JOT
JOURNAL OF
OBJECT TECHNOLOGY

Extensible Tooling for Reactive Programming Based on Active Expressions

Stefan Ramson*, **Markus Brand***, **Jens Lincke***, **and Robert Hirschfeld***
*Hasso Plattner Institute, University of Potsdam, Germany

ABSTRACT Reactive programming uses dedicated *language concepts* such as signals, data bindings, and constraints, so developers can better express behavior that is triggered by data changes and user interactions. As applications also contain aspects that cannot be easily expressed through reactive programming, reactive concepts are often integrated into more generally applicable imperative programming languages. Although such language integrations are readily available, working on reactive code with tools designed for imperative code is hard, because without dedicated tool support implementation details may leak unintentionally. There are special tools for reactive programming available, however, they are expensive to make. Further, a tool typically supports only a single language concept and cannot be applied to others even though they build on similar ideas. Consequently, control flow or data flow cannot be followed between concepts.

We propose to leverage the commonalities found in reactive programming concepts to create reusable tool components for data gathering and visualization. To do so we create a toolset working on a generalization of reactive programming concepts, *Active Expressions*. By building upon the generic tool components, tool developers can create tool support for specific reactive concepts. Furthermore, multiple reactive concepts and their potentially complex interaction can be explored in one shared environment. We implemented the approach in the Lively4 Web-based JavaScript development environment using its Active Expression framework. Our toolset gathers relevant data about the reactive system and visualizes it using *code annotations*, an *overview tree*, an *event timeline*, and a *dependency graph*. We evaluate the reusability of this toolset by adapting it to two more concepts: signals, and implicit layer activation, known from context-oriented programming. We found that most of the functionality provided by the toolset can be reused, thus, reducing the implementation effort. Further, we show that multiple reactive concepts can be supported by the same common toolset. Programmers can use and debug multiple different reactive concepts simultaneously, without requiring new tools for each one. For future work, we believe our common toolset provides a starting point for researching the interplay between multiple reactive programming concepts.

KEYWORDS Programming Tools, Development Environments, Reactive Programming, Active Expressions, Lively Kernel

1. Introduction

The reactive programming (RP) paradigm (Bainomugisha et al. 2013) established itself as a viable method for programming event-driven and interactive applications by relieving programmers from manually updating outputs when inputs change. RP

JOT reference format:
Stefan Ramson, Markus Brand, Jens Lincke, and Robert Hirschfeld. *Extensible Tooling for Reactive Programming Based on Active Expressions*. Journal of Object Technology. Vol. 23, No. 1, 2024.
http://dx.doi.org/10.5381/jot.2024.23.1.a4

Parts of this paper covering “Explicit Tool Support for Implicit Layer Activation” were previously presented at“COP’22, the 14th ACM International Workshop on Context-Oriented Programming and Advanced Modularity” (Brand et al. 2022).

has many manifestations, such as signals, data bindings, and constraints, which we will henceforth refer to as *reactive programming concepts*. These concepts overcome well-known issues of other change detection mechanisms, such as the observer pattern (Gamma et al. 1995), by providing dedicated abstractions for detecting and responding to change, including external user inputs or internal state changes.

Compared to imperative programming, the declarative style of RP eases expressing data dependencies and cause-action relationships in code. This effect becomes apparent by looking at an example. Say we want to express a data dependency, where the variable `sum` should always contain the sum of all values in the array `a`. Using a signal, a common RP concept, allows us to express this desired behavior as a one-liner:

```
signal: sum = a.reduce((acc, curr) => acc + curr)
```

This signal will automatically recalculate the value of `sum`, whenever the array changes. In contrast, an imperative implementation has to manually update the sum at all places where the array was changed. This manual task costs the programmer time, complicates the code, and can easily lead to bugs when forgotten (Salvaneschi & Mezini 2016).

While being inferior to RP for some use cases, imperative programming is still the most common programming paradigm. One of the reasons for this is that imperative programming is conceptually close to the way the hardware executes the code. Due to this direct mapping, many tasks are most easily expressed using imperative programming. To get the best of both worlds, reactive programming concepts are often embedded in an imperative programming environment (Van den Vonder et al. 2020). By extending the available linguistic repertoire, this integration allows programmers to choose the paradigm best suited for their current use case.

While language integrations are readily available, tool support lags behind. One primary reason is that debugging tools designed for these imperative environments are unable to adequately capture the declarative nature of RP. Coming back to the sum-of-an-array example, any code location that changes the contents of the array will also trigger a re-computation of the `sum` variable. Debugging this behavior by stepping through the code with a symbolic debugger would cause it to jump from a location where the array is changed into the implementation of the reactive system which internally recomputes the `sum`. A symbolic debugger involuntarily reveals implementation details at run-time. As a result, the carefully constructed abstraction in static source code does not match the available dynamic information, necessitating additional mapping steps that impede comprehending the data dependencies. These inadequate tools thus hinder programmers from embracing RP concepts and make them resort to a less concise and more error-prone, albeit easier to debug, imperative implementation.

To not lose the advantages of RP, dedicated debugging tools for RP are required. These tools need to be able to depict the reactive system and connect the imperative and reactive worlds. While first promising results for dedicated RP tooling exist (Banken et al. 2018; Salvaneschi & Mezini 2016), it is only available for a small subset of reactive programming concepts, and the tools are not frequently used in practice (Alabor & Stolze 2020). In conclusion, the lack of proper tool support makes it hard for many programmers to embrace RP concepts and can lead to frustration for those who do. To tackle this absence of proper tool support, we identify the lack of reusable components for RP debugging tools as one main cause that we want to address. Without reusable components, tools have to be implemented from scratch for every new RP concept. This increased effort can cause developers of RP concepts to not provide tools in the first place.

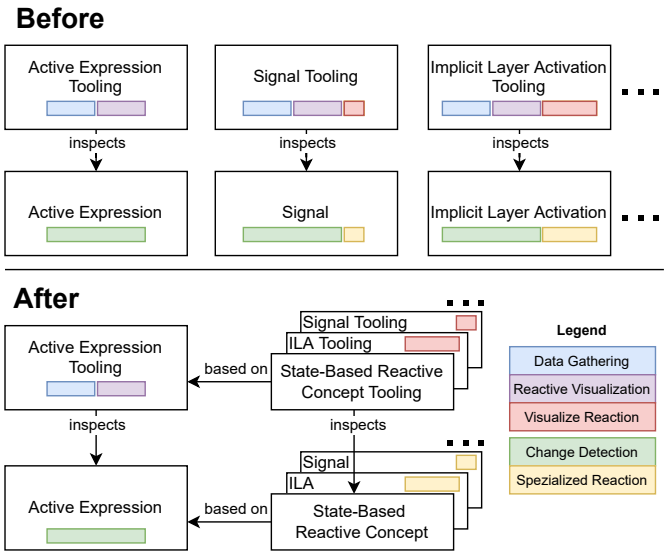

Figure 1 Overview of how Active Expressions and their tooling can be used to implement State-Based Reactive Concepts and their tooling

To overcome this issue, we propose to leverage commonalities found in many RP concepts to extract shared components like visualizations or data-gathering methods. These shared components could provide a reusable basis for new tools, reducing the overhead of implementing proper tool support. By utilizing this reusable basis, RP concept developers can focus on the required debugging functionality specific to their RP concept and do not have to worry about common RP debugging functionality, like highlighting the interface between the reactive and imperative worlds.

Due to this reduced overhead, debugging support can easily be implemented by the RP concept developers from the beginning. This co-development holds multiple potential benefits. To begin with, debugging tools can be provided to users from the very beginning, easing the usage of the RP concept as described above. Moreover, having tools early on can help RP concept developers validate that the RP concept behaves the way they intended and can help build a deeper understanding of the RP concept. Further, different RP concepts are commonly only used in isolation and the interaction between them hasn't been thoroughly researched yet. Basing the debugging tools on shared components enables a common toolset, which offers the chance to investigate this interaction between multiple RP concepts. In conclusion, extracting shared components for common functionality eases the implementation of new tools, allows for

co-development of the RP concept and its tools, and enables a shared toolset that can help investigate the interaction between multiple RP concepts.

As for the commonality we want to leverage for shared components, we choose a common change detection mechanism found in many RP concepts: detecting changes in the evaluation result of an expression (Ramson & Hirschfeld 2017). This mechanism automatically detects the dependencies of the expression, i.e. the variables that might change its evaluation result. Whenever a variable is updated, the mechanism automatically finds all expressions that depend on it, checks if their evaluation result changed, and triggers their concept-specific reactions. We call the subset of RP concepts with this change detection mechanism State-Based Reactive Concepts (SBRCs). As we want to supply debugging functionality for all SBRCs, the most primitive one, the Active Expression (AE), is of special interest to us. An AE reacts to a detected change by executing arbitrary callbacks. By specifying these arbitrary callbacks to more concrete domain-specific reactions, AEs can be used as a basic building block to implement other SBRCs (Ramson & Hirschfeld 2017).

Analogously, we claim that the tooling for AEs can be reused as a basic building block for tooling for other SBRCs. To support this claim, we provide debugging tools for AEs and extend them to also support more specialized SBRCs, like signals and implicit layer activation (ILA) (Brand et al. 2022) (see figure 1). This toolset is depicted in figure 2 and consists of *code annotations*, an *overview tree*, an *event timeline*, and a *dependency graph*. All of these tools can be used to simultaneously depict multiple different reactive concepts and their interactions.

We show that the toolset can be adapted to support additional concepts while only requiring changes related to the specific reaction behavior of these concepts. This, AEs as well as our debugging toolset and data gathering methods provide a reusable basis for developers of a new concept to simultaneously create debugging tools as depicted in figure 1.

We make the following contributions:

- The design of an AE debugging toolset is based on common challenges that occur when working with AEs.
- An exemplary adaption of the aforementioned toolset for two SBRCs: signals and ILA.
- An implementation of the aforementioned tools as well as their integration into the Lively4 system (Lincke et al. 2017).
- An evaluation of the reusability of the AE toolset as well as the usability and performance of both, the basic and adapted tools.

The remainder of this paper is organized as follows. Section 2 provides necessary background information. Section 3 describes the AEs debugging toolset. Section 4 identifies and executes on necessary adaptations of the toolset to support other SBRCs. Section 5 discusses implementation details for collecting the required debugging data and visualizing it in the tools. Section 6 evaluates the reusability, usability, and performance of the provided toolset. Finally, section 7 outlines possible future research opportunities and concludes.

2. Background and Motivation

RP has a long history, in which a multitude of different approaches have been produced. Survey papers (Bainomugisha et al. 2013; Johnston et al. 2004; Benveniste et al. 2003) summarize this long research history by classifying the reactive approaches and extracting properties that constitute a taxonomy. Two distinguishing features of reactive programming are behaviors and events (Bainomugisha et al. 2013). Behaviors (sometimes also called signals) are first-class abstractions of continuous time-varying values. Events on the other hand are values that occur at discrete points in time. Most reactive programming languages and concepts define reactive primitives (Bainomugisha et al. 2013), which include both these features and are used as basic abstractions to help express reactive behavior (Elliott & Hudak 1997).

2.1. Dependency Graphs

Graph visualizations similar to data flow graphs (Davis & Keller 1982) are commonly used to depict reactive primitives and how they interact (Bainomugisha et al. 2013; Salvaneschi & Mezini 2016; Banken et al. 2018; Salvaneschi et al. 2014). As an example, consider a temperature measuring application that displays the apparent temperature t_A in degrees Celsius. The apparent temperature depends on the actual temperature t and the relative humidity h[1] using the formula $t_A = t + (h - 0.3) * t * 0.25$. Whenever the sensor values t or h change, we should update the apparent temperature accordingly. The *dependency graph* in figure 3 shows the desired relation between the apparent temperature t_A and its dependencies t and h. Nodes highlighted in red represent objects in the reactive system. Dependencies and dependents of a node are all its highlighted ancestors and descendants, respectively. Note that most RP approaches assume these graphs to be a directed acyclic graph (DAG) (Bainomugisha et al. 2013).

Note that the degree of information depicted in a dependency graph may vary depending on the use case. As an example, figure 3 provides a detailed view of how data flows through various computation nodes. Such detailed dependency graphs can grow quickly, especially when analyzing the entire graph of a complex system. When systems grow bigger, it is often more practical to use a collapsed view as shown in figure 4. This condensed view allows developers to focus only on values and dependencies involved. By adapting the granularity and focus of the visualization, developers can align dependency graphs to their current needs and level of understanding.

Dynamic and Static Another aspect to consider is that some frameworks allow the dependency graph of a reactive primitive to be dynamic, i.e. it changes its topology at run-time. Consider the example `x = a ? b : c` using the ternary operator. When `a` is true, `b` is relevant for the value of `x`, while `c` is not. This swaps when `a` becomes false. There are two ways to represent this in the dependency graph. In the static method, `b` and `c` are always dependencies of `x`, even though one of them is always irrelevant at any given moment. In the dynamic method, the graph can change over time and always contains the precise

[1] ranging from 0 to 1

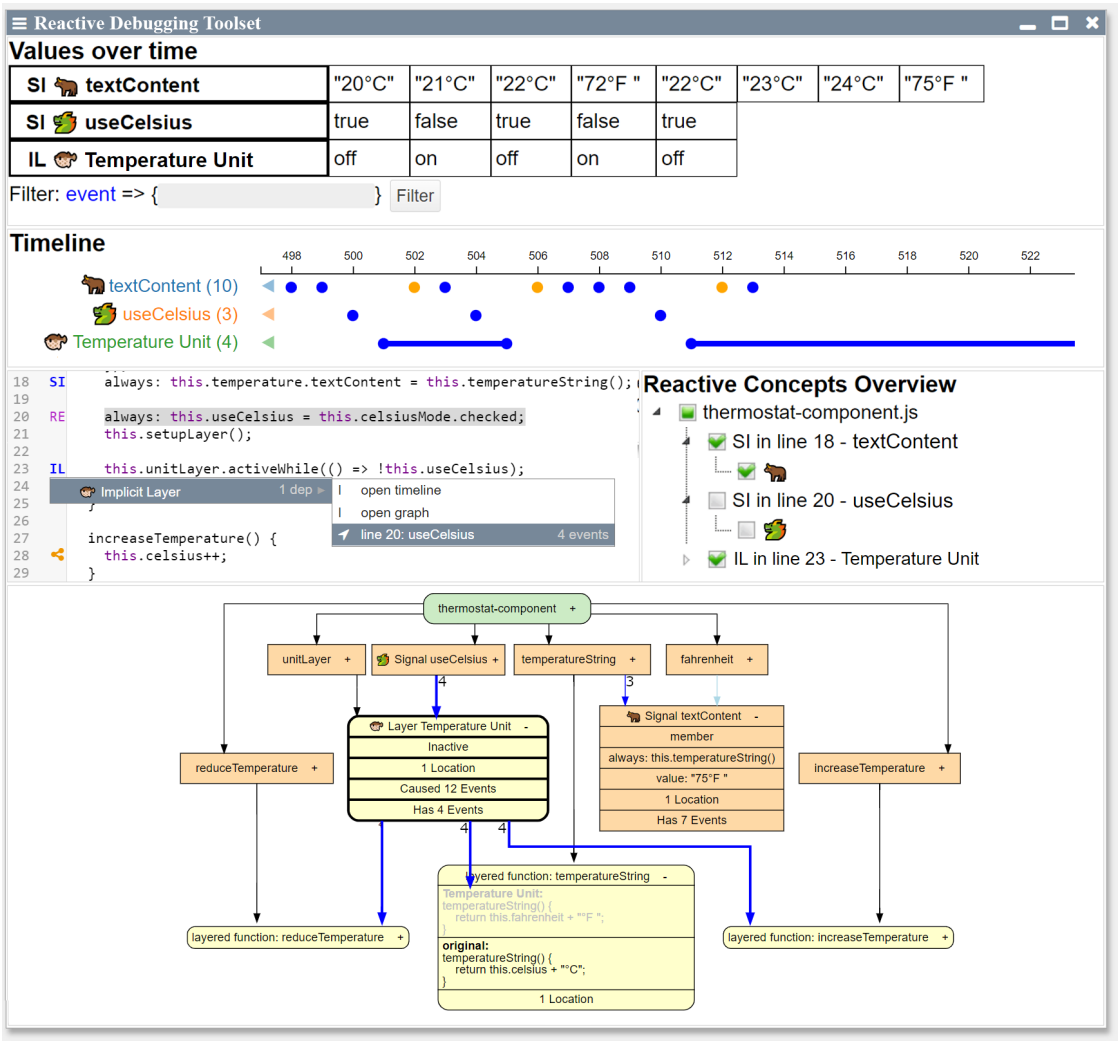

Figure 2 Overview of the toolset depicting signals (SI) and implicit layer activations (IL) simultaneously

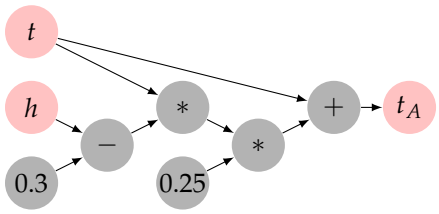

Figure 3 A detailed dependency graph of the apparent temperature application

objects that can change the expression at any given time. The dynamic method, therefore, contains fewer false positives when searching for variables that can influence the calculation of a reactive primitive at any given time.

2.2. Active Expressions

The AE (Ramson & Hirschfeld 2017) is a reactive primitive that automatically detects changes in the return value of an expression. When a change occurs, previously registered callbacks are automatically called with the new value of the expression.

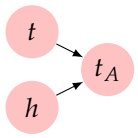

Figure 4 A collapsed dependency graph of the apparent temperature application

```
1  aexpr(expression).onChange(callback);
```

Listing 1 Most basic form of an Active Expression

Listing 1 displays the syntax of this concept in its original implementation in the Lively4 system in JavaScript.

AEs are designed to ease the detection of state changes while integrating well into existing object-oriented programming (OOP) languages. To achieve this effortless integration, every variable, including local, global, and member variables, that is used in an AE is automatically used as a dependency for this AE, without the need to manually mark it as a dependency.

```
val t = Var(0)
val h = Var(0)
val tA = Signal{t() + (h() - 0.3) * t() * 0.25}
```

Listing 3 Example of wrapping objects in ReScala

```
let t = 0;
let h = 0;
aexpr(() => t + (h - 0.3) * t * 0.25)
  .onChange(tA => println("Apparent Temperature: " + tA));

readTemperatureFromSensor(val) {
  t = val;
}
readHumidityFromSensor(val) {
  h = val;
}
```

Listing 2 Active Expression version of a reactive temperature sensor application

The fact that AEs support arbitrary expressions with arbitrary objects has a significant impact on the dependency graph. While all nodes in the graph represent a reactive primitive in most other reactive frameworks, most nodes in the AE's graph are just normal objects. These objects do not depend on each other directly, but AE nodes depend on the objects and can write other objects in their callback functions. This new bipartite dependency graph for the temperature sensor application of listing 2 is depicted in figure 5.

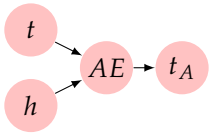

Figure 5 Active Expression version of the dependency graph of the apparent temperature calculator

The AE reactive programming concept *implicitly detects dependencies*, which can *dynamically change*, from an arbitrary expression. AEs *react to changes* in the result of the expression by executing an arbitrary callback. Consequently, their dependency graphs are not restricted. Further, the AEs concept emphasizes a seamless *integration with object-oriented languages*.

Implicit Dependency Detection A reactive system can achieve automatic detection of change in several ways. While repeatedly reevaluating the reactive primitive, also known as polling, is possible, it can quickly become inefficient and makes it hard to guarantee an appropriate evaluation order. Therefore, most reactive systems require explicit knowledge about the dependency graph to trigger the correct reactive behavior when a dependency changes. A common way of obtaining this knowledge is to wrap objects into specialized meta objects as shown in Listing 3 for ReScala (Salvaneschi et al. 2014). However, meta objects are incompatible to ordinary functions and operations. To overcome this incompatibility, the process of lifting converts these functions and operators to versions that support said meta objects (Bainomugisha et al. 2013). AEs differ from most other reactive systems in that they do not require dependencies of a reactive primitive to be wrapped by an object. Instead of using a lifting strategy, AEs use dynamic analysis to determine dependencies and their write accesses. To achieve this, AEs assume every variable that is currently used in the expression to be a dependency. Unlike other frameworks, the decision about which objects should be dependencies is, therefore, the responsibility of the system rather than the user. Since objects in the AE system are reactive without being wrapped by a dependency class, the interaction with them remains unchanged and they can be combined in arbitrarily complex ways, including code branching and function calls, by using arbitrary JavaScript code.

A common source of bugs in reactive applications is forgetting to establish all required dependencies (Salvaneschi & Mezini 2016). While this source of bugs is eliminated by the AE system, the automatic declaration of dependencies may also lead to bugs, e.g. if the user wants to reuse a variable for another purpose and forgets that it is used in an AE. Note the inherent trade-off: while automatic detection of dependencies eliminates the problem of missing dependencies and takes that responsibility from the user, it also takes away control from the user and might introduce new bugs.

Dynamic Change As with most other reactive approaches, the dependencies of an AE can dynamically change over time. In listing 4 `mode` and `a` are dependencies of the AE at the beginning. When `mode` changes in line 6, `a` is no longer a dependency for the AE but `b` becomes one. In other words, only the objects used in the last evaluation of the expression are its current dependencies. Changing dependencies also implies that the dependency graph of an AE changes over time.

```
let mode = true;
let a = 5;
let b = 10;
aexpr(() => mode ? a : b)
  .onChange((value) => /* Reaction */);
mode = false;
```

Listing 4 An Active Expression with changing dependencies

An alternative to this dynamic approach is to statically assume `mode`, `a`, and `b` to be dependencies at all times. Detecting the dependencies in this way requires static analysis, as it is not guaranteed that a given execution of an expression uses all variables it could possibly use. However, a complete static analysis is very complicated with unrestricted JavaScript code, as expressions like `obj["TEST".toLower()]` can hide property names.

Types of Reactions In addition to how they detect change, reactive concepts also differ in the way they react to a detected change. While section 2.3 describes a multitude of specialized reactions, AEs act as a basic building block to help implement these specialized reactive concepts. Thus, AEs themselves provide a generic reaction: invoking associated callbacks every time a change occurs. As these callbacks can be arbitrary JavaScript code, it is hard to predict the effects of a callback before its execution. This also makes it hard to determine the full dependency graph. As seen in figure 6, the AE system is only aware of the connections from dependencies to their AEs and not from the AEs to the objects they write.

```
1  ae1 = aexpr(() => a).onChange(val => b = 2 * val)
2  ae2 = aexpr(() => b).onChange(val => c = 2 * val)
```

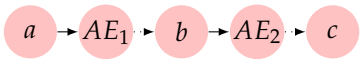

Figure 6 Example of an Active Expression writing a variable that triggers another Active Expression and the dependency graph of this example. The Active Expression system is unaware of the dashed connections.

Unrestricted Dependency Graph The most common restriction to the dependency graph is to enforce it to always be a directed acyclic graph (DAG) and therefore cycle-free. This restriction guarantees that a topological order exists. Evaluating behaviors in this order guarantees that all dependencies are up-to-date when calculating the new value of a behavior, avoiding glitches (Bainomugisha et al. 2013). However, this property also restricts the possible use cases of the system: constraint systems (Felgentreff et al. 2014) and bidirectional data bindings (Weiher & Hirschfeld 2016) inherently contain circular dependencies. Since AEs try to be a basic building block for a wide variety of reactive concepts, which may depend on a specific dependency graph form, AEs do not restrict the dependency graph themselves. Instead, AEs rely on specialized implementations to enforce these restrictions, if deemed necessary.

Object-Oriented Language Integration Frameworks like REScala or AEs focus on reducing the gap between the reactive system and the object-oriented environment around it. They, therefore, provide the user with abstractions that can detect and react to changes, without reducing the available language features. Signal expressions in REScala and even more so AEs are as close as possible to ordinary Scala or JavaScript code, respectively. In contrast, many frameworks like ReactiveX (Meijer 2010), Flapjax (Meyerovich et al. 2009), or Frappé (Courtney 2001) focus on transforming data using classic functional programming functions like `map`, `filter` or functions specialized for data streams that merge or combine multiple streams. These explicit operators provide a simple foundation to insert hooks required to detect dependencies. While explicit reactive approaches are easier to implement, a more object-oriented approach offers better integration with existing code and thus omits the need for time-consuming refactorings (Köhler & Salvaneschi 2019).

2.3. State-Based Reactive Concepts

Reactive concepts are often implemented as language extensions or libraries for various programming languages. Examples are shown in listing 5. The reactions in these examples contain a variety of use cases like writing to a variable, manipulating the document object model (DOM), or triggering a constraint solver. All concepts share that they react to changes in the return value of an expression. Concepts that share this property are called State-Based Reactive Concepts (SBRCs), which are originally defined as "all reactive concepts in which dependencies are specified implicitly as expressions over program

```
1  // Active Expression: Callback (JS)
2  aexpr(expression() -> any)
3    .onChange((value) => /* Reaction */);
4
5  // REScala Signal: Assign variable (Scala)
6  val s = Signal{expression() -> any}
7
8  // Flapjax: DOM Manipulation (JS)
9  insertValueB(expression() -> any, DOMElement, Field);
10
11 // Implicit Layer Activation (JS)
12 layer.activeWhile(expression() -> bool)
13
14 // Reactive Object Query (JS)
15 select(Class, expression(Object) -> bool) -> View
16
17 // Constraints (JS)
18 constraint: expression() -> bool
```

Listing 5 A summary of the basic form of different reactive concepts highlighting types of reactions. The reactive part for all of these concepts can be expressed by an expression and is highlighted in green. Some concepts extend the underlying language, e.g. constrains (Felgentreff et al. 2014) use JavaScript's label syntax to declare themselves.

state" (Ramson & Hirschfeld 2017). All shown SBRCs react to changes in the state of a program by identifying dependencies of its expression, like local, global, or member variables, and reevaluating the expression if one of them changed. The difference between SBRCs and AEs is the reaction to a detected change. While AEs execute associated callbacks, SBRCs specialize this behavior and sometimes adapt it to a specific domain: *signals* update a graph of interconnected, time-varying values (Gautier et al. 1987), *implicit layer activation* enables or disable dynamic behavior adaptations according to a boolean expression (Costanza & Hirschfeld 2005; von Löwis et al. 2007; Kamina et al. 2016), *reactive object queries* update groups of objects to match the current system state (Lehmann et al. 2016), and *interactive constraint systems* resatisfy desired properties of a system if necessary (Freeman-Benson 1990; Grabmüller & Hofstedt 2004).

AEs generalize these concepts by providing a freely programmable reaction. Therefore, AEs can often be used as building blocks to implement other SBRCs as exemplified in the following.

Signals Just like AEs, Signals automatically detect changes in the return value of an expression. But instead of triggering arbitrary callbacks on a change, signals always assign the new value of an expression to a variable. Signals are typically connected (i.e. the value of one signal is used in the expression of another signal) and therefore form an explicit dependency graph. When a change occurs that affects multiple signals, it is guaranteed that signals are executed in an order that is free of race conditions. Figure 7 shows a basic example of such a race condition. When `a` changes in line 4, signals should guarantee that `b` is reevaluated before `c`, which prevents `c` to update its state twice and having an unwanted intermediate state. This glitch avoidance can be achieved by sorting the graph topologically (Bainomugisha et al. 2013), starting from the changing

```
let a = 5;
signal: b = a + 1;
signal: c = a + b;
a++;
```

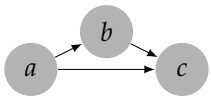

Figure 7 A potential glitch a signal system should avoid

dependency (nodes that cannot be reached by this dependency are ignored).

Implicit Layer Activation The context oriented programming (COP) programming paradigm allows developers to define behavior adaptations as partial methods (Costanza & Hirschfeld 2005). These partial methods are grouped in layers that can be activated dynamically at run-time.

```
let layer = new Layer(); // Layer definition
const obj = {
  getProp() { // getProp is a layered method
    return 17;
  }
};

layer.refineObject(obj, {
  getProp() { // Partial method definition
    // proceed calls the original getProp
    return 42 + proceed();
  }
})
```

Listing 6 Basic example of defining a layer refining a method

Listing 6 exemplifies this functionality by defining a layer (`layer`) with the partial method (`getProp`), which executes code and can use the underlying partial method using a `proceed()` call. COP behaves similarly to overriding methods using inheritance except that the adaptation is in effect if and only if the corresponding layer is active. Before the invocation of a layered method, the system, therefore, checks all the layers and augments the code with the additional functionality from all active layers. There are various proposed activation means for method layers (Kamina et al. 2016), most of which require developers to model context switches explicitly. In contrast, the concept of implicit layer activation (von Löwis et al. 2007) describes the dynamic activation or deactivation of method layers based on a boolean expression using the `activeWhile` method. The layer is active if and only if a specified boolean expression returns true.

```
layer.activeWhile(/* condition */);
```

To implement such a reactive behavior, an AE can be used to watch the boolean expression and activate or deactivate the layer whenever the expression changes (Ramson et al. 2017):

```
activeWhile(condition) { // in class Layer
  aexpr(condition)
    .onBecomeTrue(() => this.activate())
    .onBecomeFalse(() => this.deactivate());
}
```

2.4. Debugging Reactive Programming

RP concepts are declarative. Conventional tools, however, are built for imperative concepts. Since reactive and imperative concepts require different mental models, integrating reactive debugging functionality in existing debuggers is challenging. One example of this is that the declarative nature of RP implies that the effects of a reactive statement, like a signal definition, are not immediate, but leave a lingering effect in the system. Compared to the instantaneous nature of imperative programming, the order of operations is thus often unclear in RP. To make the order of operations discoverable, an interactive exploration of reactive behavior over time is suitable to debug RP concepts. Omniscient debuggers are also known as back-in-time debuggers (Pothier & Tanter 2009; O'Callahan et al. 2017) like the Whyline (Ko & Myers 2008) record the entire history of a debugged program to explore it freely after execution. This approach has the advantage that past activity is saved and that behavior can be navigated interactively, forward and backward. This approach can also be applied to RP debugging by recording the relevant events during execution, as shown by RxFiddle (Banken et al. 2018), a debugger designed for ReactiveX (Meijer 2010). Further, running temporal assertions over a time series of states can result in even faster identification of root causes (Perez & Nilsson 2017).

2.5. Extensible Debugging Tools

The need for reuse and adaptability in debugging functionality is not new and appears in various related research. Examples are integrated development environments (IDEs) that want to support and debug multiple programming languages. IDEs and languages form a many-to-many relation in software development. When m IDEs each want to support n languages, $m * n$ many debugging sets need to be implemented. In these many-to-many relations, an abstract protocol with an adapter (Gamma et al. 1995) for each IDE and each language debugger is commonly used to reduce the effort to m + n adapters. One particular example of this approach is the debug adapter protocol (DAP)[2], which allows IDE developers to automatically support a multitude of language debuggers by implementing only one abstract protocol. Similarly, we also have such a many-to-many relationship between RP concepts and debugging functionalities. The general idea of reusing as many components as possible applies to our case as well. This is why we chose AE as a generalization, which is broad enough to support many concepts while being specific enough that many functionalities can be reused. However, each concept that builds on AEs has special properties and reactions. As it is impossible to predict every possible reaction and incorporate them into the generic protocol, the need for possible adaptations always remains. This issue also applies to the DAP and other language protocols, where domain-specific languages (DSLs) (Merino et al. 2020; Prähofer et al. 2013) offer unique constructs that are not supported in the generic protocol (Jeanjean et al. 2021).

One approach to combat this abstraction gap in debugging DSLs is the moldable debugger (Chis et al. 2014) that, similar to the Reactive Inspector, automatically adapts itself to the domain of the currently debugged code at run-time. The moldable debugger uses domain-specific extensions to make it easy to

[2] https://microsoft.github.io/debug-adapter-protocol (February 21, 2022)

```
const data = new LinkedList();
let mean, median, sd, skew;

// AE4
aexpr(() => (mean - median) / sd).onChange(v => skew = v);

// AE1
aexpr(() => data.median()).onChange(v => median = v);

// AE2
aexpr(() => data.average()).onChange(v => mean = v);

// AE3
aexpr(() => {
  return Math.sqrt(
    data.map(x => (x - mean) ** 2).sum() / data.length)
}).onChange(v => sd = v);

aexpr(() => Math.sign(skew))
  .onChange(v => lively.notify("New skew of data: " + v));

data.pushBack(4);
data.pushBack(1);
data.pushBack(1);
```

Listing 7 Example usage of a `LinkedList` with Active Expressions

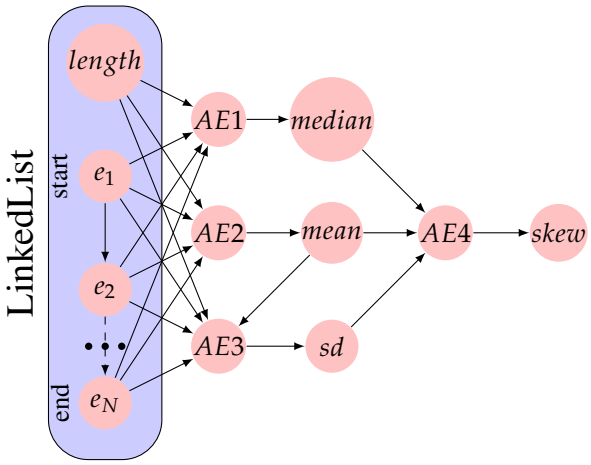

Figure 8 Dependency graph of the `LinkedList` example in listing 7

provide additional views and operations depending on the current domain. However, the lingering effects of the declarative nature of RP simply make it hard to define the current domain in reactive applications.

3. Debugging Active Expressions

Dedicated RP tooling often has to be built from scratch. To reduce this effort, we propose to leverage commonalities in reactive concepts to extract reusable components for debugging tools. We implement this approach by using AE debugging tools as a reusable component since AEs are a generalization of many reactive concepts.

3.1. Challenges

To design AE debugging tools, we analyze a selection of challenges that commonly occur when working with AEs. Namely, we will analyze *dangling AEs*, *unexpected dependencies*, *unexpected evaluation orders*, and *errors in AE statement evaluations*. These challenges were chosen, as they are perceived as the most common problems in personal experience and because they are well fit to highlight the need for debugging tools.

Running Example A `LinkedList` implementation (Full version in listing 15) is a good example to demonstrate the challenges that can come up when working with AEs. The `LinkedList` has a `pushBack(value)` method to add an item to the back of the list as well as `average()` and `median()` methods, to calculate the average and median value in the list, respectively. The `LinkedList` also tracks its `length` by incrementing a counter every time an item is added.

Listing 7 shows how AEs can be used to automatically track the `median`, `mean` and `sd` (standard deviation) and nonparametric `skew`[3] of a `LinkedList` and always output whenever the skew of the data changes. The `sd` and `skew` are calculated as a function of the other variables, as the `LinkedList` implementation does not provide it directly. The dependency graph in figure 8 depicts this logic. The nodes on the left represent the state of the `LinkedList`, with its `length` and e_1 to e_N as its elements.

In the following, we use this example to demonstrate common challenges when working with AEs.

3.1.1. Dangling AEs

AEs do not manage their lifetime automatically and have to be disposed of manually by calling `.dispose()` on AE object if the AE is no longer needed. This manual task can easily be forgotten, as done with the five AEs in the `LinkedList` example in listing 7, where no reference to the AE object is held. These AEs are "dangling" - meaning that they still exist in the system, but are never deleted. Dangling AEs are especially dangerous in a self-sustaining environment, as code is automatically rerun when changed. If no proper migration is performed, this leads to dangling AEs from previous iterations. These dangling AEs can produce unwanted side effects, like debug output or even changes in the object graph. They can also extend the lifetime of objects that are no longer referenced anywhere else and could therefore be deleted otherwise, thus clogging memory. This problem could be circumvented with weak references (which were not supported by JavaScript, as of ES7), but even then the question of whether an AE has the right to extend the lifetime of its dependencies remains.

What do tools need to provide? Currently, to find dangling AEs, programmers need to resort to reflective access through a management class called `AERegistry` to get a list of all AEs currently in the system. This yields multiple disadvantages. First, this method requires detailed knowledge about the AE API. Second, a plain list of all AEs is unfiltered and contains too much information, which makes the searched AEs hard to find. Third, the AE list will either be shown by print-debugging or in a hierarchical object inspector, which only provides text output without any visual preparation. These disadvantages make it inconvenient to work with the reflective access, which motivates the need for a dedicated tool for finding dangling AEs. This tool

[3] https://en.wikipedia.org/wiki/Nonparametric_skew (February 24, 2022)

needs to be able to show all AEs that are active in the system. Moreover, it should make the AEs discoverable, by allowing overview, filtering, and details on demand functionality(Shneiderman 1996). To fix bugs related to dangling AEs, the tools should also be able to dispose of AEs.

3.1.2. Unexpected Dependencies

Another common problem that occurs when working with AEs is unexpected dependencies, which include both missing dependencies and dependencies triggering an AE at an unexpected time. Both of these cases are mostly due to the implicitness of dependencies in AEs (See section 2.2) as we will discuss in the following.

Missing Dependencies Multiple scenarios can make programmers believe a variable is a dependency of an AE, while it is not. Contrary to the belief of the programmer, the AE does not detect a change and does not call its callbacks in these scenarios. One small example of missing dependencies is highlighted in the `minimum` method of the LinkedList example, where variables in the test expression of a for loop are not tracked, due to a limitation in the AEs system implementation, as seen in listing 8.

```
1 // this.length not detected as a dependency,
2 // when used in the test expression
3 for(let i = 0; i < this.length; i++) { /* ... */ }
4
5 // this.length is detected as a dependency
6 const l = this.length;
7 for(let i = 0; i < l; i++) { /* ... */ }
```

Listing 8 Missing dependency in LinkedList code due to a limitation in the Active Expression implementation

However, the two main scenarios that cause missing dependencies are that they change over time and the inability to detect changes in external code.

As described in section 2.2, the dependencies of an AE can change over time. This is caused by the code taking different branches on reevaluations of the AE. If a variable was not used in the path of the last evaluation, it can not possibly change the result of the AE when changed and is therefore not tracked. Programmers may not realize that a variable can no longer influence an expression and therefore be confused that the expression does not trigger its callbacks when the variable changes. To help programmers with this, the system should help them understand which dependencies an AE has at a given moment in time. As dependencies change over time, the programmer should have the possibility to explore how dependencies are changed over time or be informed if a variable is not currently a dependency but was in the past.

The second scenario that causes missing dependencies is external code and native code — in both, the AE system can not detect changes. These are actual shortcomings of a specific AE implementation and not misunderstandings of the programmer. The used AE implementation rewrites code to insert hooks. When the programmer uses an external library, the imported code also needs to be rewritten by the system, to support automatic change detection. This rewriting process is often not feasible: external libraries may suffer too much from the reduced performance due to the additional hooks. For native code, the source code is not available in the first place. Therefore, external libraries and native code are often not supported by the AE system. Implementing the `LinkedLists minimum()` method using e.g. lodash's `min()`[4] function does therefore not work, since lodash's internal access to the elements of the list do not contain the hooks to mark these variables as dependencies if lodash is not rewritten. For this case, programmers need to see all dependencies an AE ever had. This way they can identify that the dependencies they expected are not present and start reasoning why. It is also conceivable to warn programmers if their AEs use non-rewritten code.

Unexpected Evaluation When working with AEs, it can often happen that an AE is reevaluated at an unforeseen moment at which the dependencies of the AE are in an inconsistent state. This can cause the expression to evaluate to an erroneous value, which can propagate through the reactive system and cause unexpected behavior.

We will now show one of these unexpected evaluations that occur in the `LinkedList` example in the AE calculating the average: `aexpr(() => data.average())`.

```
1  average() {
2    let sum = 0;
3    let current = this.start;
4    for(let i = 0; i < this.length; i++) {
5      if(!current) break;
6      sum += current.value;
7      current = current.child;
8    }
9    return sum / this.length;
10 }
```

Listing 9 The `average` method in the `LinkedList` class

```
1  pushBack(value) {
2    this.length++;
3    const oldEnd = this.end;
4    if(!this.start) {
5      this.start = new Node(value);
6      this.end = this.start;
7    } else {
8      this.end.child = new Node(value);
9      this.end = this.end.child;
10   }
11 }
```

Listing 10 The `pushBack` method in the `LinkedList` class

The `average` method, shown in listing 9, depends on the list entries and its length. When the `pushBack(value)` method (listing 10) increases the length of the list by one at the beginning of the method, the AE immediately recognizes a change and reevaluates the expression. The `average` method is called with an increased length, but before the new item was added to the list. The `average` implementation in listing 9 at least prevents a run-time error in line 5, but still returns a wrong average calculation as the length is increased, while the sum is not. The author of the `pushBack` method assumed that the method is used as an atomic operation and that the order of operations is not relevant, as long as the state of the list is valid after the complete evaluation of the method. This assumption is broken

[4] https://lodash.com/docs/4.17.15 (January 17, 2022)

by the AE system because the AE system tracks dependencies across abstraction boundaries. While this behavior is essential for immediate feedback and e.g. constraint systems, it can lead to unexpected intermediate values.

Such problems can often be resolved by changing the order of operations to keep the internal state consistent at all times or by manually suppressing updates of the reactive system, which can be implemented in multiple ways. However, the problematic behavior can only be resolved once the inconsistent state is found and understood in the first place. Finding such problematic behavior is often the most challenging task. One reason for this is that the code locations of these write operations to a dependency can potentially be located far away from the code location of the AE declaration, in different files or even modules of the program. Even if programmers know which dependencies an AE has, it is already hard for them to find all code locations that can change this AE, as the write accesses to its dependencies may be scattered across multiple files.

Unlike other reactive systems, the implicit dependency detection of AEs implies that the programmer does not have to mark the dependencies of an AE explicitly, but they are determined automatically. This reduces the complexity for the user as it allows them to not be aware of the precise dependencies of an AE. The `LinkedList` example can demonstrate this higher level of abstraction. Let's have a look at the AE the user installed to keep track of the average: `aexpr(() => data.average())` in line 5. A programmer who writes this code is likely not too concerned with how the `average()` method looks like and will not look up the variables that are used in the method. They will therefore be unaware of the dependencies of the AE. Even if they were to check the method, it is not trivial to extract the dependencies from the method implementation.

What do tools need to provide? As demonstrated above, the unawareness of the dependencies of an AE can cause the developer to overlook unexpected erroneous behavior. At first glance, it seems like the improved ease of use that AEs provide by automatically handling dependency detection is diminished by these challenges it can cause. It is therefore important for programmers to have the ability to quickly get an overview over the dependencies of an AE. This overview should include showing all dependencies for an AE and vice versa as well as listing all code locations that write a dependency. The advantage of this approach is that in most cases, programmers do not need to worry about dependencies, but they can easily get the required information when they need it to investigate the reactive system. As described above, dependencies that change over time can lead to confusion. Thus, an appropriate visualization of how and when dependencies change over time is required.

3.1.3. Order of Evaluation

As discussed in section 2.2, AEs do not have a specialized execution model. This means that the order in which multiple AEs execute their callbacks when they simultaneously detect a change, is not resilient to glitches. We assume the most common evaluation order, which executes AEs in the order of their declaration if multiple AEs simultaneously detect a change. This can lead to unfortunate evaluation orders, which can cause unwanted behavior. Invalid intermediate values that are propagated are the source of the problem. Such glitches are avoided by most reactive concepts by evaluating them in topological order. Even without these glitches, knowing the precise order in which the reactive primitives are evaluated and invoking their callbacks can be imperative for programmers to ensure correct behavior. Tools should therefore be able to communicate the order of evaluation and which AEs trigger which other AEs.

What do tools need to provide? To learn the precise order in which reactive primitives are evaluated, the tools should be able to provide a temporal overview. This temporal overview should include the evolution of all values an AE assumed over time, as this overview can often show erroneous behavior. However, in some cases, this temporal overview does not suffice to show the cause of an erroneous value, as the timeline's ability to show the relation between multiple AEs is very limited. For these cases, a structural overview of the object graph and the AEs in it at the time of these events should also be provided. This structural overview can provide additional information about the way AEs interact and the state of related objects in the system.

3.1.4. Error in Statement Evaluation

The evaluations of AEs are interleaved with the imperative execution model, possibly at any write operation. To not raise errors at confusing code locations, the AE system, therefore, catches all errors that occur during the evaluation of AEs by default. This design decision makes the errors invisible to programmers, who just observe that their AEs do not seem to trigger callbacks. To show why these invisible errors are problematic, assume the unexpected evaluation bug above still exists and the guard clause `if(!current) break;` in line 5 of the `average` method in listing 9, does not exist. When the `average` method is called in an invalid intermediate state an error occurs that is not visible to the programmer. This completely hides the unexpected evaluation bug in the system from the programmer. AE tools should therefore communicate these errors to the user.

What do tools need to provide? Next to recording and presenting the errors to the programmer on demand, it is also essential that the tools inform the programmer that an error exists. Informing the programmer should preferably be done close to the code location of the AE definition, so they can easily relate the error with the AE and minimize context switches.

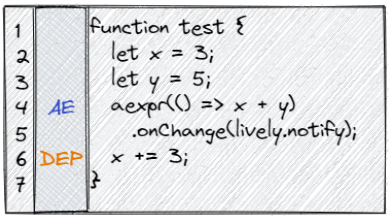

Figure 9 Sketch of code annotations marking lines that contain an Active Expression definition or a dependency change

3.1.5. From Challenges to Design

Our designed tools need to be able to display structural and temporal information

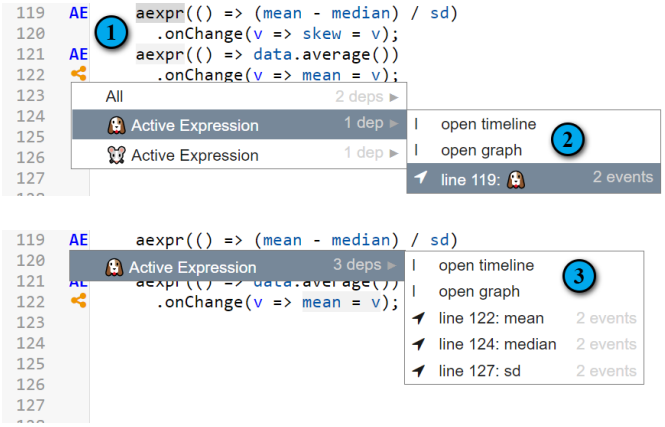

Figure 10 The code annotations add indicators at lines of code that define or trigger reactive behavior (1). The context menu of a line, that changes a dependency, links to all Active Expression written by the dependency (2) and vice versa (3).

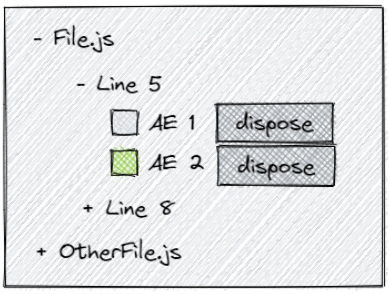

Figure 11 Sketch of an hierarchical overview tree for Active Expressions structured by file, line and instance

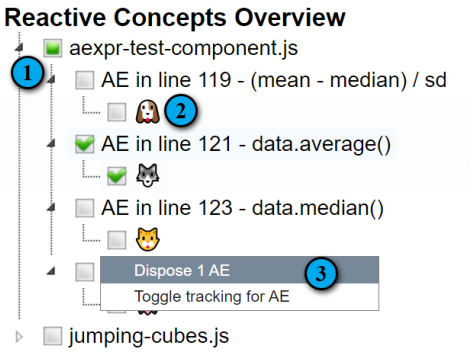

Figure 12 Overview showing all Active Expressions in a hierarchical tree

of the reactive system. Further, they need to bridge the gap between the reactive and the imperative world by linking the tools with the dynamic run-time data of the AEs to the code and vice versa and by displaying information at their interface.

To find good representations that can support these challenges, previous studies (section 2.4) suggest that reifying the mental model of a concept is a key task for a debugging tool. This reification is especially important, since answering potential questions programmers have about the reactive system is

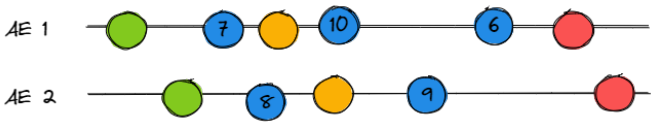

Figure 13 Sketch of a timeline tool with different color-coded event types

easiest, when the explanation is in a representation close to the mental model of the programmer. To provide proper tool support, it is, therefore, necessary to find a representation that captures the programming model of RP and AEs in particular.

We designed four tools: the **code annotations** (section 3.2) augment the code to highlight the interface between the imperative and reactive worlds and provide an entry point into the other tools as well as code navigation capabilities. The AE **overview** (section 3.3) shows all AEs that are currently in the system. The **event timeline** (section 3.4) provides a temporal overview of the reactive system. The **dependency graph** (section 3.5) gives a structural overview. To make the information explorable, the presented tools follow the information-seeking mantra by Shneiderman 1996 (Overview first, zoom and filter, then details-on-demand).

3.2. Code Annotations

Depicting the code locations that write a dependency also requires depicting structural information. However, the dependency graph is not the best fit for this task. Instead, it would be more suitable to highlight the code locations directly in the text editor, as it also shows the surrounding context in which the code is called. These annotations in the margin of the code, sketched in figure 9, can then act as an entry point for debugging by providing functionality related to the code location, like navigating to the affected AEs, or showing the dependency in the graph. On top of that, immediately warning a programmer writing an AE that it produced errors should also be close to the code.

Annotating code at the interface between the imperative and reactive worlds bridges the gap between these worlds and provides an entry point for debugging. As seen in figure 10, lines that contain an AE, as well as lines that change a dependency of an AE are annotated with a respective icon in a UI gutter on the left (1). The annotations project the dynamic run-time dependency information back onto the static code, which is an essential task in debugging (Lieberman & Fry 1995). The icons can be clicked for additional information, code navigation and to open the other reactive tools with information relevant to the selected line (2 and 3). The code navigation helps to understand which lines in the imperative code interact with the declarative behavior of an AE. One advantage of augmenting the code directly is that it provides a very small feedback cycle since there is no context switch for the programmer, who is already working with the code. Furthermore, the concept of additional information in the sidebar is familiar from established IDEs (e.g. any JetBrains IDE[5]). The annotations can also be used to

[5] https://www.jetbrains.com/ (August 8, 2021)

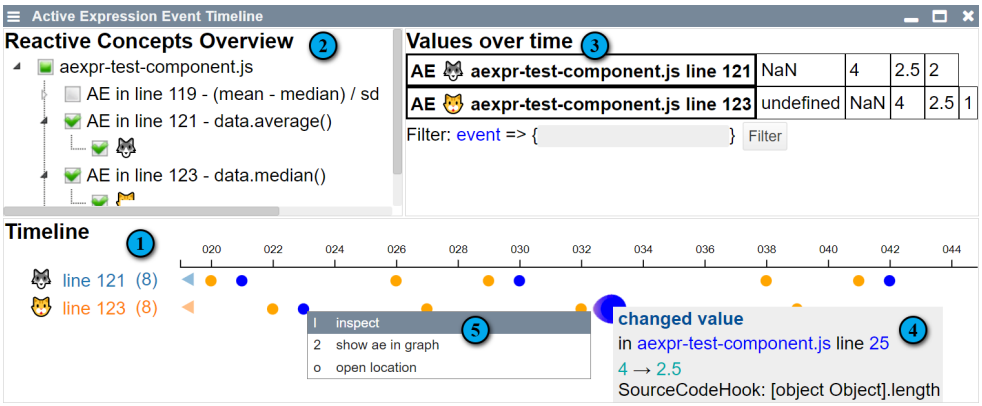

Figure 14 Event timeline tool depicting events and values of selected Active Expressions over time

display critical information to the user as fast as possible. If the evaluation of an AE fails with an error, the code locations of the AE and the triggering dependency display an additional warning icon, which provides additional information about the error.

3.3. Overview

While an overview of all AEs in the system is given by a graph, it may not always be the best tool for this use case, as it also contains a lot of other information. An additional overview over all AEs, hierarchical by file and line of declaration, for easy discoverability, should also be provided (see figure 11). Such an overview is also a fitting place for disposing of AEs.

An overview of the AEs in a system is an important first step in investigating reactive behavior. As seen in figure 12, we chose a hierarchical tree that groups the AEs by file, then line, and then instance, to structure the AEs and ease searching for specific AEs (1). Each AE instance has a corresponding emoji (2) that eases tracking this AE across multiple visualizations. The context menu of each item also allows the user to perform actions on the AEs in this subtree (2). We designed the AE overview as a reusable selection tool that is integrated in the timeline (figure 14) as well as the graph tool (figure 16).

3.4. Event Timeline

One possibility to provide temporal information is to reconstruct a dependency graph (section 3.5) at different points in time and give the user a means of selecting the desired timestamp. This is great for understanding the system behavior at any given time and showing changes in a single timestamp, like which dependencies are added or removed. However, this approach is not enough to give an actual overview over time, because it only ever shows the state of the reactive system at one timestamp at a time. An additional timeline view, as sketched in figure 13, could provide an overview of how the values of an AE changed over time or to temporally relate multiple events.

Our timeline component visualizes events that happened during the lifetime of an AE, as displayed in figure 14. These events include value and dependency changes, (de-)registration of callbacks, and created and disposed of events (1). The tool also incorporates the overview component (2). The timeline will always be filtered to only the AEs selected in this component. The history of the values an AE evaluated over time as well as filter functionality for events can be found in the upper right corner (3). Hovering an event shows additional information relevant to that event type (4) and clicking the event allows for additional debugging actions like jumping to the line in the code responsible for emitting the event, or opening the event in an object inspector (5). This tool provides a good entry point for debugging unexpected behavior because relevant AEs and events can quickly be found and displayed. It also helps to relate multiple AEs temporally, as events of multiple AEs can be displayed in the same timeline.

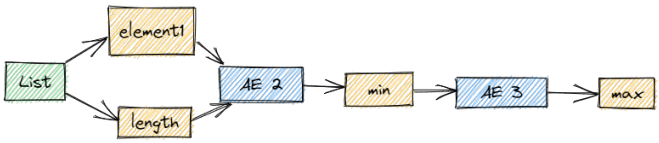

Figure 15 Sketch of a graph tool with objects in green, identifiers in orange, and AEs in green.

3.5. Dependency Graph

Graph visualizations are commonly used to reason about RP concepts. This representation visualizes the interplay between an AE and the object graph via a bipartite graph with AE nodes and variables nodes. An edge from an AE to and variable means that a callback of the AE writes to that variable. An edge from a variable to an AE means that the variable is a dependency. We use a dependency graph as shown in figure 15 as a starting point for our design. A graph is a good tool for depicting the dynamic run-time data of AEs.

In our approach, a dependency graph provides a structural overview for understanding the state of the reactive system. As shown in figure 16, the graph can visualize the dependencies between AEs and the object graph (1) at a specific timestamp during execution, e.g. upon the evaluation of an AE. The graph can therefore help understand the cause of an event as well as the state of the program. It also uses the same overview component as the timeline to allow the user to filter and select relevant AEs to show in the graph (2). One of the main features

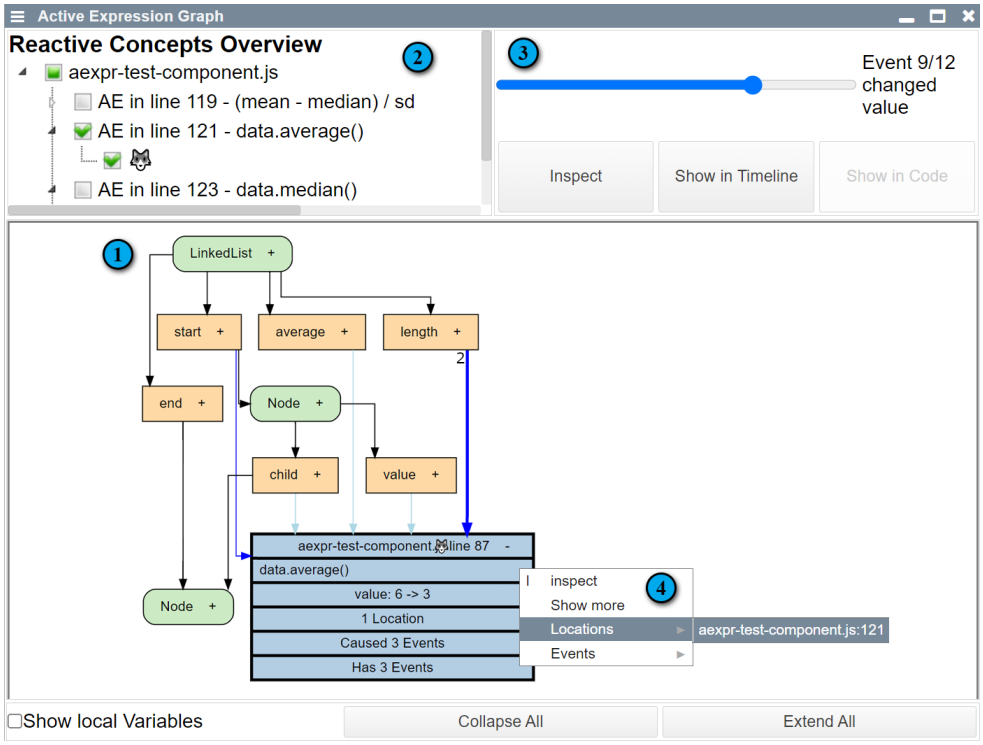

Figure 16 The Dependency graph (1) with Active Expression overview for selection (2) and event selection for time travel (3)

of the graph is that it combines the temporal and structural dimensions of the dependency graph. An event selection slider allows the user to select an event from one of the displayed AEs (3). The graph will always show the state of the system at the time of the selected event, allowing the user to travel through time and inspect the evolution of the dependency graph over time. The graph also highlights the changes the selected event caused (shown as a thick blue line).

The graph can be separated into two parts: a relevant subgraph of the object graph (Lange & Nakamura 1997), and the reactive part containing AEs and their callbacks. The partial object graph is a bipartite graph with nodes for keys (orange) and value objects (green) and arrows marking referential relationships. An edge from a value to a key means that the object has a member with the name of the key. An edge from a key to a value means that the variable points to that object. The separation of keys and their values allows the system to make it visible if two variables reference the same value, as both key nodes can connect to the same value node. This notation can be extended to data structures as well. If values are primitives, they are not displayed in a separate value node but directly within the key node.

The reactive part contains nodes for AEs (blue rectangles) and their registered callbacks (purple rounded rectangles), as shown in figure 17. AEs have an additional emoji as an easy identifier to distinguish similar AEs. The two sides are connected by dependency and event arrows. Dependency arrows are edges that show which keys are currently dependencies of an AE. Dependency arrows change their color and turn into event arrows when the dependency causes a change in the AE. Its multiplicity shows the number of times this dependency caused the AE to change. By adding additional event arrows, the dependency graph shows the relation between multiple AEs. These additional arrows are used when an AE's callback writes to a dependency of another AE and is shown by additional arrows from the AE to its callback and from the callback to the dependency, as shown in the graph displayed in figure 17. Each node type also has a context menu that links the element to the other tools, inspects the data shown in the node, or collapses/expands the node (4).

3.6. Tool Integration

As stated in section 2.4, a solid integration of these new tools into the existing workflow is necessary for them to be used in practice (Alabor & Stolze 2020). Since the write locations of dependencies are the interface between the reactive and imperative execution models, they should be annotated. These locations should be the entry point to all other debugging tools. Further, good integration between all reactive tools is essential to provide a holistic understanding of the reactive system and its integration into the imperative environment.

4. Debugging State-Based Reactive Concepts

Naturally, the RP concepts that can be implemented with AEs are the most promising candidates for being able to be debugged with the AE tools. We, therefore, analyze the subset of RP concepts called State-Based Reactive Concepts (SBRCs), described in detail in section 2.3. The additional knowledge about the specialized reactive behavior provides the opportunity for the tools to better reify a concept's reaction.

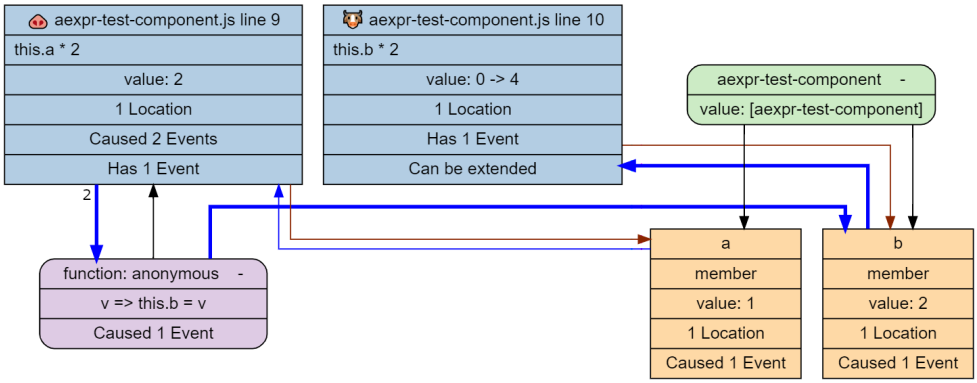

Figure 17 Dependency graph that shows how changes can propagate

We chose to adapt our toolset to signals and ILA, because they are quite different albeit representative of how they specify AEs: signals set additional restrictions on change propagation order and specialize the reactive behavior in a non-domain-specific manner. In contrast, implicit layer activations limit the relevant value range to a boolean value and have a more specialized reaction. Both of these concepts are implemented using AEs, as described in section 5.4.1 and section 5.5.1.

4.1. Changed Challenges

The challenges for debugging AEs discussed in section 3.1 change when accommodating for more SBRCs. To achieve this we first revisit these challenges and analyze how they apply to signals and ILA. Then, we analyze the specific reaction behavior and properties of these concepts and how they change the tools.

Running Example: Online Editor To build a better understanding of signals and ILA and to better demonstrate how the debugging requirements of these concepts change compared to AEs, we introduce a running example of a text editor. This text editor will have an online mode, where it syncs all changes with a remote server, and an offline mode where the text content is exclusively saved locally.

The offline text editor has two functions of interest, shown in listing 11: `render` returns an `HTMLElement` with the content of the text editor and `save` writes the current content of the text field into local storage.

A signal is used to append the content of the text editor into the DOM. It automatically detects when the return value of the `render` method changes and therefore relieves the programmer of updating the DOM when a change occurs. For the online mode, changes are saved on a server to allow for a shared text editor. Whenever a change is made, it is submitted directly to the server. However, if no internet connection is established, changes are saved locally and get synchronized with the server as soon as the connection is back up.

To implement this functionality, a method layer (see section 2.3) called `onlineLayer` (see listing 12) is used to augment the functions of the text editor with the required additional functionality. This allows all server-specific code to be defined at one central point, increasing modularity (Parnas 1972) and

```
class TextEditor {
  /* ... */
  render() {
    return <input
      type="text" id="text"
      value={this.localStorage.read(this.file)}></input>;
  }
  save(text) {
    this.localStorage.write(this.file, text);
  }
}
let editor = new TextEditor();
// content is the div containing the editor
signal: content.innerHTML = editor.render().outerHTML;
```

Listing 11 The basic text editor and a signal that keeps the DOM up-to-date

```
this.onlineLayer = new Layer("onlineEditor");

this.onlineLayer.refineObject(this.editor, {
  render() {
    return <div style="border:2px solid blue">{
      proceed()
    }</div>
  },
  save(text) {
    this.server.send(this.file, text)
    proceed(text)
  }
})
```

Listing 12 Remote editor method layer definition

```
this.onlineLayer.activeWhile(() =>
  this.workRemote && this.server.connected);

this.server.onFileChange(this.file, () =>
  this.mergeServer())

this.onlineLayer.onActivate(() => this.mergeServer());

this.onlineLayer.onDeactivate(() => {
  if(this.workRemote) {
    lively.notify("Lost server connection")
  }
});
```

Listing 13 Method layer activation and event handling

improving separation of concerns (Dijkstra 1982; Reade 1989). The `render` method adds a blue border around the text field to indicate that it is synced. The `save` method sends the content of the text editor to the server.

This additional behavior should only be active while a connection to the server is established and the programmer chooses to work remotely, which is specified with the `activeWhile` function (see listing 13). Further, we specify that when the server connection comes back online, or a change occurs on the server, the server and client texts should be merged. When the connection to the server is lost, but the programmer still wishes to work online, a message is printed. The full implementation of this online editor can be found in appendix B.

4.1.1. How do AE Challenges Change for ILA and Signals

The current implementation of AEs does not come with automatic garbage collection of AEs instances. During development in a live programming environment, it can happen that instances of AEs, signals, and ILAs can accidentally be "lost" in the system, without any reference to dispose of them. They can therefore clog memory and cause unwanted side effects. An overview of all instances of signals and ILAs, as well as the possibility to dispose of them via the tools, is therefore still required. A structural overview of their interactions with each other and the object graph is also useful for this task.

All SBRCs share the same core of reacting to changes in the return value of an expression. Hence, the challenge of unexpected dependencies (section 3.1.2) still requires linking the concepts to all its dependencies and vice versa, showing all code locations that write these dependencies, and visualizing how the dependencies change over time. The same goes for the challenge of handling errors in statement evaluation where making the errors explorable still applies.

The problem of unexpected evaluation order does not apply to signals, as glitch-avoidance is part of the concept, but can still occur for ILA. Giving the possibility for a temporal overview of the events in the system still applies as a requirement for both concepts, though, as it also helps to relate the events of multiple concepts. Showing the graph at a given timestamp also stays important for both concepts, as it is vital for understanding the state of the reactive system at a given time, which is often required to properly understand the reactive behavior. An overview of the values the expressions evaluated to over time is still important for signals, but less so for ILA, as they are just alternating true and false values indicating that a layer was activated or deactivated.

4.1.2. Specialized Reaction

As discussed, the main difference between SBRCs is the way they react to changes in the expression. These specialized reactions imply additional information that should be visualized directly instead of leaking implementation details by visualizing the underlying AE.

When a signal changes, it always sets the value of a variable. A debugging view should therefore also be able to show the changed variable directly and not expose the underlying AE. For the dependency graph, this means that it should assume the form displayed in figure 4 instead of the AE visualization from figure 5.

For implicit layer activations, a visualization of when the layer was activated and deactivated over time – perhaps within a timeline – is a more appropriate granularity than stating that an arbitrary callback was invoked.

Since implicit layer activations always enable or disable dynamic behavior, the tools should be able to visualize and link to this toggleable code.

4.1.3. Additional properties

As discussed in section 2.3, signals form an explicit dependency graph that is traversed in a topological order starting at the point of a detected change to avoid glitches. For signals, this eases the challenge of evaluation order that was discussed for AE. This reduces the significance of a temporal overview, but it can still be beneficial. A much more common way of displaying the evaluation order of Signals is using a layout for the dependency graph that captures the topological order of the graph, like placing all dependencies left to their dependents.

For ILA too, a precise temporal overview is less important, as the activation or deactivation of a layer usually does not immediately propagate changes and is thus less prone to glitches. However, in a system that uses signals and ILA at the same time, the activation of a layer may easily change a method used in a signal expression and therefore trigger that signal. It would therefore still be beneficial to show the precise temporal order of these events.

Instead of revealing the AE-based implementation, the tools should visualize the specialized change directly. Apart from adapting names and icons, this implies for signals to directly show which variable will be written and to depict the dependency graph in a way that reflects its topological order.

Almost all the challenges from section 3.1 still exist for signals. As none of these new challenges requires a completely new perspective, adapting the four previously presented tools suffices to properly capture signals and ILA.

4.2. Adapted Code Annotations

The code view mainly focuses on the change detection aspect, which is the same for all SBRCs and therefore needs the least amount of adaptations. However, it should also be able to visualize the specialized reaction behavior. All SBRCs should change the AE icon in the line that defines the concept to an icon that represents this concept, to not reveal to the user that the concept is implemented using an AE. As shown in figure 18, an implicit layer activation's declaration (3) links to the method layers it can activate or deactivate (4) and vice versa. Further, layered methods link to the original method and vice versa (2). Since the reaction of a signal, writing a variable, is already visible in the signal declaration and the possible change propagation is captured in the already displayed dependencies, no additional information is required.

4.3. Adapted Overview

Similar to the code annotations, the overview component that is integrated into the timeline and the dependency graph tools mainly has to change names to hide the underlying AE-implementation of the concepts (see figure 19). For an ILA, we

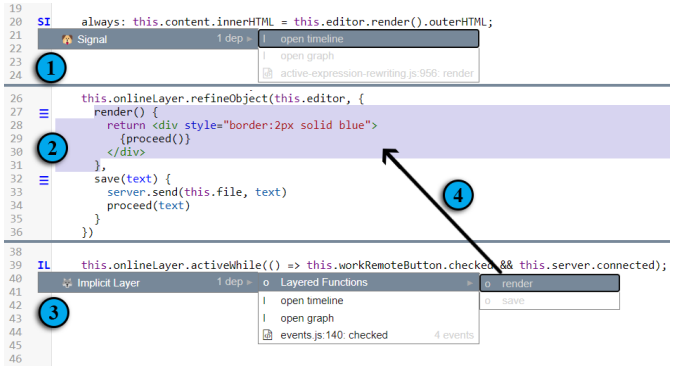

Figure 18 This figure shows the code view adjusted for signals (1) and implicit layer activation (2 and 3). Methods with an overview icon in the gutter can navigate to other partial layers of the method (2). The ILA definition (3) can navigate to partial methods (4) it refines.

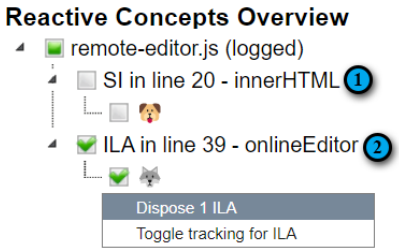

Figure 19 Reactive concepts overview for signals (1) and implicit layer activation (2) adapted from the Active Expression overview.

show the corresponding method layer name instead of the AE code as its identifier (2). For signals, we show the name of the dependency it writes to (1).

With these small changes, the tool is equipped to give the programmer an overview of all reactive concepts currently used in the system, and thereby both shows which layers are currently active and gives an overview of all signals.

4.4. Adapted Event Timeline

Other SBRCs usually require different event types compared to AEs. This leads to three scenarios: events can be completely hidden from the user, they can be adapted to better capture the specialized behavior, or new events can be added. Most commonly, the added and removed callback events no longer apply to all previously discussed SBRCs, since the reaction to change is fixed and can no longer be set by the user. These events should therefore be filtered in the timeline.

ILA benefits from additional events, which are shown in figure 20. As the creation of a method layer does not necessarily happen at the same moment when the implicit activation condition is set, an additional event is required. We reused the AE creation for the implicit condition creation and introduced a new layer created event depicted in mint (1). Moreover, refining and un-refining methods can be done at run-time too, and should therefore also be captured by events (1). Having these additional events also helps the dependency graph in reconstructing an accurate picture of the ILA system at any point in time. As the values the expression of an ILA can evaluate to are restricted, the values over time view can be specialized: since the return values are always interpreted as boolean, the intervals in which a layer is active can be marked directly inside the timeline (2), instead of a row of alternating true and false values. Due to the additional knowledge of the reaction, specialized information can be displayed. As seen in figure 20 the changed value event for ILA (3) also shows that two partial methods in one object were disabled. The event can jump to those partial methods in the code.

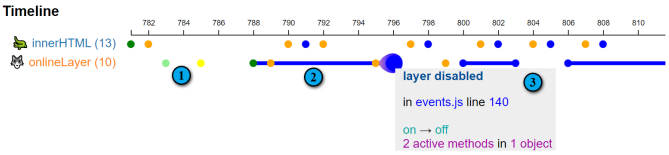

Figure 20 Timeline view depicting a signal and an implicit layer activation with new event types (1), intervals that depict when a layer is active (2), and a specialized event (3)

Signals can also benefit from specialized events. The changed value event of a signal can be augmented with the variable that was set. This small change helps to better show the effect of the changed value event.

4.5. Adapted Dependency Graph

Since SBRCs based on AEs specialize the reactive behavior, the current visualization of generic callback nodes no longer captures this behavior properly, but accidentally reveals implementation details irrelevant to the user. Instead, the specialized behavior should be shown directly. Thus, when the boolean expression of an ILA changes, the specialized reaction, which is the activation or deactivation of a method layer, needs to be visualized. The tool has to be able to link to the respective code and highlight which code is active and which is not. As seen in figure 21, we achieved this by introducing the yellow *layer* (1) and *layered function* (2) nodes. The *layer* nodes are specialized AE nodes that show the state of the layer and have the *layered function* nodes as children. The *layered function* nodes show the current interface of the layered method by listing the layers of a function in execution order from top to bottom while graying out inactive layers and highlighting the `proceed` calls.

AE-based signals always set the value of one variable when they detect a change in their expression. The AE node and the callback node can therefore be omitted, and the reactive information can be added directly to the dependency node which will be written by the signal. Collapsing these three nodes into one simplifies the graph, makes the reaction that will happen and the connections between signal variables clearer, and hides implementation details. We can see this new aggregated node in figure 21 for the *innerHTML* signal node (3). We see that it has an `HTMLDivElement` as its parent and depends on the `outerHTML` and `editor` members and the `render` function. We can also observe (by the highlighted edges) that the render function was just changed by deactivating the `onlineEditor`

layer, which will activate the signal in the next timestamp of the graph timeline.

5. Implementation

The debugging toolsets are implemented in the Lively4 system (Lincke et al. 2017; Ingalls et al. 2008, 2016). Lively is a live object computing environment (Ingalls et al. 2016; Sutherland 1964; Ingalls et al. 1988, 1997) for the web implemented using JavaScript. Like Squeak/Smalltalk (Ingalls et al. 1997), the Lively system is a self-sustaining system (S3) (Hirschfeld & Rose 2008), which means that it is based on a small kernel that is used to implement the entire user interface, including programming and debugging tools from within itself. This results in both the applications and the tools used to create these applications being implemented in the same environment. Lively was chosen for this project because the combination of live programming and an S3 allows us to implement and execute both a programming concept and its debugging tools in the same environment (Niephaus et al. 2020). Figure 23 shows how many parts of the system had to be adapted, ranging from language rewriting, providing new basic components, and creating new and adapting existing tools[6].

5.1. System Overview

To make the required data clearer, an overview of the given AE system is required, which is depicted in figure 23. The Lively4 system provides multiple implementations of AEs, which all share `BaseActiveExpression` as a common base. We only focus on the `RewritingActiveExpression`, which implements the change detection behavior by rewriting the source code via abstract syntax tree (AST) transformation with babel[7] and injecting hooks whenever the state is accessed. We also inject code that analyses an AE on registration, to determine its `Dependencies`. A `Dependency` has a type which can be either `local`, `global` or `member` and `DependencyKey` which uniquely identifies the `Dependency`. Each `DependencyKey` does this by converting all three types of `Dependencies` into a context object and an identifier string, which can then be used to access the value with a computed member expression. To achieve this for local and global dependencies, scope objects are generated, which converts a variable x in an expression like `_scope1["x"]`. For a member Dependency like `vector.x`, this is trivially achieved by using `vector` as the context and `x` as the identifier which results in a `vector["x"]` member expression to compute the value of the `Dependency`. Whenever a `Dependency` registers a change, all its `affectedAEs` are notified and reevaluated. If the evaluation result of an AE changed, it proceeds to call its `callbacks` with the new value. All `BaseActiveExpressions` in the system are stored in a `ActiveExpressionRegistry` singleton, which is a management class that allows reflective access to the AEs in the system.

[6] All tools and their adaptations became part of the Lively4 GitHub project https://github.com/LivelyKernel/lively4-core/tree/gh-pages/src/client/reactive/components/basic (January 19, 2023)

[7] https://babeljs.io/ (January 19, 2023)

5.2. Required Debugging Data

Apart from an overview of the system, figure 23 also shows the additional information that is saved for debugging purposes, highlighted in bold. There are two types of additional information.

On the one hand, there is meta information. This includes the AEs' source code and locations before rewriting, as well as the locations of each dependency, which are all code locations where a dependency is assigned a value. This information is required for the code annotations to navigate from an AE to its dependencies and vice versa, and for the other tools to link back into the code. There is also a global cache, with all dependencies and AEs per source file, for faster access.

On the other hand, there are the events recorded during the lifetime of an AE. These events allow the graph and timeline tools to display the history of an AE over time. The different events are described in more detail in section 5.2.2.

5.2.1. Rewriting

To obtain the necessary location and (original) source code information, we adapted the rewriting process of AEs as seen in figure 24. It can be seen that next to the normal insertion of hooks for variable accesses, like `_getMember`, location and source code information is provided to the method calls, which is then passed to the dependency and AE objects respectively. An alternative to this process is to analyze the call stack at a moment of interest, e.g. inside the constructor of an AE, find the stack frame that created the AE, and use the automatically generated sourcemap files to map the rewritten code back to the original code. This process, however, is quite expensive at run-time and has the disadvantage that the stack frames do not provide the end position of a statement.

DependenciesChanged		**ValueChanged**	**EvaluationFailed**
newDependencies removedDependencies matchingDependencies		previousValue newValue triggeringDependency triggeringLocation aeStack	error
Created	**Disposed**	**CallbackAdded**	**CallbackRemoved**
		callback	callback

Table 1 Event subclasses and their stored information

5.2.2. Events

An event system is used to record the relevant history of AEs. Whenever relevant code is run, a corresponding event is generated, which holds a timestamp and event data, and is saved into a list of events for this AE. There are seven types of events that are depicted in table 1: one each for the creation and deletion of an AE. One each for adding or removing a callback. One event occurs when the value of the AE changes. This event also stores the dependency and location that triggered the reevaluation as well as a shallow copy of the current AE evaluation stack, meaning all AEs that currently evaluate a callback. At the time of a `ValueChangedEvent` the stack, therefore, contains all AEs that are responsible for the change. This information is

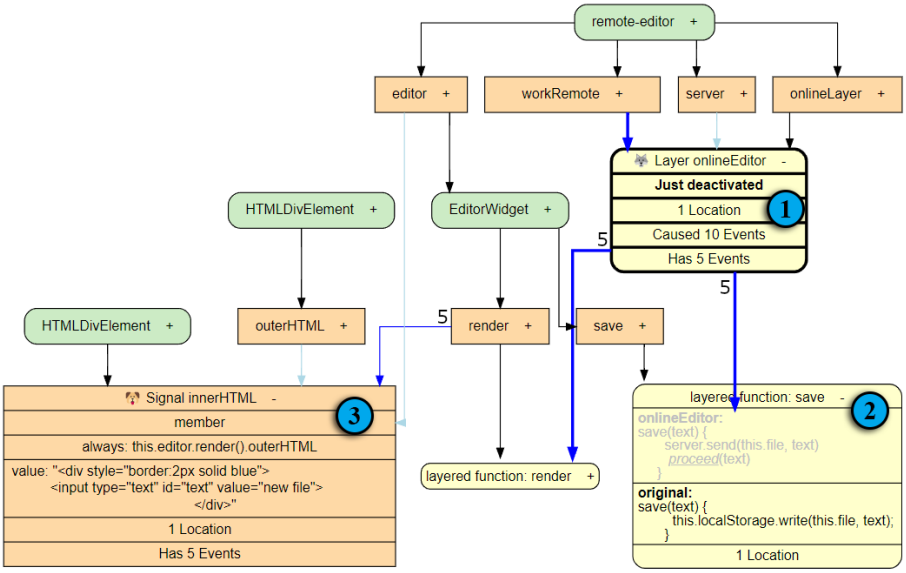

Figure 21 The figure shows the dependency graph adjusted for signals and implicit layer activations. The internal Active Expression nodes are hidden and replaced with specialized nodes that still display dependencies as usual. The yellow nodes (1 and 2) show a layer that was just deactivated. The additional layered code in the save function is therefore grayed out. The signal node (3) directly displays the variable that is written to.

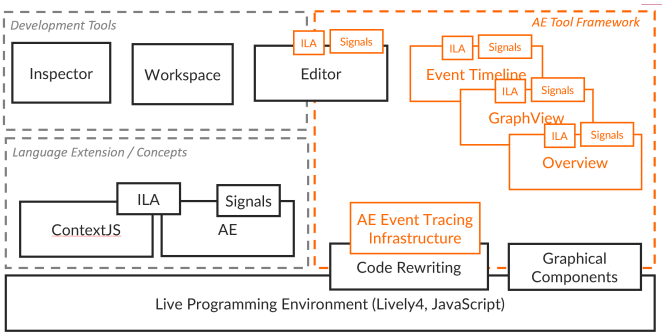

Figure 22 Implementation of AE Tool Framework, the ILA, and signals extensions in the Lively4 environment

vital for understanding the interplay of multiple AEs. It should be emphasized that this event is only emitted when the return value of the AE changes after the re-evaluation. Programmers that try to answer the question of why an AE did not trigger, might also be interested in re-evaluations that did not change the AE. We decided to not record these, due to the high amount of additional memory they would require. Another event is emitted when dependencies change. This event is different in that it is not atomic, but accumulates multiple changes. The main reasons for that are to reduce the visual clutter and to make it easier for a user to see all changed dependencies caused by one event together, as this is the level of detail a user thinks about AEs. The last event occurs when the evaluation of an AE failed. It saves the error and makes it possible to inspect it in one of the debugging tools.

5.2.3. Memory Management

Recording events of each AE in the system can quickly produce a significant memory overhead, especially since the value each AE had at any point of its lifetime is saved. Even though this information is crucial while debugging AEs, the information is not required once everything is running as expected, which ideally should be the majority of the time. Thus, a system to dynamically activate and deactivate the recording of events is needed. There are many possible granularities on how to control which AEs should record events: a global switch for all AEs, as well as a switch per folder, file, or line. Finding good default values for this decision is also important: AEs in a file where a recent change occurred are more likely to be relevant to the programmer at that time and could therefore be activated for event recording automatically. We decided on a per-file approach where everything is off by default unless there was a change in the file in the current session or the programmer explicitly activated tracking for this file.

5.3. Using the Data in the Active Expression Tools

All tools are implemented as components in the Lively4 system and share common functionalities. They all use the Lively4 inspector component, which allows the inspection of any given object by recursively displaying all of its members in a tree view. The graph and the timeline also share the AE overview component described below. The last common functionality is the connection between the tools: each tool provides a method to open it with a filter. If this method is called and a matching window already exists, the window will be reused; a new one is created otherwise. The code annotations allow jumping to a certain location in the code, which is used to find the definition of an AE or the write access to a dependency that caused an

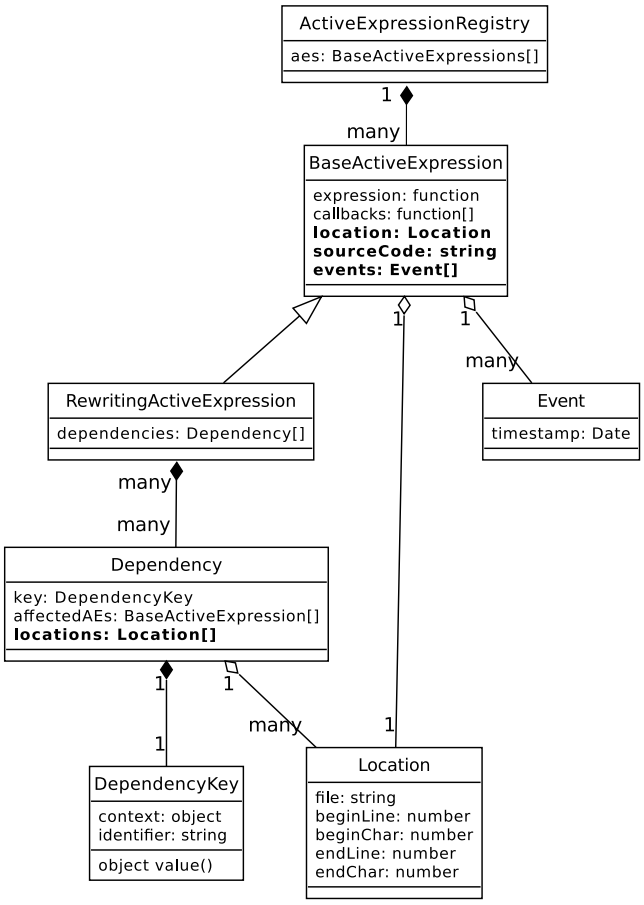

Figure 23 Class diagram of the AE system with additional debugging information depicted in bold

```
1 signal: let a = x + y;
2 signal: b = x + y;
3 signal: obj.c = x + y;
```

Listing 14 Signal syntax in the Lively4 system

event. The timeline and the graph can both be opened for a list of AEs that will be selected in the AE overview component, and optionally an event to select.

5.4. Adapting for Signals

The implementations of the signal concept (see section 2.3) and its debugging capability in the Lively4 environment are based on AEs and the AE debugging toolset respectively.

As seen in listing 14, the Lively4 signal implementations support any left hand side (LHS) of an assignment expression. The value a signal can write to can be a new variable `a`, an existing variable `b`, or a member `obj.c`. If a signal is created for a LHS, which is already the LHS of another signal, or if a signal introduces a cycle in the dependency graph, a run-time error is thrown.

Whenever the signal expression `x + y` changes, the signal system triggers the dependency graph and writes the new value to the LHS. When the dependency graph is triggered, the signal system traverses it along a topological sort to guarantee glitch freedom.

5.4.1. Implementing Signals

Implementing this signal behavior requires three steps:

Rewrite the signal expression into an AE As seen in figure 25 each signal declaration is transformed into an AE declaration with the same expression and a callback that sets the given variable to the changed value. This transformation is achieved via AST rewriting with a babel plugin. The rewriting also adds some additional information that is relevant for the debugging tools later: the fact that this AE represents a signal and the context and identifier of the set variable. The different LHSs, require a case differentiation, but all three cases are transformed into a context and identifier pair. The property and object of a member expression of a signal can be a computed expression instead of an identifier. For example, in `object.array[x * 2]`, `object.array` is the context and the evaluated value of `x * 2` is the identifier. In these cases, both are evaluated once upon the signal declaration, and the evaluated context and identifier are used from there on. This ensures that a signal always writes to the same dependency.

Ensure cycle-free property and no re-registration of signal objects at creation In order to implement the glitch freedom of signals, we need to prohibit cycles in the signal graph. Since the object that will be written to with a signal is known and can not change during the signal's lifetime, the entire signal graph can be reconstructed by the signal system. This reconstruction is used to determine, whether the creation of a new signal would create a cycle in the dependency graph. This check can be achieved by trying to compute a topologically sorted order for the graph. Further, a check whether the signal object is already used by another signal is introduced. Both of these checks throw a run-time error if they fail.

Ensure glitch freedom when traversing the signal graph. To ensure that signals are evaluated in the correct order, the internal dispatching of the reactive system needs to be adapted. When a dependency changes, the reactive system in Lively4 calls a `notifyAEs` method, with all AEs (including signal AEs) that are affected by this dependency. Instead of just triggering the AEs in order of their declaration, the logic is adapted to first trigger the affected signals and their dependent signals in the previously determined topological order. Only after the entire signal graph is evaluated, the rest of the AEs are triggered, the same as before.

5.4.2. Tool Adaptations for Signals

Some tools need to be adapted to be able to properly support signals. Screenshots of the resulting tools and a conceptual discussion of the changes can be found in section 4.

In general, all tools try to hide the underlying AE-based implementation of the signal logic. One important aspect of hiding the AE system is code navigation. Navigating to interesting positions in the code is a core feature of the toolsets and is important for relating dynamic information from the tools

Before transformation

```
aexpr(() => this.c);
```

After transformation

```
_aexpr(() => _getMember(this, "c"), {
  location: /* collapsed for brevity */,
  sourceCode: "() => this.c"
});
```

Figure 24 This listing shows Active Expression code before and after rewriting. The location object contains the filename and begin and end objects that mark the line and character index of the code.

Before transformation

```
signal: obj.c = x + y;
```

After transformation

```
aexpr(() => x + y, {
  isSignal: true,
  signalContext: obj;
  signalIdentifier: "c"
}).onChange(value => obj.c = value);
```

Figure 25 Rewritten signal from line 3 of listing 14. The used AE is annotated as representing a signal.

back to the code. The AE system handles this by annotating the source code location of AE declarations and writing operations in the rewriting step. Since the signal system introduces an additional rewriting step, we need the system to use the code locations before this rewriting instead of the intermediate AE code locations. This is achieved by annotating the locations in the signal rewriting step and then reusing these locations in the AE rewriting step.

Code Annotations The only required change for the code annotations is the introduction of a new signal icon. The specified reaction of a signal, writing a variable, is already quite clear by looking at the signal definition and does not need an additional representation in the code annotations.

Overview The overview now differentiates between AEs and signals. Moreover, the object written by a signal is a better identifier for the signal than the code of the expression and should therefore be displayed.

Event Timeline The event timeline requires a similarly small adaption: since callbacks can no longer be freely registered and deregistered, the timeline can filter the corresponding events to reduce visual clutter.

Dependency Graph The dependency graph requires the most adaptations. The specialized reaction behavior of signals allows for a more concise representation in the graph, which hides the fact that the signal was implemented using an AE: since a signal AE always has a single callback and can only affect a single dependency in the graph, these three nodes - the AE, the callback and the dependency - can be combined into one. To achieve this, a reusable reactive node extension was extracted from the AE node class. The extension provides information and behavior for nodes that have an AE. This extension is then reused in the AE node class as well as the signal node class, which is otherwise a dependency node. This allows the new node to act as both a dependency and as an AE.

5.5. Adapting for Implicit Layer Activation

As with our signal implementations, the implicit layer activation (ILA) concept (see section 2.3) and its debugging capability are based on AEs and the AE debugging toolset respectively. Figure 26 shows how to create an implicit layer activation in the Lively4 system. In this example, the `landscapeModeLayer` is active if and only if `width` is greater than `height`.

5.5.1. Implementing Implicit Layer Activation

The rewriting process for implementing ILA is quite straightforward. Figure 26 shows how code is transformed using a babel AST transformation. The expression is placed in an AE which uses the utility methods `onBecomeTrue` and `onBecomeFalse` to activate or deactivate the layer, when the return value of the expression becomes false or true, respectively. The rewriting also adds some additional information that is relevant for the debugging tools later: the fact that this AE represents an ILA, and the layer it activates. Instead of rewriting, the library that provides the method layer functionality could also internally call `aexpr` with the arguments shown in figure 26. For our purposes, this would complicate detecting the lines of code that create an ILA and annotating this information for our debugging tools. To patch this information, an additional rewriting would be required anyway. Unlike signals, ILA AEs are not prioritized for the evaluation order but follow the same rules as other AEs.

5.5.2. Tool Adaptations for Implicit Layer Activation

The AE tools need to be adapted to support ILA. Screenshots of the resulting tools and a conceptual discussion of the changes can be found in section 4. In general, all tools try to hide the underlying AE-based implementation of the ILA logic by renaming and augmenting visualizations. Some domain-specific events are added, which are relevant to multiple tools. The refine and unrefine events are triggered when the layer refines or unrefines a function. These events are required to determine the interface of a layered function at any point in time. Moreover, next to the ILA created event, which is a derived version of the AE created event, a layer created event is added. The first is triggered when the ILA condition is registered, while the latter is triggered directly at layer creation. Since events can occur, before the AE, which usually stores the events, is created, they can temporally be stored in the layer object and are transferred to the AE on its creation. Also, the callback register and deregister

Before transformation

```
landscapeModeLayer.activeWhile(() => width > height);
```

After transformation

```
aexpr(() => width > height, {
  isILA: true, ila: landscapeModeLayer
}).onBecomeTrue( () => landscapeModeLayer.activate())
  .onBecomeFalse(() => landscapeModeLayer.deactivate());
```

Figure 26 implicit layer activation syntax in the Lively4 system, which is rewritten using AEs.

events are removed, as these events always happen together with the creation of the ILA. To enable navigating to ILA definitions and partial layers, we add additional information during the rewriting that is then stored in the corresponding events.

Code Annotations The code annotations introduce a new icon for ILA. The context menu at the code locations that define the ILA are now linked to the methods they activate and deactivate and vice versa. To achieve this, code locations for refining and unrefining methods as well as defining an ILA are stored in the respective events. We also added a layer overview with an additional annotation to each partial method definition, as seen in figure 35. This overview shows all partial method and their corresponding layers in execution order and with the current partial layer highlighted. Clicking one of the partial layers navigates to the corresponding definition.

Overview The overview should now differentiate between AEs and ILA AEs. Moreover, the layer activated by an ILA is a better identifier for the ILA, than the code of the expression and should therefore be displayed.

Event Timeline Next to displaying the new event types, the event timeline is also extended with intervals. These intervals mark the time intervals at which a layer was active. They therefore start and end at value changed events of the underlying AE. These events as well as the interval are adapted to link to the methods that are affected by the layer during this interval.

Dependency Graph We extended the graph with *layer* and *layered function* nodes. The *layer* nodes are specialized AE nodes that show the state of the layer and have the *layered function* nodes as children. The *layered function* nodes show the interface of the layered method at the currently selected timestamp. The current interface is recreated by querying the current interface from the method layer system and going back in time through refine and unrefine events of all layers. Inactive layers are shown grayed out at the position in which they would be executed if they were activated.

6. Evaluation

To evaluate both the AE and the derived toolsets we analyze the reusability of the AE toolset for debugging other concepts. Then, we discuss the usability of the different toolsets and state the limitations of the reusable toolset approach.

6.1. Reusability

The AE toolset is reused and adapted for signals and ILA. The adaptations are done by modifying and extending the base tools resulting in a toolset that supports AE, signals, and ILA at the same time. We compare and discuss the effort of implementing both adaptations by counting source lines of code (SLOC) where we excluded comments and blank lines.

	AE Tools	Signals	ILA
code annotations	374	+0	+51
overview tree	171	+0	+27
event timeline	497	+0	+38
dependency graph	1511	+30	+128
misc		+6	+31
TOTAL	2553	+36	+275

Figure 27 Code size of AE tools and size of adaptation for signals and ILA (in SLOC)

As demonstrated by the small list of changes described in section 5.4.2, the signal tools were able to reuse most of the AEs debugging tools (see figure 27). For the graph view, which required the most adaptions, less than fifteen SLOC were changed in both the central graph and the dependency node class. The overview, code annotation, and timeline classes did not require any changes, but overriding the `getIdentfier`, `getName`, `getType`, and `getTypeShort` methods from the `BaseActiveExpression` class to return signal specific values added nine lines of code. Not including the small refactoring for extracting logic from the `AENode` class, we counted a total of 36 changed SLOC.

Combining the number of changes required for the new ILA node types (128 SLOC), method layer code annotations (51 SLOC), intervals (27 SLOC in the timeline and 60 SLOC to implement the d3 functionality, which we exclude here as it is library code), and new event types (38 SLOC), as well as miscellaneous small adaptions like reconstructing the state of a layered method based on the events (31 SLOC) totals at 275 changed SLOC.

The amount of changes for supporting ILA is almost an order of magnitude higher than the changes for supporting signals. This difference is mostly caused by the fact that more consequences can be directly inferred by the ILA reaction of activating a layer than by writing a variable for signals. For ILA, the activated/deactivated code, and the new interface of all changed layered methods can be visualized. In contrast, visualizing that a variable changed can simply be achieved by showing the variable name and the old and new values. The more specific the reaction behavior becomes, the more effort has to be directed toward properly capturing this mental model. The increased amount of required changes for supporting ILA is therefore not surprising, as all adaptations were highly specific for the concept. The fact that almost no functionality for

detecting the reason for changes, giving an overview of the state of the reactive system, or navigating through time is required shows the advantage of reusing the AE toolset.

The means of implementing the adaptations are currently direct manipulation of the debugging tool code. This has several advantages and disadvantages as opposed to a pre-defined interface: a programmer who wants to support a new concept needs knowledge of the debugging tools code architecture to be able to integrate their changes and needs to make sure that the code still supports all other concepts. On the upside, this leaves full flexibility for adapting the visualizations at any given point. Highly specialized concepts often require visualizations at unexpected places, which makes defining a catch-all interface close to impossible. To mitigate this, modular code, like the extension system described in section 5.4.2, was used in the implementation of the tools, to make this direct adaption easier. Maintaining the ability to visualize different RP concepts and their interplay in the same tools poses an additional challenge for a pre-defined interface. Unlike the Moldable Debugger(Chis et al. 2014), it is not an option to provide a separate view for each concept, but the views need to maintain compatibility for all concepts at the same time. We decided to favor this compatibility over the disadvantage that additional knowledge about other concepts is required to directly manipulate the code.

Based on the knowledge gained by expanding the tools to signals and ILA, an extension Application Programming Interface (API) for common adaptations to the tools could be defined in the future. This interface could ease defining new node types and extensions in the graph, as well as defining new event types and their visualizations in the timeline. While such an interface could improve modularity and enable adaptations to the tool without knowledge of the tool implementation, it also limits the types of visualizations a tool developer can choose from.

6.2. Usability

To analyze the usability of the presented toolsets, we describe how AE toolset eases overcoming the challenges of working with AEs. The signal toolset is compared to established signal debugging tools (Salvaneschi & Mezini 2016). The usability of the ILA toolset is validated by demonstrating, how it can be used to answer the six questions for debugging context oriented programming (COP) (Taeumel et al. 2014).

6.2.1. Active Expressions

To evaluate the usability of the AE toolset, we revisit the challenges of working with AEs from section 3.1. We describe how the toolset can help overcome each challenge and subsequently discuss the usability toolset as a whole.

Dangling AEs As described in section 3.1, to find dangling AEs, a good overview of all AEs in the system is helpful. For this, the AE overview, with its easy-to-search hierarchical structure, provides the required discoverability functionality, as well as a method of disposing of AEs. Figure 28 shows an AE overview, where the `mean`, `median`, `sd`, and `skew` AEs from the LinkedList example have additional instances, because AEs from previous iterations were not properly disposed. This erroneous behavior can easily be spotted in the overview, as the

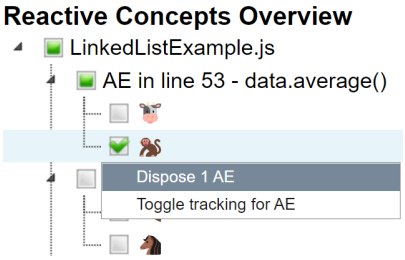

Figure 28 Dangling AEs shown in the overview

programmer knows that each AE is supposed to only have one instance. The context menu of one of these additional AEs contains an entry to dispose of it. However, when a lot of AEs exist in the system and the programmer is unaware of the code location the dangling AEs originate from, it may be hard for them to identify which AEs are dangling using the overview alone. For this case, the values over the table in the timeline can also help, to find the AEs that produce the unwanted side effects, as they can often be identified by the value an AE assumed. Further, the filter provided in the timeline can be used to search all events, which further eases identifying the events that caused unwanted side effects.

Unexpected Dependencies To demonstrate the capability of the toolset to help the programmer with unexpected dependencies, we show how the toolset can be used to debug the erroneous `average` method from section 3.1. To reiterate, the `average` AE is reevaluated in an inconsistent state, in which the length of the list is already increased, even though the new item has not been added yet. These inconsistent intermediate values are mostly spotted due to erroneous outputs or by looking at the values over time overview.

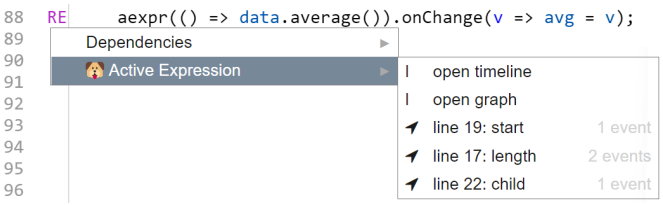

Figure 29 Code annotation of the `average` AE

Figure 29 depict the code annotation of the `average` AE in question, which is a likely starting point for debugging the erroneous behavior. As the line contains both an AE declaration and a dependency write location, for the `maximum` AE, the line is marked with the generic `RE` icon, for reactive, and has a submenu for both of these annotations. The AE submenu lists all dependencies of the AE, which are `start`, `length`, and `child`, which are all properties of the `LinkedList`. From here on, the programmer could already navigate to the code that changed the AE and probably find the bug, given enough time.

However, the tools can provide additional help to understand what is going on. The timeline (depicted in figure 30) with its values over the timetable quickly reveals values that are out of place. In this example, we pushed values 6 and 5 into the list.

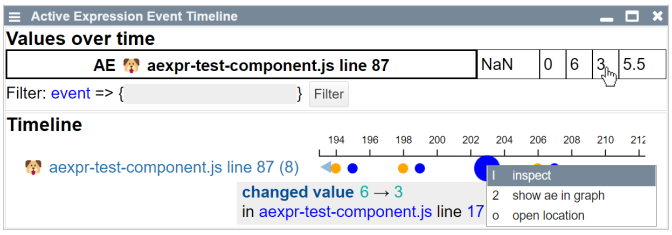

Figure 30 Timeline of the `average` AE

Therefore, the average should never become 3 which immediately reveals an error. Clicking the 3 in the table highlights the corresponding event, which shows where this change was triggered and allows the programmer to navigate there. By navigating to this code location (by clicking "open location") the programmer now knows that increasing the length causes the wrong intermediate value.

To understand what is happening at this moment, the programmer can navigate to the graph at this timestamp, which is depicted in figure 31. Here, the programmer can see that the length just changed and caused an update. They can also observe that the AE only has dependencies to the first node in the `LinkedList`, but not the second. This missing dependency becomes even more obvious by navigating to the next timestamp, where the two dependencies are added, and the timestamp after that, where the value is updated. By navigating to the cause of these two events, the programmer sees that it is just after the line of code that increases the length of the list.

All these visualizations help the programmer to find the code that triggered the erroneous behavior and to understand the order of events and the state of the system at these events. This information greatly eases the error-finding process.

Unexpected Evaluation Order The workflow for finding errors in the evaluation order of AEs is very similar to the process described for unexpected dependencies. We will therefore not go into as much detail but focus on the differences instead. Wrong AE evaluation orders can not be found by looking at the code, as the operation order is determined by the reactive system. Analyzing the behavior in the timeline and dependency graph thus becomes way more important. Especially, exploring the graph through time is best suited for understanding the evaluation order and the state of the system at each timestamp.

Unexpected evaluation orders can lead to inconsistent intermediate values, which are mostly spotted, due to erroneous outputs or by looking at the values over time overview. The first unexpected value of this kind in the `LinkedList` example occurs when the `mean` of the list is updated, which triggers a recalculation of the `skew`. As `median` and `sd` are not yet updated, the `skew` calculating has inconsistent inputs and wrongly evaluates to -0.33. After spotting this value in the timeline and navigating to the graph view at this timestamp, the user sees the graph depicted in figure 32. At this timestamp, 4 arrows in the graph are highlighted. They form a path from the `average` AE to its callback, to the `mean` variable, and finally to the `skew` AE. This not only depicts the variable that caused the change but also why it was changed. These additional highlights are detected by analyzing which AEs currently trigger callbacks when a change occurs, and are especially important for understanding relations between AEs in the graph. The graph also shows the values of the `median` and `sd` are still the old values, which hints at the fact that variables are updated in the wrong order. Exploring the subsequent timestamps in the graph with the other AEs quickly reveals that they are updated right after the event we currently observe, giving even more insight into the order of evaluations and how the AEs are connected.

In conclusion, finding unexpected values in the timeline and then exploring this timestamp, as well as surrounding timestamps, in the graph greatly eases finding unexpected evaluation orders.

Errors in AE Statement Evaluations We identified making errors visible to the programmer as the main task the tools need to support for handling errors thrown during AE evaluation. Figure 33 depicts the code annotation and timeline of the `average` method if the unexpected dependency bug still exists but without the `if(!current) break;` guard clause that prevents access from the not-yet-existing item in the list. The evaluation of the `average` AE then fails, whenever an item is added to the list, as the method tries to access a list item that was not added yet. To make this error visible, both the line that defines the AE and the dependency code location that triggers the reevaluation get highlighted in an attention-grabbing red. The context menu lists all errors, with options to show them in the timeline and to open the error in the Lively4 error view. From there, the programmer can execute the usual AE debugging tools and can get some additional information about the state of the system at the time of the error in the timeline. In conclusion, the toolset makes it hard to miss the existence of errors and the graph helps to understand the system state at the time of the error.

Conclusion The above examples show, how each of the tools offers a unique perspective on the problem at hand. While the timeline helps understand the temporal correlation between events, the overview, and the graph present the run-time state with different levels of detail. The code annotations map the information back onto the static code, which is especially important, as it bridges the gap between the reactive system and the imperative environment.

As all tools provide a unique perspective, it is important to be able to switch between them to get the whole picture. Because this process inherently comes with a lot of context switches, it is vital to preserve as much context as possible between the tools. We achieved this by introducing recognizable emojis as unique identifiers for each AE and by linking the views closely. These links are implemented by giving almost every item in the views, like nodes in the graph or events in the timeline, the possibility to be clicked and inspected in one of the other views.

In conclusion, the tools can preserve the computational model of AEs without revealing its implementation details. Our toolset, therefore, helps overcome the leaking abstractions that arise when debugging reactive code with conventional imperative tools. The implementation details that are hidden are the internals of AEs and their implementation themselves, the implementation of signals, and ILA using AEs is visible and

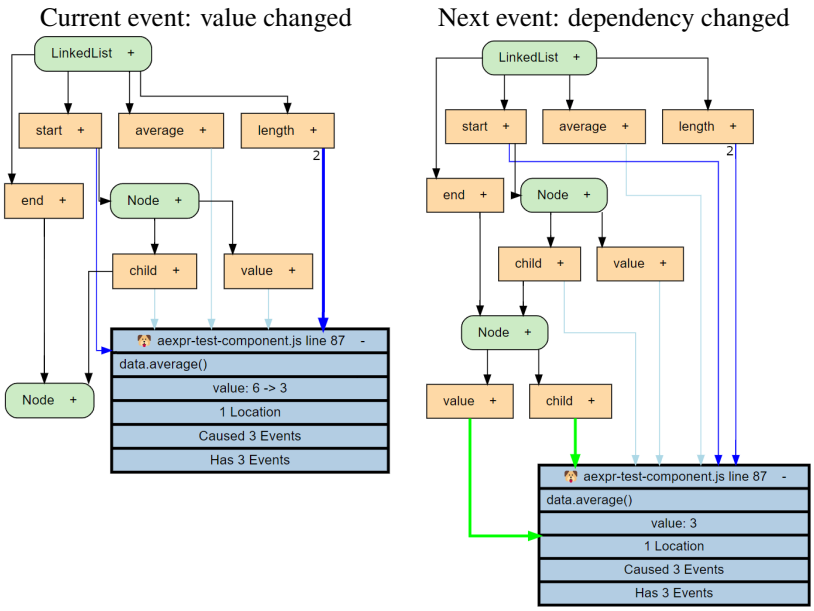

Figure 31 Dependency graph of the `average` Active Expression and the next timestamp

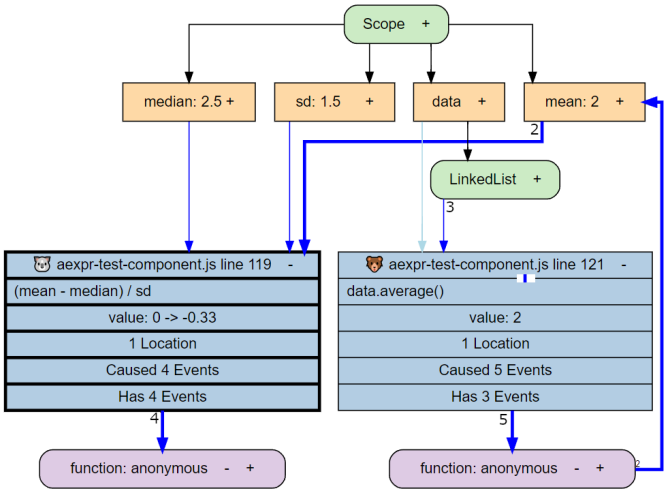

Figure 32 The dependency graph of the `average` and `skew` AEs at the time of the first wrong intermediate value

wanted to see how they interact with each other. Without the AEs in the middle, program behavior that involves different reactive concepts could otherwise not be understood.

6.2.2. Signals To comprehend the dynamic behavior of signals, we compare the signal toolset with an established debugging tool, the reactive inspector (Salvaneschi & Mezini 2016). We compare how both approaches depict the signal graph, how they integrate into the debugging workflow, how they bridge the gap between the reactive behavior and the imperative environment, and which filtering and collapsing features they provide.

Depicting the signal graph Both the reactive inspector and the graph tool depict the signal graph (see figure 34) with the option to explore it over time to inspect the evolution of the graph and the changing dependencies and program state. They are therefore both able to capture the mental model of a dependency graph.

Since signals always form a DAG, the reactive inspector uses a specialized layouting algorithm that is good at depicting the graph in a topologically sorted way, i.e. every dependency of every variable can be rendered to the left of that variable. Since our toolset can also depict concepts, whose dependency graphs can not necessarily be topologically sorted, we decided against such a layouting algorithm. In the future, using a specialized layouting algorithm, just for the signal part of the dependency graph is conceivable and might improve depicting dataflow. As none of the graphviz layouting engines [8] supports such a layouting mechanism, a manual adaptation of one of them is required to support this feature. However, as seen in figure 34, the layout engine often chooses appropriate layouts automatically, albeit the direction of the data flow may vary.

Integration into Debugging Workflow Another difference between the reactive inspector and our toolset is the way they are integrated into the debugging workflow. The reactive inspector focuses on augmenting the step-by-step debugger with reactive tools and integrates the signal graph by offering conditional breakpoints that can be set for each node. The advantage of this approach is a close integration into the existing debugging workflow, where the usual debugger can be used to inspect the run-time state. In comparison, our toolset mostly focuses on post-mortem back-in-time debugging, where the execution of reactive is recorded first and explored afterward. The advantage of this approach is that it focuses on the temporal aspect and thus makes it easier to analyze how the reactive system changes over time. While both approaches yield advantages and disadvantages, we chose the purely back-in-time approach mostly due to the integration of our toolset into the Lively4 system.

[8] https://graphviz.org/docs/layouts/ (February 15, 2022)

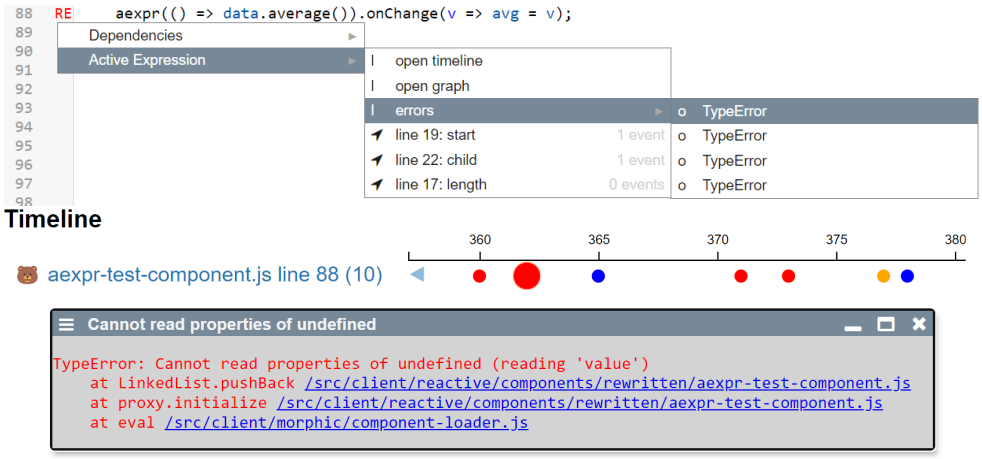

Figure 33 An Active Expression which threw errors shown in the code annotations and the timeline

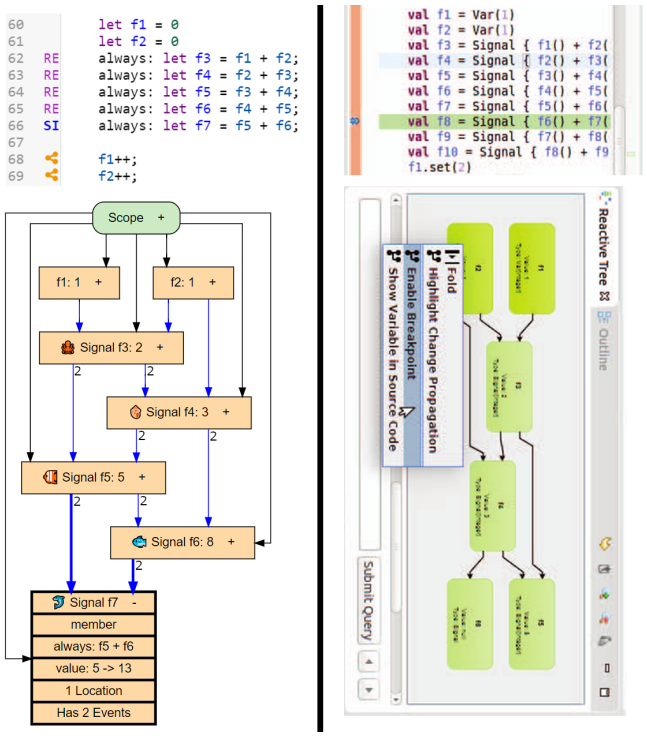

Figure 34 Comparison between the reactive inspector and our graph tool using a Fibonacci example

As a closer integration of our toolset into a classical debugger is desirable, we will now discuss how this could be achieved and which challenges the integration into the Lively4 yields. To declare conditional breakpoints in the graph, each node type in the graph could offer fitting debugging points. For example, an AE node could support breaking when the AE is evaluated or just when it changes its value. The main reason, we do not currently support this but mostly rely on back-in-time debugging is that breaking the code also freezes the debugging tools, as all UI in Lively4 runs in the same thread. Further, the Lively4 system does not provide a dedicated debugger but relies on the debugging functionality of the browser. Integrating debugging functionality into the Chrome debugger would be possible, but limits the debugging functionality to a single browser. Using web workers[9] with a second browser tab might be another solution to offer a dedicated Lively4 debugger where our tools could be integrated.

Representing the interface between reactive and imperative programming To minimize context switches between programming and debugging, it is important for debugging tools to link the information they provide back into code. The reactive inspector mainly links back into the code using conditional breakpoints and by just analyzing the graph at the time that the code is currently paused. This way, the programmer can combine the run-time state information of a classical debugger, with the static code at the position the debugger breaks as well as the additional information about the signals from the reactive inspector. As discussed before, in the Lively4 system, the programmer can not use the reactive tools when the debugger breaks. To link the reactive behavior back to the code we, therefore, use the code annotations, as well as the fact that almost any element in the views can jump to related positions in the code, like definitions of an AE or the change locations of a dependency. This behavior is enabled by annotating each event in the reactive system with corresponding code locations, which are kept up-to-date automatically.

Filtering and Collapsing Dependency graphs in general can quickly become big and therefore unclear to the user. To mitigate this, it is important to provide appropriate functionality to filter the shown data. This includes both choosing which nodes should be displayed and what information each node should show. Both tools provide the functionality to filter the graph and to collapse subgraphs for scalability. Further, our graph tool can toggle how much information should be shown in each node, so that the user can select the information they want to focus on.

Unlike the graph view, the reactive inspector provides a query language to locate events in the execution, with a twofold usage: setting the above-mentioned conditional breakpoints and

[9] https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers (February 15, 2022)

finding specific events in the past of a graph. While conditional breakpoints are currently not supported by the presented views, the timeline view with its overview and filter abilities can make events explorable.

Conclusion Many features of the reactive inspector are supported by the dependency graph, and the other tools and possible integration of missing features were discussed. Further, our toolset and the reactive inspector differ in the way they are integrated into the debugging workflow. Despite these differences, we argue that the similarity between the tools indicates the usability of our signal toolset.

6.2.3. Implicit Layer Activation A study (Taeumel et al. 2014) that showcased data-driven tool building for COP came up with a list of questions to evaluate debugging tools for method layers (de-)activation:

Q1 Which layers refine method M?
Q2 Which methods are refined by layer L?
Q3 In which methods can layer L be activated?
Q4 In which methods can layer L be deactivated?
Q5 Which layers are currently active in process P?
Q6 What is the current interface for object O considering active layers?

By evaluating if our adapted ILA toolset can be used to answer the six questions for debugging method layers, we hope to demonstrate its usefulness. We reuse the online editor example from section 4.1 with the `onlineLayer` which layers the `render` and `save` functions of a text editor to add online synchronization functionality. To better highlight the features of the tools, we added a second method layer called `darkThemeLayer`, which also layers the `render` function and toggles the dark theme.

```
28      this.onlineLayer.refineObject(this.editor, {
29        save(text) {
30          server.send(this.file, text)
31          proceed(text)
32        },
33        render() {
34          return <div style="border:2px solid blue">
35            {proceed()}
36          </div>
37        }
38      })
39
40      this.darkThemeLayer.refineObject(this.editor, {
41        render() {
42  line 33: onlineEditor                                  );
43  line 41: dark theme
44  mock-editor.js:17: original
45
46      })
47
```

Figure 35 Code annotations at a layered method

Q1: which layers refine method M? There are two points in the tools that can answer this question. The first point in the tools that can show which layers refine a method is the dependency graph, as shown in figure 36. A `layeredFunction` node shows all layered methods and has a connection to each layer node that refines it.

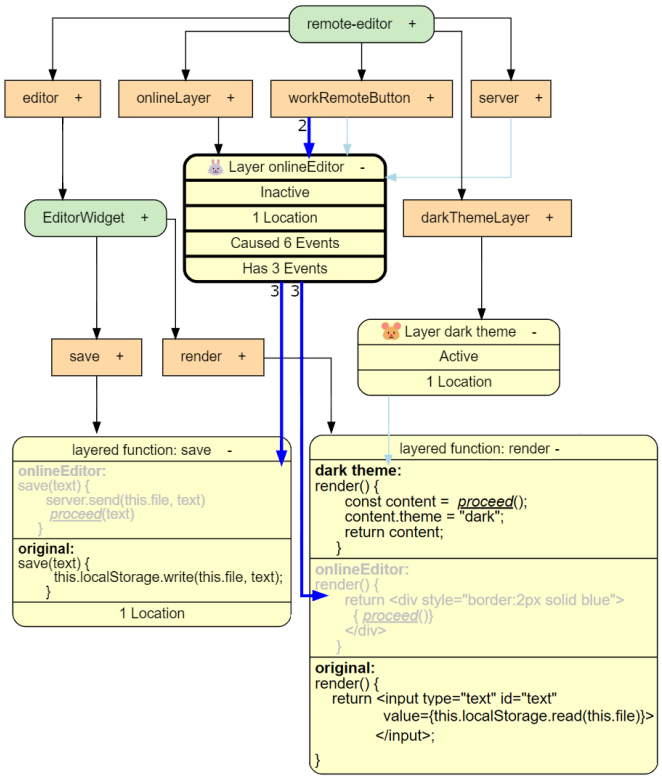

Figure 36 Dependency graph for dark theme and online layers

The second point is the code annotations at each layered method. Figure 35 shows these annotations and their context menu, which gives an overview of all partial layers of the respective method in order of execution, as well as the layer that created this partial layer. Hovering a partial layer highlights it in the code when it is in the same file and selecting an item will navigate to the method. Unlike the dependency graph, which shows the dynamic run-time state of the layers, the code annotations offer a static mapping of this data onto the code. This static mapping aggregates information and presents it close to the code, to make the data more accessible for the programmer.

Q2: which methods are refined by layer L? Analogous to Q1, the information on which methods are refined by a layer can be answered by the code annotations and the dependency graph. The code annotations add the capability to navigate from an ILA definition to all its layered methods, as seen in figure 37. In the dependency graph, the methods refined by a layer are simply its children.

Q3/Q4: in which methods can layer L be (de-)activated? This question directly maps to the question of which dependencies an AE has and where they are written. These write locations of dependencies are the precise locations at which the layer condition can change. Therefore, the same two methods as with AEs can be used. First, the code annotations link AEs, or in our case, the ILA definitions, to their dependency write locations. Second, the dependency graph shows the dependencies of an AE, or layer in our case (see figure 36).

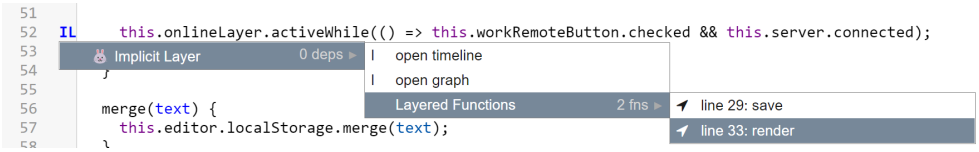

Figure 37 Code annotations at an ILA declaration.

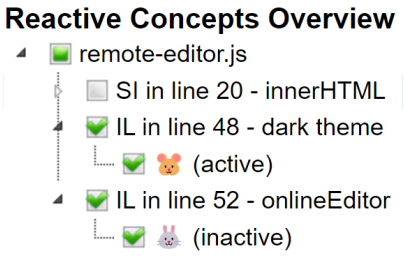

Figure 38 Concepts overview of ILA and a signal

Q5: which layers are currently active in process P? All currently registered layers are listed in the reactive concept overview, as seen in figure 38. Each instance of a layer, also shows if it is active or not. This is a good entry point for investigating ILA behavior, as the hierarchical structure gives a quick, but not overwhelming overview.

Q6: what is the current interface for object O considering active layers? This question can best be answered by the `layeredFunction` nodes in the dependency graph. It shows all partial methods of the method in the order they are executed, with the `proceed` calls highlighted. Disabled layers are greyed out but appear at the position where they would be executed if they were activated.

Conclusion All six questions can completely be answered by the adapted tools set, which indicates that the toolset can adequately depict the specialized behavior of ILA.

6.3. Limitations

Next to the possible improvements to the integration of the toolset into classical debuggers described in section 6.2.2 and the means of adaptation discussed in section 6.1, possible threats to validity remain. The memory and run-time overhead of collecting the information required for debugging was not analyzed in detail, but the approach of only collecting information when required, described in section 5.2.3, helps to mitigate this limitation. Moreover, the exact borders of which RP concepts can be supported by the tool ecosystem is not yet fully explored: highly domain-specific concepts might need entirely new views, while it is possible that even RP concepts that are not SBRCs, like an event system, could reuse some described views. Further, an extensive empirical study (Ko & Fincher 2019) of the actual usability of the tools is not provided. However, evidence for the suitability of the tools is provided in section 6.2, e.g. by the high similarity to established tools like the reactive inspector, which did perform a study to show their usability, as well as the usage of established UI elements like code annotations. Lastly, the performance impacts of the tooling on run-time and memory are not measured. Multiple things could be inspected. While the performance of rendering the visualizations themselves is important to guarantee a minimum frame rate to not impact usability, we do not think that they should be the main focus of performance analysis. To start, we never experienced any lag while using the tools. Moreover, the debugging tools are generally only opened, when the code is inspected closely due to erroneous or unexpected behavior. In these scenarios, programmers are often in a debugging mindset and do not expect maximum performance. We feel that the performance impact that should be analyzed more closely is the performance impact of the additional debugging annotations added in the rewriting, as it always impacts performance, no matter if the code is debugged or not. Also, the memory overhead of storing the debugging information should be analyzed, to better inform memory management decisions.

7. Conclusion and Future Work

To combat the lack of debugging tools for reactive programming concepts, we propose leveraging commonalities in debugging concepts to provide reusable components that ease tool development. For such reusable components to be possible, a common foundation needs to exist. We chose AEs as this foundation and developed a reusable debugging toolset for them. By providing code annotations, an overview component, a dependency graph tool, and an event timeline, we were able to offer different perspectives on the reactive system that builds a holistic representation of AEs. A tight connection of these tools with the functionality to switch between them without losing the context proved to be vital to forming this holistic picture.

This AE toolset was then adapted to signals and implicit layer activation (ILA) to demonstrate its reusability and extensibility. This adaptation process showed that using our toolset as a building block eases the process of providing tool support for a concept because few changes to the tools are sufficient to capture the new mental model. These few changes produced tools that are comparable to existing tools for signals and that can answer relevant questions for debugging ILA. Based on the type of required changes, we also found that the tool developer can focus on representing the unique aspects of their concept and fully reuse the common functionality.

Further, our common toolset allows developers to use and visualize multiple RP concepts in conjunction with the same code base. Since the concepts are embedded in the same system, their interplay can be observed and programmers can use the same toolset for all of them, minimizing context switches.

The co-evolution of a reactive concept and its tools may result in tools that are tailored for the language developer and may

be too powerful for the application developer. E.g. being able to debug lost instances of signals is important for the language developer, but the actual user of the new signal concept should in theory rely on that such things are not necessary. Thus the ideal toolset and its capabilities and with it its complexity should be tailored to the actual role of the developer.

The presented toolset is an exploration in the design space of state-based reactive programming concepts and their tool support. This work should be perceived as a feasibility study of what tools are needed and helpful and how they work together. Reactive programming concepts are very different from each other. Because of that, the tools we designed and explored are very generic. In that, they represent aspects that are shared by all reactive concepts: exploring the state, seeing behavior over time, and mapping to source code. On that level, we discovered that the tools are very similar and be expressed in common visualizations and interactions. This discovery might imply a common protocol for tooling on AE based reactive concepts, which is possible for some aspects like labeling lines of code with, e.g. “AE“, “SI“, or “IL“. By prematurely restricting the creation of tools to a rigid protocol we cannot explore the full range of possible concept-specific user interaction, especially since there are few best practices and experience with reactive concept tooling yet. This is also a reason why a more thorough investigation of how the tools themselves can be better modularized or automatically generated is future work.

Our approach of an extensible reactive toolset enables multiple directions of further research.

Visualizing multiple concepts and their interplay in the same tools enables a closer investigation of this interplay. The extent to which multiple reactive concepts can be used in conjunction is still understudied. We previously discussed that we evaluate all signals before ILA and other AEs, to guarantee glitch avoidance for signals. Integrating signals with, for example, a constraint system that also requires immediate reaction might become challenging. Our toolset could be used to investigate and visualize the interactions between these concepts.

Further, studies suggest that not using dedicated tools in application development produces a mental gap, which can limit programmers’ debugging capabilities (Chis et al. 2014; Sillito et al. 2008). When expanding this empirical body of research to system development, it would be interesting to see whether the simultaneous development of RP concepts and tooling leads to more robust RP concepts and better tools. It is also conceivable that this co-development eases the development of the concepts, as the tools can provide additional insight and understanding during development.

Despite these open research areas, we believe that our approach contributes to making RP more accessible for two reasons. First, our reusable debugging toolset eases the development of tooling for new concepts, which can result in more tools being provided. Second, our toolset supports multiple RP concepts, which facilitates their simultaneous usage, thus increasing the programmer’s expressiveness.

A. LinkedList Example Implementation

```
class Node {
  constructor(value) {
    this.value = value;
  }
}
class LinkedList {
  constructor() {
    this.length = 0;
  }
  pushBack(value) {
    // this.length++; // increasing the length here produces an invalid internal state
    if(!this.start) {
      this.start = new Node(value);
      this.end = this.start;
    } else {
      this.end.child = new Node(value);
      this.end = this.end.child;
    }
    this.length++; // This is the correct position for increasing the length
  }

  average() {
    let sum = 0;
    let current = this.start;
    for(let i = 0; i < this.length; i++) {
      if(!current) break;
      sum += current.value;
      current = current.child;
    }
    return sum / this.length;
  }

  median() {
    if(!this.start) return undefined;
    let list = [];
    let current = this.start;
    for(let i = 0; i < this.length; i++) {
      if(!current) break;
      list[i] = current.value;
      current = current.child;
    }
    list.sort();
    if(list.length % 2 === 1) {
      return list[(list.length - 1) / 2];
    } else {
      return (list[list.length / 2 - 1] + list[list.length / 2]) / 2;
    }
  }


  map(f) {
    if(!this.start) return [];
    let current = this.start;
    let list = [];
    const l = this.length;
    for(let i = 0; i < l; i++) {
      list[i] = f(current.value);
      current = current.child;
    }
    return list;
  }

  mininum() {
    if(!this.start) return undefined;
    let min = this.start.value;
    let current = this.start;
    // Missing dependency if we do not extract this
    const l = this.length;
    for(let i = 0; i < l; i++) {
      if(!current) break;
      if(current.value < min) {
        min = current.value;
      }
      current = current.child;
    }
    return min;
  }
}

const data = new LinkedList();
```

```

let mean, median, sd, skew;

aexpr(() => (mean - median) / sd) // Wrong order, as this is now prioritized leading to wrong intermediate values
  .onChange(v => skew = v);
aexpr(() => data.average())
  .onChange(v => mean = v);
aexpr(() => data.median())
  .onChange(v => median = v);
aexpr(() => {
  return Math.sqrt(
    data.map(x => (x - mean) ** 2).sum() / data.length)
}).onChange(v => sd = v);

aexpr(() => Math.sign(skew))
  .onChange(v => lively.notify("New skew of data: " + v));

data.pushBack(4);
data.pushBack(1);
data.pushBack(1);
```

Listing 15 LinkedList implementation

B. Remote Editor Example Implementation

```
"enable aexpr";
import { Layer, proceed } from 'src/client/ContextJS/src/Layers.js';
import Morph from 'src/components/widgets/lively-morph.js';
import {Server, EditorWidget} from './mock-editor.js'

export default class RemoteEditor extends Morph {
  async initialize() {
    this.windowTitle = "RemoteEditor";
    this.file = await lively.prompt("file to edit");
    this.save.addEventListener("click", () => {this.editor.save(this.text.value)})
    this.server = new Server(() => this.serverCheckbox.checked);
    this.editor = new EditorWidget(this.file);
    this.server.onFileChange(this.file, (text) => this.merge(text))

    always: this.content.innerHTML = this.editor.render().outerHTML;

    this.onlineLayer = new Layer("onlineEditor");
    this.onlineLayer.onActivate(() => this.merge(this.server.read(this.file)));
    const server = this.server;
    this.onlineLayer.refineObject(this.editor, {
      render() {
        return <div style="border:2px solid blue">
          {proceed()}
        </div>
      },
      save(text) {
        server.send(this.file, text)
        proceed(text)
      }
    })
    this.onlineLayer.activeWhile(() => this.workRemoteButton.checked && this.server.connected);
  }

  merge(text) {
    this.editor.localStorage.merge(text);
  }
  get workRemoteButton() {
    return this.get("#workRemoteButton");
  }
  get content() {
    return this.get('#content');
  }
  get save() {
    return this.get('#save');
  }
  get text() {
    return this.get('#text');
  }
  // Button that "enables"/"disables" our mock server
  get serverCheckbox() {
    return this.get('#server');
  }
}
```

Listing 16 Remote editor script

```
<template id="remote-editor" >

<input type="checkbox" id="workRemoteButton" name="work remote" checked>
<label for="workRemoteButton">Work Remote</label>
<input type="checkbox" id="server" name="enable server" checked>
<label for="server">Toggle server</label>
<button id="save">
  Save
</button>

<div id="content">
  <input type="text" id="text">
</div>
</template>
```

Listing 17 Remote editor HTML

```
export class LocalStorage {
  constructor() {
    this.files = new Map();
  }
  read(file) {
    if(!this.files.has(file)) {
      this.write(file, "new file");
    }
    return this.files.get(file);
  }
  write(file, text) {
    this.files.set(file, text);
  }
  merge(file, text) {
    this.files.set(file, text);
  }
}

export class Server {
  constructor(enabledSource) {
    this.enabledSource = enabledSource;
    this.storage = new LocalStorage();
  }
  send(file, text) {
    this.storage.write(file, text);
  }
  read(file, text) {
    return this.storage.read(file, text);
  }
  onFileChange() {
    // no remote changes for now
  }
  get connected() {
    return this.enabledSource();
  }
}

export class EditorWidget {
  constructor(file) {
    this.file = file;
    this.localStorage = new LocalStorage();
  }
  render() {
    return <input type="text" id="text" value={this.localStorage.read(this.file)}></input>;
  }
  save(text) {
    this.localStorage.write(this.file, text);
  }
}
```

Listing 18 Additional mock classes for the remote editor

References

Alabor, M., & Stolze, M. (2020). Debugging of RxJS-based applications. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems.* ACM. doi: 10.1145/3427763.3428313

Bainomugisha, E., Carreton, A. L., van Cutsem, T., Mostinckx, S., & de Meuter, W. (2013). A survey on reactive programming. *ACM Computing Surveys*(4). doi: 10.1145/2501654.2501666

Banken, H., Meijer, E., & Gousios, G. (2018). Debugging data flows in reactive programs. In *Proceedings of the 40th International Conference on Software Engineering.* ACM. doi: 10.1145/3180155.3180156

Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Guernic, P. L., & Simone, R. D. (2003). The Synchronous Languages 12 Years Later. *Proceedings of the IEEE*. doi: 10.1109/JPROC.2002.805826

Brand, M., Ramson, S., Lincke, J., & Hirschfeld, R. (2022). Explicit tool support for implicit layer activation. In *Proceedings of the 14th acm international workshop on context-oriented programming and advanced modularity.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3570353.3570355

Chis, A., Gîrba, T., & Nierstrasz, O. (2014). The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers. In B. Combemale, D. J. Pearce, O. Barais, & J. J. Vinju (Eds.), *Software Language Engineering.* Springer. doi: 10.1007/978-3-319-11245-9_6

Costanza, P., & Hirschfeld, R. (2005). Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium.* doi: 10.1145/1146841.1146842

Courtney, A. (2001). Frappé: Functional Reactive Programming in Java. In I. V. Ramakrishnan (Ed.), *Practical Aspects of Declarative Languages.* Springer. doi: 10.1007/3-540-45241-9_3

Davis, A., & Keller, R. (1982). Data Flow Program Graphs. *Computer*. doi: 10.1109/MC.1982.1653939

Dijkstra, E. W. (1982). On the role of scientific thought. In *Selected writings on computing: A personal perspective.* Springer-Verlag.

Elliott, C., & Hudak, P. (1997). Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming.* ACM. doi: 10.1145/258948.258973

Felgentreff, T., Borning, A., Hirschfeld, R., Lincke, J., Ohshima, Y., Freudenberg, B., & Krahn, R. (2014). Babelsberg/JS. In R. Jones (Ed.), *ECOOP 2014 – Object-Oriented Programming.* Springer. doi: 10.1007/978-3-662-44202-9_17

Freeman-Benson, B. N. (1990). Kaleidoscope: Mixing objects, constraints, and imperative programming. *ACM SIGPLAN Notices*(10). doi: 10.1145/97946.97957

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc.

Gautier, T., Le Guernic, P., & Besnard, L. (1987). SIGNAL: A declarative language for synchronous programming of real-time systems. In G. Kahn (Ed.), *Functional Programming Languages and Computer Architecture.* Springer. doi: 10.1007/3-540-18317-5_15

Grabmüller, M., & Hofstedt, P. (2004). Turtle: A Constraint Imperative Programming Language. In F. Coenen, A. Preece, & A. Macintosh (Eds.), *Research and Development in Intelligent Systems XX.* Springer. doi: 10.1007/978-0-85729-412-8_14

Hirschfeld, R., & Rose, K. (Eds.). (2008). *Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15-16, 2008, Proceedings*. Springer-Verlag. doi: 10.1007/978-3-540-89275-5

Ingalls, D., Felgentreff, T., Hirschfeld, R., Krahn, R., Lincke, J., Röder, M., … Mikkonen, T. (2016). A world of active objects for work and play: The first ten years of lively. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* ACM. doi: 10.1145/2986012.2986029

Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., & Kay, A. (1997). Back to the future: The story of Squeak, a practical Smalltalk written in itself. *ACM SIGPLAN Notices*(10). doi: 10.1145/263700.263754

Ingalls, D., Palacz, K., Uhler, S., Taivalsaari, A., & Mikkonen, T. (2008). The Lively Kernel A Self-supporting System on a Web Page. In R. Hirschfeld & K. Rose (Eds.), *Self-Sustaining Systems, First Workshop, S3.* Springer. doi: 10.1007/978-3-540-89275-5_2

Ingalls, D., Wallace, S., Chow, Y.-Y., Ludolph, F., & Doyle, K. (1988). Fabrik: A visual programming environment. *ACM SIGPLAN Notices*(11). doi: 10.1145/62084.62100

Jeanjean, P., Combemale, B., & Barais, O. (2021). IDE as Code: Reifying Language Protocols as First-Class Citizens. In *14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference).* ACM. doi: 10.1145/3452383.3452406

Johnston, W. M., Hanna, J. R. P., & Millar, R. J. (2004). Advances in dataflow programming languages. *ACM Computing Surveys*(1). doi: 10.1145/1013208.1013209

Kamina, T., Aotani, T., & Masuhara, H. (2016). Generalized Layer Activation Mechanism for Context-Oriented Programming. In S. Chiba, M. Südholt, P. Eugster, L. Ziarek, & G. T. Leavens (Eds.), *Transactions on Modularity and Composition I.* Springer International Publishing. doi: 10.1007/978-3-319-46969-0_4

Ko, A. J., & Fincher, S. A. (2019). A Study Design Process. In A. V. Robins & S. A. Fincher (Eds.), *The Cambridge Handbook of Computing Education Research.* Cambridge University Press. doi: 10.1017/9781108654555.005

Ko, A. J., & Myers, B. A. (2008). Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering.* ACM. doi: 10.1145/1368088.1368130

Köhler, M., & Salvaneschi, G. (2019). Automated refactoring to reactive programming. In *34th IEEE/ACM international conference on automated software engineering, ASE 2019,*

san diego, ca, usa, november 11-15, 2019 (pp. 835–846). IEEE. doi: 10.1109/ASE.2019.00082

Lange, D. B., & Nakamura, Y. (1997). Object-oriented program tracing and visualization. *Computer*(5). doi: 10.1109/2.589912

Lehmann, S., Felgentreff, T., Lincke, J., Rein, P., & Hirschfeld, R. (2016). Reactive object queries: Consistent views in object-oriented languages. In *Companion Proceedings of the 15th International Conference on Modularity.* ACM. doi: 10.1145/2892664.2892665

Lieberman, H., & Fry, C. (1995). Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM Press/Addison-Wesley Publishing Co. doi: 10.1145/223904.223969

Lincke, J., Rein, P., Ramson, S., Hirschfeld, R., Taeumel, M., & Felgentreff, T. (2017). Designing a live development experience for web-components. In *PX/17.2*. doi: 10.1145/3167109

Meijer, E. (2010). *Subject/observer is dual to iterator.* Conference on Programming Language Design and Implementation (PLDI), Fun Ideas and Thoughts Session.

Merino, M. V., Vinju, J., & van der Storm, T. (2020). Bacatá: Notebooks for DSLs, Almost for Free. *The Art, Science, and Engineering of Programming*(3). doi: 10.22152/programming-journal.org/2020/4/11

Meyerovich, L. A., Guha, A., Baskin, J. P., Cooper, G. H., Greenberg, M., Bromfield, A., & Krishnamurthi, S. (2009). Flapjax: A programming language for ajax applications. In *24th acm sigplan conference on object-oriented programming systems, languages, and applications (OOPSLA), 2009.* (pp. 1–20). New York, NY, USA: ACM. doi: 10.1145/1640089.1640091

Niephaus, F., Rein, P., Edding, J., Hering, J., König, B., Opahle, K., … Hirschfeld, R. (2020). Example-based live programming for everyone: Building language-agnostic tools for live programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* ACM. doi: 10.1145/3426428.3426919

O'Callahan, R., Jones, C., Froyd, N., Huey, K., Noll, A., & Partush, N. (2017). Engineering Record And Replay For Deployability: Extended Technical Report. *CoRR*.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*(12). doi: 10.1145/361598.361623

Perez, I., & Nilsson, H. (2017). Testing and debugging functional reactive programming. *Proceedings of the ACM on Programming Languages*(ICFP). doi: 10.1145/3110246

Pothier, G., & Tanter, É. (2009). Back to the Future: Omniscient Debugging. *IEEE Software*. doi: 10.1109/MS.2009.169

Prähofer, H., Schatz, R., Wirth, C., Hurnaus, D., & Mössenböck, H. (2013). Monaco—A domain-specific language solution for reactive process control programming with hierarchical components. *Computer Languages, Systems & Structures*(3). doi: 10.1016/j.cl.2013.02.001

Ramson, S., & Hirschfeld, R. (2017). Active Expressions: Basic Building Blocks for Reactive Programming. *The Art, Science, and Engineering of Programming*(2). doi: 10.22152/programming-journal.org/2017/1/12

Ramson, S., Lincke, J., & Hirschfeld, R. (2017). The declarative nature of implicit layer activation. In *Proceedings of the 9th International Workshop on Context-Oriented Programming.* ACM. doi: 10.1145/3117802.3117804

Reade, C. (1989). *Elements of functional programming*. Addison-Wesley Longman Publishing Co., Inc.

Salvaneschi, G., Hintz, G., & Mezini, M. (2014). REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th international conference on Modularity.* ACM. doi: 10.1145/2577080.2577083

Salvaneschi, G., & Mezini, M. (2016). Debugging for Reactive Programming. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE).* doi: 10.1145/2884781.2884815

Shneiderman, B. (1996). The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages.* doi: 10.1109/VL.1996.545307

Sillito, J., Murphy, G. C., & De Volder, K. (2008). Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering*(4). doi: 10.1109/TSE.2008.26

Sutherland, I. E. (1964). Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop.* ACM. doi: 10.1145/800265.810742

Taeumel, M., Felgentreff, T., & Hirschfeld, R. (2014). Applying Data-driven Tool Development to Context-oriented Languages. In *Proceedings of 6th International Workshop on Context-Oriented Programming.* ACM. doi: 10.1145/2637066.2637067

Van den Vonder, S., Renaux, T., Oeyen, B., De Koster, J., & De Meuter, W. (2020). Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model. In R. Hirschfeld & T. Pape (Eds.), *34th european conference on object-oriented programming (ecoop 2020).* Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.ECOOP.2020.19

von Löwis, M., Denker, M., & Nierstrasz, O. (2007). Context-oriented programming: Beyond layers. In *Proceedings of the 2007 international conference on Dynamic languages.* ACM. doi: 10.1145/1352678.1352688

Weiher, M., & Hirschfeld, R. (2016). Constraints as polymorphic connectors. In *Proceedings of the conference on modularity (modularity) 2016* (pp. 134–145). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2889443.2889456 doi: 10.1145/2889443.2889456

About the authors

Stefan Ramson is a member of the Hasso Plattner Institute's Software Architecture Group at the University of Potsdam. He regards the design of programming systems as the intersection of notation, interface design, psychology, and ergonomics. His current research interests include live and exploratory programming systems, alternative input methods, visual languages, and natural programming. You can contact him at stefan.ramson@hpi.uni-potsdam.de.

Markus Brand is a graduate student at the Software Architecture Group of the Hasso Plattner Institute. His research interests include programming tools and reactive programming in particular. You can contact him at Markus.Brand@student.hpi.uni-potsdam.de.

Jens Lincke is a member of the Hasso Plattner Institute's Software Architecture Group. His research interests include live and exploratory programming. Lincke received a PhD in IT-Systems Engineering from the Hasso Plattner Institute at the University of Potsdam. You can contact him at jens.lincke@hpi.uni-potsdam.de.

Robert Hirschfeld leads the Software Architecture Group of the Hasso Plattner Institute at the University of Potsdam. His research interests include dynamic programming languages, development tools, and runtime environments to make live, exploratory programming more approachable. Hirschfeld received a PhD in computer science from Technische Universität Ilmenau. You can contact him at robert.hirschfeld@hpi.uni-potsdam.de.