A Type System for Flexible User Interactions Handling

Arnaud Blouin

▶ **To cite this version:**

Arnaud Blouin. A Type System for Flexible User Interactions Handling. Proceedings of the ACM on Human-Computer Interaction , 2024, EICS, pp.27. hal-04485762v2

HAL Id: hal-04485762

https://inria.hal.science/hal-04485762v2

Submitted on 29 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

A Type System for Flexible User Interactions Handling

ARNAUD BLOUIN, Univ Rennes, INSA Rennes, IRISA, Inria, France

```
1  val i1: Interaction[DnDData[MousePointData]] = new MouseDnD() ! new KeyTyped("ESC")
2  // val i1: Interaction[DnDData[MousePointData]] = new ReciprocalDnD()
3  val i2: Interaction[DnDData[TouchPointData]] = new TouchDnD()
4  val i3: Interaction[TouchPointData,TouchPointData] = new LongTap(2000) + new Tap()
5  val dnd = i1 | i2
6
7  dnd.on(canvas)
8   .toProduce(data => new MoveShape(data.src, canvas))
9   .when((data: DnDData[MousePointData] | DnDData[TouchPointData]) =>
10     usesLeftButton(data.tgt))
11 i3.on(canvas.children)
12 .toProduce(data:[TouchPointData,TouchPointData]=> new DeleteShape(data[1].target)
```

Listing 1. Handling user interactions with flexibility, an example in Scala-based pseudo-code. Lines 1 to 5, a developer creates user interactions. The operators |, !, + allow developers to handle user interactions without writing boilerplate code. The operator | is interaction union and allows the execution of one of the two involved interactions (`i1` or `i2` on line 5). The binary operator ! allows to cancel a first interaction when the second one is executed (the execution of `KeyTyped("ESC")` cancels a running `MouseDnD`, on line 1). The operator + appends one interaction to a first one (`Tap` is appended to `LongTap` to form a new user interaction, on line 4). When processing a running user interaction (Lines 7 to 12), the developer handles the data the user interactions produce (the argument *data* on line 9) and not the interaction itself. User interactions that produce the same type of data are part of the same user interaction family (*e.g., MouseDnD*, *ReciprocalDnD*). This permits the developer to switch between user interactions part of a same family without changing the processing code. For example, switching from a classical DnD to another kind of DnD (reciprocal DnD) by commenting line 1 and uncommenting line 2 does not require code change as these two DnD interactions are part of the same family and expose the same data (`DnDData[TouchPointData]`).

Engineering user interfaces involves the use of multiple user interactions. Developers may struggle with programming and using those user interactions because of a lack of flexibility that affects the current user interface programming approaches. First, developers may want to switch from one user interaction to another close one or combine multiple user interactions without changing much code. Second, developers may also want to use several user interactions to concisely produce the same user command. Third, developers may want to be warned about conflicts between involved user interactions. Currently, developers can hardly perform these first two cases without applying numerous code changes or producing boilerplate code. Regarding the third case, developers can only observe such issues during the execution of the interactive systems, which prolongs the detection time. To overcome these three issues this paper proposes a user interaction type system. This user interaction type system reifies user interactions as first-class concerns with typing facilities for enabling user interactions substitutability and union. It also allows the writing of type checking rules to check

Author's address: Arnaud Blouin, firstname.lastname@irisa.fr, Univ Rennes, INSA Rennes, IRISA, Inria, France.

© 2024 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the ACM on Human-Computer Interaction*, https://doi.org/XXXXXXX.XXXXXXX.

for possible issues related to user interactions at compile time. We implemented the type system within the *TypeScript* version of *Interacto*, a framework for processing user interactions. We evaluated the soundness and the expressiveness of our approach through several implemented use cases. This demonstrates the feasibility of the proposed approach and its ability to overcome the three mentioned issues.

CCS Concepts: • **Human-centered computing → User interface programming**.

Additional Key Words and Phrases: typing system, user interaction, user interface engineering, object-oriented programming

ACM Reference Format:
Arnaud Blouin. 2024. A Type System for Flexible User Interactions Handling. *Proc. ACM Hum.-Comput. Interact.* 1, 1 (March 2024), 27 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

When programming user interfaces, developers need to implement and use multiple user interactions to enable end-user actions on the system. Developers also have to define the user commands those interactions will produce. Current software engineering practices for such development activities tend to imply numerous code changes notably because of constantly evolving requirements [13, 30]. Research work also highlighted the need for flexibility in customization of user interactions [33]. Moreover, the increasingly interactivity of systems, with various devices to support (*e.g.,* touch screen, pen-based devices), makes the development of user interactions more complex. To better work on user interactions, developers need: flexibility to substitute user interactions without changing much code and thus ease code changes; expressiveness to reduce the boilerplate code required to use or combine multiple user interactions; and type checking capabilities to identify potential programming issues related to user interactions. Approaches have been proposed to better code the behavior of interactive objects and to ease the handling of user interactions [2, 7, 24, 34]. All these approaches, however, have no or little support for the three above-mentioned concerns.

To address these concerns, this paper details a type system for user interactions. In programming languages, "*a type can be viewed as a property of an expression that describes how the expression can be used*" [15]. A type system is "*a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute*" [29]. So, typing user interactions allows expressing how developers can use them. A dedicated type system would prevent misuse of user interaction data types. Pierce [29] also argued that types lead to better abstractions. Our proposal leverages this last point by providing developers with programming abstractions operating with the proposed type system to overcome the mentioned issues. Indeed, considering a user interaction as a typed object permits one to leverage object-oriented concerns, such as substitutability and union type, to overcome the aforementioned flexibility and expressiveness concerns. In addition, reifying user interactions as objects allows one to write type checking rules that verify potential issues, such as conflicts between user interactions. We implemented this type system in *TypeScript* and extended an existing framework to demonstrate the feasibility of the proposal. We also implemented a proof-of-concept in Scala used in the illustrative examples as pseudo-code. Our evaluation also discusses the expressiveness and the soundness of the proposal. We completed this evaluation with representative use cases we implemented and used to illustrate the approach. This permits us to discuss the ability of the proposal to overcome the mentioned issues. To sum up, the contributions of the paper are as follows:

- a type system for user interactions that enables user interaction substitution and union to provide developers with more flexible features while handling user interactions;

- an open-source and freely available implementation that demonstrates the feasibility of the proposal, in particular through several implemented use cases discussed while presenting the approach;
- an evaluation that includes discussions on implemented use cases, the soundness of the proposal, and its expressiveness.

The rest of the paper is organized as follows: Section 2 motivates this work with detailed examples. Section 3 presents the proposed approach. Section 4 details the evaluation of the proposal. Section 5 discusses the related research work. Section 6 concludes and discusses research opportunities.

2 MOTIVATING EXAMPLES AND BACKGROUND

This section introduces illustrative examples that motivate the current limitations of programming and processing user interactions in terms of reuse, conciseness, and reliability.

2.1 Motivating Examples

```
1  const dragListener = (evt: MouseEvent) => {
2    svgShape.setAttribute('x', evt.clientX); // Moving the shape
3    svgShape.setAttribute('y', evt.clientY);
4  }
5  svgShape.addEventListener('mousedown', (e: MouseEvent) => {
6    (e.target as HTMLElement).style.cursor = 'pointer';
7    svgShape.addEventListener('mousemove', dragListener);
8  });
9  svgShape.addEventListener('mouseup', (e: MouseEvent) => {
10   (e.target as HTMLElement).style.cursor = 'default';
11   svgShape.removeEventListener('mousemove', dragListener);
12 });
```

Listing 2. Using a mouse-based DnD for moving SVG shapes.

The examples are part of an SVG drawing Web application. Figure 2 gives a glimpse of such an application, where users can edit SVG shapes using different mouse-based and touch-based user interactions. The examples are written in *TypeScript* and use the standard Web API. We chose *TypeScript* and the Web API for two reasons: this language and API are representative of the current development practices we discussed in this section; our implementation is also implemented in *TypeScript*, thus avoiding the use of multiple languages. The different examples involve the same start of scenario depicted by Listing 2: this listing contains the code a developer can write to use a drag-and-drop (DnD) interaction to move SVG shapes (*e.g.,* SVG rectangles) in an SVG canvas. This code classically assembles three event listeners (mouse down, mouse move, and mouse up) to move SVG shapes[1]. Note that classically, developers need to register the mouse move handler only after a mouse down to avoid the processing of unrelated mouse move events.

2.2 Example #1: Several user interactions for the same command

Modern interactive systems may need to support both mouse and touch screen devices. So, in this first scenario the developer needs to support touch screens and therefore needs to use a touch DnD, in addition to the mouse-based DnD, to move SVG shapes. This leads to a new version of the code

[1]Note that the Web API provides drag listeners to be used on objects the browser can drag natively (so, not applicable in our case).

(Listing 3). In this version, the three listeners are factorized by unifying (using *union type*) both touch and mouse event parameters: `(e: TouchEvent | MouseEvent)` means that *e* is either a *TouchEvent* or a *MouseEvent*. This permits the reuse of these three listeners during the registration.

```
1  const startListener = (e: TouchEvent | MouseEvent) => {
2    (e.target as HTMLElement).style.cursor = 'pointer';
3    if ('button' in e) { // if is a mouse event
4      e.target.addEventListener('mousemove', dragListener);
5    }
6  }
7  const dragListener = (e: TouchEvent | MouseEvent) => {
8    // Moving the shape
9    if('button' in e) {
10     svgShape.setAttribute('x', e.clientX);
11     svgShape.setAttribute('y', e.clientY);
12   }else {
13     svgShape.setAttribute('x', e.changedTouches[0].clientX);
14     svgShape.setAttribute('y', e.changedTouches[0].clientX);
15   }
16 }
17 const releaseListener = (e: TouchEvent | MouseEvent) => {
18   (e.target as HTMLElement).style.cursor = 'default';
19   if ('button' in e) { // if is a mouse event
20     e.target.removeEventListener('mousemove', dragListener);
21   }
22 }
23 svgShape.addEventListener('touchstart', startListener);
24 svgShape.addEventListener('mousedown', startListener);
25 svgShape.addEventListener('touchmove', dragListener);
26 svgShape.addEventListener('touchend', releaseListener);
27 svgShape.addEventListener('mouseup', releaseListener);
```

Listing 3. Using either a mouse DnD or a touch DnD for moving SVG shapes.

Changing from the initial mouse-based DnD to the dual mouse and touch-based DnD requires developers to apply numerous changes, making the code more complex (twice as long in our case). Developers may need facilities to ease the use of several user interactions to perform same user commands, without making the code more complex (*e.g.*, longer). For example, Listing 1 gives an example of such facilities with our user interaction typing operator | (not to confuse with the union type operator | *TypeScript* provides, such as on line 1 of Listing 3).

2.3 Example #2: Switching from one interaction to another similar one

In this second scenario, the developer wants to experiment the use of different user interactions of the DnD family: the mouse-based DnD, a drag-lock, a touch-based DnD. The developer first adapts the initial code of Listing 2 to use a drag-lock instead (Listing 4). A drag-lock is a kind of DnD where the user double-clicks on a first location, to then move to a second location and double-click again to end the interaction. This new code shows that such a switch requires numerous code changes, in particular for setting the behavior of the new user interaction (the drag-lock). In this

example, the listener that contains the user command execution (Lines 3 and 4) does not change since both the DnD and the drag-lock use mouse events.

```
1  const lockedListener = (evt: MouseEvent) => {
2    // Moving the shape
3    svgShape.setAttribute('x', evt.clientX);
4    svgShape.setAttribute('y', evt.clientY);
5  }
6  let locked = false;
7  svgShape.addEventListener('dblclick', (e: MouseEvent) => {
8    locked = !locked;
9    if(locked) {
10     (e.target as HTMLElement).style.cursor = 'pointer';
11     svgShape.addEventListener('mousemove', lockedListener);
12   }
13   else {
14     (e.target as HTMLElement).style.cursor = 'default';
15     svgShape.removeEventListener('mousemove', lockedListener);
16   }
17 });
```

Listing 4. Switching from a DnD to a drag-lock.

Now, the developer wants to switch to a touch DnD (for the possible reason that the system will actually run on a tablet). Listing 5 contains this new code, in which the developer had to change both the behavior of the interaction and the command. Regarding the command (Lines 6 and 7), the developer had to adapt the code while both interactions (drag-lock and touch DnD) are part of the same family, the DnD family. We define a family of user interactions as a set of interactions that expose the same kind of data (*e.g.,* a click and a double-click are part of the same family that exposes data detailing a position, see Section 3.1). The reason is that current development approaches rely on UI events (*e.g.,* touch, mouse events) instead of relying on user interactions grouped by family. So, changing from one interaction that uses one device to another one from the same family that uses a different device requires numerous code changes.

```
1  svgShape.addEventListener('touchstart', (e: TouchEvent) => {
2    (e.target as HTMLElement).style.cursor = 'pointer';
3  });
4  svgShape.addEventListener('touchmove', (e: TouchEvent) => {
5    // Moving the shape
6    svgShape.setAttribute('x', e.changedTouches[0].clientX);
7    svgShape.setAttribute('y', e.changedTouches[0].clientX);
8  });
9  svgShape.addEventListener('touchend', (e: TouchEvent) => {
10   (e.target as HTMLElement).style.cursor = 'default';
11 });
```

Listing 5. Switching from a drag-lock to a touch DnD.

Changing from the initial mouse-based DnD to a drag-lock or a touch-based DnD requires developers to apply numerous changes to adapt both the interaction behavior and the user command execution. Such changes are currently mandatory while these three user interactions are part of the same family of DnD interactions. Developers may need facilities to ease the switch between user interactions of the same family without applying such multiple changes. In the illustrative code of Listing 1, a developer can replace the `MouseDnD` interaction with a `DragLock` interaction with no additional change.

2.4 Example #3: Combining several user interactions

The developers now want to make it possible to abort a running mouse DnD by pressing the '*escape*' key. This requires changes in the original mouse DnD as illustrated by Listing 6. This code modifies the *mousedown* callback to listen for *keydown* events (Line 10). If the '*escape*' key is pressed, the interaction is cancelled (Line 3). So, if developers want to investigate some customization of user interactions, they may need to make code changes, making their investigations more complex.

```
1  //...
2  const cancelListener = (evt: KeyboardEvent) => {
3    if(event.keyCode === 27) { // escape key
4      // cancelling
5    }
6  }
7  svgShape.addEventListener('mousedown', (e: MouseEvent) => {
8    (e.target as HTMLElement).style.cursor = 'pointer';
9    svgShape.addEventListener('mousemove', dragListener);
10   svgShape.addEventListener('keydown', cancelListener);
11   //...
12 });
```

Listing 6. Making the mouse DnD cancellable when pressing the key 'escape'.

Combining user interactions requires developers to manually assemble the behavior of those interactions to form a new one. Developers may need flexible facilities to speed-up such combinations. Listing 1 gives an example of such facilities with the operators + and !.

2.5 Example #4: Conflicts between user interactions

Starting from the code of Listing 2, that uses the mouse-based DnD, the developer adds code to use a double click on SVG shapes to remove them. Listing 7 depicts this new code. Now, the developer switches from the DnD interaction to the drag-lock one. The developer runs the interactive system and spots that there is a conflict between the double-click and the drag-lock interactions since they start with the same sequence of events (the double-click): a double-click to start the drag-lock removes the concerned SVG shape.

```
1 svgShape.addEventListener('dblclick', (e: MouseEvent) => {
2   // Removing the shape
3   this.svgdoc.removeChild(e.target as SVGElement);
4 });
```

Listing 7. Double-clicking on an SVG shape to remove it.

Conflicts between user interactions can occur, causing some user commands to be executed unexpectedly: this may be the case for user interactions that run on the same interactive object and start with the same sequence of events. Current development approaches do not alert developers to such issues. Developers may expect features that check for such development issues, similar to what a code linter can do.

3 A USER INTERACTION TYPE SYSTEM

This section details the proposed type system that relies on the following principles:

- User interactions are reified as objects (in the sense of object-oriented programming), allowing developers to handle user interactions as any object of the code;
- User interactions can be grouped into families, *i.e.,* user interactions that have related behaviors and are used for related purposes;
- User interactions of the same family are fully or partly substituable depending if they involve the same input device;
- Reifying user interactions as objects permits static code analyzes to check for conflicts between user interactions

The examples of Section 2 illustrate that a user interaction is composed of two parts: its behavior, that consists in an assembly of UI events (*e.g.,* the mouse down, mouse move, and mouse up events for the DnD); their data. Interaction data are updated when the interaction is running (*e.g.,* positions of the pointing device, mouse button). So formally, given a user interaction i_x, then i_x is composed of i_{dx} and i_{bx}, formally denoted $i_x ::= (i_{dx}, i_{bx})$, where i_{dx} is its user interaction data and i_{bx} its behavior.

The proposed type system makes no assumptions about the formalism used to model interaction behavior (*e.g.,* Petri nets, finite-state machines).

3.1 Family of user interactions

Classically, each UI event provides data. For example, a mouse down event provides a *MouseEvent* object that contains data describing the mouse down (*e.g.,* X-Y positions, mouse button). UI events of the same family provide the same type of data. For example, mouse down, mouse move, *etc.* provide *MouseEvent* data. The same applies for key down, key up, *etc.* events that supply *KeyEvent* data. There exist UI events that correspond to the triggering of a widget (*e.g.,* button, checkbox). Such UI events are, in fact, independent of the user interaction used to trigger them: one can trigger a button by clicking on it or by pressing the key '*enter*' if the button has the focus. The data of such *InputEvent* events usually contain the widget that triggered the event. Interviews with expert developers identified this issue as "*Inconsistent/unintuitive API*", where, for example, "*a long keypress should result in an event with duration rather than a sequence of events*" [33].

Such a practice has a strong limitation. When using a DnD, as the one depicted by Listing 2, a developer handles data that each UI event provides: at least three different *MouseEvent* instances (one for mouse down, one for mouse up, and one for each mouse move). Those data instances are created independently of the goal of the user interaction so that developers have to process those event data to extract interaction data. For example the DnD is a kind of user interaction that starts at a given position with a pointing device (*e.g.,* mouse, touch, eye) to then move to a final position. Note that we consider here the DnD as a gesture independently of its usages (*e.g.,* dragging an object, swiping for changing the screen page). For example, the swipe interaction follows the DnD gesture with specific constraints: vertical or horizontal DnD, with a possible minimal distance or speed to reach. So, when handling a DnD, developers may expect to handle a single data object that represents this move, as depicted by Figure 1: the interface *DnDData* is, in our approach, the base

interaction data type for all the DnD interactions. It contains data related to the source and target positions (*src* and *tgt*), and data corresponding specifically to DnDs: distances, duration, velocity, etc. This *DnDData* interface forms the root type of the DnD family that groups the different forms of DnD interactions for each concerned pointing device: the classical DnD (*e.g.,* using a mouse, a touch screen); the drag-lock, the drag-and-pop [4], the drag-and-pick [4], the reciprocal DnD [3], the swipe. For mouse-based DnDs, the exact type of *src* and *tgt* is *MousePointData* that contains the mouse button used. For touch-based DnDs, the exact type of *src* and *tgt* is *TouchPointData* that contains touch properties (*e.g.,* touch ID, force). This leads to the following definition:

Definition 3.1 (Family of user interactions). From a typing viewpoint, a family of user interactions groups user interactions that share the same super interaction data type.

For the DnD family, this super type is `DnDData<PointData>`. Similarly, the family of click user interactions groups the click, double-click, tap, long touch, and long pressure user interactions (non-exhaustive list). This family groups user interactions that share the *PointData* super interaction data type (see Figure 1). Note that the double-click here corresponds to a double-click at the same location, and the tap interaction corresponds to a single (touch) tap. For mouse-based click interactions, the exact type is *MousePointData*. For touch-based click interactions, the exact type is *TouchPointData*.

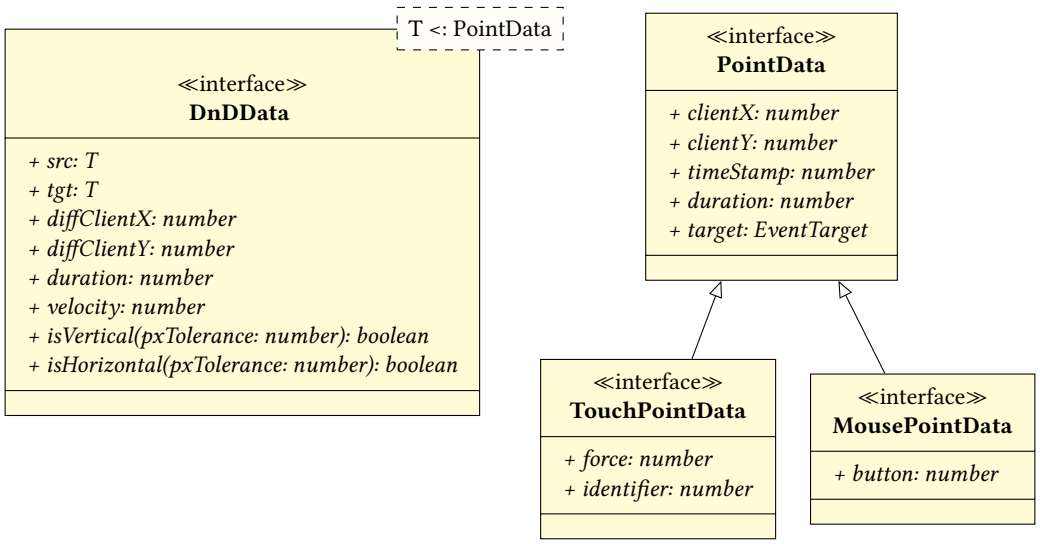

Fig. 1. Interaction data types for DnD interactions. All DnD interactions produce interaction data of type `DnDData<PointData>`. Touch-based DnD interactions can also be handled as `DnDData<TouchPointData>` and mouse-based DnD interactions as `DnDData<MousePointData>`.

3.2 User interaction substitution

Based on the concept of user interaction family, we define two kinds of user interaction substitution: total and partial substitution. Formally, i_1 and i_2 are two user interaction objects of type $\mathcal{I}_1$ and $\mathcal{I}_2$ defined as follows, where $\mathcal{I}_d$ corresponds to user interaction data type and $\mathcal{I}_b$ to user interaction

behavior type:

$$\Gamma \vdash i_1 : \mathcal{I}_1 \qquad \Gamma \vdash i_2 : \mathcal{I}_2$$
$$\Gamma \vdash i_{d1} : \mathcal{I}_{d1} \qquad \Gamma \vdash i_{d2} : \mathcal{I}_{d2}$$
$$\Gamma \vdash i_{b1} : \mathcal{I}_{b1} \qquad \Gamma \vdash i_{b2} : \mathcal{I}_{b2}$$
$$i_1 ::= (i_{d1},\ i_{b1}) \qquad i_2 ::= (i_{d2},\ i_{b2})$$

Definition 3.2 (Total user interaction substitution). An interaction i_2 can substitute an interaction i_1 (noted $i_1 := i_2$) if both interactions share the same exact user interaction data type. This leads to the following rule:

$$\frac{\mathcal{I}_{d1} \equiv \mathcal{I}_{d2}}{i_1 := i_2}$$

The following Listing 8 depicts an example of total user interaction substitution using our Scala-based pseudo-code. In this pseudo-code, a developer instantiates a DnD interaction (Line 1). The type of this interaction is `Interaction[DnDData[MousePointData]]`, which means that the data of this interaction is of type `DnDData[MousePointData]`. Then, a developer can define the processing of this user interaction (Lines 3 to 9). At the end of each execution, this interaction produces a command (here the command `MoveShape` that moves the targeted SVG shape) as defined by the method `toProduce` (Line 7). The DnD will operate on the widget *canvas* (method *on*, Line 5). The method *when* (Line 9) defines a predicate that must be fulfilled to produce the command. Here, the left button of the mouse must be used. In this method, *data* refers to the ongoing user interaction data: our approach has the requirement that developers handle user interaction data, not user interaction behavior when processing an interaction. In this example the developer replaces the standard DnD interaction with a reciprocal DnD [3] (Line 2). As *DnD* and *ReciprocalDnD* share the exact same user interaction data type (`DnDData[MousePointData]`), the rest of the code requires no change to compile and work.

```
1 val dnd: Interaction[DnDData[MousePointData]] = new DnD()
2 // val dnd: Interaction[DnDData[MousePointData]] = new ReciprocalDnD()
3 dnd
4   // The DnD will operate on each SVG element the canvas will contain:
5   .on(canvas)
6   // The command to produce:
7   .toProduce(data => new MoveShape(data.src, canvas))
8   // The DnD must be used using left mouse button
9   .when(data => data.tgt.button == MouseButton.LEFT)
```

Listing 8. Total substitution of mouse-based DnD interactions, in Scala-based pseudo-code. Switching between the standard DnD interaction (line 1, to comment in) and another kind of DnD (reciprocal DnD, line 2, to comment out), requires no change in the code that processes the interaction (from line 3).

Benefits of total user interaction substitution

(1) Exposing user interaction data that specifically concerns the interaction (*e.g.,* `diffClientX`, `duration`, `velocity` for `DnDData`) prevents developer from building such data by hand.
(2) After having selected a user interaction, developers handle its user interaction data for producing commands. Thus, switching from one user interaction to another one from the same family does not require any change thanks to user interaction data substitution.

Definition 3.3 (Partial user interaction substitution). An interaction i_2 can partly substitute an interaction i_1 (noted $i_1 :\approx i_2$) if both interactions do not share the exact same user interaction data type but share a same super user interaction data type (<: refers to subtyping):

$$\frac{\mathcal{I}_{d1} \not\equiv \mathcal{I}_{d2} \qquad \exists\, \mathcal{I}_{dx} \mid \mathcal{I}_{d1} <: \mathcal{I}_{dx} \ \wedge\ \mathcal{I}_{d2} <: \mathcal{I}_{dx}}{i_1 :\approx i_2}$$

Listing 9 illustrates the partial user interaction substitution based on Listing 8. Now, the developer wants to switch from the mouse-based DnD interactions to use a touch-based DnD interaction. So, the developer changes the user interaction in line 2. The single change the developer has to apply is line 6: this line must be commented out since `button` is a mouse property that is no longer relevant. The rest of the code uses DnD properties shared by both mouse and touch DnDs, and therefore requires no change.

```
1 // val dnd: Interaction[DnDData[MousePointData]] = new DnD()
2 val dnd: Interaction[DnDData[TouchPointData]] = new TouchDnD()
3 dnd.on(canvas)
4    .toProduce(data => new MoveShape(data.src, canvas))
5 // The following 'when' had to be removed since specific to mouse interactions
6   // .when(data => data.tgt.button == MouseButton.LEFT)
```

Listing 9. Partial substitution of DnD interactions, in Scala-based pseudo-code

Benefits of partial user interaction substitution

(1) Similarly to total user interaction substitution, interaction data expose properties that specifically concern the involved user interaction.
(2) Even with user interactions that involve different input devices, our approach permits to reduce code changes developers have to perform when switching from user interactions from a same family.

3.3 User interaction union

As illustrated in Section 2.2, a developer can use two different user interactions to produce the same output command. There are two possible cases:

(1) The developer wants to support several input devices for the same user interaction. For example the developer wants to use both a mouse-based and a touch-based DnD interaction to move the SVG shapes. We call this case *Device Union*.
(2) The developer wants different interactive objects to produce the same output command. For example, the developer wants to remove all the shapes using a swipe interaction or by clicking on a button. We call this case *Widget Union*.

These two cases consist of having an *xor* operator on the involved user interactions: either the user uses a mouse-based DnD, or the user uses a touch-based DnD. This leads to the same typing rule we call *user interaction union*, based on the union type concept: an object *ab* of type `A | B` means that at a given run-time instant, *ab* can be either an *A* object or a *B* object (a logical '*or*'). We thus define user interaction union as follows:

Definition 3.4 (User interaction union). Given, a user interaction i_3, that at run-time has either the behavior of interaction i_1 or the behavior of interaction i_2 (so an *xor*, noted $\oplus$). Then, the user interaction data type of i_3, $\mathcal{I}_{d3}$, is the union type of the user interaction data type of i_1 and i_2, so $\mathcal{I}_{d1} \mid \mathcal{I}_{d2}$.

$$\frac{\Gamma \vdash i_3 : \mathcal{I}_3 \qquad i_3 ::= (i_{d3},\ i_{b1} \oplus i_{b2})}{\Gamma \vdash \mathcal{I}_{d3} : \mathcal{I}_{d1} \mid \mathcal{I}_{d2}}$$

```
1 val dnd: Interaction[DnDData[MousePointData] | DnDData[TouchPointData]]
2          = new TouchDnD() | new MouseDnD()
3 dnd.on(canvas)
4  .toProduce(data => new MoveShape(data.src, canvas))
5  .when((data: DnDData[MousePointData] | DnDData[TouchPointData]) => data match
6   {case m: MousePointData => m.button==MouseButton.LEFT; case default => true})
```

Listing 10. Device Union Example, in Scala-based pseudo-code

Listing 10 illustrates the user interaction union, and the specific case of device union, using our Scala-based pseudo-code. This example still uses a DnD to move SVG shapes. Compared to the previous versions of the example, this code now involves the mouse-based and the touch-based DnD interactions (Line 2). So, now the type of the user interaction data is the union type `DnDData[MousePointData] | DnDData[TouchPointData]`. Most of the code remains untouched as it is based on properties that both data types share (*e.g.,* `tgt`). The only places that need changes are lines 5 and 6: the `when` function only concerns the mouse-based DnD so that the developer checks that the ongoing data type is produced by the mouse-based DnD to then check the involved mouse button.

```
1 val i = new Swipe() | new ButtonPressed()
2 i.on(delete, canvas)
3  .toProduce((data: MultiTouchData | WidgetData) => new DeleteAll())
```

Listing 11. Widget Union Example, in Scala-based pseudo-code

Listing 11 illustrates the specific case of widget union, still using our Scala-based pseudo-code. In this example the developer wants to use a button or a swipe on the canvas to remove all the shapes of the canvas. To do this, the developers select both the swipe and the button interactions (Line 1). These two interactions run on the `delete` button and the `canvas` (Line 2). Now, the type of *data* in the interaction processing code is the union type `MultiTouchData | WidgetData`, as illustrated on line 3.

Benefits of user interaction union

User interaction union allows developers to factorize code for producing a command using different user interactions. This concerns the two kinds of user interaction union we have identified, namely: device union and widget union. In both cases, developers handle a union of the user interaction data.

3.4 User interaction assembly

Developers may want to assemble user interactions to form a new one. We focus on two specific cases of assembly:

(1) *Appending*, that consists in adding a second user interaction at the end of a first one. For example with Listing 12, the operator `+` appends the interaction `Tap` to the interaction `LongTap` to create a new user interaction `i2` (Line 2).
(2) *Cancelling*, that consists in joining a secondary user interaction i_2 in parallel of the first one, and if i_2 ends while the first one is running then the user interaction stops and cancels

its effects. For example with Listing 12, the operator ! creates a new user interaction `i1` composed of the main interaction `MouseDnD`, which execution is cancelled if the interaction `KeyTyped` with the key '*escape*' is performed (Line 1).

```
1 val i1: Interaction[DnDData[MousePointData]] = new MouseDnD() ! new KeyTyped("ESC")
2 val i2: Interaction[TouchPointData,TouchPointData] = new LongTap(2000) + new Tap()
3 i1. ...
4   .when((data: DnDData[MousePointData] | DnDData[TouchPointData]) =>
5     usesLeftButton(data.tgt))
6 i2...
7  .toProduce(data:[TouchPointData,TouchPointData]=> new DeleteShape(data[1].target))
8 }
```

Listing 12. User interaction assembly

Definition 3.5 (User interaction appending). Given, a user interaction i_3, which behavior consists of appending (noted ::) the behavior of the two user interactions i_1 and i_2. Then, the user interaction data type of i_3, $\mathcal{I}_{d3}$, is a new data structure composed of the user interaction data of i_1 and i_2.

$$\frac{\Gamma \vdash i_3 : \mathcal{I}_3 \qquad i_3 ::= (i_{d3}, i_{b1} :: i_{b2})}{\Gamma \vdash \mathcal{I}_{d3} : \{\mathcal{I}_{d1}\ ,\ \mathcal{I}_{d2}\}}$$

Definition 3.6 (User interaction cancelling). Given, a user interaction i_3, which behavior consists of the behavior of a user interaction i_1, and for which the execution of a secondary user interaction i_2 stops and cancels the execution of i_3 (noted !). Then, the user interaction data type of i_3, $\mathcal{I}_{d3}$, is the interaction data of i_1: i_2 only impacts the behavior of i_3.

$$\frac{\Gamma \vdash i_3 : \mathcal{I}_3 \qquad i_3 ::= (i_{d3}, i_{b1} \,!\, i_{b2})}{\Gamma \vdash \mathcal{I}_{d3} : \mathcal{I}_{d1}}$$

Note that this typing feature does not aim at replacing approaches that create user interactions by, for example, assembling UI events or treating flows of UI events [2, 7, 19, 24, 27]. Instead, this feature aims at providing developers with high-level operators to ease development tasks where developers want to customize the behavior of user interactions.

3.5 Type checking rules

The goal of type checking rules is to, based on types, forbid certain expressions in the code that may lead to erroneous behaviors. The user interaction type system described in the previous section already defines implicit type checking rules. For example, when a developer selects a DnD interaction, the interaction data type this developer will then handle is of type `DnDData<MousePointData>`. If the developer handles such DnD data using an incompatible type, the type system of the host programming language will raise an error stating that the two types are incompatible.

In this section, we detail three additional rules, that developers may activate or not. These rules aim at finding issues related to the category *Action* defined in [21]: issues related to the execution of the wrong command(s). We focus specifically on conflicts between user interactions that operate on the same widget to produce different commands (actions). These rules are similar to linter rules, such as *Eslint*[2] or the *TypeScript* compiler[3], that aim at helping developers in finding issues in their code. The three rules rely on the type system detailed in this section, and on code analyses to get

[2] https://eslint.org
[3] https://www.typescriptlang.org/tsconfig

additional information. Developers should be able to configure each rule following the classical scheme: ignore (the rule is disabled), warning (it raises a warning message), error (it raises an error). Indeed, depending on the use case, developers may want to disable specific rules if they are not relevant to their current development project. Rules can be checked either at: compile time, for example as a linter that developers execute when building the system; run time, for example on the interactive system start-up (in this case raised errors prevent the system to start).

These three rules are representative examples of type checking rules that a linter dedicated to user interactions and their processing can check based on our typing system. Given w_1 and w_2, two interactive objects (widgets) on which, respectively, the interactions i_1 (of type $\mathcal{I}_1$) and i_2 (of type $\mathcal{I}_2$) operate. The statement $w_1 \cap w_2 = \emptyset$ is a shortcut for stating that there is no kinship between w_1 and w_2, and corresponds to this exact statement:

$$\begin{aligned} w_1 \neq w_2 \wedge (w_2 \in \mathit{children}(w_1) &\implies \mathit{consume}(w_2, i_2)) \\ \wedge (w_1 \in \mathit{children}(w_2) &\implies \mathit{consume}(w_1, i_1)) \end{aligned}$$

This statement means that: w_1 and w_2 are not the same object; w_1 (resp. w_2) is not a child in the scene graph of w_2 (resp. w_1) or consumes its interaction i_2 (resp. i_1). Indeed, thanks to event bubbling a parent also catches UI events its children produce. So, a user interaction that operates on w_2 also concerns w_1 if w_1 is a parent of w_2. A developer can stop the bubbling of events by consuming an event/interaction being processed. With the Web API, this is the goal of the method `stopPropagation`. So, if a user interaction is consumed, there is no more kinship between a widget and its parent widgets.

3.5.1 Rule same-interactions*.* The first rule *same-interactions*, defined below, states that if two user interactions of the same type are used, it must not exist any kinship between their involved widgets. Indeed, in many cases if two same interactions are used on related widgets this is a programming issue: for example, interacting on w_1 will also execute the interaction on w_2, then producing two different commands. Listing 13 shows code that triggers this rule. The two widgets involved, `canvas` and `shape`, are related as `canvas` contains `shape` (not illustrated in the code). The same type of user interactions (click) operates on both widgets. So, when a user clicks on `shape`, this also triggers the click on `canvas` so that both commands are produced on lines 3 and 7.

$$\frac{\mathcal{I}_1 \equiv \mathcal{I}_2}{w_1 \cap w_2 = \emptyset} \qquad \text{(same-interactions)}$$

```
1 new Click()
2   .on(canvas)
3   .toProduce(...)...
4
5 new Click()
6   .on(shape)
7   .toProduce(...)...
8   // .consume()...
```

Listing 13. Code example that triggers the rule *same-interaction*, in Scala-based pseudo-code.

There are several cases where developers may want to disable this rule. In some specific cases, a developer may want to use the same user interactions to produce different commands. For example, typing a sequence of characters may produce one command that changes the text and another command that logs it.

3.5.2 Rule same-data. The second rule, called *same-data*, is more restrictive than the first one: given two user interactions, if the type of their user interaction data are the same the rule is triggered.

$$\frac{\mathcal{I}_{d1} \equiv \mathcal{I}_{d2}}{w_1 \cap w_2 = \emptyset} \quad \text{(same-data)}$$

```
1 new Tap()
2   .on(canvas)
3   .toProduce(...)...
4
5 new LongTouch(2000)
6   .on(canvas)
7   .toProduce(...)...
```

The Scala-based pseudo-code above illustrates this rule. On the same widget `canvas`, the developer registers two user interactions that expose the exact same user interaction data type: `tap` and `longTouch` that are both touch interactions that consists in pressing on one position. Their user interaction data type is `TouchPointData`. They belong to the same family of user interactions and involve the same input device. So, when a user performs a long tap, this produces both commands of lines 3 and 7.

Note that the efficiency of this rule depends on how the families of user interactions are designed. If a family is too generic and encompasses a large set of user interactions, this rule will not be efficient. For example, interaction data types of interactions that use different devices should differ: in Figure 1 the mouse-based and touch-based DnD interactions have different data type. We discuss in this point in detail in the evaluation section (Section 4).

3.5.3 Rule included. A large variety of user interactions contain in fact simpler user interactions. For example, a *double-click* includes the *click* interaction. Written differently, when a user performs a *double-click*, this user performs (two) *click* interactions. Such an inclusion is nested: the *drag-lock* interaction includes the *double-click* that includes the *click* that includes the *press* interaction.

We define two kinds of user interaction inclusion: *full user interaction inclusion*; *partial user interaction inclusion*.

Definition 3.7 (Full user interaction inclusion). Full user interaction inclusion occurs when the behavior i_{b2} of a given user interaction i_2 is fully included in the behavior i_{b1} of another user interaction i_1. The above-mentioned example of the double-click and the click typifies full user interaction inclusion: in all the cases, doing a double-click also implies doing a click (in fact two).

Definition 3.8 (Partial user interaction inclusion). A user interaction may be partly included into another one. Given i_1 and i_2, a partial inclusion occurs when there exists one full execution path (*i.e.,* the interaction has completed) of i_{b1} that is also part of i_{b2}. For example, the long pressure is partly included into the mouse-based DnD: if a user performs a long pressure (i_1) to then drag and release the mouse button to perform a DnD (i_2), this DnD execution includes the long pressure. However, performing the same DnD but without holding the pressure does not execute the long pressure. So, not all the executions of the DnD include the execution of the long pressure.

3.5.4 Properties. We now discuss three specific properties that affect the three rules.

Bubbling. In mainstream approaches, developers can prevent event bubbling by consuming the event. For example using the Web API, this can be done using the method `event.stopPropagation()`. In JavaFX this can be done using `event.consume()`. The Scala-based pseudo-code of Listing 13

(consume on line 8) illustrates this principle. If uncommented, this line will prevent canvas to catch and process the click made on shape. In this case the linter will raise no issue.

Interaction Parameters. Interactions can have parameters, such as the long touch that has a duration as parameter. To be sound, the *same-interactions* implies that interaction parameters must not correspond to different types of user interactions. Following this principle, a swipe interaction, for example, should not have a parameter *vertical* that defines whether the swipe must be vertical or horizontal. These two interactions should be separated into two different types. Indeed, a vertical swipe cannot be in conflict with an horizontal one. These two interactions are frequently used together, for example, to move to the next page (horizontal swipe) or to refresh or scroll the current page (vertical swipe).

Conditions. Developers can constrain a user interaction execution with conditions. For example, a developer can use two tap interactions but with two different and non-conflicting levels of pressure. The method when refers to such conditions in our Scala-based pseudo-code (see Line 5 in Listing 10). The type checking rules we propose ignore user interactions that use the when method since such conditions can hardly be analyzed.

Benefits of type checking rules

The three type checking rules leverage the user interaction type system we propose to analyse code and find potential user interaction conflicts that may lead to the incorrect execution of user commands. The efficiency of these rules depends on the user interaction type system implementation, and in particular on the design of the user interaction families.

4 EVALUATION

Section 3 illustrates the proposal with a dedicated use case: a Web interactive system for drawing SVG shapes. This use case illustrated in Scala-based pseudo-code our proposal. In this section, we go further in the evaluation of the proposal by discussing the three following research questions:

- RQ1 – To what extent can our proposal be implemented in a mainstream programming language?
- RQ2 – To what extent is the proposed type system sound?
- RQ3 – What is the expressiveness of the proposal?

4.1 Implementation

4.1.1 Implementation in Interacto. The listings of Section 3 show Scala-based pseudo-code we developed for the specific purpose of illustrating our proposal. We implemented our proposed type system within the research approach *Interacto* that aims at processing user interactions [7]. We have extended the *TypeScript* implementation of *Interacto*, version 7.4.0[4]. Before detailing the implementation, we introduce major concepts required to understand how *Interacto* works. To do so, we use Listing 14 that gives the *Interacto* code for performing a total user interaction substitution, as originally illustrated with the Scala-based pseudo-code of Listing 8. Figure 2 is a screenshot of an Angular application implementing the illustrative examples of this paper (a drawing application) we developed using our *Interacto* implementation.

[4]https://github.com/interacto/interacto-ts

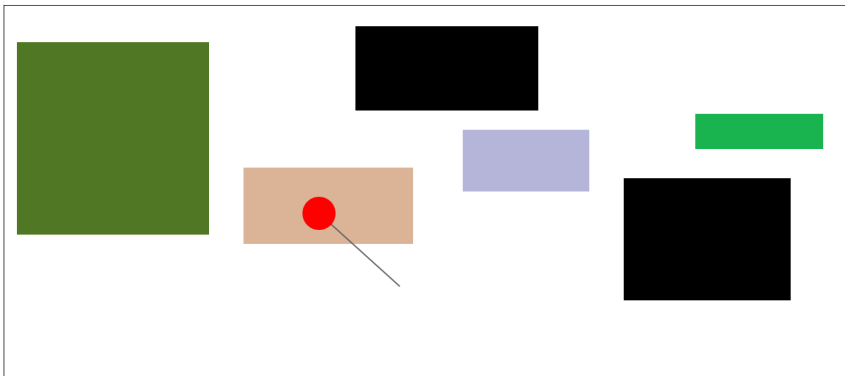

Fig. 2. Screenshot of an Angular application developed using the extended version of *Interacto*. A user can, for example, move shapes using a reciprocal DnD (either touch-based or mouse-based). In this picture, a user is dragging the beige rectangle and makes a stop. The red circle then appears and corresponds to the zone where the user can release the DnD to cancel the move.

```
binders
  .dndBinder()
  // Changing the standard DnD interaction, by commenting out the previous
  // line, to then uncommenting the following line that selects a reciprocal
  // DnD does the change the following code that processes the interaction
  // .reciprocalDndBinder(this.dwellSpring.handle, this.dwellSpring.spring)
  // .touchDnDBinder()
  // The Dnd will operate on each SVG element the canvas will contain:
  .onDynamic(this.canvas)
  // The command to produce:
  .toProduce(i => new MoveShape(i.src.target, this.canvas))
  .first((_, i) => { // Called on each DnD start
    if(i.src.target instanceof HTMLElement)
      i.src.target.style.cursor = 'pointer';
   })
  .then((c, i) => {  // Called on each DnD update
    c.vectorX = i.diffClientX;
    c.vectorY = i.diffClientY;
  })
  .endOrCancel(i => {  // Called on each DnD end
    if(i.src.target instanceof HTMLElement)
      i.src.target.style.cursor = 'default';
  })
  // The DnD must be used using mouse button '0'
  .when(i => i.tgt.button === 0)
  .continuousExecution()
  .bind();
```

Listing 14. Total substitution of mouse-based DnD interactions in *Interacto*

Using *Interacto*, a developer selects a predefined user interaction (*e.g.,* `dndBinder` line 2) and then configures how the executions of this user interaction will produce and execute output user commands. Such a configuration is called a *binder*. In this binder, the developer has selected the command `MoveShape` that the lambda function of `toProduce` (Line 11) will produce. This lambda function will be executed every time the DnD starts in order to create the command to be executed. The DnD will operate on all the objects contained in the SVG canvas (Line 9). The lambda function defined in `first` will be executed each time the DnD starts after `toProduce` (Lines 12 to 15). Similarly, `end` is executed each time the DnD stops (Lines 20 to 23), and `then` on each update of the DnD during its execution (Lines 16 to 19). The command is produced only if the predicate of the `when` (Line 25) returns `true` (if the mouse button used for the DnD is the left button).

In *Interacto*, a developer selects the user interaction once, to then handle its user interaction data in the different methods of the binder (`first`, `then`, *etc.*). Such user interaction data correspond to the parameter *i* that each lambda function of the binder provides. The type of *i* is inferred from the selected user interaction. Here a DnD provides a `DnDData` object. During one execution of the user interaction, the *i* object remains the same and is automatically updated. It is cleared when the user interaction ends. The other parameter *c* refers to the ongoing user command. This code also illustrates that using *Interacto*, a developer selects a user interaction and then never handles it directly but handles its data (the *i* parameter).

In this example, switching from a mouse DnD (Line 2) to a reciprocal DnD (Line 6) does not require any code change as these two interactions are part of the same family. Switching from a mouse DnD to a touch DnD (Line 7) requires to comment out the `when` method (Line 25), as in the Scala-based pseudo-code.

Listing 15 is the *Interacto* code that corresponds to the Scala-based pseudo-code of Listing 10. The method `xorBinder` (Line 2) initializes a binder with the touch or mouse DnD interactions (user interaction union). The type of the underlying user interaction data is the union of `DnDData<MousePointData>` and `DnDData<TouchPointData>`, as depicted on Line 4.

```
1 binders
2   .xorBinder(new TouchDnD(), new DnD())
3   // ...
4   .when((i: DnDData<MousePointData | TouchPointData>) =>
5     isTouchPointData(i.tgt) || i.tgt.button === 0)
6   .bind();
```

Listing 15. Device Union Example in *Interacto*

```
1 binders
2   .combine([new LongTap(2000), new Tap()])
3   .when((i: ThenData<Array<TouchPointData | TouchPointData>>) => ...)
4   // ...
5 binders
6   .cancel(new MouseDnD, new KeyTyped(27))
7   .when((i: DnDData<MousePointData>) => ...)
8   // ...
```

Listing 16. Appending and canceling operators in *Interacto*

Listing 16 shows *Interacto* code combining user interactions. Line 2 shows the appending operator where a `Tap` is appended to a `LongTap`, creating a novel user interaction. As explained in Section 3.4,

the type of the data of this new interaction is a new data structure composed of the data of `Tap` and `LongTap`. In *Interacto*, this new structure corresponds to `ThenData`, which is in this case an instance of `ThenData<Array<TouchPointData | TouchPointData>` (used for example on Line 3).

Line 6 shows the canceling operator where the `KeyTyped` cancels the `MouseDnD`. The data of this new user interaction is the data of `MouseDnD`, as illustrated on Line 7.

The implementation of the proposed user interaction type system within *Interacto* involved several changes[5]. First, we modified the *Interacto* user interaction system to add user interaction families and novel user interactions. This allows support for user interaction substitution (*Interacto* has a limited support of substitution that concerns user interactions using the same input device). To implement user interaction union, we added a new user interaction that consists of an *xor* between two given user interactions. We modified the *Interacto* binder API to expose this new feature. For example, we added the method `xorBinder` used in Listing 15.

To support the three type checking rules, we developed dedicated code analyses that currently operate on system startup (*i.e.,* dynamic code analyses). A developer can configure the rules directly through the *Interacto* instance module, such as in the following code example. Such a configuration affects all the *Interacto* binders of the system.

```
1 binders
2   .configureRules('same-interactions', 'err')
3   .configureRules('same-data', 'warn');
```

Developers can also configure the rules per *Interacto* binder. For example, the following code configures the two rules for the binder under creation only.

```
1 binders
2   .dndBinder()
3   .configureRules('same-interactions', 'err')
4   .configureRules('same-data', 'warn')
5   ...
```

Regarding the code analyses that implement the rules *same-interactions* and *same-data*, we have implemented them using the *Interacto* feature that registers all the created binders (in the main *Interacto* object `binders`). Each binder explicitly identifies: the widgets on which the user interaction will operate, thanks to the routine on; the selected user interaction using the different dedicated routines (*e.g.,* `clickBinder`, `tapBinder`).

```
1 function checkBinder(binder, binders, ruleName) {
2   if(binder.linterRules.get(ruleName) !== "ignore" && !binder.isWhenDefined()){
3     const conflicts = binders.filter(b =>
4       b.linterRules.get(ruleName) !== "ignore" && !b.isWhenDefined() &&
5       hasKinship(binder, b) && checkRule(binder, b, ruleName));
6     if(conflicts.length > 0) {
7       raiseIssue(binder, conflicts[0]);
8     }
9 }
```

Listing 17. Algorithm of the static code analyses, in pseudo-code

[5]All the materials are freely available on this code repository: https://github.com/interacto/research/EICS2024. The web page also lists all the user interactions and interaction data types.

The code of Listing 17 details the main steps of the algorithm of the code analyses. Given all the binders (of the system or of a component), we first exclude the binders with a `when` routine or that ignore the ongoing rule. We then select the other binders that have a kinship with the given one (`binder`) and that are in conflict with it (based on the current rule to check). If binders are selected, this means that an issue must be raised.

Regarding the rule *included*, checking full user interaction inclusion with *Interacto* is simple in our approach. When a user interaction contains another one in *Interacto*, the former reuses the class of the latter. For example, the double-click interaction explicitly uses two click interaction instances. So, testing whether the double-click contains the click interaction is direct.

Implementing partial user interaction inclusion is more complex and depends on how user interaction behaviors are represented (FSMs, Petri nets, *etc.*). We have hard-coded inclusion checking into our checker since the number of user interactions is limited and their behavior stable.

4.1.2 Families of user interactions. Table 1 gives representative examples of families of user interactions we designed based on the proposed type system.

Table 1. Representative examples of user interaction families. The user interaction data types are given between parentheses. The lists of user interactions are not exhaustive. Native widget interactions are ignored (*e.g.,* input, scroll events). Swipe and pan interactions include all the similar interactions (*e.g.,* swipe left, vertical pan).

Families	**User interactions**
One-key	KeyTyped, KeyDown (KeyData)
Multi-keys	KeysTyped (KeysData)
One-tap	Click, TimedClick, MouseDown, LongPress, DoubleClick (PointData<MousePointData>); Tap, TimedTap, TouchDown, LongTouch, DoubleTap (PointData<TouchPointData>)
Multi-taps	Taps (PointsData<TouchPointData>); Clicks (PointsData<MousePointData>)
DnD	MouseDnD, Draglock, DnPick, DnPop, ReciprocalDnD (DnDData<MousePointData>); TouchDnD, Pan, Swipe (DnDData<TouchPointPoint>)
2-touch	2-pan, 2-swipe, 2-touch (TwoTouchData); Rotate (RotateData); Scale (ScaleData)
3-touch	3-pan, 3-swipe, 3-touch (ThreeTouchData)
4-touch	4-pan, 4-swipe, 4-touch (FourTouchData)
Multi-touch	Multi-touch (MultiTouchData)

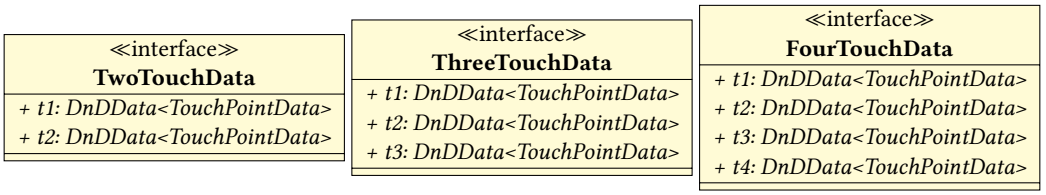

Fig. 3. Excerpts of the 2-touch, 3-touch and 4-touch data types, that support partial user interaction substitution.

The *one-key* family groups keyboard interactions that involve a single keyboard symbol. In complement, the *multi-keys* family involves a sequence of keyboard symbols. The *one-tap* family

groups user interactions that involve a single point, either using a touch device or a mouse. The *TimedClick* interaction is a click with a timeout. This permits the use of *TimedClick* and *longPress* together if the two timers do not conflict. The *multi-taps* family involves several successive points produced using the same touch or mouse button. The number of points is a parameter of these user interactions. The DnD family groups user interactions that involve a start and an end positions, using either a touch device or a mouse. We have designed three touch families that strictly involve two, three, or four touches. Such families are important to, at compile time, being able to check conflicts between touch interactions. The multi-touch family has the same named user interaction, where the number of touches is a parameter. Note that partial user interaction substitution operates on 2-touch, 3-touch and 4-touch interactions since we implemented their interaction data type in a similar way: as illustrated by Figure 3, a developer can replace a 2-touch interaction with a 3-touch one without changing the code (both interfaces share the same attributes `t1` and `t2`). Replacing a 3-touch with a 2-touch interaction, however, may require code changes to adapt the use of the third touch.

RQ1 – Implementation

We fully implemented our approach in *TypeScript* and the Web API. We freely provide the code of our extended version of *Interacto* and use cases. We have provided and discussed representative examples of user interaction families, showing the ability of our proposal to support mainstream user interactions.

4.2 Soundness

4.2.1 Soundness of the type system. A type system is sound if it prevents run-time errors related to types. Our proposal relies on the programming language developers use. In the case of our implementation, this language is *TypeScript*, that has a structural, static, and strong typing system[6]. So, there are two facets of our type system soundness: the host programming language one; the intrinsic soundness of our proposal. Because it relies on *JavaScript*, *TypeScript* has unsound features[7]. For example, the following *TypeScript* code uses an unsound feature: the use of the operator `as` incorrectly casts an object of type `number` into a variable of type `string`.

```
1 const value: Object = 1;
2 const str: string = value as string;
```

To limit the possible use of the cast operator in our type system, we leverage the *Interacto* binder API to add new features. For *device union*, we added new routines in the *Interacto* API, such as the `touchOrMouseDnDBinder` routine that hides the explicit creation of the user interactions. This routine hides the call to `xorBinder(new TouchDnD(), new DnD())` so that developers cannot cast these user interactions to incorrect types. We implemented the following other routines: `ClickOrTapBinder`, `ClicksOrTapsBinder`, `LongTouchOrPressBinder`.

Despite such typing features, developers can still perform some, yet limited, unsound operations when handling user interactions. A developer can cast a user interaction data object, for example `i as unknown as string`. However, developers can forbid the use of the operator `as` by activating linter rules[8].

Regarding the type of user interaction data objects used in *Interacto* binders, the typing rules are sound since automatically inferred at compile time from the selected user interactions. For example,

[6]We assume that the type 'any' is disabled, as it is the nominal case when using *TypeScript*, to prevent weak and dynamic typing.
[7]https://www.typescriptlang.org/docs/handbook/type-compatibility.html
[8]https://typescript-eslint.io/rules/consistent-type-assertions

in the following code chunk the type of i (`DnDData<MousePointData | TouchPointData>`) is inferred from the types of the selected user interactions (`touchOrMouseDnDBinder`).

```
1 binders
2   .touchOrMouseDnDBinder()
3   .when((i: DnDData<MousePointData | TouchPointData>) => ...)
4   ...
```

Table 2. Representative examples of rules decisions. ✓ means the rule passes while ✗ means the rule fails (raises an issue).

i_1	i_2	*same-interactions*	*same-data*	*included*
mouse DnD	touch DnD	✓	✓	✓
zoom	rotate	✓	✓	✓
mouse or touch DnD	touch DnD	✓	✓	✗
click	clicks	✓	✓	✗
double click	clicks	✓	✓	✗
long pressure	mouse DnD	✓	✓	✗
double-click	drag-lock	✓	✓	✗
swipe left	swipe right	✓	✗	✓
vertical pan	horizontal pan	✓	✗	✓
swipe	touch DnD	✓	✗	✓
long touch (500ms)	timed tap (400ms)	✓	✗	✓
long touch (500ms)	timed tap (500ms)	✓	✗	✗
long touch	tap	✓	✗	✗
vertical pan	vertical swipe	✓	✗	✗
click	double click	✓	✗	✗
swipe right	horizontal swipe	✓	✗	✗
click	double click	✓	✗	✗
tap	tap	✗	✗	✗

4.2.2 Soundness of the type checking rules. The soundness of the three type checking rules is defined by their ability to detect issues and avoid false positives. False positives are issues raised while they do not correspond to real issues. As numerous type checking rules linters provide, this is subjective (developers may have different opinions on rules) and may depend on the interactive system under development. For this reason, developers can disable the proposed rules. However, we discuss in Table 2 representative examples of results of the three proposed rules. These examples assume that the listed user interactions operate on the same widget (so they have a kinship). An example of issues subject to discussion is *long touch* with *tap* used, for example, with Android buttons to either enable/disable a feature or open its parameters. Both user interactions provide the same data (`TouchPointData`, see Figure 1), so this raises an issue if the rule *same-data* is enabled. Moreover, if using a tap interaction that does not have a timeout in the case of a too-long touch, there is a conflict between these two user interactions: doing a long touch will, at the touch release, also disable/enable the feature. In this case, the tap is included in the long touch. The use of both user interactions requires the design of a tap interaction with a timeout. This is the goal of the user interactions *TimedTap* and *TimedClick* we have developed: using a timed tap with a timeout of 400 ms is not in conflict with a long touch with a duration of 500 ms. However, the '*included*' rule cannot check at compile time such dynamic parameters, as discussed in the scope section

(Section 4.3.1). Our implementation operates at run time so that our checker can perform such a dynamic analysis.

Table 2 also shows examples of swipe interactions used together. When using a swipe left and a swipe right (touch-based), the user interaction data types are the same, but the other two rules pass. Similarly, the vertical pan and the horizontal pan, used in media applications to scroll content or switch to a new content, share the same data but are not included in each other. This shows that the *same-data* rule is the strictest of the three. When using a *horizontal swipe* and a *left swipe*, the former includes the later so that the rule *included* raises an issue. The same reasoning applies to clicks and click, and double click and click. However, using a swipe and a touch DnD raises a *same-data* issue. They are part of the same family and a touch DnD can be interpreted as a swipe.

An interesting case concerns touch interactions that start with the same sequence of events, such as the scale and rotate touch interactions. From a typing viewpoint, these two user interactions differ, do not expose the same user interaction data type, and are not included in each other. So using both user interactions does not lead to a typing issue. However, Kin et al. motivated with this example the problem of conflicts between touch interactions that start similarly [20]. Using our approach, the implementations of the provided user interactions must be consistent with their typing: if our typing system declares no conflict between the scale and zoom user interactions, their implementations must behave accordingly (which is the case using *Interacto*).

RQ2 – Soundness

The soundness of our proposal depends on the soundness of the underlying programming language type system. Our implementation limits unsound usages of *TypeScript* thanks to the *Interacto* API we have extended.
As with any linter, the relevance of the proposed type checking rules may depend on the system under development. We have introduced various representative examples to discuss the sound behavior of the proposed rules.

4.3 Expressiveness

We discuss the expressiveness of the proposal through the lens of its scope and its generalization to other programming languages.

4.3.1 Scope. First, the proposal has been developed using Web technologies. It is therefore not currently intended to be used beyond the scope of traditional desktop, Web, or mobile interactive systems and their input devices. Second, the proposed type system fully works with user interactions that correspond to a specific semantic, *i.e.,* discrete user interactions. User interactions that interpret events to infer their meaning dynamically (non-discrete user interactions or user-defined gestures, such as in [27, 28, 37]) are beyond the scope of the proposal. For example, a generic multi-touch interaction that at run-time dynamically infers whether it is a swipe, a pan, or a zoom interaction will not fully work with our type system. Another example is the use, on the same widget, of a click with the shift mode, another click with the control mode, and a last click with no mode, to perform three different commands. If these three clicks are instances of the same click interaction, then the type system will rise an issue (same interaction). To prevent this situation, three different interactions (*click*, *ctrl-click*, *shift-click*) must be defined. The reasons are the same as the difference between dynamic typing and static typing: our type system is static and cannot operate on 'dynamically typed' user interactions. Note that in *Interacto*, a user interaction starts as soon it receives an acceptable event but stops and cancels if it receives non-acceptable events. For example, a touch-DnD interaction will stop if the user starts using a second touch.

4.3.2 Generalization. Our current implementation supports *TypeScript*. We also implemented a proof of concept in Scala 3, we used as an illustrative purpose. Our proposed type system relies on the host language type system. So, porting our approach to other programming languages requires those languages to have the following features: union type, object substitutability, strong typing. Our approach can work with both nominal typing (Scala) and structural typing (*TypeScript*). For example, *Interacto* has an implementation in *Java* with the *JavaFX* toolkit[9]. Porting our proposal to *Java* and *JavaFX* would face the challenge that *Java 21* (the latest version) does not support *union type*.

Regarding the generalization of the proposal, our Scala proof of concept shows that our proposal is implementable beyond *Interacto*. Based on this Scala implementation, we did not identify barriers for porting our proposal to other standard UI toolkits (*e.g., Android*, *JavaFX*, *Qt*, *WPF*). However, the families of user interactions may depend on the UI event these UI toolkits support. Porting our proposal to other frameworks than *Interacto* has the following requirements: considering user interactions as objects; separating user interaction behavior and user interaction data. For example, a port to *SwingState* [2] would consider *SwingState* as the engine for the behavior of user interactions. Our Scala proof of concept can use SwingState (developed in Java) for defining user interactions. It would also require to add the concept of command and the families of user interactions. This is also the case for the frameworks that propose user interaction formalisms, such as *Proton++* [19] or dataflow approaches, such as *Djnn/Smala* [24].

RQ3 – Expressiveness

The current scope of our proposal encompasses standard input devices (mouse, keyboard, touch screen), used to develop traditional desktop, Web, and mobile applications. The proposed type system is static so that its support of dynamically typed user interactions is limited. Regarding the generalization of our proposal, a targeted programming language must have a strong type system and support union type and object substitutability. Porting our proposal to frameworks that define and process user interactions, would require to consider them as the engine for defining the behavior of user interactions. Developers must then manually port UI commands, families of user interactions and the proposed API.
We discussed the generalization of our proposal with as example the *Java* language. To fully work, our proposal requires a programming language supporting *union types*.

5 RELATED WORK

5.1 Types in UI Events

Mainstream UI frameworks, such as Angular or the Web API, rely on UI events and callbacks to process user interactions. For example using the Web API, the following *TypeScript* code processes a mouse move event:

```
1 element.addEventListener('mousemove', (e: MouseEvent) => {
2  // Callback executed on each mouse move on 'element'
3 });
```

The provided callback methods have one argument that corresponds to the UI event to process. With strongly typed programming languages, those events are typed based on the kind of the triggered UI events. In our example, the type of the UI event `e` is `MouseEvent`. In *TypeScript*, the UI event type is inferred from the value of the first argument of `addEventListener` (here the string '*mousemove*'). Typed UI events allow partial user interaction substitution: a developer can replace a

[9] https://github.com/interacto/interacto-javafx

mouse move event with any mouse event without changing the code. However, a UI event is not a user interaction. A user interaction is composed of a sequence of UI events. So, relying on UI events is not as expressive as defining user interactions [7].

5.2 UI frameworks

The closest research work is *Interacto* [7], a framework for processing user interactions. *Interacto* reifies user interactions as objects and allows full user interaction substitution (*i.e.,* only on user interactions of the same family that use the same input device). Our work relies on and extends these two *Interacto* concepts by defining a user interaction type system that enables: full and partial user interaction substitution; user interaction combination; user interaction code analysis. Our implementation extends the *TypeScript Interacto* implementation to demonstrate the ability of our proposal to overcome the issues we discussed in Section 2.

Different research works promoted the use of user interactions as objects (in object-oriented programming). *Interacto* [7], as we just discussed. *ICOs* models UI behavior using Petri nets [25]. *SwingStates* codes user interactions as finite-state machines [2]. A *SwingStates* FSM contains both the user interaction definition and the changes applied on the system during its execution. So, switching between user interactions for a same user command requires various code changes. A *SwingStates* user interaction has the generic type *StateMachine*, so that this approach does not focus on typing facilities. *Proton* and *Proton++* are approaches that represent multi-touch interactions as regular expressions [19, 20]. Similarly to *SwingStates*, *Proton++* focuses on describing user interactions and mixes both the behavior of an interaction and its actions on the system. This approach does not provide any typing facility. *Malai* is an architectural design pattern that considers user interactions as objects [6, 9]. Closely related, *Loa* is an approach for developing interactive systems [5]. It relies on Malai (user interaction are first class-objects) with templating and data binding facilities. The *Djnn/Smala* programming language [24] is interaction-oriented and aims at reducing the programming distance between processing interactions and producing the output commands. This approach also aims at leveraging reactive programming to limit the spaghetti of callbacks that classical event processing approaches suffer from. Using this approach, a developer can associate an FSM to a widget for defining its behavior. Such FSMs can trigger UI commands. Oney et al. proposed an interaction model for easing the development of multi-touch user interactions [27]. This model may be used as back-end of multi-touch interactions in *Interacto* and thus of our proposal. They also provide mechanisms for detecting and resolving conflicts between touch interactions at run time. Our proposed approach relies on representing user interactions as objects to propose a user interaction type system: a concern these other approaches do not consider. Our proposal also aims at being integrated within existing and widely used programming languages such as *TypeScript*.

Raffaillac and Huot investigated the benefits of using the Entity-Component-System (ECS) architectural model for programming interactive objects [32]. This work operates at the architectural level of interactive systems while our work focuses on programming and processing user interactions.

Hudson et al. proposed an approach for easing UI development tasks by augmenting UI toolkits with interactive objects (called *interactors*) developers can customize [18]. Our approach does not focus on interactive objects, but on how developers can handle and process user interactions that can be part of such interactive objects.

Schwarz et al. proposed an approach for supporting the uncertainty that input devices may imply [35]. Our approach does not tackle this problem yet. Uncertainty is particularly challenging for the type checking rules to spot issues.

Compared to our proposal (and as discussed in Section 4.3.2), research approaches that define and process user interactions (*e.g., SwingStates*, *Proton++*, *Djnn/Smala*) can be used as engines for

defining the behavior of user interactions. Moreover, some of these approaches tackle a larger problem than defining user interactions. For example, *Djnn/Smala* also aims at defining the behavior of interactive objects based on reactive programming concepts.

5.3 Detecting issues in user interaction code

Regarding UI listeners, researchers have proposed approaches for detecting issues in listeners (aka. callbacks) on Web or Android interactive systems [1, 8, 17, 22]. These research works demonstrate two facts: handling listeners is a complex task that may lead to issues; analyzing listeners and user interactions is important for detecting potential bugs.

Regarding identifying issues in user interaction code, formal methods are longstanding approaches for verifying user interface and interaction properties [10, 11, 26, 31]. These approaches complement the code analyses we propose, similarly as formal methods and linters co-exist in software engineering. Our proposal focuses on classical interactive systems (*e.g.,* Web applications) while formal methods are usually used for critical interactive systems.

Gardey et al. proposed an approach to assist non-developer stakeholders in fixing or changing user interactions during the interactive system execution [14]. This approach complements ours by focusing on different stakeholders: developers for our approach, and non-developers for them.

5.4 Type systems in other domains

Other domains in software engineering have leveraged type system concepts to a specific domain. For example in the model-driven engineering (MDE) community, research works have focused on typing (and sub-typing) models [12, 16, 36]. These works brought flexibility while handling models and metamodels in MDE. Similar work was conducted in the database domain [23]. Our work aims to achieve similar purposes but for user interactions.

6 CONCLUSION

This paper introduces a type system for user interactions. This type system permits three advances compared to the current state of the art: user interaction substitution; user interaction union; the definition of dedicated user interaction type checking rules. We implemented our proposal in *TypeScript* as an improvement of the *Interacto* library. This implementation demonstrates the ability of our proposal to be fully operational and sound, and to overcome: flexibility issue by allowing user interactions substitution; expressiveness issue by reducing the boilerplate code required to use multiple user interactions to produce the same user command; reliability issue by enabling type checking capabilities to identify potential programming issues related to user interactions.

In our future work, we aim to investigate to what extent our approach can be adapted to specific domains such as virtual reality. We will also investigate the support of dynamic typing to consider non-discrete user interactions, and how to support uncertainty as investigated in [35].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their numerous comments that helped improve the paper.

REFERENCES

[1] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2016. Understanding JavaScript event-based interactions with Clematis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 2 (2016), 1–38. https://doi.org/10.1145/2876441

[2] Caroline Appert and Michel Beaudouin-Lafon. 2008. SwingStates: Adding state machines to Java and the Swing toolkit. *Software: Practice and Experience* 38, 11 (2008), 1149–1182. https://doi.org/10.1002/spe.867

[3] Caroline Appert, Olivier Chapuis, Emmanuel Pietriga, and María-Jesús Lobo. 2015. Reciprocal drag-and-drop. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 6 (2015), 1–36. https://doi.org/10.1145/2785670

[4] Patrick Baudisch, Edward Cutrell, Dan Robbins, Mary Czerwinski, Peter Tandler, Benjamin Bederson, Alex Zierlinger, et al. 2003. Drag-and-pop and drag-and-pick: Techniques for accessing remote screen content on touch-and pen-operated systems. In *Proc. of Interact'03*. IOSPress, NL, 57–64.

[5] Olivier Beaudoux, Mickael Clavreul, Arnaud Blouin, Mengqiang Yang, Olivier Barais, and Jean-Marc Jezequel. 2012. Specifying and running rich graphical components with loa. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, USA, 169–178. https://doi.org/10.1145/2305484.2305513

[6] Arnaud Blouin and Olivier Beaudoux. 2010. Improving modularity and usability of interactive systems with Malai. In *EICS'10: Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, USA, 115–124. https://doi.org/10.1145/1822018.1822037

[7] Arnaud Blouin and Jean-Marc Jézéquel. 2021. Interacto: A Modern User Interaction Processing Model. *IEEE Transactions on Software Engineering* 48, 9 (2021), 1–20. https://doi.org/10.1109/TSE.2021.3083321

[8] Arnaud Blouin, Valéria Lelli, Benoit Baudry, and Fabien Coulon. 2018. User Interface Design Smell: Automatic Detection and Refactoring of Blob Listeners. *Information and Software Technology* 102 (May 2018), 49–64. https://doi.org/10.1016/j.infsof.2018.05.005

[9] Arnaud Blouin, Brice Morin, Olivier Beaudoux, Grégory Nain, Patrick Albers, and Jean-Marc Jézéquel. 2011. Combining Aspect-Oriented Modeling with Property-Based Reasoning to Improve User Interface Adaptation. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, Pisa, Italy, 85–94. https://doi.org/10.1145/1996461.1996500

[10] José Creissac Campos, Camille Fayollas, Michael D Harrison, Célia Martinie, Paolo Masci, and Philippe Palanque. 2020. Supporting the analysis of safety critical user interfaces: an exploration of three formal tools. *ACM Transactions on Computer-Human Interaction (TOCHI)* 27, 5 (2020), 1–48. https://doi.org/10.1145/3404199

[11] José C Campos and Michael D Harrison. 1997. Formally verifying interactive systems: A review. In *Design, Specification and Verification of Interactive Systems' 97: Proceedings of the Eurographics Workshop in Granada, Spain, June 4–6, 1997*. Springer, Vienna, 109–124. https://doi.org/10.1007/978-3-7091-6878-3_8

[12] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2017. Safe model polymorphism for flexible modeling. *Computer Languages, Systems & Structures* 49 (2017), 176–195. https://doi.org/10.1016/j.cl.2016.09.001

[13] Tore Dybå and Torgeir Dingsøyr. 2008. Empirical studies of agile software development: A systematic review. *Information and software technology* 50, 9-10 (2008), 833–859. https://doi.org/10.1016/j.infsof.2008.01.006

[14] Juan Cruz Gardey, Alejandra Garrido, Sergio Firmenich, Julián Grigera, and Gustavo Rossi. 2020. UX-painter: an approach to explore interaction fixes in the browser. *Proceedings of the ACM on Human-Computer Interaction* 4, EICS, Article 89 (2020), 21 pages. https://doi.org/10.1145/3397877

[15] Carl A Gunter. 1992. *Semantics of programming languages: structures and techniques*. MIT press, USA.

[16] Clément Guy, Benoit Combemale, Steven Derrien, Jim RH Steel, and Jean-Marc Jézéquel. 2012. On model subtyping. In *Modelling Foundations and Applications: 8th European Conference, ECMFA 2012*. Springer, Vienna, 400–415. https://doi.org/10.1007/978-3-642-31491-9_30

[17] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for Android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM/IEEE, USA, 532–542. https://doi.org/10.1145/3238147.3238181

[18] Scott E Hudson, Jennifer Mankoff, and Ian Smith. 2005. Extensible input handling in the subArctic toolkit. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM New York, NY, USA, USA, 381–390. https://doi.org/10.1145/1054972.1055025

[19] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. 2012. Proton++ a customizable declarative multitouch framework. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*. ACM, USA, 477–486. https://doi.org/10.1145/2380116.2380176

[20] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. 2012. Proton: multitouch gestures as regular expressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, USA, 2885–2894. https://doi.org/10.1145/2207676.2208694

[21] Valéria Lelli, Arnaud Blouin, and Benoit Baudry. 2015. Classifying and Qualifying GUI Defects. In *8th IEEE International Conference on Software Testing, Verification and Validation*. IEEE, USA, 1–10. https://doi.org/10.1109/ICST.2015.7102582

[22] Valéria Lelli, Arnaud Blouin, Benoit Baudry, Fabien Coulon, and Olivier Beaudoux. 2016. Automatic Detection of GUI Design Smells: The Case of Blob Listener. In *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS'16)*. ACM, USA, 263–274. https://doi.org/10.1145/2933242.2933260

[23] Yuri Leontiev, M Tamer Özsu, and Duane Szafron. 2002. On type systems for object-oriented database programming languages. *ACM Computing Surveys (CSUR)* 34, 4 (2002), 409–449. https://doi.org/10.1145/592642.592643

[24] Mathieu Magnaudet, Stéphane Chatty, Stéphane Conversy, Sébastien Leriche, Celia Picard, and Daniel Prun. 2018. Djnn/Smala: A conceptual framework and a language for interaction-oriented programming. *Proceedings of the ACM*

on Human-Computer Interaction 2, EICS, Article 12 (2018), 27 pages. https://doi.org/10.1145/3229094

[25] David Navarre, Philippe Palanque, Jean-Francois Ladry, and Eric Barboni. 2009. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Trans. on CHI* 16, 4 (2009), 1–56. https://doi.org/10.1145/1614390.1614393

[26] Raquel Oliveira, Sophie Dupuy-Chessa, Gaelle Calvary, and Daniele Dadolle. 2016. Using formal models to cross check an implementation. In *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, USA, 126–137. https://doi.org/10.1145/2933242.2933257

[27] Steve Oney, Rebecca Krosnick, Joel Brandt, and Brad Myers. 2019. Implementing multi-touch gestures with touch groups and cross events. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, USA, 1–12. https://doi.org/10.1145/3290605.3300585

[28] Vik Parthiban, Pattie Maes, Quentin Sellier, Arthur Sluÿters, and Jean Vanderdonckt. 2022. Gestural-Vocal Coordinated Interaction on Large Displays. In *Companion of the 2022 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, USA, 26–32. https://doi.org/10.1145/3531706.3536457

[29] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press, USA.

[30] Rachel Potvin and Josh Levenberg. 2016. Why Google stores billions of lines of code in a single repository. *Commun. ACM* 59, 7 (2016), 78–87. https://doi.org/10.1145/2854146

[31] Daniel Prun and Pascal Béger. 2022. Formal Verification of Graphical Properties of Interactive Systems. *Proceedings of the ACM on Human-Computer Interaction* 6, EICS, Article 167 (2022), 30 pages. https://doi.org/10.1145/3534521

[32] Thibault Raffaillac and Stéphane Huot. 2019. Polyphony: Programming interfaces and interactions with the entity-component-system model. *Proceedings of the ACM on Human-Computer Interaction* 3, EICS, Article 8 (2019), 22 pages. https://doi.org/10.1145/3331150

[33] Thibault Raffaillac and Stéphane Huot. 2022. What do Researchers Need when Implementing Novel Interaction Techniques? *Proceedings of the ACM on Human-Computer Interaction* 6, EICS, Article 159 (2022), 30 pages. https://doi.org/10.1145/3532209

[34] Thiago Rocha Silva. 2022. Towards a Domain-Specific Language to Specify Interaction Scenarios for Web-Based Graphical User Interfaces. In *Companion of the 2022 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, USA, 48–53. https://doi.org/10.1145/3531706.3536463

[35] Julia Schwarz, Scott Hudson, Jennifer Mankoff, and Andrew D Wilson. 2010. A framework for robust and flexible handling of inputs with uncertainty. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*. ACM New York, NY, USA, USA, 47–56. https://doi.org/10.1145/1866029.1866039

[36] Jim Steel and Jean-Marc Jézéquel. 2007. On model typing. *Software & Systems Modeling* 6 (2007), 401–413. https://doi.org/10.1007/s10270-006-0036-6

[37] Jacob O Wobbrock, Meredith Ringel Morris, and Andrew D Wilson. 2009. User-defined gestures for surface computing. In *Proceedings of the SIGCHI conference on human factors in computing systems*. ACM, USA, 1083–1092. https://doi.org/10.1145/1518701.1518866