Hook: An Embedded Domain-Specific Language for Fusing Implicit Interactions to Explicit Event Handlers*

TOMOKI SHIBATA, Tufts University, USA
MATTHEW AHRENS, Tufts University, USA
ROBERT J.K. JACOB, Tufts University, USA

Emerging physiological sensing technologies are leading to an interaction design paradigm—namely implicit interactions—in which computers have the ability to implicitly perceive and respond to the physiological understandings of their users. However, a noticeable challenge remains in integrating such implicit interactions into a conventional event-based interactive system, which typically responds to the user's explicit events. (e.g., key presses, mouse clicks, etc.) The challenge is due to multiple input sources of different types, and, by extension, demands programmers to be responsible for prescribing how interactions of the two different types, implicit and explicit, interfere with each other. To address this challenge, we introduce a domain-specific language (DSL), Hook, that allows programmers to declaratively express *when* to perform implicit interactions with respect to desired explicit ones and *how* to fuse the computational effects of the interactions in a modular way. The Hook language treats interactions as a first-class abstraction and provides three types of fusion strategies, which assist programmers in gluing the two types of interactions together. This paper describes an implementation of the Hook language as a DSL embedded in Haskell and formalizes the language as a computational model for future replication and extension. We also demonstrate the utility of the language through three case studies, each of which implements an implicit interaction that extends behaviors of a published interactive system. Lastly, we discuss limitations of the language in its current state and how the language could further aid programmers to achieve separation of concerns while maintaining algorithmic precision in implementing a complex combination of different types of interactions.

CCS Concepts: • **Human-centered computing → Human computer interaction (HCI)**; **Systems and tools for interaction design**; • **Software and its engineering → Domain specific languages**.

Additional Key Words and Phrases: Implicit User Interfaces, Implicit Interactions, Explicit Interactions, Domain-Specific Languages, Metaprogramming, Template Haskell, Haskell, Hook

1 INTRODUCTION

Emerging physiological sensing technologies provide a multitude of data about the user's physiological status. Such data combined with advanced computational capabilities, such as machine learning techniques, opens up a door for designing interactive systems that are better aware of their users. However, introducing an additional input source to an existing interactive system often increases complexity. In particular, this is the case where the type of additional input is heterogeneous, compared to other conventional input types. (For example, imagine user status as an input in contrast to key press events.) As a result, the problem calls for programmers to take on greater responsibility for managing this increased complexity in their implementation.

To illustrate this challenge concretely, let us consider the following scenario. Suppose a programmer is implementing a small Graphical User Interface (GUI) application that responds to two different types of inputs: the user's explicit mouse actions, namely events, and implicitly perceived user's busyness, namely user status information. This application displays a graphical object on the screen. The object is initially painted in black; the object color changes to cyan when the user

*(Draft version. 2020)

Authors' addresses: Tomoki Shibata, Tufts University, Medford, MA, USA, tshibata@cs.tufts.edu; Matthew Ahrens, Tufts University, Medford, MA, USA, mahrens@cs.tufts.edu; Robert J.K. Jacob, Tufts University, Medford, MA, USA, jacob@cs.tufts.edu.

presses the mouse; and it changes back to black when the user releases the mouse. Simultaneously, the object color changes by increasing the value of the red channel and decreasing the value of the green channel when the user is found to be busier.

While there are many ways to encode such interactive behaviors, this scenario introduces two competing desires in the programmer's mind. First, when managing multiple input sources, the programmer wishes to write maintainable code by making it small and modular. Second, the programmer must also ensure all computations responding to inputs result in the expected program states. In other words, the programmer ends up needing to force themselves to choose one of the following approaches: a) prioritizing "*separation of concerns*" [7] by putting a logical boundary between how the application responds to user's explicit events and how it does to implicitly perceived user status information; or b) prioritizing the *algorithmic precision* of interactive behaviors by employing compound conditionals to list procedures that cover all possible input combinations.

To expose the pros and cons of both approaches, Figure 1 implements the example GUI application using JavaScript-like syntax, in which the interactive behaviors are represented as handler logic. As for the approach a), the functions `p` and `q` define the event handler logic and user status handler logic, respectively. It achieves separation of concerns, and its benefits include code reuse of the

```
const Events = [MouseDown, MouseUp, ...];
let world = {..., red : 0 , green : 0 , blue : 0, ..., x : 0, ...};
let user = {busy : true};
let p =
  function(event){
    if(event == MouseDown){
      world.red = 0;
      world.green = 255;
      world.blue = 255;
    }else if(event == MouseUp){
      world.blue = 0;
      world.green = 0;
      world.red = 0;
    }else{
      ... //for other events
    }
  };

let q =
  function(user){
    if(user.busy == true){
      world.red = 255;
      world.green = 0;
    }
  };
```

```
let r =
  function(event, user){
    if(event == MouseDown
       && user.busy == true){
      world.red = 255;
      world.green = 0;
      world.blue = 255;
    }else if(event == MouseDown
             && user.busy != true){
      world.red = 0;
      world.green = 255;
      world.blue = 255;
    }else if(event == MouseUp){
      world.red = 0;
      world.green = 0;
      world.blue = 0;
    }else{
     ... //for other events
    }
  };
```

Fig. 1. Interactions as handler logic written in a JavaScript-like syntax for the example GUI scenario. The `world` captures the application state, and the (`red`, `green`, `blue`) represents the color of the object on the screen. The `user` describes the user status information, which indicates whether the user is busy or not in this example.

handlers improving modularity. However, this approach does not explain how these handlers are synchronized, which implies multiple possible results based on the application run-time. As for the approach b), the function `r` precisely combines both handler logics into a single logic. This approach ensures the algorithmic precision by specifying exactly when and how the two handler logics interact to produce a single, final outcome. However, the procedure to respond to any particular input event ends up being distributed over every possible user status. This distribution often forces the programmer to implement boilerplate logic redundantly.

While common abstractions from general-purpose programming languages allow programmers to choose a trade-off, we argue that there should exist a supportive tool for programmers to achieve separation of concerns without giving up algorithmic precision when writing an interactive system that handles both explicit user events and implicitly perceived user status information.

To that end, we designed a domain-specific language (DSL), named Hook, that allows programmers to express event handler logic and user status handler logic separately and compose them to produce the single handler that the application run-time can use. The Hook language comprises the two key ideas: (1) the abstraction layer to enclose the handler logics in the concept of explicit and implicit interaction, and (2) the fusion strategies to precisely glue the results of computations expressed as explicit and implicit interactions.

Figure 2 illustrates how the programmer would implement the same example GUI application using the Hook language. The code in the quasi-quotes, `[hook| ... |]`, encodes the interactive behaviors using the primitives of the language. The expression bound to the variable `e` uses the

```
data Event = MouseDown | MouseUp | ...
data World = World { ..., red :: Int, green :: Int, blue :: Int
                   , ..., x :: Int, ...}
data User = User { busy :: Bool }

[hook|
  e = explicit world where
        MouseDown -> do { returnIO world { red = 0, green = 255, blue = 255 } }
      | MouseUp   -> do { returnIO world { red = 0, green = 0, blue = 0 } }
      | _         -> do { returnIO world } --for other events

  i = implicit world where
        User { busy = True } -> do { returnIO world { red = 255, green = 0 } }
      | _                    -> do-nothing

  c = (after e
       handles MouseDown
       do i
       then merge)
|]
```

Fig. 2. Interactions expressed with the Hook language for the same example GUI scenario. The explicit interaction, bound to `e`, encloses the event handler logic. The implicit interaction, bound to `i`, encloses the user status handler logic. The composed interaction, bound to `c`, is the handler that will be registered to the eco-system that the GUI application runs on.

`explicit` keyword to enclose the event handler logic in the concept of explicit interaction, and it modifies the application state, `world`, based on explicit user input events. (e.g., mouse events produced by the user.) Similarly, the expression bound to `i` uses the `implicit` keyword to enclose the user status handler logic in the concept of implicit interaction, and it modifies `world` based on implicitly perceived user status information. (e.g., the user's current busyness.) Together, these primitives `e` and `i` achieve the separation of concerns that the functions `p` and `q` achieved in Figure 1.

To also achieve algorithmic precision, the expression bound to `c` is composed of `e` and `i`, in which *when* and *how* `i` runs with respect to `e` are precisely declared by the keywords `after`, `handles`, and `merge`. First, the `after` keyword specifies that the implicit interaction, `i`, modifies `world` after the explicit interaction, `e`, does. Then, the `handles` keyword followed by the `MouseDown` specifies that `i` modifies `world` only when `e` handles `MouseDown` events. Lastly, the `merge` keyword specifies that the modifications made by `e` and `i` are glued together using the merge strategy—one of the three fusion strategies that the Hook language provides. The later sections cover this composition process in more detail and show additional variations.

This paper contributes to methods of integrating implicit interactions into conventional interactive systems by providing: 1) A set of abstractions that help programmers to describe explicit and implicit interactions as parameterized "world transformation" [9] functions; 2) An implementation of the Hook language, as an embedded DSL in Haskell, that allows programmers to express explicit and implicit interactions independently and specify their compositions declaratively; 3) The computational model of the Hook language compiler for future language replication and extension; and 4) The case studies that demonstrate how the Hook language assists programmers in achieving both separation of concerns and algorithmic precision.

2 RELATED WORK

2.1 Explicit and Implicit Interactions

Referring to the field of Human-Computer Interaction (HCI), Hornbæk and Oulasvirta [20] state that "the term interaction is field-defining, . . ." There is a substantial body of work on the design of explicit interactions. This work includes Direct Manipulation [55], and, for example, Graphical User Interfaces (GUI) appear widely in our daily life. Meanwhile, researchers explore other forms of interactions beyond Direct Manipulation as more sensing technologies become available. One of the forms is an interaction design that concerns environmental contexts as a source of input. For example, a location-aware computer system, discussed in the context of ubiquitous computing [60], and context-enabled applications [47] are such examples. By extension, Schmidt [48] introduces the term "implicit human-computer interaction" for such the design and defines it as "an action performed by the user that is not primarily aimed to interact with a computerised system but which such a system understands as input."

Aligning with recent works providing an overview [13], introducing a framework [27], and discussing challenges [50] in the design of implicit interactions, we echo that implicit and explicit interactions are different in nature. We consider that explicit interaction involves a process where a human explicitly sends a command (or equivalents) to a computer and then the computer responds to it. Meanwhile, implicit interaction involves a process where, while a human and computer are in an interactive relationship, the computer implicitly perceives *thing* related to the human, treats it as input, and then responds to it. The word "implicitly perceive," which we will use through this paper, implies that the computer ideally demands no effort from the human when understanding the *thing* as input and that the human is unconscious of producing the input. In this sense, cognition-aware computing [3] and some Brain-Computer Interfaces (BCI) works would be user interface designs that involve implicit interactions. For example, Brainput [57] implicitly measures the user's "brain

activity patterns," utilizes them as input to a human-robot interactive system, and then controls the autonomy level of robots. Similarly, BACh [62] implicitly measures the user's "cognitive workload," leverages it as input, and then controls the difficulty levels of tasks in learning contexts. On the other hand, if a BCI translates brain activities that the user intentionally created to commands for performing computer tasks (e.g., typing [41]), then it would be a design involving explicit interactions.

While Schmidt [48] generalizes the *contexts* and proposes the "situational context," our work further expands it to include implicitly perceived user status information—in particular physiological understandings of the user—and treats it as a viable input for an interactive system.

2.2 Formal Specifications

In the field of HCI, formal specification becomes increasingly important as novel user interface designs emerge. There is considerable work on the methods for formally describing behaviors of interactive systems. This work includes state diagram based approaches, such as network definition language [37], terminal state transition diagram [42], a language for visual programming [24], statecharts [15], and grammar based approaches, such as action languages [46], SYNGRAPH system [40], BNF-based notations [23], and so on.

Besides, with abstraction techniques seen in software engineering, the formal specification research in HCI introduces User Interface Management System (UIMS) [28], many of which provide higher-level abstractions specific to definite purposes and aim to facilitate software developments. For example, ALGAE [12] is a UIMS that abstracts and specifies events and event handlers for dialogues, and Sassafras [18] comes with a language for specifying concurrent dialogues. MIKE [39] is also a UIMS that describes "interactive interface" using data types of the application. Meanwhile, UIMS is not limited to dialogue-style interfaces. For example, some work introduces the concept of constraints to specify relationships among data and views [2, 19]. Garnet [35] has its own constraint system and uses one-way constraints to specify relationships among objects and their interactive behaviors. Also, UAN [16] concerns "the behavioral aspects" of interactive systems and provides notations to describe a series of user actions.

The formal specification research also extends to non-WIMP user interface designs as hardware and software architectures advance. Some work aims to capture characteristics of non-WIMP including "continuous interaction between user and computer via several parallel, asynchronous channels or devices" [25]. For example, PMIW [26] combines "data-flow" and "token-based event handlers" to describe interactions found in a virtual environment. HephaisTK [8] concerns the synchronization issue in a combination of Tangible User Interfaces and other modalities and discusses fusion techniques. Hasselt UIMS [5] concerns multimodal interactions and provides an integrated system that combines events from multiple input modalities and specifies handlers for "composite events."

Past work suggests that providing abstractions that suit specific interaction types facilitates user interface design, while the level of abstraction is often a trade-off with the level of expressiveness. Our work considers "interaction" as the highest level of abstraction and utilizes it as the lens to capture interactive behaviors involved in user interface designs. In particular, our work defines "interaction" as a first-class primitive in the proposed language and studies a method to compose two types of interactions (explicit and implicit) derived from the "interaction" abstraction.

2.3 Domain-Specific Language (DSL)

In the field of HCI, the term DSL covers the concept of User Interface Description Language (UIDL). The foundation of UIDL lies within the development of UIMS; therefore, some work on the designs of UIDL shares parts of the aims UIMS sets forth. This work includes model-based user interface

designs, some of which aim to cope with the increasing diversity in input/output modalities and often leverage the form of XML to describe specifications and constructs of user interfaces. For example, USIXML [30] describes models related to various levels of user interface design processes and makes user interface designs "device-, platform-, modality-, and context-independent." MARIA [43] covers the domain of web services, whereas PUC [38] covers interfaces for remotely controlling devices. Not limited to XML-based techniques, ICOs [36] provides "Petri-net-based" notations. In addition, other forms of modeling include such as: Interactors in Garnet [34] model input event handling using Lisp-family syntax; Schwarz et al. [49] model "probabilistic events"; Proton [29] uses regular expressions for touch gestures; and ProbUI [4] defines declarative expressions along with "probabilistic gesture models."

UIDLs in HCI offer various levels of formalism in modeling user interfaces, yet it does not necessarily imply they are programmable per se. Meanwhile, in the area of Programming Language research, a DSL is often designed to be more restrictive than general-purpose programming languages but better to optimize, formalize, and prove properties of programs written in that DSL. For example, PADS [11] is a DSL for specifying, parsing, and pretty-printing ad-hoc file formats. Forest [10] is also a DSL for specifying and traversing ad-hoc file systems. Moreover, DSL could be stand-alone. For example, Elm [6] is a programming language that is domain-specific to interactive web applications and provides an eco-system including design patterns, syntax, compiler (transpiler to JavaScript code), and run-time. Djnn/Smala [31] proposes "Interaction-Oriented Programming" and provides syntax and semantics to express relationships between "events and reactions." Flapjax [33] provides a compiler (transpiler to JavaScript code) and adds "reactive semantics" to JavaScript. Similarly, Mobl [17] provides syntax and a compiler to unify "design, styling, data modeling, querying and application logic" in mobile web application developments.

While the Functional Reactive Programming (FRP) paradigm (e.g., "signal" in Elm, "event stream" in Scala.React [32], etc.) captures discrete events coming from multiple input sources and allows programmers to merge them, our proposed language captures handler logics using the "interaction" abstraction and helps programmers to glue the computational effects of the two types of interactions, each of which responds to a different type of input.

3 THE HOOK LANGUAGE

The Hook language is a programming language for expressing explicit and implicit interactions and their compositions; it consists of a surface syntax, run-time libraries, and a compiler. In this section, we focus on the surface syntax of the language and describe our implementation[1] of the Hook language as an embedded DSL in Haskell.

3.1 Abstracting Interactions as World Transformation Functions

The benefit of using the Hook language is that programmers can express two types of interactions, explicit and implicit, independently and can compose them declaratively. For this, the Hook language provides the abstraction layer where programmers can extract interactions as a data type and compose them via a higher-order function. In other words, interaction is a first-class citizen in the Hook language, and is the unit of programmers' interest. The following code fragment is the declaration of `Interaction` type in our host language, Haskell, and is provided as a part of the Hook language run-time libraries.

```
data Interaction kind user event m world
```

The `Interaction` type makes it possible to represent interactive behaviors of arbitrary applications by exposing application-specific information as type variables. This means that when using

[1]The code for this implementation is provided at (URL).

the Hook language, the programmer will be responsible for implementing a concrete type for each of the five type variables per application. For example, the `kind` type variable specifies the kind of *this* interaction, and it will be either `Explicit` or `Implicit` in this research. The `event` and `user` type variables parameterize the explicit user events and implicitly perceived user status information, respectively, that *this* interaction responds to. The `m` type variable exposes application-specific contexts that could involve side-effects. For instance, `m` may be `IO`, `State`, or even their combination, for monadic (e.g., sequence-dependent) computations, or it may be `Identity` for pure computations. Lastly, the `world` type variable parameterizes the application state. For example, suppose a programmer is implementing a GUI application involving a counter. Then, `world` for this application will include fields for the current count, the appearance of the counter, and so on.

The `Interaction` type also helps us to formally define the terms "explicit interaction" and "implicit interaction" in the context of this research. As the foundation, we first argue that "interaction" can be explained as a process that takes input and the current (application) state and then modifies the (application) state based on the given input within a specific context. Needless to say, it assumes humans will be a source of inputs and perceive the updated state. To be specific, using the vocabularies from the `Interaction` type, we define "interaction" as an anonymous function of type (*input* `-> world -> m world`). This type signature discloses that it is a function taking *input* and the (current) `world` and then returning `m world`, in which the resulting `world` may include some modifications made in the context `m`. Using the "interaction" as the umbrella term, we then define that "explicit interaction" is an "interaction" where *input* $\in$ `event` and that "implicit interaction" is an "interaction" where *input* $\in$ `user`.

Lastly, we refer to the modification of the application state—represented as (`world -> m world`) in the "interaction"—as world transformation function, whose idea stems from the functional IO programming [9]. Leveraging this set of abstractions, programmers now can describe explicit interactions as world transformation functions parameterized over explicit user events and implicit interactions as ones parameterized over implicitly perceived user status information. Next, we present how programmers would express the two types of interactions using the syntax of the Hook language.

3.2 Expressing Interactions

As stated, explicit and implicit interactions are parameterized world transformation functions in the Hook language. Hence, expressing an explicit interaction is to describe how explicit user events change the current application state; similarly, expressing an implicit interaction is to describe how implicitly perceived user status information change the current application state.

Figure 3 shows the grammar of the Hook language, and the following example implements one explicit and one implicit interaction using the syntax of the Hook language. In this example, the explicit interaction is bound to the variable `e` and the implicit one is bound to the variable `i`.

```
e = explicit (World color text) where
      MouseDown     -> do { returnIO (World Red text) }
    | MouseUp       -> do { returnIO (World Black text) }
    | (StdIn words) -> do { returnIO (World color (text ++ words)) }

i = implicit (World color text) where
      (Happy True)  -> do { returnIO (World color (text ++ ":)")) }
    | (Happy False) -> do { returnIO (World color (text ++ ":(")) }
```

The definition of the explicit and implicit interaction begins with the keyword, `explicit` and `implicit`, respectively. Then, the keyword is followed by the pattern expressing the application

```
Exp ::= When Exp Condition do Exp How
      | x
      | explicit p where {p -> e |}+
      | implicit p where {p -> e |}+
      | Exp and Exp
      | Exp or Exp
      | it

p ::= <HOST LANGUAGE PATTERN>
e ::= <HOST LANGUAGE EXPRESSION>
    | do-nothing

Decl ::= x = Exp

When ::= before | after

Condition ::= handles p | ϵ

How ::= then merge
      | then overwrite
      | then tweak
      | ϵ
```

Fig. 3. The grammar describing well-formed programs written in the Hook language. The programs are a collection of type level declarations (Decl), each of which could be bound to an identifier x. Patters p and expressions e are provided by the host language Haskell and defined by the Template Haskell [52] metaprogramming library.

state, namely world patterns. In this example, `(World color text)` is the one, and it reveals that the world has two fields, `color` and `text`, and brings those variables into scope.

The `where` keyword, following the world pattern, indicates the beginning of the pattern matching, and each case match is a pair of pattern and world transformer delimited by the right arrow, `->`. In the explicit interaction, the pattern ought to express explicit user events, namely event patterns. In this example, each of `MouseDown`, `MouseUp`, and `(StdIn words)` is an event pattern. Similarly, in the implicit interaction, the pattern ought to express implicitly perceived user status information, namely user status patterns. In this example, each of `(Happy True)` and `(Happy False)` is a user status pattern. Lastly, the world transformer, following the `->`, describes how the given world will be modified. For instance, in `e`, the `color` part of the world will change to `Red` when the user presses the mouse. For instance, in `i`, the `text` part of the world will change by appending a smiley face to the text when the user is happy.

3.3 Composing Interactions

In the Hook language, composing interactions refers to producing a new interaction based on two other interactions. (See also implications in Discussion section.) For this, the Hook language provides a special syntax—in fact the namesake of the language—that guides programmers to *hook* an implicit interaction into an explicit one, which results in producing a new explicit interaction. The syntax is designed to express the internal behavior of the newly composed interaction and requires programmers to declare *when* to perform the given implicit interaction concerning the given explicit one and *how* to fuse the computational effects of the two interactions.

The following example exhibits two composed interactions, `c1` and `c2`, where the variable `e` and `i` refers to the explicit and implicit interaction presented above.

```
c1 = (before e do i)
c2 = (after  e do i)
```

The definition of the composed interaction begins with either the `before` or `after` keyword. Then, the keyword is followed by the explicit interaction, the `do` keyword, and then the implicit interaction. As it sounds, the `before` and `after` keyword declare when to perform the implicit interaction with respect to the explicit interaction. For example, `c1` is a composed interaction that behaves in the way that "`before` running `e`, (`do`) run `i`." This means that given the current world,

`i` will first modify the given world, and then `e` will modify the world modified by `i`. Similarly, `c2` is a composed interaction that behaves in the way that "`after` running `e`, (`do`) run `i`." This means that given the current world, `e` will first modify the given world, and then `i` will modify the world modified by `e`. Recall that one explicit interaction can handle more than one event pattern. For instance, `e` will respond to the three events, `MouseDown`, `MouseUp`, and `(StdIn words)`. As a result, in `c1` and `c2`, whenever `e` runs on any of the three events, `i` also runs.

The Hook language has the `handles` keyword for narrowing down when to perform the implicit interaction based on event patterns of the explicit interaction. In the syntax, the `handles` keyword is always followed by one event pattern, and those are placed between the explicit interaction and the `do` keyword, when used. The following example exhibits three composed interactions, `c3`, `c4` and `c5`, using the keyword; therefore, `i` runs only when does `e` run on the specified event.

```
c3 = (after e handles MouseDown     do i)
c4 = (after e handles (StdIn words) do i)
c5 = (after e handles (StdIn "\n")  do i)
```

As in, `c3` is a composed interaction, where `i` runs after `e` handles `MouseDown` but doesn't when `e` handles any other events. In `c4`, `i` runs only when does `e` respond to `(StdIn words)`, which represents key press events and `words` captures characters typed. In addition, the event pattern, following the `handles` keyword, can be more specific than ones from the explicit interaction. For instance, `c5` is still well-formed and is more specific than `c4` because `"\n"` (a newline) is more specific than `words`, referring to any characters. Note that when the character typed is not `"\n"`, `c5` can still respond to `(StdIn words)` events in the same way how `e` does.

So far, the syntax explains *when* to run the implicit interaction concerning the explicit one. Meanwhile, the following example exhibits three composed interactions, `c6`, `c7` and `c8`, that declare not only the application order of the two interactions but also *how* to fuse the effects of the two interactions.

```
c6 = (after e handles MouseDown do i then merge)
c7 = (after e handles MouseDown do i then overwrite)
c8 = (after e handles MouseDown do i then tweak)
```

To glue the two interactions together, the Hook language provides the three types of fusion strategies—merge, overwrite, and tweak. (See the details below.) As seen in this example, each of the `merge`, `overwrite`, and `tweak` keywords directly corresponds to the name of fusion strategies and declares which one to use. In the syntax, all the three follow the `then` keyword, which follows the implicit interaction. As a side note, if no fusion strategy is specified, then the Hook language compiler chooses the merge strategy as the default behavior. That is, the composed interaction `c3` and `c6` behave in the same way.

Fusion strategy details. The fusion strategies are methods to conclude the final outcome of the composed interaction based on the computational results of the two interactions ran internally. The fusion strategies recognize which interaction changed which fields of the world and then decide which of the changes, the two interactions made, will persist in the final outcome.

There are three types of fusion strategies and each of them employs a unique way to produce its outcome. The **merge** strategy keeps all changes made by the first and second interactions. (Recall that the two interactions run in the specified order.) This strategy is the most general form of fusion and is equivalent to function composition. On the other hand, the **overwrite** strategy keeps only changes made cooperatively by the first and second interactions and ones made solely by the second interaction, yet it throws away changes made solely by the first interaction. The intuition behind this strategy is that the second interaction overwrites the effects of the first interaction. Similarly,

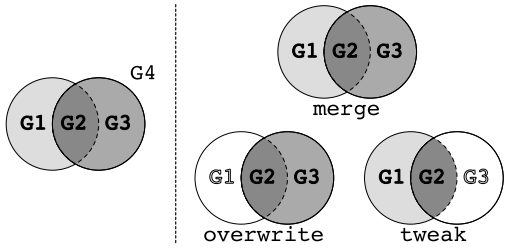

Fig. 4. Venn diagrams illustrating the fusion strategies. Left: The fields of the "world" are categorized into four groups. Right: Shaded groups include the effects that persist in the resulting world when using that particular fusion strategy.

Table 1. The possible outcomes from using the three fusion strategies in Figure 2.

Fusion Strategy	channel			
	red	green	blue	(color)
merge	255	0	255	magenta
overwrite	255	0	0	red
tweak	0	0	255	blue
(`do-nothing`)	0	255	255	cyan

Note: `do-nothing` case is for when the user is not busy; thus, no fusion strategy is applied.

the **tweak** strategy keeps only changes made cooperatively by the first and second interactions and ones made solely by the first interaction, yet it throws away changes made solely by the second interaction. The intuition behind this strategy is that the second interaction tweaks the effects of the first interaction. Note that in all the three fusion strategies, changes made by the second interaction may depend on changes made by the first interaction, regardless of whether they persist in the final outcome.

To make the descriptions transparent, Figure 4 uses Venn diagrams to visualize the notion of "which changes will persist." First of all, Figure 4 (Left) shows the premise that after running the two interactions in the specified order, each field of the world will always belong to one of the following four groups: G1 if the field was changed only by the first interaction; G2 if the field was changed by the first interaction and then by the second interaction; G3 if the field was changed only by the second interaction; and G4 if the field was changed by none of the first and second interactions. Using this grouping, Figure 4 (Right) illustrates the possible outcomes when applying the three fusion strategies. In short, applying each of the merge, overwrite, and tweak strategies results in producing the world that preserves effects made to fields belonging to (G1, G2, and G3), (G2 and G3), and (G1 and G2), respectively.

The fusion strategies empower programmers to take control of the field-level changes made to the world. To illustrate the benefit of using the fusion strategies, let us consider the example scenario presented in Introduction section once again. Recall the functions `p`, `q`, and `r` shown in Figure 1. Because the value of the green channel in `r` corresponds to the result from `q` and not `p`, we could guess that the programmer subtly implied that the user status information is handled after the event is done and merged their outcomes. However, this intention is only clear in the mind of the programmer and is not always apparent in the implementation. Furthermore, in case the programmer needed to change the logic that `r` currently employs, the programmer would probably need to re-implemented `r` from scratch. In contrast, recall the implementations of `e`, `i`, and `c` shown in Figure 2. In the Hook language, the syntax of the composed interaction directly explains the programmer's intention on how the implicit and explicit interactions interfere with each other. Moreover, changing the logic for composing interactions can be accomplished simply by choosing one of the other fusion strategies.

To find the effects of changing the fusion strategy, Table 1 shows the possible outcomes when `c` in Figure 2 employs each of the three fusion strategies, and it reveals that each case results in producing a different outcome in this example scenario. Also, Figure 5 illustrates the run-time

behavior of the composed interaction `c` that employs the overwrite strategy instead of the merge. Assume that the user of the GUI application pressed the mouse while the user was busy. Assume also that the initial world (w0), representing the current application state, contains the color black with initial fields set to 0. Following the definition of `c`, `e` first receives w0 and changes the green and blue fields to 255, which results in w1. Then, `i` receives w1 and changes the red field to 255 and the green field to 0, which results in w2. Lastly, the overwrite strategy considers w0, w1, and w2 to produce the resulting world, w3. By applying the overwrite strategy, w3 preserves the effects made to the red and green fields but not ones made to the blue field; that is, `i` overwrites the effects of `e`.

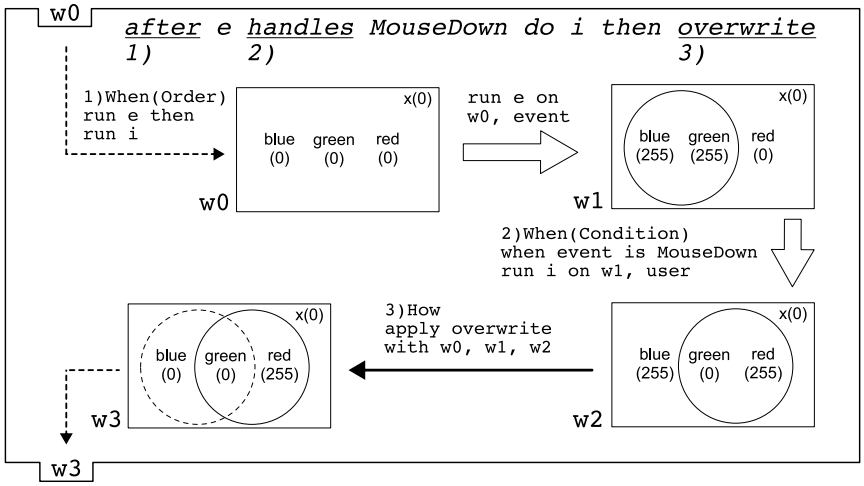

Fig. 5. The run-time behavior of the composed interaction shown in Figure 2 that employs the overwrite strategy instead of the merge.

3.4 Complex Compositions

All the composed interactions presented so far hook a single implicit interaction into a single explicit one. Recall, meanwhile, that composed interactions are also explicit interactions at the type level because a composed interaction responds to explicit user events, in which the computational effects of implicit interaction are already fused into it. It means that programmers can further hook another implicit interaction into a composed one, allowing them to express even more complex compositions.

For facilitating this, the Hook language prepares the three keywords—`and`, `it`, and `or`. The following code fragments exhibit example uses of the keywords. (As a side note, the `do-nothing` keyword, replacing a world transformer, is useful when programmers think of complex compositions. Programmers can use the keyword when they want to make the pattern matching exhaustive, for example, by adding a wildcard pattern, but do not want that case to be considered in the process of applying the fusion strategy.)

```
e1 = explicit w where
       MouseDown -> do {...}
     | MouseUp   -> do {...}

e2 = explicit w where
       (StdIn words) -> do {...}

e3 = (e1 or e2)

c4 = ((after e3 handles MouseDown do i1 then merge)
       and (after it handles MouseUp do i2 then tweak))

c5 = (((after (e1 or e2) handles MouseDown do i1 then merge)
        and (after it handles MouseUp do i2 then tweak))
             and (before it handles (StdIn words) do i1 then overwrite))
```

```
i1 = implicit w where
       (Happy True) -> do {...}
     | _            -> do-nothing

i2 = implicit w) where
       (Happy False) -> do {...}
     | _             -> do-nothing
```

The `or` keyword combines two explicit interactions and produces a single explicit interaction that unions all the case matches from the two interactions. For example, `e3` is an explicit interaction that has all the three case matches from `e1` and `e2`. When using the `or` keyword, the programmer is responsible for ensuring no patterns are redundant. Yet, the order of the case matches is taken care of by the Hook language compiler so that a more specific pattern matches first.

The `and` and `it` keywords are always used together and declare when and how another implicit interaction runs with the composed interaction. The `and` keyword follows the composed interaction that is the base of the newly composed interaction, plus the `it` keyword refers to the base interaction. At the same time, the `and` keyword is followed by the definition hooking another implicit interaction into the interaction that the `it` keyword is referring to. For example, `c4` is a newly composed interaction hooking `i2` into a composed interaction that hooks `i1` into `e3`. Furthermore, because the newly composed interaction is also explicit interaction at the type level, the programmer can further hook another implicit interaction into it. For instance, `c5` is another newly composed interaction hooking `i1` into a composed interaction, which is `c4` equivalent, that hooks `i2` into a composed interaction that hooks `i1` into `(e1 or e2)`, which is `e3` equivalent. In this manner, programmers can continue hooking as many implicit interactions as needed.

3.5 Implementation Considerations

This section lists some implementation details of the current Hook language compiler, which will be essential and particularly useful to programmers.

Capture-avoiding substitution for scoping. When bringing host language expressions and patterns into the same scope, the Hook language compiler applies capture-avoiding substitution. The algorithm walks over the pattern algebraic data type provided by Template Haskell to collect all the variables bound by the pattern and then renames them in both the pattern and the expression, except when sub-patterns shadow them.

Overlaps in patterns. As seen in Composing Interactions and Complex Compositions sections, a composed interaction could involve cases where more strict matches are specified than any matches available and cases where more than one implicit interactions are hooked with potential overlaps in the patterns. To find whether a pattern shadows another, the Hook language compiler checks the property using the syntactic structure of the pattern with no type information for most cases. For example, a variable or a wildcard matches anything, a literal matches on `Eq` equality, and

everything else matches on inductive structure or name equality. When finding overlaps, the Hook language compiler sorts the patterns so that the most specific pattern matches first in the original declaration order.

Retrieving user status information. Most modern eco-systems already provide high-level descriptions for low-level hardware events, so that programmers can deal directly with events (e.g., mouse clicks) instead of constantly checking a particular part of the computer memory to find the current state of the hardware (e.g., a mouse). While the programmer can leverage the tokens of events provided by the eco-system of their choice as the event patterns in the Hook language, the programmer is responsible for defining their own representation of the user status information and a context-specific way to retrieve that information. To be specific, using the vocabularies from the `Interaction` type, the Hook language compiler expects the programmer provides a function of type `m user`, written in the host language, which returns a user status pattern of type `user` when called. Note that `m` will involve `IO` in most cases to accommodate side-effects.

Trackable. The `Interaction` type makes the Hook language agnostic to the types of events, user status, and worlds. At the same time, the Hook language compiler enforces the programmer to make their world type to be an instance of the `Trackable` typeclass[2], so it can computationally keep track of the field-level changes made to any type of world. The following code fragment is the declaration of the `Trackable` typeclass in the host language and is provided as a part of the Hook language run-time libraries.

```
class Trackable world where
  type Diff world
  delta       :: world -> world -> Diff world
  appDiff     :: Diff world -> world -> world
  undoSubDiff :: Diff world -> Diff world -> world -> world
```

The `Trackable` typeclass will provide an interface to specify how a world can change. To make a world type an instance of the `Trackable` typeclass, the programmer defines the representation of the difference between two worlds, `Diff world`, and three methods: (1) `delta` for how to take the difference between two worlds, (2) `appDiff` for how to apply a difference to a world, and (3) `undoSubDiff` for how to undo the set subtraction of two differences on a world. In fact, the fusion strategies are defined in terms of these three methods. (See Δ and `fuse` in Computational Model section for details.) Note that `Trackable` allows the programmer to define whether a field of the world is considered "changed" in case the field is overwritten with the same value. Note also that effects made outside the world (e.g., writing logs to a file) cannot be undone by `undoSubDiff`.

4 COMPUTATIONAL MODEL

In this section, we formalize a high-level description of the Hook language as a computational model. The model consists of mathematical notations, abstract syntax, and semantics, describing the translation rules that the Hook language compiler employs. We aim the model to enable arbitrary host languages, which meet some preconditions, to implement the Hook language compiler as well as to be helpful means when extending the language in the future. Note that the computational model itself does not specify variable scoping, side-effects, application context, nor run-time system dependencies. Therefore, those details should be implemented based on idioms of the host language chosen.

[2]Typeclasses are a mechanism in Haskell that enables ad-hoc polymorphism.

4.1 Mathematical Notations

We shall begin by showing the mathematical notations of the computational model. Figure 7 summarizes the notations, and the details are as follows.

$$
\begin{aligned}
\text{(pattern) } p &::= x \mid \_ \mid \mathit{all}\ \vec{p} \\
\text{(world transformer) } t &::= \lambda p_w.w \mid f \\
\text{(world) } w &::= t(w) \mid \Delta \cdot w \mid x_w \\
\text{(delta) } \Delta &::= w - w \mid \Delta - \Delta \mid \Delta_\emptyset \\
\text{(associated transformer) } m &::= (p_e, p_u) \to t \\
\text{(model) } M &::= \vec{m}
\end{aligned}
$$

Fig. 7. Mathematical notations of the computational model.

(1) $(w_2 - w_1) \cdot w_1 = w_2$
(2) $w - w = \Delta_\emptyset$
(3) $\Delta - \Delta = \Delta_\emptyset$
(4) $\Delta_\emptyset \cdot w = w$
(5) $\Delta \cdot (\Delta \cdot w) = \Delta \cdot w$

Fig. 8. The algebraic laws for Δ.

Patterns. Let p represent a language of patterns that, at minimum, includes variable bindings x, a wildcard pattern _ (underscore), and an inductively defined pattern $\mathit{all}\ \vec{p}$ for matching the same expression with multiple sub-patterns. As a convenience, subscripts indicate particular sets of patterns, such that p_e for events, p_u for user status, and p_w for world states.

World transformers. Let t represent world transformers and have the abstraction form $\lambda p_w.w$. t may include other arbitrary host language functions, notated f, that modify the world. While allowing f, t abstracts all host language expressions—for example, arithmetic, if-then-else statements, record operations, and so on—away from uses of events and user status information.

Worlds. Let w represent world expressions and have the following forms: world transformation function application $t(w)$, applying and undoing of changes $\Delta \cdot w$, and variable references x_w.

Deltas. Let Δ represent changes in the world state that can be applied or undone. In addition, let them be constructed by taking the differences between two worlds or by removing changes between two Δs via set subtraction. For example, $w_1 - w_2$ notates the difference between w_1 and w_2. For example, $\Delta_1 - \Delta_2$ notates the parts of the world changed by Δ_1 not also changed by Δ_2. Lastly, when given $\Delta \cdot w_1 = w_2$, we read this as "applying Δ to w_1 gives us w_2" and "undoing Δ from w_2 gives us w_1."

To restrict the use of Δ, the following algebraic laws describe how this abstraction behaves when well-formed. Figure 8 lists the laws in a compact form, and the details are as follows.

(1) $(w_2 - w_1) \cdot w_1 = w_2$: Applying the change between two worlds to the first world should result in the second world.
(2) $w - w = \Delta_\emptyset$: Taking the difference between a world and itself should result in an empty delta, $\Delta_\emptyset$.
(3) $\Delta - \Delta = \Delta_\emptyset$: Taking the difference between a Δ and itself should also result in an $\Delta_\emptyset$.
(4) $\Delta_\emptyset \cdot w = w$: Applying or undoing the $\Delta_\emptyset$ with a world should result in that world.
(5) $\Delta \cdot (\Delta \cdot w) = \Delta \cdot w$: Applying or undoing a Δ should be idempotent.

To illustrate the use of w and Δ, it may be helpful to imagine them as records. For example, if w_1 = {red: 100, green: 50, blue: 0}, w_2 = {red: 50, green: 100, blue: 0}, then $w_1 - w_2$ = {red: (100, 50), green: (50, 100)}, which is a Δ. In this Δ, the value associated with each key of the world is a pair of old and new values. Then, set subtraction on Δ would be performed over the keys. For instance, if Δ_1 = {red: (100, 50), green: (50, 100)}, Δ_2 = {green: (100, 0)}, then $\Delta_1 - \Delta_2$ = {(red: (100, 50)}. We read this as "get the parts of the world changed via Δ_1 and remove any that were also changed via Δ_2."

Associated world transformers and Model. Let m represent a match in a case statement that binds event and user status patterns to a world transformer, $(p_e, p_u) \rightarrow t$. Then, the model comprises an ordered collection of m, represented as $\vec{m}$, in which the first match on an event and user status pattern short-circuits the rest. Note that the model omits how to obtain the user status information as a room to apply implementation specific optimizations.

4.2 Abstract Syntax and Semantics

Figure 9 shows the abstract syntax that describes the core expressions of programs written in the Hook language. The abstract syntax boils the grammar, presented in Figure 3, down to the three key expression forms: explicit expressions, implicit expressions, and hook expressions. The three forms directly correspond to their counterparts in the concrete syntax. (i.e., explicit, implicit, and composed interactions). Also, complex compositions and variable references can be statically rewritten in the three expressions. Below we shall show the processes to translate given Hook program code to the three key expressions and then to the computational model.

Exp ::= Explicit p_w ExplMatch+
| Implicit p_w ImplMatch+
| Hook Exp Exp When How

ExplMatch ::= MatchE p_e ($w \mid \epsilon$)
ImplMatch ::= MatchI p_u ($w \mid \epsilon$)

When ::= (Order, Condition)
Order ::= Before | After
Condition ::= Handles p_e | All
How ::= Merge | Overwrite | Tweak

Fig. 9. The abstract syntax representing the core expressions for programs written in the Hook language.

4.2.1 Explicit Expressions. Let us consider the code fragment below, which is written using the concrete syntax. The code fragment represents an explicit interaction, which, in this example, is a handler responding to mouse events. The `w` next to the `explicit` keyword is the variable for the world pattern introducing the identifier into scope. The `w`s in the `do` blocks are the forms of record updates in the host language functions on worlds, $f(w)$. Lastly, `MouseDown` and `MouseUp` are members of the event patterns, p_e.

```
explicit w where
  MouseDown -> do { return w { color = Red } }
| MouseUp   -> do { return w { color = Black } }
```

This code fragment is translated to the following explicit expression in the abstract syntax.

```
Explicit w [ MatchE MouseDown (do { return w { color = Red } })
           , MatchE MouseUp (do { return w { color = Black } }) ]
```

Then, this explicit expression is expectedly translated to the following $\vec{m}$.

```
{ (MouseDown, _) → λ w . w{color = Red}
, (MouseUp, _) → λ w . w{color = Black}}
```

In this translation process, $\vec{m}$ preserves the order of the abstract matches, MatchE, shown in the explicit expression. Plus, for every abstract match MatchE, a world transformer, given the world pattern p_w, is associated with the p_e from the MatchE and the wildcard pattern (_) for p_u, since this expression does not depend on user status information. As in, this $\vec{m}$ represents a case statement that will first match on the `MouseDown` event and any user status information to produce a world transformer that sets the `color` to `Red`. If that pattern does not match, then it will try to match the `MouseUp` event and any user status information to produce a world transformer that sets the

`color` to `Black`. Note that the model leaves checking for exhaustive pattern matching as a host language implementation detail. Lastly, Figure 10 includes this translation semantics, labeled as explicit expression rule.

4.2.2 Implicit Expressions. Let us consider the code fragment below, which is written using the concrete syntax. The code fragment represents an implicit interaction, which, in this example, is a handler responding to the user status information that indicates whether the user is happy. The wildcard pattern indicates any other user status, which implies the user is not happy in this example. Lastly, `User { happy = True }` and the wildcard pattern are members of the user status patterns, p_u.

```
implicit w where
  User { happy = True } -> do { return w { color = Green } }
| _                     -> do { return w }
```

This code fragment is translated to the following implicit expression in the abstract syntax.

```
Implicit w [ MatchI (User { happy = True }) (do { return w { color = Green } })
           , MatchI _ (do { return w })]
```

Then, this implicit expression is translated to the following $\vec{m}$.

```
{ (_, User{happy = True}) ⟶ λ w . w{color = Green}
, (_, _) ⟶ λ w . w}
```

In this translation process, for every abstract match, MatchI, a world transformer, given the world pattern p_w, is associated with the p_u from the MatchI and the wildcard pattern (_) for p_e, since the implicit interaction does not bind to any particular event. Recall that unlike explicit interactions, implicit interactions by themselves do not specify when they will be triggered. However, the computational model omits to restrict what kinds of interactions can be run as an implementation specific detail. Lastly, Figure 10 includes this translation semantics, labeled as implicit expression rule.

4.2.3 Hook Expressions. A hook expression is a composition of an explicit expression with an implicit one, which results in a new explicit expression. Let us consider the code fragment below. In this example, the code between `after` and `do` is the explicit interaction, and the one between `do` and `then` is the implicit interaction. This code fragment discloses a composed interaction that internally performs the following: "`after` running the explicit interaction, (`do`) run the implicit interaction, and `then` apply the `tweak` fusion strategy to glue the results of computations."

```
after ( explicit w where
          MouseDown -> do { return w { color = Red, brightness = Max } }
        | MouseUp   -> do { return w { color = Black } } )
do    ( implicit w where
          User { happy = False } -> do { return w { brightness = Min } }
        | _                      -> do { return w } )
then tweak
```

The code fragment is translated to the following hook expression in the abstract syntax.

```
Hook (Explicit w [ MatchE MouseDown (do{ return w{color = Red, brightness = Max} })
                 , MatchE MouseUp (do{ return w{color = Black} })])
     (Implicit w [ MatchI (User{happy = False}) (do{ return w{brightness = Min} })
                 , MatchI _ (do { return w })])
     (After, All) Tweak
```

Enforced by the concrete syntax, hook expression assumes the first sub-expression to be an explicit expression and the second sub-expression to be an implicit expression. (After, All) is a form of (Order, Condition), which declares *when* the implicit sub-expression should be evaluated concerning the explicit sub-expression. The Order abstract syntax specifies whether the implicit sub-expression happens Before or After the explicit sub-expression does whereas the Condition abstract syntax narrows down the occurrence based on event patterns in the explicit sub-expression. The Handles p_e condition binds every user status pattern in the implicit sub-expression with all the event patterns in the explicit one that match the specified p_e. On the other hand, the All condition, used in this example, binds every user status pattern with every event pattern (i.e., like a cross product). Lastly, Tweak is a form of the How abstract syntax, which specifies what changes to the world (application state) get persisted in terms of the Δ algebra.

Then, this hook expression is translated to the following $\vec{m}$. For readability, assume that the Δ algebra includes a let form to enable undoing changes, meaning that `w'` is the resulting world.

$$
\begin{aligned}
&\{\ (\texttt{MouseDown}, \texttt{User\{happy = False\}}) \longrightarrow \lambda\,\texttt{w}\,.\ \text{let}\ ((t_3(t_1(\texttt{w}))\text{-}t_1(\texttt{w}))\text{-}(t_1(\texttt{w})\text{-}\texttt{w})) \cdot \texttt{w'} = t_3(t_1(\texttt{w}))\ \text{in}\ \texttt{w'}\\
&,\ (\texttt{MouseDown}, \_) \longrightarrow \lambda\,\texttt{w}\,.\ \text{let}\ ((t_4(t_1(\texttt{w}))\text{-}t_1(\texttt{w}))\text{-}(t_1(\texttt{w})\text{-}\texttt{w})) \cdot \texttt{w'} = t_4(t_1(\texttt{w}))\ \text{in}\ \texttt{w'}\\
&,\ (\texttt{MouseUp}, \texttt{User\{happy = False\}}) \longrightarrow \lambda\,\texttt{w}\,.\ \text{let}\ ((t_3(t_2(\texttt{w}))\text{-}t_2(\texttt{w}))\text{-}(t_2(\texttt{w})\text{-}\texttt{w})) \cdot \texttt{w'} = t_3(t_2(\texttt{w}))\ \text{in}\ \texttt{w'}\\
&,\ (\texttt{MouseUp}, \_) \longrightarrow \lambda\,\texttt{w}\,.\ \text{let}\ ((t_4(t_2(\texttt{w}))\text{-}t_2(\texttt{w}))\text{-}(t_2(\texttt{w})\text{-}\texttt{w})) \cdot \texttt{w'} = t_4(t_2(\texttt{w}))\ \text{in}\ \texttt{w'}\}
\end{aligned}
$$

where

t_1: λ `w . w{color = Red, brightness = Max}`
t_2: λ `w . w{color = Black}`
t_3: λ `w . w{brightness = Min}`
t_4: λ `w . w`

In this translation process, each of the explicit and implicit sub-expressions first translates to their respective model, $\vec{m_{expl}}$ and $\vec{m_{impl}}$. Then, the translator would bind every match in $\vec{m_{impl}}$ to every match in $\vec{m_{expl}}$, which results in the final $\vec{m}$ that comprises four matches in this example. Note that the matches are sorted, so the most specific matches apply first. As for the world transformation part, in every Before composition, the world transformer from the implicit sub-expression t_{impl} runs on the given world first. Similarly in every After composition, the world transformer from the explicit sub-expression t_{expl} runs on the given world first. In both cases, the resulting world is then given to the other sub-expression's world transformer. In this example, the After abstract syntax enforced that t_3 and t_4 from $\vec{m_{impl}}$ always come after t_1 and t_2 from $\vec{m_{expl}}$. Note that in case m includes a pair of ϵ, corresponding to the `do-nothing` keyword, and non-ϵ, then the translator produces a world transformer that simply passes through the result of the non-ϵ world transformation. In case m includes a pair of two ϵs, then the translator produces an identity world transformer. Lastly, the composition using the Tweak fusion strategy applies the specified world transformers and then undoes the difference.

Figure 10 shows this translation semantics, labeled as hook expression rule, where the rule uses the following three constructs: `fuse`, `satisfy`, and `sort`.

(Explicit expression rule)

$$\frac{}{\texttt{Explicit}\ p_w\ [\ldots, \texttt{MatchE}\ p_e\ w, \ldots] \Rightarrow \{\ldots, (p_e, \_) \rightarrow \lambda\ p_w\ .\ w, \ldots\}}$$

(Implicit expression rule)

$$\frac{}{\texttt{Implicit}\ p_w\ [\ldots, \texttt{MatchI}\ p_u\ w, \ldots] \Rightarrow \{\ldots, (\_, p_u) \rightarrow \lambda\ p_w\ .\ w, \ldots\}}$$

(Hook expression rule)

$$\frac{\begin{array}{l}\texttt{explE} \Rightarrow \vec{m_{expl}} \\ \texttt{implE} \Rightarrow \vec{m_{impl}} \qquad \forall\ (p_{e_e}, p_{u_e}) \rightarrow t_{expl} \in \vec{m_{expl}}, (\_, p_{u_i}) \rightarrow t_{impl} \in \vec{m_{impl}}\end{array}}{\begin{array}{l}\texttt{Hook explE implE (order,condition) how} \Rightarrow \\ \quad \texttt{sort}(\{(p_{e_e}, \mathit{all}\ [p_{u_e}, p_{u_i}]) \rightarrow \texttt{fuse}(t_{expl}, t_{impl}, \texttt{order}, \texttt{how}) \mid \texttt{satisfy}(p_{e_e}, \texttt{condition})\} \\ \qquad \cup \{(p_{e_e}, p_{u_e}) \rightarrow t_{expl} \mid \neg\ \texttt{satisfy}(p_{e_e}, \texttt{condition})\})\end{array}}$$

Fig. 10. The translation semantics of the Hook language compiler.

Let `fuse(`t_{expl}`, `t_{impl}`, Order, How)` be a syntactic translation that, $\forall t_{expl}, t_{impl}$, it produces a t' such that:

Order	How	t'	where
any	any	$\lambda x.x$	$t_{expl} = \epsilon, t_{impl} = \epsilon$
		t_{expl}	$t_{expl} \neq \epsilon, t_{impl} = \epsilon$
		t_{impl}	$t_{expl} = \epsilon, t_{impl} \neq \epsilon$
Before	Merge	$\lambda x.\Delta_2 \cdot (\Delta_1 \cdot x)$	$\Delta_1 = t_{impl}(x) - x;$
Before	Overwrite	$\lambda x.\Delta_2 \cdot x$	$\Delta_2 = t_{expl}(t_{impl}(x)) - t_{impl}(x);$
Before	Tweak	$\lambda x.w'$	$(\Delta_2 - \Delta_1) \cdot w' = \Delta_2 \cdot (\Delta_1 \cdot x)$
After	Merge	$\lambda x.\Delta_2 \cdot (\Delta_1 \cdot x)$	$\Delta_1 = t_{expl}(x) - x;$
After	Overwrite	$\lambda x.\Delta_2 \cdot x$	$\Delta_2 = t_{impl}(t_{expl}(x)) - t_{expl}(x);$
After	Tweak	$\lambda x.w'$	$(\Delta_2 - \Delta_1) \cdot w' = \Delta_2 \cdot (\Delta_1 \cdot x)$

Let `satisfy(`p`, Condition)` be a predicate, such that it evaluates to true when Condition is All and it tests $p' \subseteq p$ when Condition is Handles p'. The pattern comparison operator $\subseteq$ is defined as $\forall p1, p2.p1 \subseteq p2$ iff $\forall e.p1$ matches $e \rightarrow p2$ matches e, and if two patterns are unrelated (i.e., the set of expressions they match are disjoint) then both $\neg p_1 \subseteq p_2$ and $\neg p_2 \subseteq p_1$ will be true.

Lastly, assume that p below is (p_e, p_u) from m, and then let `sort(`$\vec{m}$`)` be a function which sorts $\vec{m}$ and produces $\vec{m}'$ such that:

$\vec{m}$	$\vec{m}'$	where
[]	[]	
$[p]$	$[p]$	
$[p_1, p_2, \ldots]$	$[p_1, \texttt{sort}([p_2, \ldots])]$	$\neg(p_2 \subseteq p_1)$
$[p_1, p_2, \ldots]$	$\texttt{sort}([p_2, \texttt{sort}([p_1, \ldots])])$	$p_2 \subseteq p_1$

5 EVALUATION

As a first suggestive form of validation, we present three case studies that demonstrate some of the language features, usage, and capabilities. The goal of the case studies is twofold: (1) demonstrate our ability to express explicit and implicit interactions and their compositions using the Hook language; and (2) show that the current Hook language compiler, working together with the host language compiler, can produce a runnable program. In each case study, we used the Hook language to replicate a published interactive system and implement an implicit interaction that extends its original behavior. To highlight our motivation for studying "Implicit Human-Computer Interactions" [48] in our case studies, we designed the implicit interactions so that the resulting interactive system—i.e., computer—ought to be more considerate of its user in each case.

To implement Graphical User Interface (GUI) applications in functional programming settings, we leveraged the eco-system that the gloss Haskell library[3] provides. In this eco-system, programmers are responsible for implementing how to render graphics on a screen and how to handle conventional input events (e.g., keyboard, mouse events) in addition to periodic timer events. For space, the following case studies will focus on presenting interactions handling events and user status information.

To retrieve user status information, we implemented a backend server program that emulates an oracle system providing the current status of the user at any moment. (e.g., How happy the user is now, etc.) The backend server program ran as a different process from the frontend GUI application written in the Hook language. When the frontend application needed to have access to user status information, it communicated with the backend server by exchanging strings in the JSON format over the TCP/IP connection. The procedure of these message exchanges follows the form of "implicit dialogue" [53] and was implemented as a series of `IO` actions in the host language, Haskell.

5.1 Case Study 1: Target Acquisitions

GUI applications contain many target acquisition tasks. People use a mouse cursor to select file icons, to respond to a dialogue by selecting yes/no buttons, and so on. For efficiency, Grossman and Balakrishnan invented Bubble Cursor [14], which is an advanced target acquisition technique that updates the cursor's selection area dynamically based on the arrangement of target objects.

We shall begin by implementing a version of Bubble Cursor as an explicit interaction. Figure 11 shows the code fragment for this case study, and the explicit interaction is bound to `e`. The `World` data type contains two circle objects to be selected, and each of them consists of a set of `x` and `y` coordinates on screen, a radius, and a flag indicating whether or not it is currently selected. As seen in `e`, when the user moves the mouse, an `EventMotion` event is triggered, and the current selection of target gets updated using the `selectTarget` function. (Note that in our simplified version, the mouse cursor always selects the closest target.)

Next, we shall extend this behavior by taking user status information into account. Following the idea presented by Afergan, et al. [1], let us assume a scenario where one of the target objects has higher priority than another, and the programmer wants to make it easier for the user to select the prioritized target based on the user's current busyness. (Note that `busyness` is ranged from 0.0 to 100.0; higher is busier.) The implicit interaction bound to `i` encodes this behavior. As the user becomes busier, the size of the prioritized target becomes at most twice as large as its original size. Thus, the cursor is more likely to select the prioritized target when the user is busier, which could appear to be more considerate, compared to the original behavior.

[3] https://hackage.haskell.org/package/gloss

Suppose the programmer only wishes to apply the effects of the implicit interaction `i` during the target selection procedure of the explicit interaction `e` but not during the rendering of the target objects. The composed interaction `c` achieves this goal. `c` declares that 1) before running `e` which handles `EventMotion` events, run `i`; and 2) the effects of `e`'s world transformation overwrite those of `i`'s. Since `i` modifies `r1`, which is the radius of the prioritized target, `e` refers to the modified target size when selecting a target. Then, the overwrite fusion strategy produces the resulting world by taking the result of `e`, which only changed `isSelected1` and `isSelected2` but not `r1`. Therefore, both targets are rendered in their original sizes. Figure 12 shows the composed interaction `c` in action.

To sum up, this case study demonstrated a seamless integration between the Hook language and host language code by calling the host language function from the explicit interaction. This case study also demonstrated that a composed interaction can express a staging effect where an implicit interaction prepares the world for the explicit interaction.

```
data World = World { x1 :: Float, y1 :: Float, r1 :: Float, isSelected1 :: Bool
                   , x2 :: Float, y2 :: Float, r2 :: Float, isSelected2 :: Bool
                   , ... }
data User = User { busyness :: Float }

[hook|
  e = explicit world where
        (EventMotion (x, y)) -> do { returnIO $ selectTarget x y world }
      | _                    -> do { returnIO world }

  i = implicit world where
        User { busyness = b } -> do { let r = (r1 world) * (1.0 + b / 100.0)
                                      in returnIO world { r1 = r }
                                    }
  c = (before e
       handles (EventMotion (x, y))
       do i
       then overwrite)
|]

selectTarget :: Float -> Float -> World -> World
selectTarget x y w =
  let d1 = distance (x, y) (x1 w, y1 w) - r1 w
      d2 = distance (x, y) (x2 w, y2 w) - r2 w
   in if d1 < d2
        then w { isSelected1 = True,  isSelected2 = False }
        else w { isSelected1 = False, isSelected2 = True }

--distance :: (Float, Float) -> (Float, Float) -> Float
--returnIO :: World -> IO World
```

Fig. 11. A version of Bubble Cursor + a staging effect written in the Hook language.

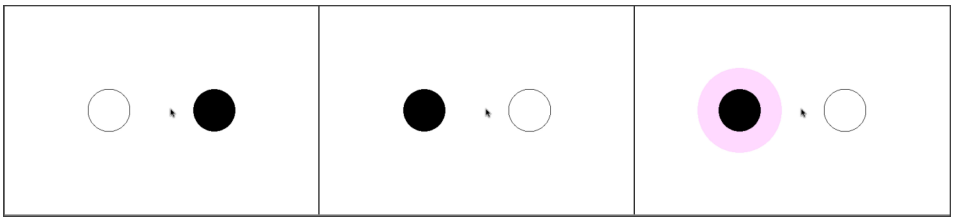

Fig. 12. A version of Bubble Cursor + the staging effect in action. Assume the left circle is prioritized. When selected, the target is rendered as a solid circle. Left: User `busyness` is 0.0, which resulted in keeping the size of the prioritized target its original size. Thus, the cursor selects the right target which is closer. Middle: User `busyness` is 100.0, which resulted in doubled the size of the higher priority target. Thus, the cursor now selects the left target from the same location as before. Right: (supplemental figure) The pink circle shows the size of the prioritized target expanded by the fact that user `busyness` is 100.0.

5.2 Case Study 2: Readability Enhancement

People read articles on computer screens everyday, but non-native readers seem to be intimidated when encountering articles that ". . . include unfamiliar vocabulary, complicated grammatical structure, and long, crowded or otherwise intimidating content display" [61]. To that end, Yu and Miller invented Jenga Format [61], which transforms the sentence layout to enhance web page readability.

We shall implement a feature of Jenga Format as an explicit interaction, which allows the reader to change the space between sentences using a special slider. Figure 13 shows the code fragment for this case study, and the explicit interaction is bound to `e`. The `World` data type contains the sentences to be displayed, layout parameters, and other application state. As seen in `e`, when the user moves the slider to increase or decrease the space between sentences, the `HMoveKnob` event matches and updates the `gap` field.

Next, we shall extend this behavior by taking user status information into account. Let us assume a scenario where the programmer thinks that when the user is frustrated, increasing the space between sentences even more and enlarging characters might further improve the readability. The implicit interaction, bound to `i`, encodes this behavior. As the user is frustrated, the space between sentences (`gap`) and the character scale (`scale`) gets doubled, which could appear to be more considerate, compared to the original behavior.

Suppose the programmer—after implementing `i`—noticed that the user of this GUI application only expects the space between sentences (`gap`) to change when using the slider. The composed interaction `c` can respect this desire. `c` declares that 1) after `e` handles `HMoveKnob`, run `i` and 2) the effects of `i`'s world transformation tweak `e`'s. Since `e` only modifies `gap` and the position of the knob, the tweak fusion strategy produces the resulting world by applying `i`'s doubling of the gap size from `e`, but not taking the `scale` changes from `i`. So, the user of this GUI application does not see the character scale changes when they did not expect it to be. Figure 14 shows the composed interaction `c` in action.

While the first case study showed a staging effect, this case study demonstrated that a composed interaction can express a decorating effect that an implicit interaction only modifies the parts of the world that an explicit interaction modified.

```
data World = World { ... (sentences, graphic information for rendering) ...
                   , gap :: Float , scale :: Float
                   , isGrabbingKnob :: Bool, knobX :: Float }
data User = Busy | Happy | Frustrated | Unspecified

[hook|
  e = explicit world where
        HGrabKnob -> do { returnIO world { isGrabbingKnob = True } }
      | HReleaseKnob -> do { returnIO world { isGrabbingKnob = False } }
      | (HMoveKnob x) -> do { let (knobXMin, knobXMax) = (0.0, (sliderWidth world))
                            ;     (gapMin, gapMax) = (0.0, 40.0)
                            ;     ((topLeftX, _), _) = locateSlider world
                            ;     knobX' = x - topLeftX
                            ;     gap' = gapMin + (knobX' - knobXMin) *
                                          (gapMax - gapMin) / (knobXMax - knobXMin)
                               in returnIO world { gap = gap', knobX = knobX' }
                            }
      | _ -> do { returnIO world }

  i = implicit world where
        Frustrated -> do { let gap' = (gap world) * 2.0
                         ;     scale' = (scale world) * 2.0
                            in returnIO world { gap = gap', scale = scale' }
                         }
      | _ -> do-nothing

  c = (after e handles (HMoveKnob x)
       do i then tweak)
|]

--locateSlider :: World -> ((Float, Float), (Float, Float))
--returnIO :: World -> IO World
```

Fig. 13. A feature from Jenga Format that changes the space (gap) between sentences + a decorating effect written in the Hook language.

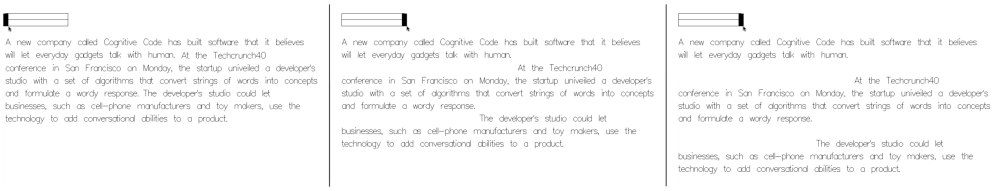

Fig. 14. A feature from Jenga Format that changes the space (gap) between sentences + the decorating effect in action. Left: The normal space (gap); the slider knob is at leftmost. Middle: As the slider knob was moved rightward, the space (gap) is increased. Right: The knob was moved to right with the same amount, but the space (gap) was increased more compared to before, since the user is frustrated. (Note: The sentences in this figure were taken from [61].)

5.3 Case Study 3: No explicit user events

In the previous case studies, the source of events to the explicit interactions came directly from the user's action. This case study, on the other hand, considers a different scenario where no explicit user event is expected, yet the programmer can still use the Hook language to fuse implicit interactions to programmatic event handlers, such as a periodic timer. For example, expressing implicit interactions in relation to a periodic timer makes it possible for the programmer to detect changes in the user status and trigger specific actions accordingly.

Below we consider Zero Shutter Camera [54] as an example and use the Hook language to replicate its behavior—i.e., the application automatically takes a picture for the user when the user became busy. Figure 15 shows the code fragment for this case study. We first prepare the `stepHandler` function, in the host language, which responds to periodic timer events. The function simply consumes timer events and does not change the world in this example. Note that we used `liftE`, a function from the Hook language run-time libraries, to treat the `stepHandler` function as the explicit interaction and then bound it to `e`. Then, we encode the behavior of Zero Shutter Camera as the implicit interaction `i`. `i` uses the host language code that presumably takes a picture when the user status switches from not busy to busy and then saves the file location of the saved picture to the `filePath` field. Finally, we declare `c` that hooks `i` into `e` and simply merges the effects since `e` does not change the world.

```
data World = World { filePath :: FilePath }
data User = User { busy :: Bool }

stepHandler :: Float -> World -> IO World
stepHandler _ = return

e = liftE stepHandler

[hook|
  i = implicit world where
        User { busy = True } -> do { if (filePath world) == ""
                                       then do filePath' <- takePicture
                                               returnIO world { filePath = filePath' }
                                       else returnIO world
                                   }
      | User { busy = False } -> do { if (filePath world) /= ""
                                        then returnIO world { filePath = "" }
                                        else returnIO world
                                    }
  c = (after e do i then merge)
|]

--takePicture :: IO FilePath
--returnIO :: World -> IO World
```

Fig. 15. Zero Shutter Camera mock-up written in the Hook language.

6 DISCUSSION AND FUTURE WORK

Overall, our case studies have demonstrated that the Hook language is capable of independently expressing the explicit interactions of the existing interactive systems and the implicit interactions for augmenting the behaviors of the systems. Also, because of the fusion strategies and the declarative manner in expressing composed interactions, the programmer was able to specify the interference between the explicit and implicit interactions in their implementation without ambiguity. Thus, we argue that programmers using the Hook language can achieve separation of concerns without giving up algorithmic precision in implementing the two types of interactions and their compositions.

6.1 Design Choices

Agnostic. We designed the Hook language to be agnostic to the events, user status, and world states so it can express as many interactive systems as possible. In the meantime, this design choice also contributed to making the language flexible enough to handle more event patterns than ones the eco-system provides. For example, we observed in Case Study 2 that the language was able to handle a custom application-specific event pattern (`HMoveKnob x`) defined on top of the generic event pattern (`EventMotion (x, y)`) that the gloss Haskell library provides. This observation implies that finely grinded custom patterns encourage programmers to break down their world transformers into even smaller and modular pieces. As a result, programmers will be able to declare their composed interactions in even finer granularity. Plus, the observation also implies that the concept of the Hook language could apply to more domains than GUI—e.g., Augmented Reality (AR), Virtual Reality (VR) system, and so on, as long as programmers can map event-equivalent tokens in the system to their own event patterns and program states of the system to their own world pattern. Meanwhile, there should be no new restrictions on defining the user status pattern.

Fusion strategies. From the Venn diagram illustrations shown in Figure 4 and the Δ algebra formalism shown in Computational Model section, we discovered eight possible fusion strategies of which we picked three because the other five seemed to be detrimental. For example, there exists a fusion strategy that keeps all changes other than ones made to fields belonging to G2—algebraically represented as $\lambda x_w.(\Delta_1 - \Delta_2) \cdot ((\Delta_2 - \Delta_1) \cdot x_w)$. However, this strategy—unlike the merge, overwrite, and tweak strategies—is not compelling for explaining the interference of the two interactions in human language. Therefore, we decided to provide only fusion strategies that come with a compelling analogy, which, we believe, reduces the chances for programmers implementing unintended interactive behaviors.

6.2 Limitations

The following limitations, currently restricting the set of programs that can benefit from being implemented in the Hook language, suggest directions for future work on improving the design of the language.

World state. The Hook language compiler asks programmers to make their world type to be an instance of the `Trackable` typeclass, and the current language design trades this requirement for gaining the flexibility of the world type. However, there are indeed commonly used data structures, such as inductive data structures (e.g., array, list, tree, etc.), that require more effort from programmers to satisfy this requirement. Future work should consider providing a metaprogram as a part of the Hook language run-time libraries that generates the corresponding `Diff` type and `Trackable` instance with the programmers' world type programmatically.

Pattern matching. The current Hook language compiler does not prevent explicit and implicit interactions from having redundant or non-exhaustive pattern matching. This is because the patterns are currently defined in the host language. To perform this check at compile-time, future work should consider extending the current language grammar to provide syntax for defining events, user status, and world patterns directly in the Hook language.

Static analysis. Compilers ought to generate correct programs from correct code ("build the thing right"). At the same time, compilers should also support programmers for writing correct code from their desired logic ("build the right thing"). The current Hook language compiler, in cooperation with the type system of Haskell, provides a lightweight static analysis. For example, it checks the `kind` of every interaction and ensures that only implicit interactions are hooked into explicit interactions, but not the other way around. Future work should enhance the static analysis as well as implement its own type system specific to the Hook language. It would be desirable for the type system of the Hook language to enable a compile-time check, such that it can locate which interaction within composed interactions makes changes that overlap with others. This check would be especially worthwhile when programmers define complex compositions.

Optimizations. In the example shown in Figure 1, the programmer was able to use their intuition to remove unnecessary procedures from the implementation of `r`. For instance, the programmer already merged the logic of `p` and `q` in their mind; therefore, the programmer was able to directly substitute 0, the result of `q`, to the `green` field and simply ignored 255, the result of `p`. Meanwhile, the current Hook language compiler does not perform any domain-specific optimizations, expect it delegates to the lazy semantics of Haskell to prevent evaluating unnecessary expressions in the generated host language code. Future work should consider employing "defunctionalization" [45] to detect and eliminate redundant expressions, possibly side-effects as well, at compile-time while the resulting code still follows the Δ algebraic laws at run-time.

Abstractions. The host language of our choice, Haskell, influenced the current language design in terms of the relationships between patterns and world transformers. There are indeed other programming languages that could satisfy preconditions for implementing the Hook language compiler. Future work should consider exploring other host language implementations, especially those commonly used for implementing interactive applications, to extend the current language design with abstractions inspired by those implementations.

User studies. Lastly, future work should consider conducting a user study to evaluate the current design of the language with programmers from various backgrounds. It would be desirable that the user study finds which groups of programmers based on their skill sets can benefit from the language as well as the economic acceptance of the language. In addition, we hope that user studies motivate researchers to explore what kind of implicit interactions that computers initiate can bring benefits to humans. For this, the Hook language is ready to handle more complex, sophisticated user status patterns than the simple ones shown in this paper.

6.3 Implications for Interaction Algebra

This paper showed the two types of interactions, explicit and implicit, are composable. Taking a step forward, we now touch on the feasibility of defining algebraic specifications for "interaction" beyond the semantics of the Hook language. In concept, the Hook language lifts the event handler and user status handler logics to the `Interaction` type so that programmers can treat them as composable units at the type level. Inspired by Polaris [58] introducing Table Algebra, the following is our early attempt to explain "interaction composition" found in the Hook language, as a part of Interaction Algebra, using the plus (+) operator.

In the Hook language, interaction is an ordered set of pairs of a pattern and world transformer. Therefore, using the vocabularies from Computational Model section, let e be an instance of explicit interaction that has the form s.t. $e = \{(p_{e1}, t_{e1}), ..., (p_{en}, t_{en})\}$ and let i be an instance of implicit interaction that has the form s.t. $i = \{(p_{u1}, t_{u1}), ..., (p_{um}, t_{um})\}$. For space, let us consider an example case where each of e and i associates with two patterns; that is, $e = \{(p_{e1}, t_{e1}), (p_{e2}, t_{e2})\}$ and $i = \{(p_{u1}, t_{u1}), (p_{u2}, t_{u2})\}$.

To explain the application order of the two interactions in using the plus operator, let us define that the left hand side of the operator is the interaction that internally runs first. Then, the interaction compositions, $e + i$ and $i + e$, will be defined as follows, where the plus operator for world transformers could be defined on top of the Δ algebraic laws and function composition because world transformers are apparently functions. For example, in $((p_{e1}, p_{u1}), (t_{e1}{+}t_{u1}))$, we could see $(t_{e1}{+}t_{u1})$ as $\lambda x_w.t_{u1}(t_{e1}(x_w))$ and (p_{e1}, p_{u1}) as a lexical environment that the λ has access to.

$$\begin{aligned} e + i = \{&((p_{e1}, p_{u1}), (t_{e1}{+}t_{u1})), ((p_{e1}, p_{u2}), (t_{e1}{+}t_{u2})) \\ &, ((p_{e2}, p_{u1}), (t_{e2}{+}t_{u1})), ((p_{e2}, p_{u2}), (t_{e2}{+}t_{u2}))\} \\ i + e = \{&((p_{e1}, p_{u1}), (t_{u1}{+}t_{e1})), ((p_{e1}, p_{u2}), (t_{u2}{+}t_{e1})) \\ &, ((p_{e2}, p_{u1}), (t_{u1}{+}t_{e2})), ((p_{e2}, p_{u2}), (t_{u2}{+}t_{e2}))\} \end{aligned}$$

At the same time, $e_x + e_y$ and $i_x + i_y$ will be defined by simply using the set union.
To specify an event pattern in e, let us leverage subscript square brackets ($_{[]}$), assume $p_{es} \subseteq p_{e1}$, and then define the interaction composition as follows.

$$\begin{aligned} e_{[p_{e1}]} + i &= \{((p_{e1}, p_{u1}), (t_{e1}{+}t_{u1})), ((p_{e1}, p_{u2}), (t_{e1}{+}t_{u2})), (p_{e2}, t_{e2})\} \\ i + e_{[p_{e2}]} &= \{(p_{e1}, t_{e1}), ((p_{e2}, p_{u1}), (t_{u1}{+}t_{e2})), ((p_{e2}, p_{u2}), (t_{u2}{+}t_{e2}))\} \\ e_{[p_{es}]} + i &= \{((p_{es}, p_{u1}), (t_{e1}{+}t_{u1})), ((p_{es}, p_{u2}), (t_{e1}{+}t_{u2})), (p_{e1}, t_{e1}), (p_{e2}, t_{e2})\} \end{aligned}$$

Similarly, complex compositions will be defined as follows.

$$\begin{aligned} (e_{[p_{e1}]} + i)_{[p_{e2}]} + i = \{&((p_{e1}, p_{u1}), (t_{e1}{+}t_{u1})), ((p_{e1}, p_{u2}), (t_{e1}{+}t_{u2})) \\ &, ((p_{e2}, p_{u1}), (t_{e2}{+}t_{u1})), ((p_{e2}, p_{u2}), (t_{e2}{+}t_{u2}))\} \\ i + (e_{[p_{e1}]} + i)_{[p_{e2}]} = \{&((p_{e1}, p_{u1}), (t_{e1}{+}t_{u1})), ((p_{e1}, p_{u2}), (t_{e1}{+}t_{u2})) \\ &, ((p_{e2}, p_{u1}), (t_{u1}{+}t_{e2})), ((p_{e2}, p_{u2}), (t_{u2}{+}t_{e2}))\} \end{aligned}$$

Lastly, to specify a fusion strategy, let us decorate[4] the plus operator and define so that we use <+> (and + as well) for merge, +> for overwrite, and <+ for tweak, where missing < or > indicates throwing away changes made solely by the world transformer placed at that side.

$$\begin{aligned} e_{[p_{e1}]} \texttt{<+>}\ i &= \{((p_{e1}, p_{u1}), (t_{e1}\texttt{<+>}t_{u1})), ((p_{e1}, p_{u2}), (t_{e1}\texttt{<+>}t_{u2})), (p_{e2}, t_{e2})\} \\ e_{[p_{e1}]} \quad \texttt{+>}\ i &= \{((p_{e1}, p_{u1}), (t_{e1} \ \texttt{+>}t_{u1})), ((p_{e1}, p_{u2}), (t_{e1} \ \texttt{+>}t_{u2})), (p_{e2}, t_{e2})\} \\ e_{[p_{e1}]} \texttt{<+} \quad i &= \{((p_{e1}, p_{u1}), (t_{e1}\texttt{<+} \ t_{u1})), ((p_{e1}, p_{u2}), (t_{e1}\texttt{<+} \ t_{u2})), (p_{e2}, t_{e2})\} \end{aligned}$$

We acknowledge that there is still room we need to fill in to define "interaction composition" in detail. (e.g., overlaps in patterns, etc.) Yet, we envision that introducing formalism brings two potential benefits in the field of Human-Computer Interaction. First, it may allow us to import theories and tools developed in the field of Programming Language researches. For example, since the presented computational model defines world transformers using the form of functions, we may be able to naturally apply and take advantage of existing tools that analyze properties of functions. Second, it may allow us to expand design spaces beyond the *kinds* of user interface designs. For example, there exist classes of user interfaces whose interactive behaviors involve explicit interactions. (e.g., GUI, Tangible User Interfaces (TUI) [22], and so on, although event-equivalent tokens, associated with "constraint" [51], in TUI may look different from ones in GUI.) Plus, there exist classes of user interfaces whose interactive behaviors involve implicit interactions.

[4]Like the winged star (<*>) operator for Applicative apply in Haskell.

(e.g., Passive Brain-Computer Interfaces (BCI) [63], Implicit BCI [56], Affective Computing [44], Attentive or Attentional User Interfaces (AUI) [21, 59], and so on.) Consequently, if we can extract explicit and implicit interactions from those user interface designs, then the algebraic specifications for "interaction" may allow us to design a combination of two *kinds* of user interfaces—e.g., (TUI + Implicit BCI)—and further compare two different combinations of user interfaces based on their complexity—e.g., (GUI + Implicit BCI) compared to (TUI + Passive BCI)—systematically, even without having their actual implementations.

7 CONCLUSION

As machine capabilities advance, opportunities to design implicit interactions will likely continue to grow beyond explicit ones. As a result, programmers using the current tools to implement combinations of the explicit and implicit interactions will face increasing modularity and clarity problems due to the growing complexity. In this paper, we have introduced the Hook language, wherein programmers using the language can treat explicit and implicit interactions independently and can specify their composition declaratively. The Hook language made this possible by providing the set of abstractions for describing the two types of interactions as parameterized world transformation functions plus the fusion strategies for gluing results of the two interactions. Our case studies demonstrated that the Hook language—agnostic to events, user status, and application states—could tackle a range of arbitrary (GUI) applications and that programmers could maintain modularity and clarity of their code when extending interactive behaviors of existing interactive systems with implicit interactions. Lastly, we discussed the design choices we made on achieving separation of concerns and algorithmic precision and the limitations in the current language implementation for future work, as well as the implications beyond the concept of the Hook language toward defining Interaction Algebra. We hope that the Hook language encourages programmers to take a step toward designing more implicit interactions and contributes to making interactive systems more aware of their users.

ACKNOWLEDGMENTS

We would like to thank Remco Chang, Jeff Foster, Kathleen Fisher from Tufts University, and Jones Yu from Wentworth Institute of Technology for their valuable insights into this work.

REFERENCES

[1] Daniel Afergan, Tomoki Shibata, Samuel W. Hincks, Evan M. Peck, Beste F. Yuksel, Remco Chang, and Robert J. K. Jacob. 2014. Brain-Based Target Expansion. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (Honolulu, Hawaii, USA) *(UIST '14)*. Association for Computing Machinery, New York, NY, USA, 583–593. https://doi.org/10.1145/2642918.2647414

[2] Alan Borning and Robert Duisberg. 1986. Constraint-Based Tools for Building User Interfaces. *ACM Trans. Graph.* 5, 4 (Oct. 1986), 345–374. https://doi.org/10.1145/27623.29354

[3] Andreas Bulling and Thorsten O. Zander. 2014. Cognition-Aware Computing. *IEEE Pervasive Computing* 13, 3 (2014), 80–83. https://doi.org/10.1109/MPRV.2014.42

[4] Daniel Buschek and Florian Alt. 2017. ProbUI: Generalising Touch Target Representations to Enable Declarative Gesture Definition for Probabilistic GUIs. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 4640–4653. https://doi.org/10.1145/3025453.3025502

[5] Fredy Cuenca, Jan Van den Bergh, Kris Luyten, and Karin Coninx. 2015. Hasselt UIMS: A Tool for Describing Multimodal Interactions with Composite Events. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (Duisburg, Germany) *(EICS '15)*. Association for Computing Machinery, New York, NY, USA, 226–229. https://doi.org/10.1145/2774225.2775437

[6] Evan Czaplicki. 2012. Elm: Concurrent frp for functional guis. (2012). https://elm-lang.org/assets/papers/concurrent-frp.pdf

[7] Edsger W. Dijkstra. 1982. *On the Role of Scientific Thought.* Springer New York, New York, NY, 60–66. https://doi.org/10.1007/978-1-4612-5695-3_12

[8] Bruno Dumas, Denis Lalanne, Dominique Guinard, Reto Koenig, and Rolf Ingold. 2008. Strengths and Weaknesses of Software Architectures for the Rapid Creation of Tangible and Multimodal Interfaces. In *Proceedings of the 2nd International Conference on Tangible and Embedded Interaction* (Bonn, Germany) *(TEI '08)*. Association for Computing Machinery, New York, NY, USA, 47–54. https://doi.org/10.1145/1347390.1347403

[9] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2009. A Functional I/O System or, Fun for Freshman Kids. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/1596550.1596561

[10] Kathleen Fisher, Nate Foster, David Walker, and Kenny Q. Zhu. 2011. Forest: A Language and Toolkit for Programming with Filestores. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 292–306. https://doi.org/10.1145/2034773.2034814

[11] Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-Specific Language for Processing Ad Hoc Data. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 295–304. https://doi.org/10.1145/1065010.1065046

[12] Mark A. Flecchia and R. Daniel Bergeron. 1986. Specifying Complex Dialogs in ALGAE. In *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface* (Toronto, Ontario, Canada) *(CHI '87)*. Association for Computing Machinery, New York, NY, USA, 229–234. https://doi.org/10.1145/29933.275635

[13] Laurent George and Anatole Lécuyer. 2010. An overview of research on "passive" brain-computer interfaces for implicit human-computer interaction. In *International Conference on Applied Bionics and Biomechanics ICABB 2010 - Workshop W1 "Brain-Computer Interfacing and Virtual Reality"*. Venise, Italy. https://hal.inria.fr/inria-00537211

[14] Tovi Grossman and Ravin Balakrishnan. 2005. The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Portland, Oregon, USA) *(CHI '05)*. Association for Computing Machinery, New York, NY, USA, 281–290. https://doi.org/10.1145/1054972.1055012

[15] David Harel. 1988. On Visual Formalisms. *Commun. ACM* 31, 5 (May 1988), 514–530. https://doi.org/10.1145/42411.42414

[16] H. Rex Hartson, Antonio C. Siochi, and D. Hix. 1990. The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs. *ACM Trans. Inf. Syst.* 8, 3 (July 1990), 181–203. https://doi.org/10.1145/98188.98191

[17] Zef Hemel and Eelco Visser. 2011. Declaratively Programming the Mobile Web with Mobl. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 695–712. https://doi.org/10.1145/2048066.2048121

[18] Ralph D. Hill. 1986. Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction—the Sassafras UIMS. *ACM Trans. Graph.* 5, 3 (July 1986), 179–210. https://doi.org/10.1145/24054.24055

[19] Ralph D. Hill. 1992. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Monterey, California, USA) *(CHI '92)*. Association for Computing Machinery, New York, NY, USA, 335–342. https://doi.org/10.1145/142750.142828

[20] Kasper Hornbæk and Antti Oulasvirta. 2017. What Is Interaction?. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 5040–5052. https://doi.org/10.1145/3025453.3025765

[21] Eric Horvitz, Carl Kadie, Tim Paek, and David Hovel. 2003. Models of Attention in Computing and Communication: From Principles to Applications. *Commun. ACM* 46, 3 (March 2003), 52–59. https://doi.org/10.1145/636772.636798

[22] Hiroshi Ishii and Brygg Ullmer. 1997. Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '97)*. Association for Computing Machinery, New York, NY, USA, 234–241. https://doi.org/10.1145/258549.258715

[23] Robert J. K. Jacob. 1983. Using Formal Specifications in the Design of a Human-Computer Interface. *Commun. ACM* 26, 4 (April 1983), 259–264. https://doi.org/10.1145/2163.358093

[24] Robert J. K. Jacob. 1985. A state transition diagram language for visual programming. *Computer* 8 (1985), 51–59.

[25] Robert J. K. Jacob. 1996. A visual language for non-WIMP user interfaces. In *Proceedings 1996 IEEE Symposium on Visual Languages*. 231–238. https://doi.org/10.1109/VL.1996.545292

[26] Robert J. K. Jacob, Leonidas Deligiannidis, and Stephen Morrison. 1999. A Software Model and Specification Language for Non-WIMP User Interfaces. *ACM Trans. Comput.-Hum. Interact.* 6, 1 (March 1999), 1–46. https://doi.org/10.1145/310641.310642

[27] Wendy Ju. 2015. The design of implicit interactions. *Synthesis Lectures on Human-Centered Informatics* 8, 2 (2015), 1–93.

[28] David J. Kasik. 1982. A User Interface Management System. *SIGGRAPH Comput. Graph.* 16, 3 (July 1982), 99–106. https://doi.org/10.1145/965145.801268

[29] 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* (Austin, Texas, USA) *(CHI '12)*. Association for Computing Machinery, New York, NY, USA, 2885–2894. https://doi.org/10.1145/2207676.2208694

[30] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, Murielle Florins, and Daniela Trevisan. 2004. USIXML: A User Interface Description Language for Context-Sensitive User Interfaces. In *Proceedings of the ACM AVI'2004 Workshop "Developing User Interfaces with XML: Advances on User Interface Description Languages"*. 55–62.

[31] 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. *Proc. ACM Hum.-Comput. Interact.* 2, EICS, Article 12 (June 2018), 27 pages. https://doi.org/10.1145/3229094

[32] Ingo Maier and Martin Odersky. 2012. Deprecating the Observer Pattern with Scala.React. (2012), 20. http://infoscience.epfl.ch/record/176887

[33] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 1–20. https://doi.org/10.1145/1640089.1640091

[34] Brad A. Myers. 1990. A New Model for Handling Input. *ACM Trans. Inf. Syst.* 8, 3 (July 1990), 289–320. https://doi.org/10.1145/98188.98204

[35] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. 1990. Garnet: comprehensive support for graphical, highly interactive user interfaces. *Computer* 23, 11 (1990), 71–85. https://doi.org/10.1109/2.60882

[36] 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. Comput.-Hum. Interact.* 16, 4, Article 18 (Nov. 2009), 56 pages. https://doi.org/10.1145/1614390.1614393

[37] William M. Newman. 1968. A System for Interactive Graphical Programming. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '68 (Spring))*. Association for Computing Machinery, New York, NY, USA, 47–54. https://doi.org/10.1145/1468075.1468083

[38] Jeffrey Nichols and Brad A. Myers. 2009. Creating a Lightweight User Interface Description Language: An Overview and Analysis of the Personal Universal Controller Project. *ACM Trans. Comput.-Hum. Interact.* 16, 4, Article 17 (Nov. 2009), 37 pages. https://doi.org/10.1145/1614390.1614392

[39] Dan R. Olsen. 1986. MIKE: The Menu Interaction Kontrol Environment. *ACM Trans. Graph.* 5, 4 (Oct. 1986), 318–344. https://doi.org/10.1145/27623.28868

[40] Dan R Olsen and Elizabeth P. Dempsey. 1983. SYNGRAPH: A Graphical User Interface Generator. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques* (Detroit, Michigan, USA) *(SIGGRAPH '83)*. Association for Computing Machinery, New York, NY, USA, 43–50. https://doi.org/10.1145/800059.801131

[41] Chethan Pandarinath, Paul Nuyujukian, Christine H Blabe, Brittany L Sorice, Jad Saab, Francis R Willett, Leigh R Hochberg, Krishna V Shenoy, and Jaimie M Henderson. 2017. High performance communication by people with paralysis using an intracortical brain-computer interface. *eLife* 6 (feb 2017), e18554. https://doi.org/10.7554/eLife.18554

[42] David L. Parnas. 1969. On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System. In *Proceedings of the 1969 24th National Conference (ACM '69)*. Association for Computing Machinery, New York, NY, USA, 379–385. https://doi.org/10.1145/800195.805945

[43] Fabio Paterno', Carmen Santoro, and Lucio Davide Spano. 2009. MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments. *ACM Trans. Comput.-Hum. Interact.* 16, 4, Article 19 (Nov. 2009), 30 pages. https://doi.org/10.1145/1614390.1614394

[44] Rosalind W. Picard. 1997. *Affective Computing*. MIT Press, Cambridge, MA, USA.

[45] François Pottier and Nadji Gauthier. 2004. Polymorphic Typed Defunctionalization. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/964001.964009

[46] Phyllis Reisner. 1981. Formal Grammar and Human Factors Design of an Interactive Graphics System. *IEEE Transactions on Software Engineering* SE-7, 2 (1981), 229–240. https://doi.org/10.1109/TSE.1981.234520

[47] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. 1999. The Context Toolkit: Aiding the Development of Context-enabled Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Pittsburgh, Pennsylvania, USA) *(CHI '99)*. ACM, New York, NY, USA, 434–441. https://doi.org/10.1145/302979.303126

[48] Albrecht Schmidt. 2000. Implicit human computer interaction through context. *Personal technologies* 4, 2-3 (2000), 191–199.

[49] 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* (New York, New York, USA) *(UIST '10)*. Association for Computing Machinery, New York, NY, USA, 47–56. https://doi.org/10.1145/1866029.1866039

[50] Barış Serim and Giulio Jacucci. 2019. Explicating Implicit Interaction: An Examination of the Concept and Challenges for Research. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. ACM, New York, NY, USA, Article 417, 16 pages. https://doi.org/10.1145/3290605.3300647

[51] Orit Shaer, Nancy Leland, Eduardo H Calvillo-Gamez, and Robert J. K. Jacob. 2004. The TAC paradigm: specifying tangible user interfaces. *Personal and Ubiquitous Computing* 8, 5 (2004), 359–369.

[52] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

[53] Tomoki Shibata, Alena Borisenko, Anzu Hakone, Tal August, Leonidas Deligiannidis, Chen-Hsiang Yu, Matthew Russell, Alex Olwal, and Robert J. K. Jacob. 2019. An Implicit Dialogue Injection System for Interruption Management. In *Proceedings of the 10th Augmented Human International Conference 2019* (Reims, France) *(AH2019)*. Association for Computing Machinery, New York, NY, USA, Article 27, 9 pages. https://doi.org/10.1145/3311823.3311875

[54] Tomoki Shibata, Evan M. Peck, Daniel Afergan, Samuel W. Hincks, Beste F. Yuksel, and Robert J. K. Jacob. 2014. Building Implicit Interfaces for Wearable Computers with Physiological Inputs: Zero Shutter Camera and Phylter. In *Proceedings of the Adjunct Publication of the 27th Annual ACM Symposium on User Interface Software and Technology* (Honolulu, Hawaii, USA) *(UIST'14 Adjunct)*. Association for Computing Machinery, New York, NY, USA, 89–90. https://doi.org/10.1145/2658779.2658790

[55] Ben Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8 (1983), 57–69. https://doi.org/10.1109/MC.1983.1654471

[56] Erin T. Solovey, Daniel Afergan, Evan M. Peck, Samuel W. Hincks, and Robert J. K. Jacob. 2015. Designing Implicit Interfaces for Physiological Computing: Guidelines and Lessons Learned Using FNIRS. *ACM Trans. Comput.-Hum. Interact.* 21, 6, Article 35 (Jan. 2015), 27 pages. https://doi.org/10.1145/2687926

[57] Erin T. Solovey, Paul Schermerhorn, Matthias Scheutz, Angelo Sassaroli, Sergio Fantini, and Robert J. K. Jacob. 2012. Brainput: Enhancing Interactive Systems with Streaming Fnirs Brain Input. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) *(CHI '12)*. Association for Computing Machinery, New York, NY, USA, 2193–2202. https://doi.org/10.1145/2207676.2208372

[58] Chris Stolte, Diane Tang, and Pat Hanrahan. 2002. Polaris: a system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics* 8, 1 (2002), 52–65. https://doi.org/10.1109/2945.981851

[59] Roel Vertegaal. 2003. Attentive user interfaces. *Commun. ACM* 46, 3 (2003), 30–33.

[60] Mark Weiser. 1999. The Computer for the 21st Century. *SIGMOBILE Mob. Comput. Commun. Rev.* 3, 3 (July 1999), 3–11. https://doi.org/10.1145/329124.329126

[61] Chen-Hsiang Yu and Robert C. Miller. 2010. Enhancing Web Page Readability for Non-Native Readers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 2523–2532. https://doi.org/10.1145/1753326.1753709

[62] Beste F. Yuksel, Kurt B. Oleson, Lane Harrison, Evan M. Peck, Daniel Afergan, Remco Chang, and Robert J. K. Jacob. 2016. Learn Piano with BACh: An Adaptive Learning Interface That Adjusts Task Difficulty Based on Brain State. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. Association for Computing Machinery, New York, NY, USA, 5372–5384. https://doi.org/10.1145/2858036.2858388

[63] Thorsten O Zander and Christian Kothe. 2011. Towards passive brain–computer interfaces: applying brain–computer interface technology to human–machine systems in general. *Journal of Neural Engineering* 8, 2 (mar 2011), 025005. https://doi.org/10.1088/1741-2560/8/2/025005