



ReactCOP Supporting Layer Parameter Management for Front-End Web Applications

Hiroki Hashimoto

Tokai University
Information and Telecommunication Engineering
Tokyo, Japan
hasihiro1117@gmail.com

Nobuhiko Ogura

Tokyo City University
Faculty of Informatics
Yokohama, Japan
ogura@tcu.ac.jp

Ikuta Tanigawa

Change Vision Inc.
Fukui, Japan
ikuta.tanigawa@change-vision.com

Harumi Watanabe

Tokai University
Information and Telecommunication Engineering
Tokyo, Japan
harumi-w@tsc.u-tokai.ac.jp

ABSTRACT

In modern software, including web applications, context-dependent behavior is one of the most important features. Context-oriented programming (COP) is a suitable programming technique for developing such software. However, we often need to work on handling parameter values in layers. This problem which we experience during the development of web application means inconvenience in setting parameters by each layer. We call it “layer parameter problem.” Especially front-end web applications use a component-based approach with a DOM tree, making the layer parameter problem more complicated because they cannot handle COPs in class-in-layer and layer-in-class models. We propose ReactCOP, an implementation of an idea that applies COP to React, one of the front-end web application libraries. ReactCOP solves the parameter problem on a layer-in-component model. As the solution to this problem, we present Layer Parameter Management that dynamically switches values in a variable within a layer. In this paper, we propose ReactCOP with Layer Parameter Management. Finally, we investigate our proposed approach through two case studies.

CCS CONCEPTS

• **Software and its engineering** → *Object oriented languages*.

KEYWORDS

React, Context-Oriented Programming, Web Application

ACM Reference Format:

Hiroki Hashimoto, Ikuta Tanigawa, Nobuhiko Ogura, and Harumi Watanabe. 2023. ReactCOP Supporting Layer Parameter Management for Front-End Web Applications. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming*



This work is licensed under a [Creative Commons Attribution International 4.0 License](https://creativecommons.org/licenses/by/4.0/).

<Programming>'23 Companion, March 13–17, 2023, Tokyo, Japan

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0755-1/23/03.

<https://doi.org/10.1145/3594671.3594684>

(<Programming>'23 Companion), March 13–17, 2023, Tokyo, Japan. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3594671.3594684>

1 INTRODUCTION

1.1 Context-Dependent Behavior Adaptation in Web Application

In front-end web application development, a technology for dynamically adapting behavior and contents on the context is required for cases such as differences in location or time. For example, a web application that automatically analyzes stock markets in real-time is expected to dynamically change analysis methods that match the market situation based on the stock price.

For the adaptation of dynamic context-dependent behavior, there is Context-oriented programming (COP) [1, 3–5]. Layers provided by COP support the modularization of context-dependent behaviors, and COP applications de-/activate layers to adapt those behaviors at runtime.

Web application development frameworks and libraries are used for effectively developing web applications. React [10] is a JavaScript library for building user interfaces and one of the most popular libraries in the field. React app builds a complex UI by combining modules called components. Rendering is the key point of React application. Rendering is a process that decides whether the component has to be reloaded when there is a change in the state of the component, React checks each component by the Reconciliation process using virtual DOM, and React reloads only the components that need to be updated.

1.2 COP in Web Application Development

Various COP languages and libraries are proposed [2, 13–15], and there are a few COP libraries proposed for web application development, such as ContextJS[9] and EMAs[8].

ContextJS is a JavaScript library based on an open implementation for layer activation, allowing customized adaptation rules. In the research, Jens Lincke et al. proposed a holistic approach that integrates new scoping strategies with existing strategies based on an open implementation [6, 7] in which layer composition strategies are encapsulated into objects.

EMAjs is a JavaScript implementation based on an expressive and modular activation mechanism for COP. The mechanism allows developers to implement their own activation mechanism matching their needs.

However, no extensions have been proposed to work directly with front-end web application libraries such as React to support dynamic behavior adaptation. React uses a component-based approach with a DOM tree. For adapting the COP mechanism using layers in React components, it is important to implement so that the rendering process and the layer operation processing coexist. Especially after each de-/activation of layers, React should execute rendering processes to switch the application's behavior and contents. `useState`, one of the functions provided by React, is effective to be adapted into the layer mechanism and associated layer update process and the rendering process call.

1.3 Layer Parameter Problem

As mentioned above, React uses rendering to update the UI caused by data changes. Components that have been updated are reloaded by rendering, and information other than values protected by `useState`, etc., in the component will be reset at each update. The prototype of ReactCOP realizes the idea of a layer in React. However, to make layers and their associated behaviors even more effective, we need the feature to protect layer-compliant variable values, which we call Layer Parameters. For implementing this feature, it is necessary to solve the problem of complex interrelationships between layer activation states and rendering processing timings. Therefore, a mechanism is required to prevent variable values from reset by switching a layer's active state. Implement this as a custom Hook and combine it with the layer mechanism to allow the implementation of dynamic variable value adapted to layers.

1.4 Implementation Goals

Our research aims to develop a COP extension of React that solves the aforementioned problems. We illustrated a rough outline of the extension in Figure 1.

1. Scope of layers
A class provides Layer information and layer controls within the extension. By wrapping the top-level component of the React application with a component (LayerProvider) that contains an instance of that class, the layer mechanism is shared throughout the application, and layer information can be accessed from all child components.
2. Layer activation
Activation and deactivation of layers are performed using class methods. Those methods can be called from anywhere in React application.
3. Behavior adaptation
Behaviors are implemented by a custom hook inspired by `useEffect`, and the callback is executed if the specified layer is active. Also, a component (`<Layer>`) that changes the content according to the layer state is provided to add behavior within JSX.

The novel point of programming experience is to solve the layer parameter problem of COP in the development of front-end web applications. The source of ReactCOP is available at [11]. The

remainder of this paper is structured as follows. Section 2 describes our approach to COP with React. Section 3 introduces the notation of ReactCOP. Its usage is presented by two case studies in Section 4. Finally, Section 5 provides the conclusion.

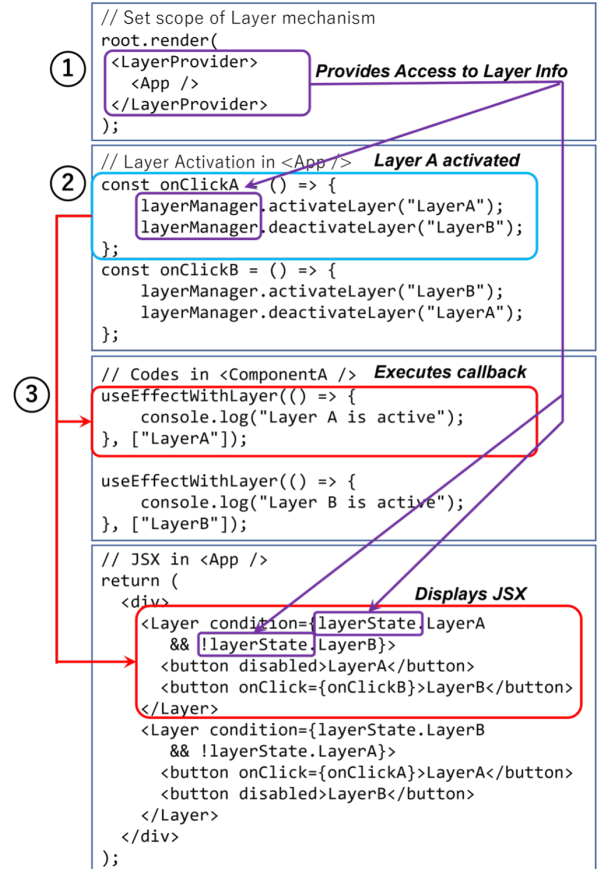


Figure 1: Goal of COP in React

2 REACTCOP

2.1 Layer Mechanism

2.1.1 Layer & De-/Activation. React application is structured as a component tree, and layer definitions are spread among components in React application. According to the general classification[12], Our approach is a layer-in-component model where layers exist within components. With the layer-in-component model, layer declarations are outside the lexical scope of the code they modify. A layer can express by defining layer information and layer operation processing in the LayerManger class. Layers and their states are stored in JavaScript objects. The layer name is used as a key, and the layer's state is expressed as a value (true/false). Activation/deactivation of a layer is a process of exchanging the values of its objects. When updating the layer's state, it is necessary to change the rendering content in the application to correspond to the activated layer behavior. There is a way to intentionally trigger React rendering by adding a counter with

useState. useState is a hook that triggers rendering when the value is updated and reflects the latest value in the application. By adding this hook to the activation process, rendering can occur after each de-/activation process, and the contents can be switched to the latest one. Importantly, the rendering process recursively renders all child components within it once the parent component has been rendered. Therefore, by adding the above processing to the top level of the React application, changes in layer activation status can be propagated throughout the application. We illustrated the relation of the layer mechanism and the rendering process in Figure 2.

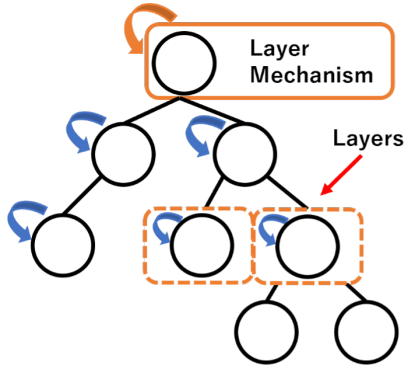


Figure 2: Relation of the layer mechanism and rendering process

2.1.2 Scoping of Layer Composition. Layers should be accessible from anywhere in the application. For extending the layer mechanism's scope to all components, React's context feature can be used to treat the layer mechanism as a global property. React applications typically allow data to be shared by passing data from parent components to child components (top-down). However, if there is a property that is needed by many components in the application, such an approach can lead to a loss of productivity. The context feature is effective for the situation, and a component is called Context.Provider and a hook called useContext are provided from React. Context.Provider provides context values as global properties to all child components. useContext is for getting the context value in the child component. Combining these two functions and the layer mechanism allows layer information to be accessed and controlled from anywhere.

2.1.3 Behavior Adaptation. In a React application, we believe that adding behavior should be done to internal processing and JSX of components. For describing the behavior in internal processing, a custom Hook (useEffectWithLayer) that applies useEffect can be used. useEffect is a function provided by React that executes a callback when the specified variable value changes. (1) Use this feature to implement a custom Hook that will execute a callback when the specified layer is activated. (2) Set the behavior as a callback, and (3) Pass the layer state as the second argument to the custom hook so that the active layer behavior (callback) will be executed when the layer state is updated. Behavior description for JSX is made possible by enclosing it in a React component (<Layer>) that operates (adds/deletes) the behavior according to

the layer status. The component returns a behavior if the layer is active and returns empty data if the layer is inactive. We believe that these processes can be easily implemented using conditional branching.

2.2 Layer Parameter Management

While discussing ReactCOP further and putting the functionality into practical applications, we encountered a situation where the value needed to revert to the previous value of the layer variable. In this section, we describe the problems extracted from the design and implementation process of ReactCOP and our solutions. In React, the values held by variables will revert to their initial values with re-rendering unless protected. In a regular React app, we can use useState to persist values after rendering. However, when we combine the concept of layers with React apps, there are values that need to protect separately for each layer. In some situations, variables within layers should be retained and changed for each layer, and variables within layers should be switched to layer-specific values by layer switching. We call such variable Layer Parameter and introduce Layer Parameter Management to solve this problem.

Figure 3 shows changes in Layer Parameter values using Layer Parameter Management. There are two layers, A and B, and a variable x is in a React component under a layer. As an initial value, 1 is stored in x of Layer A, and 10 is stored in x of Layer B. After initialization, Layer A is activated, and 2 is stored in x , and update the value to 2 with the setter method. When Layer B becomes active (Layer A is deactivated), x 's value changes to 10, which is the initial value of Layer B. When Layer A has been activated again, the value of x will be 2, which was set at the previous activation. The above process can be realized by defining an object using useState in a custom hook and implementing the process of substituting each variable value with the layer as a key.

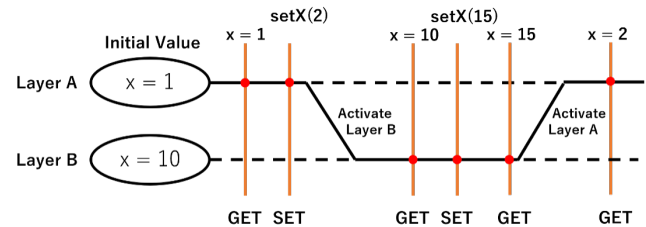


Figure 3: Value Flow with Layer Parameter

3 NOTATIONS OF REACTCOP

In the following section, we introduce ReactCOP to achieve the above goal. ReactCOP is a specialized contextual behavioral adaptation of React components that could be considered as dynamic cross-cutting concerns. ReactCOP helps to deploy context-dependent behavior at JSX and the rest of the components. For JSX to reflect de/activation, it simply checks the activation status of layers, and based on the status, it returns empty <Fragment> tag or child components to the parent component. For the rest of the components to reflect de/activation, we implement a custom

Hook named `useEffectWithLayer`. We explain the notation of the methods and other methods that support the implementation of context-aware react applications.

3.1 Layer Description

This section describes adding behavior within a layer. `<Layer>` is a method for JSX, and `useEffectWithLayer` is a method that describes the behavior for internal processing of the component.

`<Layer>`

This tag shown in Listing 1 is a method for reflecting components and tags according to the layer activation status within JSX. In the JSX description, a part enclosed by the method is valid only when the layer specified by the name attribute matches conditions which are states of layer names. The second argument is children, which are React components. By calling `layerState.LAYERNAME`, the Boolean value of whether the layer is active, will be returned. If the condition is true, the Layer method returns the child components, and if not, it returns nothing.

Listing 1: Notation of Layer method

```
<Layer condition={ layerState.LayerA
  && !layerState.LayerB }>
  <CompA />
</Layer>
<Layer condition={ layerState.LayerB
  && !layerState.LayerA }>
  <CompB />
</Layer>
```

`useEffectWithLayer` / `useEffectWithoutLayer`

This method shown in Listing 2 is a custom hook for implementing processes that trigger based on the change of layer active status. It takes a layer name as one of its arguments, and when React detects an update of the layer name, it calls two `useEffect` methods inside of this custom hook. It has three arguments, callback as the first argument, layer state as the second argument, and dependency as the third argument. A callback is executed only when the layer given in the second argument is true. The third argument is the same as the second argument of `useEffect`. This method helps to implement partial methods and the de/activation process of layers.

Listing 2: Notation of `useEffectWithLayer`

```
useEffectWithLayer(() => {
  // Some Processes here
}, layerState.LayerA, [dependencies]);
```

3.2 Layer De-/Activation

`LayerProvider`

This method shown in Listing 3 is used to set a scope of contextual information and layers deployment. It defines `LayerManager`, a class of managing layers and their status, and `layerStateCount` with `useState`. This allows all child components of `LayerProvider` to use layer information and activation processes stored in `LayerManager`. Components that contain layer definition or layer de/activation should be enclosed in `LayerProvider` as shown below. It often should be set on the top level of components such as `<App />`.

Listing 3: Notation of `LayerProvider`

```
const root =
  ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <LayerProvider>
    <App />
  </LayerProvider>
);
```

`LayerManager`

`LayerManager` provides the functionality of managing layer definitions and activate information. Layer information is stored in an object with layer names as keys and activation status as values. Field variables contain `layerStateCount` and its setter to detect de/activation calls on React application.

`activateLayer(layerName)`

A method to use for layer activation. Pass layer name as a string argument. Set value of the layer name in the object to 'true' and add 1 to `layerStateCount` using `notifyUpdatedLayerState`.

`deactivateLayer(layerName)`

A method to use for layer deactivation. Pass layer name as a string argument. Set value of the layer name in the object to 'false' and add 1 to `layerStateCount` using `notifyUpdatedLayerState`.

`getLayerState(layerName)`

A method to return `layerState`. Returns an object containing all layer information. By calling this method, layer information can be accessed from anywhere in React components.

`useLayerManager`

This method shown in Listing 4 is for getting access to `LayerManager` instance. It returns `LayerManager` instance and by using the instance, methods for de/activation and getting layer status, shown in above, can be implemented in React application. It stores `layerStateCount`'s value and `setLayerStateCount` method when it is called and integrates `layerStateCount` and its setter into `LayerManager`.

Listing 4: Notation of `useLayerManager`

```
const layerManager = useLayerManager();
useEffect(() => {
  layerManager.activateLayer("LayerA")
  layerManager.deactivateLayer("LayerB")
}, []);
```

3.3 Layer Parameter Management

`useLayerParams`

This method, shown at the top of Listing 5, is a custom React Hook that lets us add a state variable to components at each layer. It is mainly the same as `useState` hook, but the value switches with layer status changes. This method allows developers to hold different values for each layer with only one variable. The value in the variable is replaced depending on the activation status by calling getter method. The first argument is initial values, and the second argument is an array of layer names that use the defining variable. There are two return values which are getter and setter for the state

variable. To set values to the variable, requires to use of the setter, and the getter needs to be called to get the value, as shown at the bottom of Listing 5.

Listing 5: Notation of useLayerParams

```
const [getCount, setCount] =
  useLayerParams(0, ["LayerA", "LayerB"]);

const onClick = () => {
  setCount((ct) => ct + 1);
  setText((pre) => pre + getButtonLabel());
};

// JSX
return (
  <div>
    <p>CountA: {getCount("LayerA")}</p>
    <p>CountB: {getCount("LayerB")}</p>
    <button onClick={onClick}>{getButtonLabel()}</button>
    <b> {getText()}</b>
  </div>
);
```

4 CASE STUDY

In the following section, we introduce two case studies to investigate our ideas and approach with ReactCOP. First case study in Section 4.1 shows the implementation of layers and de-/activation. Second case study in Section 4.2 shows the implementation of Layer Parameter Management.

4.1 Case Study for ReactCOP

The example is a chat application built with React, shown in Figure 4. It is a simple application that users can communicate with each other using texts and calls. As a component tree diagram of the application shown in Figure 5, this application is structured with two main components, Navbar and ChatScreen. Navbar component provides access to application functionalities and has time or notification that is displayed based on context. ChatScreen component provides general chat features such as a contact list, text chat, and access to media and information.

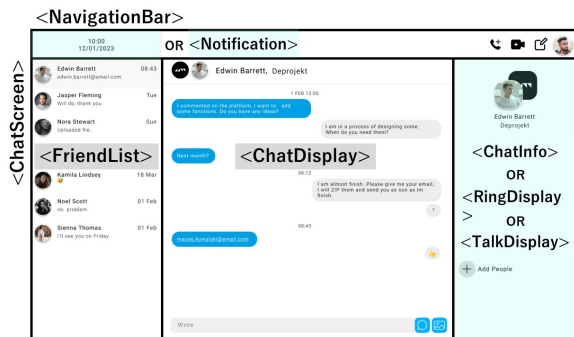


Figure 4: Image of Chat Application Display

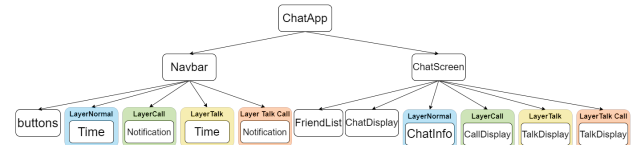


Figure 5: Component Tree Diagram of Chat Application

4.1.1 Context-Dependent Behaviors. As the application context, the presence or absence of incoming and ongoing calls can be considered. These contexts are represented by two Boolean variables: `isRinging` and `isTalking`. Context-dependent behaviors that provide functionalities for each situation can be implemented in the two main components, `Navbar` and `ChatScreen`. Also, we think that the creation of three layers is needed to express the behaviors. The definition of each layer and the conforming components are shown in the following items.

Normal layer (both `isRinging` and `isTalking` are “false”)

This layer provides functions and displays rendering when there is no incoming call or no ongoing call. It is activated when `isRinging` and `isTalking` are false. In ordinary COP systems, this layer is defined as a base layer, but in React app, due to the rendering position of components, we focus on switching child components instead of adding partial processing by layers. By activating this layer, `Time` component, which displays the current time, will be rendered in `Navbar`, and `ChatInfo` component, which provides access to media and chats information, will be added in `ChatScreen` component.

Ring layer

This layer is mainly for providing notification of incoming calls in the application display. Whether or not there is an incoming call determines whether this layer is active or inactive. It is activated when `isRinging` is set to true. By activating this layer, `Notification` component, which displays user information, will be rendered on the left side of `Navbar` component where `Time` was rendered with `Normal` layer, and `RingDisplay`, which displays the user info and provides functions for handling incoming calls in `ChatScreen` component.

Talk layer

This layer provides functions and displays rendering if the user is on a call. This layer is activated when the user accepts an incoming call and `isTalking` is set to true. By activating the layer, `Time` component will be rendered in `Navbar` component, and `TalkDisplay`, which provides In-call features such as video and mute functionality, will be rendered in `ChatScreen` component.

4.1.2 Implementation of Chat Application. A flow diagram with the layers and associated components and activation codes described above is shown in Figure 6. For implementing the context-dependent behaviors in Chat application, three steps are required. The first step is to define the context value and implement the process for updating the value. The context needs to be accessed from anywhere in our case. Therefore, we used

createContext and Context.Provider to make the context variable “global.” The second step is the addition of activation processing that activates and deactivates layers by updating context values. These activation processes can be implemented anywhere within the React component structure to update the activation status of the layer. However, this time we implemented the process in the top-level component (ChatApp component). The final step is to add the component corresponding to the layer within JSX. This step is for internal processing using useEffectWithLayer and drawing processing using components, that is, for implementing behavior for each layer. In the chat application, we only used Layer method to describe the behaviors.

4.2 Case Study for useLayerParams

This section presents an application using useLayerParams to confirm its functionality. The application is a simple counter app that consists of three components, App, CmpA, and CmpB, and has two parameters defined with useLayerParams method. There are two layers, LayerA and LayerB. When LayerA is active, CmpA will be rendered as a child component of App component. Inside CmpA and CmpB are two buttons and functions handling the button click.

4.2.1 Implementation of Counter app. The layer, activation process, and parameters described above are implemented in the App component shown in Listing 6. The parameters are implemented using the useLayerParams method and named Param1 and Param2. Since the method returns a getter and setter for each parameter, we created variables for each and passed them as arguments to the child component. Also, implemented CmpA and CmpB as child components, we will explain details using CmpA shown in Listing 7. Processes to add 1 to the previous value of Param1 and add “World” for Param2, are implemented using the getter and setter passed as arguments. Variables that hold the return value of the getter need to be added, and the setter is required to update the value at onClick function.

Listing 6: Program of App component

```
function App() {
  const layerManager = useLayerManager();
  const layerState = layerManager.getLayerState();

  const [param1, setParam1] =
    useLayerParams(0, ["LayerA", "LayerB"]);
  const [param2, setParam2] =
    useLayerParams("Hello", ["LayerA", "LayerB"]);

  const [isLayerA, setIsLayerA] = useState(true);
  useEffect(() => {
    if(isLayerA) {
      layerManager.activateLayer("LayerA");
      layerManager.deactivateLayer("LayerB");
    } else {
      layerManager.activateLayer("LayerB");
      layerManager.deactivateLayer("LayerA");
    }
  }, [isLayerA]);

  return (
    <div>
```

```
<button onClick={() => setIsLayerA(!isLayerA)}>
  {isLayerA?"LayerA":"LayerB"}
</button>
<Layer condition={layerState.LayerA}>
  <CmpA param1={param1} param2={param2}
    setParam1={setParam1} setParam2={setParam2}/>
</Layer>
<Layer condition={layerState.LayerB}>
  <CmpB param1={param1} param2={param2}
    setParam1={setParam1} setParam2={setParam2}/>
</Layer>
</div>
);
};
```

4.2.2 Result of Counter app. The purpose of this case study is to confirm that variable values that exist within a layer revert to their previous values when the same layer is reactivated. Figure 7 shows the execution result of the counter application as a flow chart. After running the application, layer A becomes active, click the button to add 1 to Param1, and add “World” to Param2. After that, when pressing the button to activate LayerB, the initial values are assigned to Param1 and Param2. When activating LayerA again, Param1 has been re-assigned 1, which was the previous value in LayerA, and “World” has been re-assigned to Param2. From the above, it can be confirmed that using useLayerParams can prevent the variable values from being reset by the layer-switching process.

Listing 7: Program of CmpA

```
function CmpA({param1, param2, setParam1, setParam2}) {
  const param1Value = param1();
  const param2Value = param2();

  const addNumber = () => {
    setParam1(param1Value + 1);
  };
  const addText = () => {
    setParam2(param2Value + " World");
  };
  return (
    <div>
      <button onClick={addNumber}>
        add 1 to param1
      </button>
      <button onClick={addText}>
        add "World" to param2
      </button>
      <p>Param1 is {param1Value}</p>
      <p>Param2 is {param2Value}</p>
    </div>
  );
};
```

5 CONCLUSION

In this paper, we introduced ReactCOP supporting to modularize contextual behavioral variations for front-end web applications. The research goals were (1) designing COP extension for front-end web application development and (2) providing a solution to the layer parameter problem. For the first goal, we implement

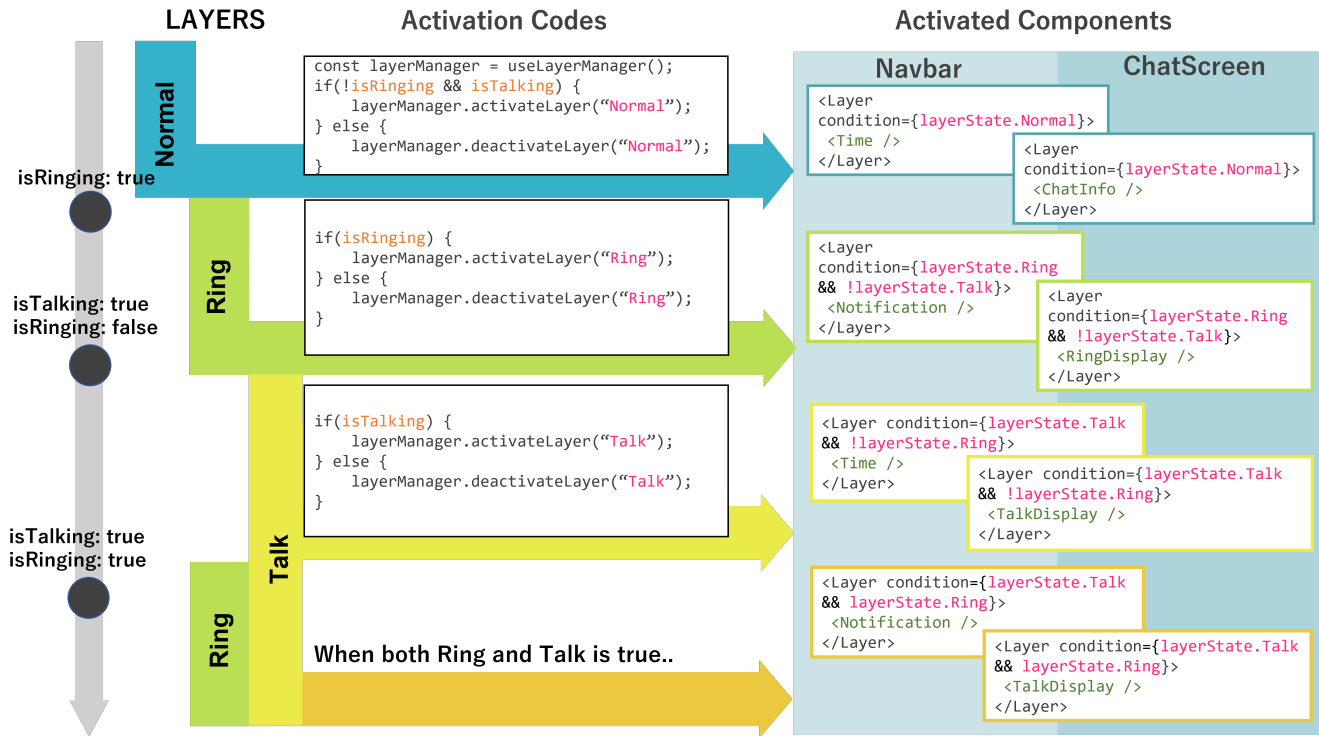


Figure 6: Layer De-/Activation Flow

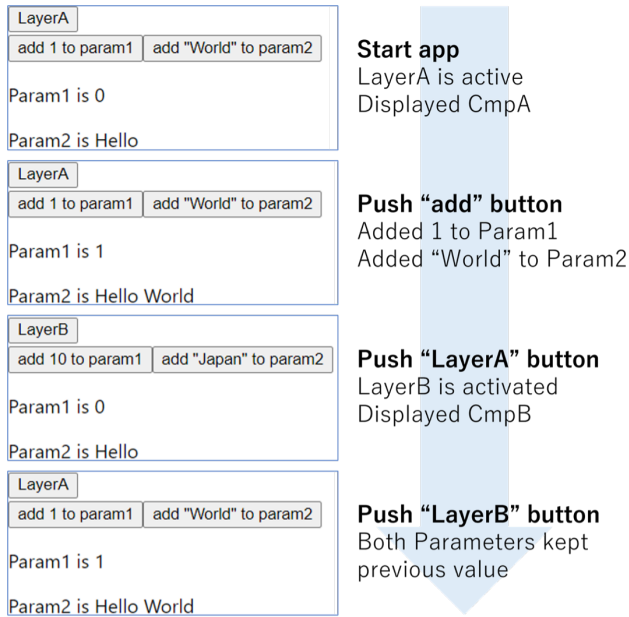


Figure 7: Execution Results of Counter App

ReactCOP with the layer-in-component model. To achieve the second goal, we proposed layer parameter management that dynamically switches values of a variable in a component on a

layer. We confirmed that ReactCOP achieved those goals with two case studies. As future work, validation of ReactCOP, especially on useLayerParam, whether it corresponds to multi-layer activation and other situations, can be considered.

REFERENCES

- [1] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. 2009. A Comparison of Context-oriented Programming Languages. *Proceedings of the 1st ACM International Workshop on Context-Oriented Programming (COP '09)*, 1–6. <https://doi.org/10.1145/1562112.1562118>
- [2] Malte Appeltauer, R. Hirschfeld, Michael Haupt, and H. Masuhara. 2011. ContextJ: Context-oriented programming with Java. *Information and Media Technologies* 6, 2 (06 2011), 399–419. <https://doi.org/10.11185/imt.6.399>
- [3] Pascal Costanza and Robert Hirschfeld. 2005. Language constructs for context-oriented programming: An overview of ContextL. *Proceedings of the Dynamic Languages Symposium*, 1–10. <https://doi.org/10.1145/1146841.1146842>
- [4] Pascal Costanza, Robert Hirschfeld, and Wolfgang De Meuter. 2006. Efficient Layer Activation for Switching Context-Dependent Behavior, D. E. Lightfoot and C. A. Szyperski (Eds.). *7th Joint Modular Languages Conference, JMLC 2006* 4228, 84–103. https://doi.org/10.1007/11860990_7
- [5] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-oriented Programming. *The Journal of Object Technology* 7, 3 (03 2008), 125–151. <https://doi.org/10.5381/jot.2008.7.3.a4>
- [6] Gregor Kiczales. 1996. Beyond the Black Box: Open Implementation. *Software, IEEE* 13, 1 (02 1996), 8–11. <https://doi.org/10.1109/52.476280>
- [7] Gregor Kiczales and Andreas Paepcke. 1996. *Open Implementations and Metaobject Protocols*. MIT Press, Cambridge, MA, USA.
- [8] Paul Leger, Nicolas Cardozo, and Hidehiko Masuhara. 2022. An expressive and modular layer activation mechanism for Context-Oriented Programming. *Information and Software Technology* 156 (12 2022), 107132. <https://doi.org/10.1016/j.infsof.2022.107132>
- [9] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. 2011. An open implementation for context-oriented layer composition in ContextJS. *Sci. Comput. Program.* 76 (12 2011), 1194–1209. <https://doi.org/10.1016/j.scico.2010.11.013>
- [10] React [n. d.]. React — A JavaScript library for building user interfaces. reactjs.org.

- [11] ReactCOP [n. d.]. https://github.com/tanigawaikuta/react_cop
- [12] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. 2012. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software* 85, 8 (08 2012), 1801–1817. <https://doi.org/10.1016/j.jss.2012.03.024>
- [13] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. 2012. ContextErlang: Introducing context-oriented programming in the actor model. *AOSD'12 - Proceedings of the 11th Annual International Conference on Aspect Oriented Software Development*, 191–202. <https://doi.org/10.1145/2162049.2162072>
- [14] Ikuta Tanigawa, Kenji Hisazumi, Nobuhiko Ogura, Midori Sugaya, Harumi Watanabe, and Akira Fukuda. 2019. RTCOP: Context-Oriented Programming Framework based on C++ for Application in Embedded Software. *ICISS 2019: Proceedings of the 2019 2nd International Conference on Information Science and Systems*, 65–72. <https://doi.org/10.1145/3322645.3322689>
- [15] Benjamin Hosain Wasty, Amir Semmo, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. 2010. ContextLua: Dynamic Behavioral Variations in Computer Games. In *Proceedings of the 2nd ACM International Workshop on Context-Oriented Programming* (Maribor, Slovenia) (COP '10). Association for Computing Machinery, Article 5, 6 pages. <https://doi.org/10.1145/1930021.1930026>