Explicit Tool Support for Implicit Layer Activation

Markus Brand
markus.brand@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Germany

Stefan Ramson
stefan.ramson@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Germany

Jens Lincke
jens.lincke@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Germany

Robert Hirschfeld
robert.hirschfeld@uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Germany

ABSTRACT

Implicit Layer Activation (ILA) is a declarative mechanism to scope behavior adaptations in Context-oriented Programming (COP). ILA binds the activation status of a layer to a Boolean condition. The layer is active as long as the given condition evaluates to true. This mechanism to scope layer activations is very powerful, but without dedicated tool support, it may be hard to debug due to its implicitness. A solution that can mitigate this is proper tool support, which is expensive to build and can be highly domain-specific. We have previously shown that by building the language extension not from scratch but by relying on a common more powerful shared concept, Active Expressions, the implementation becomes simpler and more elegant since it does not require deep integration into the ContextJS implementation. In this paper we show how providing tool support for ILA makes implicit dependencies to state changes more explicit. We show how such tool support can be implemented by leveraging the existing Active Expression tool suite. We illustrate the usage based on a catalog of COP questions from literature.

KEYWORDS

Programming Tools, Active Expressions, Context-oriented Programming, Lively Kernel, ContextJS

ACM Reference Format:
Markus Brand, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2022. Explicit Tool Support for Implicit Layer Activation. In *COP 2022: International Workshop on Context-Oriented Programming and Advanced Modularity (collocated with ECOOP) (COP '22), June 7, 2022, Berlin, Germany*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3570353.3570355

COP '22, June 7, 2022, Berlin, Germany

ACM ISBN 978-1-4503-9986-9/22/06…$15.00
https://doi.org/10.1145/3570353.3570355

1 INTRODUCTION

As the name already suggests implicit layer activation is an indirect way to control the scope of behavioral adaptations in context oriented programming. There are serveral context oriented programming (COP) implementations supporting implicit layer activation (ILA) [1, 11, 12, 17, 21]. ILA is different from other COP layer activation mechanisms such as dynamically scoped or global layer activation, in that it is not explicitly activated with a control statement, but behavior gets activated automatically whenever an expression would evaluate to be true. Consequently, not the developer, but the system is in charge of identifying context boundaries.

Using this concept may result in more declarative code. At the same time, its implicit nature makes it hard to debug. For example, it is not clear what imperative change in some program states might trigger an ILA. Dedicated tool support can mitigate such problems when for example the line that would trigger an ILA is highlighted in the code editor, the layer activation becomes visible. Building such powerful tool support that makes implicit dependencies explicit through visualizations and dynamic highlighting is hard and expensive. In a previous paper [17] we have shown that with the help of Active Expression, adding implicit layer activation to ContextJS is possible just by using the public API and without having to change the underlying layer composition mechanism itself. In this paper, we show that building on top of Active Expression (AE) further allows us to reuse the underlying AE tool-suite. The implicit layer activation tools presented in this paper are part of a bigger evaluation of language extensions and reactive concepts that build on AE [2]. In that work, the whole AE tool suite, including domain-specific extensions for implicit layer activation, Signals, and other reactive concepts (together called State-Based Reactive Concept) are presented and evaluated. In this paper, we concentrate on proper tool support for implicit layer activation that uses visualizations and highlights to make implicit dependencies explicit and thus more comprehensible.

The remaining of this paper is structured as follows. In section 2, we look at what implicit layer activation is, give an introduction to Active Expression, and an overview of the existing AE tool suite. section 3 presents our approach to a tool set for implicit layer activation. section 4 presents the implementation in Lively4. section 5 uses six questions from literature applied to an example to illustrate the usage of our tool set. Finally, section 6 concludes this work.

2 BACKGROUND

As a basis for the ILA toolset, we need to look at the COP language extension, AE, and its tools suite [2].

2.1 Implicit Layer Activation

To understand implicit layer activation, we first have to look at method layers, [3]. Method layers are a language construct from COP, where methods can be layered with toggleable functionality.

```
let baz = new Layer(); // Layer definition
const foo = {
  bar() { // bar is a layered method
    return 17;
  }
};

baz.refineObject(foo, {
  bar() { // Partial method definition
    // proceed calls the original bar
    return 42 + proceed();
  }
})
```

Listing 1: Basic example of a layer refining a method

As seen in listing 1, this functionality is specified by defining a layer (`baz`) with a partial method (`bar`), which executes code and can use the underlying partial method using a `proceed()` call. The system, therefore, behaves similarly to overriding methods in inheritance, except that the additional functionality is active 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 [12], most of which require developers to model context switches explicitly. In contrast, the concept of implicit layer activation [21] 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.

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

While this behavior can be implemented in an imperative style, an argument in favor of a reactive implementation can be made [17]. While the imperative implementation performs better in a system with frequent context switches, the reactive implementation is more suitable if the context-dependent behavior is called frequently and the condition is time-intensive to evaluate. Further, the reactive implementation allows for eager life-cycle callbacks, which can run code registered by a `onActivate` or `onDeactivate` method, as soon as a layer toggles its state.

```
baz
  .onActivate(() => lively.notify("Enabled"))
  .onDeactivate(() => lively.notify("Disabled"));
```

For such a reactive implementation, an AE can be used to watch the boolean expression and activate or deactivate the layer whenever the expression changes.

2.2 Active Expressions

The AE [16] is a reactive primitive that automatically detects changes in the return value of an expression. When a change occurred, previously registered callbacks are automatically called with

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

Listing 2: Most basic form of an Active Expression

the new value of the expression. Listing 2 displays the syntax of this concept in its original implementation in the Lively4 system in JavaScript.

There are two main approaches to implementing this thin interface: First, the explicit method reevaluates the monitored expressions and compares the result to see if it changed. The reevaluations can either be triggered by the user with `check();` calls, similar to the observer pattern [5], or automatically in small intervals by the system. This approach brings the disadvantage of either requiring a lot of manual work from the user or causing a lot of unnecessary reevaluations of the expression. The second method requires the identification of dependencies, which are variables that can change an AE, and the monitoring of the dependencies' write accesses. This can be achieved in several non-trivial ways, like modifying the VM or setting up code hooks via reflection, or performing a source code transformation. From here on, we abstract from the concrete implementation and only assume that an AE knows its dependencies.

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. An example program that makes use of this can be seen in Listing 3.

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

readTemperature(val) {
  t = val;
}
readHumidity(val) {
  h = val;
}
```

Listing 3: AE based reactive temperature sensor

2.3 Active Expression-based Implementation of Implicit Layer Activation

The AE based implementation of ILA is realized through source code transformation. The rewriting process is quite straightforward: Figure 1 shows how code is transformed using a babel abstract syntax tree (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 a 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 1. 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

Before transformation

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

After transformation

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

Figure 1: implicit layer activation syntax in the Lively4 system, which is rewritten using AEs.

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.

2.4 Active Expression Tool Set

The AE tool set consists of four tools: *The code annotations* augment the code to highlight the interface between the imperative and reactive worlds and provides an entry point into the other tools as well as code navigation capabilities. *The AE overview* shows all AEs that are currently in the system. *The event timeline* provides a temporal overview of the reactive system. *The dependency graph* gives a structural overview.

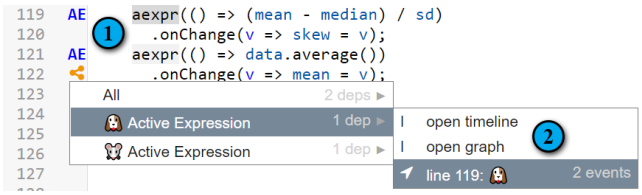

Figure 2: The code annotations add indicators at lines of code that define or trigger reactive behavior (1). The context menu of a line, changing a dependency, links to all Active Expression written by the dependency (2).

2.4.1 Code Annotations. 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 2, 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 runtime dependency information back onto the static code. 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. 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.

2.4.2 Overview. An overview of the AEs in a system is an important first step in investigating reactive behavior. As seen in figure 3, 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, like disposing or setting their logging behavior (3).

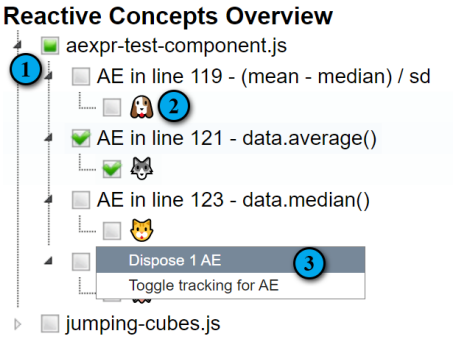

Figure 3: Overview showing all Active Expressions in a hierarchical tree

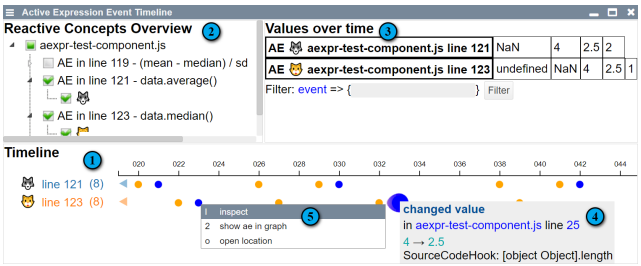

Figure 4: Event timeline tool depicting events and values of selected Active Expressions over time

2.4.3 Event Timeline. To properly reify the mental model of AEs a temporal overview of the reactive behavior is required. To achieve this, the timeline visualizes events that happened during the lifetime of an AE, as displayed in figure 4. These events include value and dependency changes, (de-)registration of callbacks, and created and disposed 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-to 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).

2.4.4 Dependency Graph. The dependency graph visualizes the reactive graph. As shown in figure 5, 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

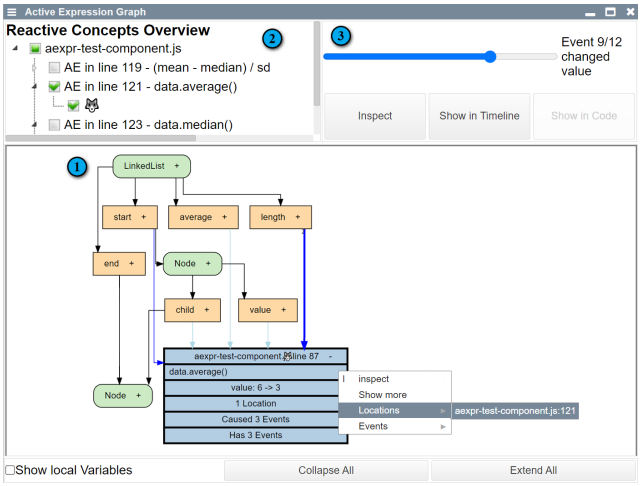

Figure 5: The Dependency graph (1) with Active Expression overview for selection (2) and event selection for time travel (3)

as the timeline to allow the user to filter and select relevant AEs to show in the graph (2). One of the main features of the graph is that it combines the temporal and structural dimensions of the reactive graph. An event selection slider allows the user to select an event from one of the displayed AEs (3). The graph always shows 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 reactive graph over time. The graph also highlights the changes the selected event caused (the edge from length to the AE is highlighted).

3 IMPLICIT LAYER ACTIVATION TOOL SET

Using the default tool suite for Active Expressions allows will show some information at run time in the development tools, but more specialized adaptations are needed. For example, implicit layer activations limits the relevant value range to a boolean value and has a more specialized reaction.

3.1 Running Example: Online Editor

To build a better understanding ILA and to better demonstrate how the debugging requirements of this concept 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. Note, that this example is part of a bigger evaluation [17] and uses also the reactive concepts "signals", demonstrating that the AE tool suite can be used to work on different reactive concepts on the same code base.

The offline text editor has two functions of interest, shown in listing 4: `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

```
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 4: 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 5: 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 6: Method layer activation and event handling

updating the DOM when a change occurred. 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, content 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 2.1) called `onlineLayer` (see listing 5) 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 [15] and improving separation of concerns [4, 18]. 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.

We want this additional behavior to only be active while a connection to the server is established and the programmer chose to work remotely, which is specified with the `activeWhile` function

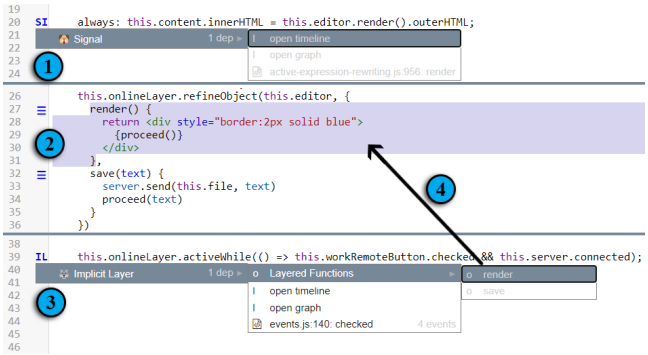

Figure 6: This figure shows the code view adjusted for signals (1) and implicit layer activation (2 and 3). Methods with an overview icon in the gutter can navigate to other partial layers of the method (2). The ILA definition (3) can navigate to partial methods (4) it refines.

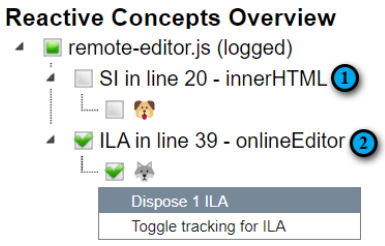

Figure 7: Reactive concepts overview for signals (1) and implicit layer activation (2) adapted from the Active Expression overview.

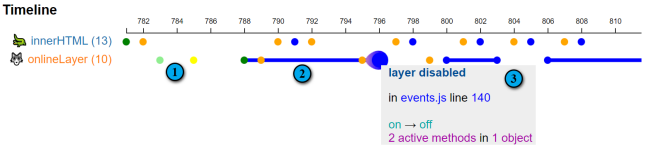

Figure 8: Timeline view depicting an implicit layer activation with new event types (1), intervals that depict when a layer is active (2), and a specialized event (3)

(see listing 6). Further, we specify that when the server connection comes back online, or a change occurred on the server, we want to merge server and client texts. When the connection to the server is lost, but the programmer still wishes to work online, a message is printed.

3.2 Code Annotations

The code view mainly focuses on the change detection aspect. However, it should also be able to visualize the specialized reaction behavior. The ILAs 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 6, 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.

3.3 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 7). For an ILA, we show the corresponding method layer name instead of the AE code as its identifier (2).

With these small changes, the tool is equipped to give the programmer an overview of all reactive concepts currently used in the system.

3.4 Event Timeline

Other State-Based Reactive Concepts (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 8. 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 runtime 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 8 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.

3.5 Dependency Graph

Since SBRCs based on AEs specializes in 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 implicit layer activations 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 9, 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.

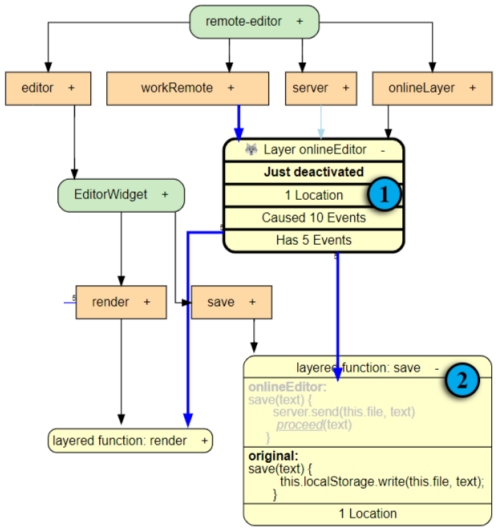

Figure 9: Dependency graph adjusted for implicit layer activations: The internal Active Expression nodes are hidden and replaced with specialized nodes. The yellow nodes (1 and 2) show a deactivated layer.

```
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 10: Code annotations at a layered method

4 IMPLEMENTATION

The debugging toolsets are implemented in the Lively4 system [7, 9, 13]. Lively is a live object computing environment [7, 8, 10, 19] for the web implemented using JavaScript. Like Squeak/Smalltalk [8], the Lively system is a self-sustaining system (S3) [6], 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 self-sustaining system (S3) allows us to implement and execute both a programming concept and its debugging tools in the same environment [14].

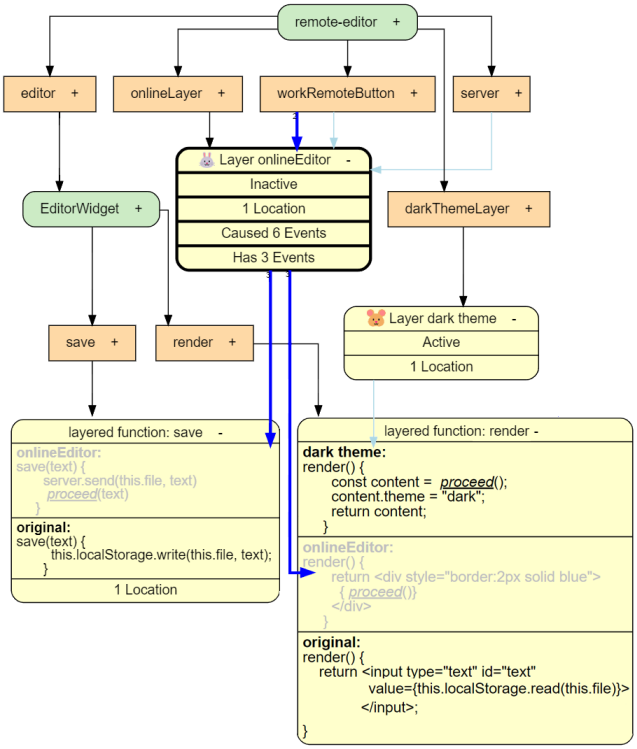

Figure 11: Dependencies of dark theme and online layers

4.1 Dynamic Analysis of AE

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 AST transformation with babel[1] 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 into 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

[1] https://babeljs.io/

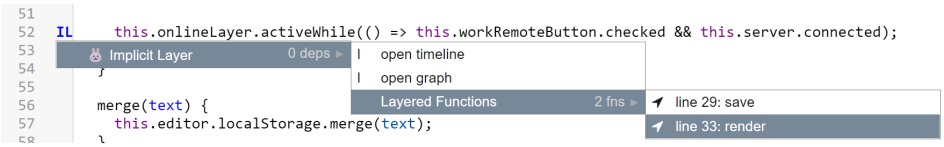

Figure 12: Code annotations at a ILA declaration.

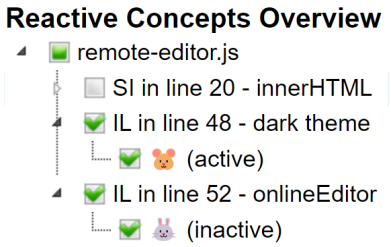

Figure 13: Concepts overview of ILA and a signal

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.

4.2 Active Expression Tools in Lively4

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 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.

4.3 Code Annotations

Lively4 uses CodeMirror[2] as its code editor. The code annotations add a gutter for reactive behavior, which displays appropriate icons in lines of code containing AE definitions or dependencies. To achieve this, it queries the global AE cache and aggregates the data per line of code. When the icon is clicked, a context menu appears that lets the user jump into one of the other two tools or navigate from an AE to its dependencies in the code or vice versa. It also shows an additional icon when the evaluation of an AE failed in the line of the AE definition and the triggering dependency. This notifies the programmer that something is probably not as expected and encourages them to use the tools to determine the reason.

[2]https://codemirror.net/ (August 8, 2021)

Further, we implemented functionality for aggregating multiple annotations in one line. Instead of the specific icon, a generic RE (for reactive) icon is shown, and the context menu adds a top layer, with an entry for each annotation.

4.4 Adapting for Implicit Layer Activation

These implementations are based on AEs and the AE debugging toolset respectively. Figure 1 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`.

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 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.

4.4.1 Code Annotations. The code annotations introduce a new icon for ILA. The context menu at the code locations that define the ILA is 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 10. 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.

4.4.2 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.

4.4.3 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.

4.4.4 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.

5 USAGE EXAMPLES

To illustrate the capabilities of the tools, we try to answer questions that occur while debugging. Taeumel et al [20] proposed six questions for debugging method layers. We use these questions and try to answer them with our toolset.

We will reuse the online editor example from section 3.1 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.

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 11. A `layeredFunction` node shows all layered methods and has a connection to each layer node that refines it.

The second point is the code annotations at each layered method. Figure 10 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 runtime 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 12. 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 11).

Q5: Which layers are currently active in process P?. All currently registered layers are listed in the overview of the reactive concept, as seen in figure 13. 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.

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 CONCLUSION

In this paper, we presented dedicated tool support for ILA. Instead of rebuilding the tools from scratch, we leveraged the existing tool for AE and extended it to cater to its domain-specific needs. In particular, the tools show the relation of layers and their dependencies in the object graph, they show the activation history of layers in the timeline, and they annotate static source code with dynamic runtime information.

We implemented the tools for ContextJS ILA in the Lively4 development environment. We illustrated the usage of the tool suite by answering typical code comprehension questions for COP systems known from the literature.

REFERENCES

[1] Engineer Bainomugisha, Jorge Vallejos, Coen De Roover, Andoni Lombide Carreton, and Wolfgang De Meuter. 2012. Interruptible Context-Dependent Executions: A Fresh Look at Programming Context-Aware Applications. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Tucson, Arizona, USA) *(Onward! 2012)*. Association for Computing Machinery, New York, NY, USA, 67–84. https://doi.org/10.1145/2384592.2384600

[2] Markus Brand, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2022. Extensible Tooling for Reactive Programming Based on Active Expressions. (2022).

[3] Pascal Costanza and Robert Hirschfeld. 2005. Language Constructs for Context-Oriented Programming: An Overview of ContextL. In *Proceedings of the Dynamic Languages Symposium*. https://doi.org/10.1145/1146841.1146842

[4] Edsger W. Dijkstra. 1982. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc.

[6] Robert Hirschfeld and Kim Rose (Eds.). 2008. *Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15-16, 2008, Proceedings*. Springer-Verlag. https://doi.org/10.1007/978-3-540-89275-5

[7] Daniel Ingalls, Tim Felgentreff, Robert Hirschfeld, Robert Krahn, Jens Lincke, Marko Röder, Antero Taivalsaari, and Tommi Mikkonen. 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 (Onward! 2016)*. ACM. https://doi.org/10.1145/2986012.2986029

[8] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. *ACM SIGPLAN Notices* 10 (1997). https://doi.org/10.1145/263700.263754

[9] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. 2008. The Lively Kernel A Self-supporting System on a Web Page. In *Self-Sustaining Systems, First Workshop, S3 (Lecture Notes in Computer Science)*, Robert Hirschfeld and Kim Rose (Eds.). Springer. https://doi.org/10.1007/978-3-540-89275-5_2

[10] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. 1988. Fabrik: A Visual Programming Environment. *ACM SIGPLAN Notices* 11 (1988). https://doi.org/10.1145/62084.62100

[11] Hiroaki Inoue and Atsushi Igarashi. 2016. A library-based approach to context-dependent computation with reactive values: suppressing reactions of context-dependent functions using dynamic binding. In *15th International Conference on Modularity (MODULARITY), 2016*. 50–54. https://doi.org/10.1145/2892664.2892669

[12] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2016. Generalized Layer Activation Mechanism for Context-Oriented Programming. In *Transactions on Modularity and Composition I*, Shigeru Chiba, Mario Südholt, Patrick Eugster, Lukasz Ziarek, and Gary T. Leavens (Eds.). Springer International Publishing. https://doi.org/10.1007/978-3-319-46969-0_4

[13] Jens Lincke, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. 2017. Designing a Live Development Experience for Web-Components. In *PX/17.2*. https://doi.org/10.1145/3167109

[14] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 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. https://doi.org/10.1145/3426428.3426919

[15] David Lorge Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 12 (1972). https://doi.org/10.1145/361598.361623

[16] Stefan Ramson and Robert Hirschfeld. 2017. Active Expressions: Basic Building Blocks for Reactive Programming. *The Art, Science, and Engineering of Programming* 2 (2017). https://doi.org/10.22152/programming-journal.org/2017/1/12

[17] Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2017. The Declarative Nature of Implicit Layer Activation. In *Proceedings of the 9th International Workshop on Context-Oriented Programming (COP '17)*. ACM. https://doi.org/10.1145/3117802.3117804

[18] Chris Reade. 1989. *Elements of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc.

[19] Ivan E. Sutherland. 1964. Sketch Pad a Man-Machine Graphical Communication System. In *Proceedings of the SHARE Design Automation Workshop (DAC '64)*. ACM. https://doi.org/10.1145/800265.810742

[20] Marcel Taeumel, Tim Felgentreff, and Robert Hirschfeld. 2014. Applying Data-driven Tool Development to Context-oriented Languages. In *Proceedings of 6th International Workshop on Context-Oriented Programming (COP'14)*. ACM. https://doi.org/10.1145/2637066.2637067

[21] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. 2007. Context-Oriented Programming: Beyond Layers. In *Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007 (ICDL '07)*. ACM. https://doi.org/10.1145/1352678.1352688