control theory. The application domain of those languages makes the dataflow computational model particularly suitable and the graphical approach simplifies the use by non programmers. Interestingly, researchers have explored synergies between graphical languages and synchronous dataflow languages. Tripakis *et al.* propose a method to translate Simulink to Lustre [73]. This approach allows model-based development of embedded systems software with Simulink, the *de facto* standard of many industrial applications domains. On the other hand, the advantages of Lustre are preserved, including formal semantics and static analysis, verification and controller synthesis tools.

4.8 Self-adjusting Computation

Self-adjusting computation [2] studies the automatic derivation of incremental programs from batch ones. This solution adopts an algorithmic approach, focusing on reducing the computational complexity of the incremental application. Self-adjusting computation has been introduced in the functional setting [2]. More recent approaches focus on applying similar concepts to imperative languages [1,39] and low-level languages like those used in stack-based abstract machines [40].

In contrast to reactive languages, which focus on explicitly modeling time-changing values using proper language abstractions, self-adjusting computation focuses on deriving an incremental version of a batch algorithm. More generally, self-adjusting computation aims at speeding up algorithms by taking advantage of incrementality. However, self-adjusting computation does not provide new abstractions to express time-changing values. As a result, it is not clear how incremental programs can be integrated into existing applications (e.g. by interfacing time-changing values with events) nor how incremental programs can support modular composition. Instead, reactive languages intentionally enforce a programming model that supports time-changing values and their composition to achieve better design. Despite the different focus, we envisage important synergies between the two approaches. In particular, the solutions explored by self-adjusting computation can be applied to generic computations, solving in an automated way the problem described in Section 2.4. For this reason, this technique can be used to incrementalize the updates performed in reactive languages.

Recent work on self-adjusting computation merges ideas from incremental computing and programming models for Big Data, like MapReduce [18]. Incoop [5] is an incremental MapReduce framework. When Incoop detects changes to the input dataset it obtains the result of the computation through a fine-grained update of previous results. A fundamental feature of Incoop is that it is *transparent* to the user, i.e. the application is not different from a traditional MapReduce interface and all the incrementalization is done under the hood.

4.9 Complex Event Processing

Complex event processing (CEP) is about performing queries over time-changing streams of data. Event streams are combined and correlated to define *complex* events triggered when a correlation condition is satisfied. Typical application scenarios include intrusion detection [16], stock trading [19] and power management [78].

CEP has similarities with databases. Event streams in CEP can be thought as time-changing database tables which are updated every time a new event occurs (i.e. a new entry in the table is added). In contrast to database software, where data change relatively rarely and the user triggers the evaluation of queries over the existing data, in CEP, data change dynamically and the queries must be reactively evaluated upon data arrival. Queries are expressed in terms of time windows to correlate only the events that are included in a given time frame [16]. Because of the similarity between recognizing an event pattern and deciding whether a word belongs to a formal language, the semantics of event correlation is often expressed in terms of finite state automata. This is the case e.g. of SASE [77], Cayuga [19] and TESLA [15].

Similarly to reactive languages, CEP operates on event streams and CEP query languages are usually very declarative. CEP systems support a semantics that includes time, and is in general more expressive than the semantics of event-based languages, like EScala, supporting only event combination. However, in contrast to event-based languages, CEP systems are typically implemented as separate applications accessed through with a proper library. For this reason, CEP query languages are usually specified as strings – a state of the affairs that is similar to databases and SQL queries. As such, they do not take advantage of in-language integration, including safety guarantees from the compiler and integration with other language abstractions, as reactive languages do.

5 A Research Roadmap

In this section, we present a research roadmap for embedding direct support of reactive applications in object-oriented programming languages. The milestones in the roadmap are ordered from the basic ones – which are ongoing, like the integration of reactive abstractions into an event model, to more elaborate ones – which address complex systems that require automatic adaptation. Beside language design, we plan to work on the improvement of the performance of reactive languages. Previous work mostly focused on language abstractions, with less attention to optimization or performance assessment. However, the high overhead of current reactive languages is also among the factors that limit the spreading of this technology.

5.1 Integration with Event-based Programming

Languages with support for event-based programming do make an important step forward in more directly supporting reactive behavior in an imperative object model. As such, they are an ideal starting point for our research. Yet, as argued, they lack the capability of declaratively expressing reactive computations dependent on changing values. In summary, both events and reactive expressions are needed. Events support fine-grained updates of mutable objects. Reactive abstractions capture reactive computations in a compact and declarative way.

Hence, a first step in our plan is to seamlessly integrate reactive abstractions into object-oriented event systems. This goal requires the design of the interface between the reactive abstractions and the abstractions for imperative events, such that they can be treated uniformly in computations and become composable. Such interface is fundamental to support a mixed programming approach and gradual migration of existing software to a more functional and declarative style.

A further step is to integrate reactive abstractions in other aspects of the language. For example, collections must react to changes of the contained elements. It has been shown that changes can be suitably provided to the clients via an event-based interface[1]. Similarly, reactive abstractions can be conveniently used in data structures to model properties which are functionally dependent on other values (e.g. the size or the head of a list, both functionally dependent on the content of the list). The result is a library of *reactivity-enabled* data structures, which expose certain values as reactive abstractions.

This work is currently ongoing in the incarnation of REScala (R*eactive*EScala), a language which integrates the advanced event system of EScala [34] and time-changing values in the style of Scala.React [52]. Our experiments show that with REScala the design of reactive applications is improved according to several metrics, including number of composable abstractions and removed callbacks [69].

5.2 Integration with Object-oriented Design

Reactive languages provide abstractions to represent time-changing values. For simplicity, we assume the Flapjax terminology established in Section 4.1 and we refer to these abstractions as behaviors. Behavior values are bound to expressions that capture the dependencies over other values. For example, in Figure 12, Line 5, the `elapsedB` value is bound to the behavior expression `nowB-clickTmsB`. It is unclear, however, how behaviors should integrate with OO design.

We believe that behaviors should be part of the interface of an object and clients should attach to public behaviors to build complex reactive expressions. Private behaviors should instead model functionally dependent values that are consumed only inside the object. Clearly, object encapsulation should be supported to hide implementation details from clients. These results can be trivially achieved by applying visibility modifies to behaviors. However, the next steps in the integration with OO design require more investigation.

An open question is whether behavior expressions can be reassigned. A negative answer leads to a design more similar to method bodies in most OO languages: They are statically defined at development time and at runtime can only be executed. On

[1] For example, the .NET framework provides the `System.Collections.ObjectModel.ObservableCollection<T>` class which exposes to the clients the `PropertyChanged` and the `CollectionChanged` events.

the contrary, modifiable behaviors imply a design similar to fields that can be accessed by getter and setter methods. In that case, behavior expressions would be changeable. Since behavior expressions capture the dependencies over other application entities, allowing their reassignment introduces a potentially excessive degree of dynamicity, especially if behavior expressions can be reassigned from outside the object. However, in a large application, it can happen that the dependencies of a long-lived component are not known when the component is instantiated and, depending on the evolution of the system, must be assigned during its lifetime.

While in the existing literature behaviors are usually assigned and not modified later, this mostly seems due to accidental circumstances rather than justified by design considerations. First, the use cases provided in literature are mostly small examples in which reassignment is not really needed. Second, the functional flavor of the existing solutions presumably favors single assignment. Third, reassignment complicates the reactive model both from an implementation and a semantic standpoint, so non reassignment has been favored also for the sake of simplicity. As a result, we still lack a broad discussion of these issues that concretely justifies the preference for a model or the other.

Another open issue concerns inheritance. Should it be possible to override a reactive value with a new dependency expression or refer to the overridden one via *super*? Intuitively, this seems desirable, but the consequences on the propagation model need careful investigation as well as the expected benefits. A final consideration is about polymorphism. We envisage a scenario in which reactive entities are late bound – like objects – and the dynamic type of the reactive value captures the dependencies over the other entities of the application.

In summary, while reactive abstractions have been applied in the context of OO languages before, previous approaches focused on reactive fragments that only superficially challenge OO design. As a result, we still lack a systematic investigation of the interaction between OO features and reactive abstractions. A starting point for our work is [44] which focuses on the specific scenario of an application in a functional reactive style interfacing with an OO graphic library.

5.3 Efficient Reactive Expressions

Reactive languages enable to define arbitrary constraints on dependencies between objects and leave the framework shouldering the recomputation of the dependent objects. However, as shown in Section 4.1, current approaches enforce immutability and recompute dependent objects every time, which negatively impacts efficiency. On the other hand, reactive collections (Section 4.5) overcome this problem by applying advanced strategies such as update incrementalization for a predefined set of operators.

Unfortunately, optimizations are provided out-of-the-box for built-in operators and are not at the fingertips of end developers. As discussed in Section 2.4, in certain circumstances, only domain knowledge enables the developer to provide a mechanism that supports incremental updates. Hence, a predefined set of operators is not sufficient.

Motivated by the above observations, a fundamental step in our research roadmap is to design a framework that combines the open approach of reactive languages, which support arbitrary reactive computations, with the efficiency of built-in reactive data structures. This solution overcomes the frustrating state of the art, where efficient reactive data structures and reactive languages are separate worlds. We will follow two lines of research.

As a first step towards this goal, we aim at bringing efficient built-in operators to reactive abstractions. We will provide a variety of efficient operators that seamlessly operate on reactive collections and reactive abstractions. Highly efficient libraries will be designed along the lines of [76,66], but integrated with the abstractions of existing reactive languages. For example, by allowing behavior-like expressions in the predicate of a filter operator or by modeling the result of the reactive operators as behaviors. Predefined operators must cover the most common applicative scenarios such as collections and relational operations.

The second step of research will aim at reconciling the openness of reactive languages with efficient reactive operators. This is only possible if the optimization of generic computations is available to application developers. Optimizing a particular step of a reactive computation process must be handy: Providing a faster version of a reactive computation must be as easy as – say – overriding an existing method.

Our idea is to separate the *creation* of dependent objects, which is performed from scratch, from the *maintenance* of objects. In the default case, both creation and maintenance are accomplished by complete recomputation, i.e. by applying the function that relates basic and derived objects, for example `filter` in the `list2 = list1.filter(x>10)` expression. However, the programmer can provide refinements for the maintenance case. Those refinements can be implemented by imperative algorithms or by taking advantage of domain-specific knowledge to efficiently update dependent objects. In this way, efficient event-based computations that apply the optimization principles well-known from the OO context can be conveniently hidden behind high level operators expressed in a functional style. Reactive objects are then connected by those operators to compose constraints. To further open the framework, the programmer should be able to refine existing operators with a more efficient version when better performance is needed. Finally, late binding can be leveraged to obtain the dynamic selection of the best operator (i.e. the best refinement for a set of types).

A second line of research concerns optimizations that must take into account a broader scope than single operators. For example, considerable performance improvement in relational expressions comes from reordering by anticipating selections and deferring joins. In addition, those optimizations must be performed, at least partially, at runtime, to allow cross-module analysis. Deeply-embedded DSLs come with powerful interfaces to support custom optimizations for DSL expressions embedded into the host languages. For example, in LINQ, the developer is provided with the raw compiler output in the form of an – internally untyped – expression tree. Scala-virtualized [55] employs a similar approach, but fosters more typing guarantees. We will investigate the applicability of these techniques to optimizing reactive expressions. However, these mechanisms are quite low-level. As a result, optimizations are hard to perform and require highly specialized skills. It has been reported that building a LINQ provider for the RavenDB database took more time than building the database [24]. Also, they do not support dynamic optimizations. We will opt for hiding the complexity of those techniques behind higher-level abstractions.

5.4 Propagation Model

To enforce the constraints defined by the user, reactive languages keep a runtime model of the dependencies in the application. Usually, this model is a directed graph in which a change in a node triggers an update over the transitive closure of the dependency relation. Reactive languages mostly enforce a push propagation model in which changes are proactively applied to dependents in the graph [13,54]. However, also lazy models with invalidation of the cached values and on-demand recomputation have been proposed [52]. The propagation of the changes along the graph has a considerable performance impact.

Optimization techniques regarding the propagation model have been already proposed. For example, *lowering* is a technique that applies static analysis to collapse several reactive nodes in the graph into one [9]. As a result, the computation is moved from the reactive model to the usual (and more efficient) call-by-value system. The Yampa FRP framework employs a similar approach but merges the computations at runtime [59]. Based on the above observations, one research direction that we plan to follow concerns optimizations related to the propagation model.

First, alternative graph constructions can be performed that lead to observationally equivalent reactive models. As a result, performance considerations can guide the choice. For example, as noted in [9], always collapsing the computations can lead to poor performance in certain cases. Consider the following code snippet from [9]. A time consuming operation depends on an operation whose output rarely changes. The second operation, instead, depends on a frequently-changing value:

```
(time-consuming-op
  (infrequently-changing-op frequent-emitter))
```

Consider the case in which this code results in three nodes: A source node `frequent-emitter`, an `infrequently-changing-op` node which depends on the first one, and a `time-consuming-op` node, which depends on `infrequently-changing-op`. Every time `frequent-emitter` emits a new value, `infrequently-changing-op` is executed. Since the outcome of `infrequently-changing-op` rarely changes, `time-consuming-op` is executed just a few times. Instead, if `infrequently-changing-op` and `time-consuming-op` are collapsed into the same node, `time-consuming-op` is executed at the same rate `frequent-emitter` changes its value. This effect is even more significant in languages like Scala.React, which apply collapsing of computations as the principal composition mechanism. In summary, collapsing should not be accepted or refused in its entirety and code analysis or dynamic techniques must be applied to detect where each solution leads the best results.

A second aspect of the propagation model that needs further investigation is the choice of a push-based implementation that is adopted by most reactive languages. According to the design space presented in Section 2, this solution favors caching over on-demand recomputation. The choice is motivated by a constraint: A push-based solution is necessary to guarantee that possible side effects in the reaction are really performed. However, change propagation does not always involve side effects. An optimization should explore the space between caching and on-demand recomputation and provide a convenient compromise. For example, reactive data structures dynamically switch to a caching strategy when the requests exceed a threshold [76].

However, this solution is quite simple and does not consider factors like the current machine load. For example, if the load is low, it may be convenient to recompute cached reactive values even if they are rarely requested. On the other hand, with heavy load, this strategy can further degrade the performances without providing significant benefits. More advanced approaches relying on concepts from control systems need to be investigated. The latter have been e.g., explored for parallel data structures to provide the best performance adapting to different machines, configurations and workloads [22].

Finally, update propagation models are common to both reactive languages and event-based systems. The former use these models to propagate updates across dependencies, the latter to trigger dependent events [34]. Since these mechanisms are more and more used in programming languages, an obvious question is if those functionalities should be supported at the VM level. In previous work, starting from similar considerations, we investigated dedicated VM support for AOP [7]. Research work on runtime environments which natively support the concept of reactive memory was recently carried out at the OS level [20]. However, implementing a similar approach in a managed environment which specifically supports propagation of changes across reactive entities is still a research challenge.

A second research direction is optimization by design. This approach is enabled by making the performance implications of the language abstractions explicit and leaving the choice in the hands of the programmer. Current reactive languages focus on expressivity rather than performance. As a result, the programmer has no clear control of how performance is affected by the design choices. Unlike reactive languages, dataflow languages, like Esterel and LUSTRE, intentionally limit the expressive power of the available abstractions to achieve memory and time-bound execution. Attempts to limit expressivity to improve performances have already been done in the reactive languages community. For example, real-time FRP [75] is a time-bound and space-bound subset of FRP. However, real-time FRP is a *closed language* [48], i.e. it is not embedded in a larger general-purpose language, which considerably limits the applicability of this approach.

In summary, programmers face a black-or-white choice: Relinquish performance for expressivity or abdicating abstraction for efficiency. Instead, reactive languages should incorporate reactive primitives that, at the cost of a reduced expressive power, have a high-performance profile. Static analysis or a dedicated static type system can ensure that those primitives are not combined with the rest of the reactive system in a way that cancels the performance improvement.

A starting point is to implement lexically scoped dependencies. In current reactive approaches, reactive dependencies are established dynamically. When, during the evaluation of an expression, a reactive value is found, the value is inserted in the dependency graph. This approach introduces considerable overhead. In fact, the evaluation is slowed down by the process of double-liking dependent values with their dependencies. Similarly, when the value of a node changes, it must be unlinked from the nodes depending on it. This behavior is required to keep the graph updated, since dependencies can change dynamically. For example, the value of the expression `if(a) b else c` depends on either `b` or `c` on the bases of `a`. As a consequence, the structure of the graph is not fixed but must be continuously restructured to capture the current dependencies [52]. In contrast, lexically scoped dependencies are fixed. This results in regions in which the graph structure does not change, avoiding the computations required to keep the graph updated and improving performances.

5.5 Seamless Integration of Reactive Programming

Current reactive programming solutions show different degrees of automation in tracking computational dependencies. In languages like Scala.React, the user specifies time-changing values – signals, in Scala.react terminology – and associates them to a reactive expression. During the evaluation of the reactive expression, when a time-changing value is encountered, it is added to the set of values the signal depends on. Time-changing values are identified by a special type, e.g. `Var[Int]`, which is used by the framework to recognize them and establish the correct dependencies. Flapjax and Fr-Time are more flexible and support *transparent programming* [13]: when a function receives a reactive value as an actual parameter, the function is *transparently* lifted to and equivalent function that produces a time-changing value. Function lifting is adopted in most implementations of functional-reactive programming (e.g. [25]). However, all these approaches require that the programmer identifies the primitive reactive values manually, by assigning them a special type.

The solutions discussed so far have several disadvantages. First, they are error-prone. Developers can forget to mark some dependencies as reactive. In that case, dependent values are not correctly updated when a change occurs. Second, reactivity must be planned in advance by identifying all the sources of reaction. Also, this approach does not support a gradual migration to a reactive style: existing applications require considerable effort to manually identify all the sources of change and wrap them with reactive types. An approach that automatically identifies the values a reactive value depends on can solve the aforementioned issues. In such a solution, when an expression marked as reactive is evaluated, it implicitly adds to its dependencies all the values that are read during the evaluation. When a change in one of those values occurs, the expression is automatically reevaluated.

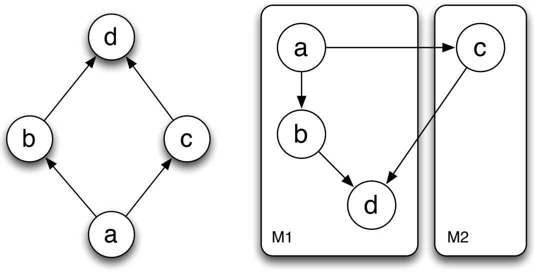
Interestingly, a similar approach has been investigated by Ostermann *et al.* [60] in the context of advanced AOP languages. In [60], the authors present Alpha, a statically typed AOP language that demonstrates the role of an expressive pointcut mechanism in enhancing software modularity. In Alpha, pointcuts are Prolog [72] logic queries. Thanks to the expressive power of the pointcut language, it is possible to define a pointcut that captures all the write operations to the fields that were evaluated during a previous computation. In this way, when one of these values changes, the computation can be triggered again to obtain the new result. However, Alpha is an experimental language based on an ad-hoc interpreter. An open problem is to make such a functionality efficiently available in mainstream languages. A first approach requires to implement a static analyzer that detects all the fields read inside a control flow and proxies write accesses to those values to detect any change.

Beside efficient implementation, seamless integration of reactive programming requires to investigate some methodological aspects. For example, it can be an invaluable

```
a = 1
b := 2*a
c := 3*a
d := b + c

a = 2 // b=4,c=6,d=10

```



(a) (b) (c)

Figure 13. A reactive program (a). The associated dependency graph (b). A possible distribution of the dependency graph over different hosts (c).

resource to refactor existing software to a reactive style. However, if the computation is repeated every time that an input changes, side effects are played again – a behavior that is typically undesired. For this reason, the migration to reactive programming also requires to refactor the application to move side effects out of signal expressions.

5.6 Distribution

Reactive applications include a wide class of distributed systems, like publish-subscribe systems [29], tuple spaces [36] and message-oriented middleware [42]. These systems typically use inversion of control to decouple the components of the application. As a result, they suffer from the shortcomings of the Observer pattern mentioned in Section 4.3. Our research roadmap includes exploring the use of reactive abstractions in the distributed setting. As a first step, we plan to make behaviors available remotely. Remote reactive objects are then similar to remote objects in Java RMI: clients can publish and lookup them through a public registry. In this scenario, the components in the distributed system expose time-changing behaviors to other components and build reactive computations by composing remote behaviors.

A sound design of such system is challenging. Reactive languages like Flapjax, FrTime or Scala.React organize the reactive values in a dependency graph in which each reactive object is connected to the objects it depends on. When a value in the graph changes, the change is transitively propagated to all the dependents to update them. However, the update order is controlled to enforce consistency properties in the propagation of the changes among behaviors. Consider the code in Figure 13(a) where `a`,`b`,`c` and `d` are behaviors and `:=` denotes a constraint among behaviors (not an assignment). When `a` changes, `b`, `c`, and `d` must be recomputed according to the dependency graph in Figure 13(b). However, if the evaluation is in the order `a-b-d-c`, `d` must be recomputed again to reflect the last change of `c`. As a result, the expression that computes `d` is unnecessarily evaluated twice. More seriously, `d` is assigned an intermediate value – reflecting the change of `b` but not yet of `c` – that is *spurious* and only due to the propagation mechanism. Temporary inconsistencies of the propagation system during the propagation are referred to as *glitches*. Reactive languages typically enforce the absence of glitches (glitch-freedom) by updating the graph in topological order [51,13]. For example, the update order `b-c-d` guarantees glitch freedom.

Current reactive languages enforce glitch-freedom only on single hosts, but inconsistencies can still arise when inter-host communication is involved. Ensuring glitch-freedom in a distributed setting is not easy because the dependency graph is distributed over different machines. In the case of Figure 13(c) the graph spans over the machines M1 and M2. In this case, the propagation order is likely to be `a-b-d-c-d` because propagation on a single machine is probably faster than the one over the network, i.e., the message `a->b` is delivered before `a->c`. Applying the algorithm used locally to the distributed environment would require to pause the computation in all hosts and guide the change propagation across the distributed graph according to the topological order. However, this solution is clearly inefficient and requires a centralized manager that schedules the updates across the distributed system.

We envisage the following research directions. A first step consists of avoiding sequential updates in the graph. We plan to develop an algorithm for distributed reactive programming that maximizes the degree of parallelism among the hosts in the recomputation of the dependent nodes: independent nodes can be updated in parallel. This problem concerns parallelism *in the same propagation phase* along the graph. Another aspect concerns the parallelization of several update phases initiated by different hosts. A starting point is to consider the reactive values in the distributed reactive system as shared values in a concurrent system. An implementation can leverage distributed software transactional memories [71], which have been proved to efficiently scale up to large-scale clusters [6].

Web applications are a special case of distributed systems where only a client and a server interact. We expect that the implementation of a glitch-free propagation system can be optimized leveraging this simplification. For example, Javascript client-side code is single threaded, which greatly reduces the conflicts that can show up in the access to shared reactive objects.

Another research direction concerns the integration of the middleware that provides reactive programming with existing enterprise containers. In those systems, developers adopt conventions on the structure of certain objects and the container automatically enforces certain properties for them. For example, Enterprise Java Beans are automatically persisted by the container. We envisage a scenario in which, in a similar way, certain objects are automatically updated by the container thanks to the changes that propagate across the distributed dependency system.

Finally, an open research question is to find a balance in the trade-off between the properties enforced by the propagation system and performance. For example, in an environment where communications are not reliable, it can be reasonable to increase the efficiency of the propagation system at the cost of introducing glitches. An interesting work in this direction is in the implementation of distributed reactive programming by Carreton *et. al.* [10] which focuses on mobile networks in the context of pervasive systems. In that framework, since the network can be extremely unreliable and hosts can experience temporary lack of connectivity, loose coupling is favored over safety and the propagation system in not glitch-free by design.

5.7 Evaluation

To make sure that our research leads to concrete results, we plan to continuously evaluate how it progresses. However, evaluating language design is not easy, because design quality is hard to capture with synthetic metrics and design choices have long term effects which are hard to predict. For example, the impact of programmers' experience or the maintainability of large systems can be evaluated only when a considerable number of projects have been developed. As a consequence, we believe that a priori reasoning and careful analysis of the available options remain fundamental steps [47]. Nevertheless, where applicable, we plan to perform objective evaluations of our results.

Performance can be evaluated effectively by running benchmarks. For example, former studies compared the performance of OO programming (Java) with the mixed OO-functional style (Scala) in the context of parallel applications and multicore environments [61]. We believe that similar experiments can evaluate the performance impact of reactive abstractions compared to the traditional solutions for reactive applications.

Other aspects of the language design can be evaluated by using software metrics. Common metrics include coupling and cohesion, lines of code, number of operations and others [30]. We plan to adopt this approach to evaluate our design choices in the advanced state of our research, when studies can comprise several artifacts. Other researchers already used metrics to validate language design choices in reactive applications. Recently, the combined use of synthetic metrics and manual inspection (to investigate specific issues) was successfully applied to evaluate event quantification in software product lines [21].

6 Acknowledgments

This work is supported in part by the European Research Council, grant No. 321217 and by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under grant No. 16BY1206E "Sinnodium".

References

1. U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 309–322, New York, NY, USA, 2008. ACM.
2. U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, Nov. 2006.
3. Android developers web site. http://developer.android.com/index.html.
4. G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87 – 152, 1992.
5. P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: MapReduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA, 2011. ACM.
6. R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 247–258, New York, NY, USA, 2008. ACM.

7. C. Bockisch, S. Kanthak, M. Haupt, M. Arnold, and M. Mezini. Efficient control flow quantification. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 125–138, New York, NY, USA, 2006. ACM.
8. E. Bodden, R. Shaikh, and L. Hendren. Relational aspects as tracematches. In *Proceedings of the 7th international conference on Aspect-oriented software development*, AOSD '08, pages 84–95, New York, NY, USA, 2008. ACM.
9. K. Burchett, G. H. Cooper, and S. Krishnamurthi. Lowering: a static optimization technique for transparent functional reactivity. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '07, pages 71–80, New York, NY, USA, 2007. ACM.
10. A. L. Carreton, S. Mostinckx, T. Van Cutsem, and W. De Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In *Proceedings of the 48th international conference on Objects, models, components, patterns*, TOOLS'10, pages 41–60, Berlin, Heidelberg, 2010. Springer-Verlag.
11. P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.
12. S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 577–589, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.
13. G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European conference on Programming Languages and Systems*, ESOP'06, pages 294–308, Berlin, Heidelberg, 2006. Springer-Verlag.
14. Microsoft Corporation. C# language specification. version 3.0. http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx, 2007.
15. G. Cugola and A. Margara. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 50–61, New York, NY, USA, 2010. ACM.
16. G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.
17. J. B. Dabney and T. L. Harman. *Mastering SIMULINK 4*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.
18. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
19. A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proceedings of the 10th international conference on Advances in Database Technology*, EDBT'06, pages 627–644, Berlin, Heidelberg, 2006. Springer-Verlag.
20. C. Demetrescu, I. Finocchi, and A. Ribichini. Reactive imperative programming with dataflow constraints. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 407–426, New York, NY, USA, 2011. ACM.
21. R. Dyer, H. Rajan, and Y. Cai. An exploratory study of the design impact of language features for aspect-oriented interfaces. In *11th International Conference on Aspect-Oriented Software Development*, AOSD '12, March 2012.
22. J. Eastep, D. Wingate, and A. Agarwal. Smart data structures: an online machine learning approach to multicore data structures. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 11–20, New York, NY, USA, 2011. ACM.
23. Eclipse IDE Web site. http://www.eclipse.org/.

24. O. Eini. The pain of implementing LINQ providers. *Queue*, 9(7):10:10–10:22, July 2011.
25. C. Elliott. Functional implementations of continuous modeled animation. In *Proceedings of the 10th International Symposium on Principles of Declarative Programming*, PLILP '98/ALP '98, pages 284–299, London, UK, 1998. Springer-Verlag.
26. C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM.
27. C. M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. ACM.
28. P. Eugster and K. R. Jayaram. EventJava: An extension of Java for event correlation. In *Proceedings of the 23rd European Conference on ECOOP 2009 – Object-Oriented Programming*, ECOOP '09, pages 570–594, Berlin, Heidelberg, 2009. Springer-Verlag.
29. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/-subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.
30. E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 261–270, New York, NY, USA, 2008. ACM.
31. N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.
32. FreeCol game Web site. http://www.freecol.org/.
33. B. N. Freeman-Benson. Kaleidoscope: mixing objects, constraints, and imperative programming. OOPSLA/ECOOP '90, pages 77–88, New York, NY, USA, 1990. ACM.
34. V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. EScala: modular event-driven object interactions in Scala. AOSD '11, pages 227–240, New York, NY, USA, 2011. ACM.
35. T. Gautier, P. Le Guernic, and L. Besnard. SIGNAL: A declarative language for synchronous programming of real-time systems. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 257–277, London, UK, 1987. Springer-Verlag.
36. D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, Jan. 1985.
37. Glazed Lists Web site. http://www.glazedlists.com/.
38. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305 –1320, sep 1991.
39. M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: a C-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 25–37, New York, NY, USA, 2009. ACM.
40. M. A. Hammer, G. Neis, Y. Chen, and U. A. Acar. Self-adjusting stack machines. In *Proceedings of the 2011 ACM international conference on Object-oriented programming systems languages and applications*, OOPSLA '11, pages 753–772, New York, NY, USA, 2011. ACM.
41. J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 161–173, New York, NY, USA, 2002. ACM.
42. M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java message service. *Sun Microsystems Inc., Santa Clara, CA*, 2002.

43. P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.
44. D. Ignatoff, G. H. Cooper, and S. Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *FLOPS*, pages 259–276, 2006.
45. JMeter developers Web site. http://jakarta.apache.org/jmeter/index.html.
46. G. W. Johnson. *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*. McGraw-Hill School Education Group, 2nd edition, 1997.
47. G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 195–213, Berlin, Heidelberg, 2005. Springer-Verlag.
48. R. B. Kieburtz. Implementing closed domain-specific languages. In *Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, SAIG '00, pages 1–2, London, UK, UK, 2000. Springer-Verlag.
49. N. R. Krishnaswami, N. Benton, and J. Hoffmann. Higher-order functional reactive programming in bounded space. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*, POPL '12, pages 45–58, New York, NY, USA, 2012. ACM.
50. LiveLinq Web site. http://www.componentone.com/SuperProducts/LiveLinq/.
51. I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.
52. I. Maier, T. Rompf, and M. Odersky. Deprecating the Observer Pattern. Technical report, 2010.
53. S. McDirmid and W. C. Hsieh. SuperGlue: Component programming with object-oriented signals. In D. Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 206–229. Springer, 2006.
54. 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 *Proceeding of the 24th ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM.
55. A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 117–120, New York, NY, USA, 2012. ACM.
56. B. A. Myers, D. A. Giuse, R. B. Dannenberg, D. S. Kosbie, E. Pervin, A. Mickish, B. V. Zanden, and P. Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, Nov. 1990.
57. B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferrency, 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, 1997.
58. R. Newton, G. Morrisett, and M. Welsh. The Regiment Macroprogramming System. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 489–498, 2007.
59. H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA, 2002. ACM.
60. K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 214–240, Berlin, Heidelberg, 2005. Springer-Verlag.

61. V. Pankratius, F. Schmidt, and G. Garreton. Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java. In *Software Engineering (ICSE), 2012 34th International Conference on*, ICSE '12, pages 123 –133, 2012.
62. D. J. Pearce and J. Noble. Relationship aspects. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 75–86, New York, NY, USA, 2006. ACM.
63. M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006.
64. H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In J. Vitek, editor, *ECOOP 2008 Paphos, Cyprus*, volume 5142 of *LNCS*, pages 155–179, Berlin, July 2008. Springer-Verlag.
65. http://www.stg.tu-darmstadt.de/media/st/publications/oo_reactive_report.pdf.
66. T. Rothamel and Y. A. Liu. Generating incremental implementations of object-set queries. In *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, pages 55–66, New York, NY, USA, 2008. ACM.
67. K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 16–25, New York, NY, USA, 2004. ACM.
68. G. Salvaneschi, J. Drechsler, and M. Mezini. Towards distributed reactive programming. In R. Nicola and C. Julien, editors, *Coordination Models and Languages*, volume 7890 of *Lecture Notes in Computer Science*, pages 226–235. Springer Berlin Heidelberg, 2013.
69. G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Aspect-Oriented Software Development*, AOSD '14, New York, NY, USA, Accepted for publication, 2014. ACM.
70. G. Salvaneschi and M. Mezini. Reactive behavior in object-oriented applications: an analysis and a research roadmap. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, AOSD '13, pages 37–48, New York, NY, USA, 2013. ACM.
71. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
72. L. Sterling and E. Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1994.
73. S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, Nov. 2005.
74. J. M. Van Ham, G. Salvaneschi, M. Mezini, and J. Noyé. JEScala: Modular coordination with declarative events and joins. In *Proceedings of the 13th International Conference on Aspect-Oriented Software Development*, AOSD '14, New York, NY, USA, Accepted for publication, 2014. ACM.
75. Z. Wan, W. Taha, and P. Hudak. Real-time frp. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 146–156, New York, NY, USA, 2001. ACM.
76. D. Willis, D. J. Pearce, and J. Noble. Caching and incrementalisation in the Java query language. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 1–18, New York, NY, USA, 2008. ACM.
77. E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 407–418, 2006.

78. Y. Xiao, W. Li, M. Siekkinen, P. Savolainen, A. Yla-Jaaski, and P. Hui. Power management for wireless data transmission using complex event processing. *IEEE Trans. Comput.*, 61(12):1765–1777, Dec. 2012.