Async/Await is an Effective Paradigm for Event Management in User Interfaces

Donatien Grolaux
ICHEC Brussels Management School
Brussels, Belgium
Université catholique de Louvain
Louvain Research Institute in Management and Organizations
Louvain-la-Neuve, Belgium
donatien.grolaux@{ichec,uclouvain}.be

Thanh-Diane Nguyen
ICHEC Brussels Business School
Brussels, Belgium
Université catholique de Louvain
Louvain Research Institute in Management and Organizations
Louvain-la-Neuve, Belgium
thanh-diane.nguyen@{ichec,uclouvain}.be

Jean Vanderdonckt
Université catholique de Louvain
Louvain Research Institute in Management and Organizations (LouRIM)
Louvain-la-Neuve, Belgium
jean.vanderdonckt@uclouvain.be

Abstract

Event Management in Graphical User Interfaces (GUIs) remains a fundamental challenge when relying on complex, domain-specific paradigms like event loops, callbacks, or reactive programming, which mix and match different paradigms. This leads to steep learning curves and difficult-to-maintain code bases. To address these limitations, this paper treats user interface events as an input/output operation that leverages the sophisticated, language-level concurrency features of the `async`/`await` paradigm, a standard for managing input/output-bound tasks in modern programming languages. Through an implemented proof-of-concept, we compare the `async`/`await` approach against existing event management alternatives. Our analysis demonstrates that the `async`/`await` paradigm is capable of covering the functional scope of existing alternatives while delivering unique and substantial benefits. It simplifies event management by utilizing native platform features like sequential flow control and exception handling, rather than implementing these concepts indirectly. The `async`/`await` paradigm offers an effective and unified model for event management that reduces the learning process and provide a powerful tool for designing complex systems and modernizing legacy code. Our findings advocate for adopting `async`/`await` as the foundational paradigm for building responsive and maintainable user interfaces.

CCS Concepts

• **Human-centered computing** → **Ubiquitous and mobile computing design and evaluation methods**; **Graphical user interfaces**; *Web-based interaction*; *User interface programming*; • **Networks** → **Application layer protocols**; • **Computing methodologies** → **Distributed computing methodologies**; • **Computer systems organization** → **Cloud computing**.

Keywords

Event handling, Event management,Asynchronous programming, User interface events.

EICS Companion '26, Patras, Greece

ACM ISBN 979-8-4007-2669-9/2026/06
https://doi.org/10.1145/3807968.3810928

ACM Reference Format:
Donatien Grolaux, Thanh-Diane Nguyen, and Jean Vanderdonckt. 2026. Async/Await is an Effective Paradigm for Event Management in User Interfaces. In *Companion Proceedings of the 18th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS Companion '26), June 30-July 03, 2026, Patras, Greece.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3807968.3810928

1 Introduction

Event management in Graphical User Interfaces (GUIs) is the process by which an interactive system detects, processes, and responds to user actions and system-generated events within a visual application environment. Event management involves [5, 8]: *event detection* (*i.e.*, capturing user or system actions, such as mouse clicks, key presses, menu selection, or functions completed); *event queuing* (*i.e.*, placing events into an event queue in the order they occur to adequately process them); *event dispatching* (*i.e.*, sending events to the appropriate component, such as a menu or a text field); *event handling* (*i.e.*, executing predefined functions or callbacks that respond to the event). Early research shifted event handling [7] from raw input loops [19, 31] to structured event languages and toolkits, laying the foundations for later architectures. Early work articulated event languages that extended the basic event model into runtime frameworks [5], enabling modular handler composition and exception management [27]. Toolkits and interface builders then standardized component libraries [31] and event dispatch patterns for desktop systems, broadening adoption and consistency of GUI event models [25]. Subsequent milestones emphasized distribution and persistence of event information. Synthesized interaction and toolkit extensions on window systems explored richer event generation and composition [5]. The need to capture and replay events spawned event repository architectures that treat events as temporal indices suitable for storage, replay and analysis [9]. At the same time the publish/subscribe paradigm [42] and broker composability emerged as a structuring mechanism for distributed, decoupled event flows [40] that need to be synchronized [1].

More recently, reactive user-event handling [2] relies on architectural patterns that decouple event producers from consumers [37] and raise interaction abstractions above callbacks [10]. Architectures range from Event-Driven Architecture (EDA) [21] for distributed systems [16] to UI-level reified models [17] that separate semantics from commands:

- Event-driven architecture decouples producers and consumers [37] using asynchronous events, brokers, and streaming analytics [21] as core blocks for event-response systems [19].
- Interaction as a first-class citizen reifies user interactions (*e.g.* sequences of actions, composition of gestures) so they become reusable objects that map to commands with built-in features, such as Undo/Redo. Interacto demonstrates this for desktop and web toolkits, improving modularity and testability [23].
- Reactive dependency graphs use graphs of reactive nodes to manage propagation, granularity, and cycle handling in object-oriented settings, enabling more precise dependency control than classic patterns [24].

Key implications of these advances reveal that GUI developers prefer decoupling for scalability and fault isolation, reify higher-level interactions for reuse and testability, and represent dependencies explicitly when fine-grained propagation and cycle management matter. While these approaches are well suited to their respective problems, they require a specific architecture for event management with some complexity [29]. They also reimplement features already existing in the underlying platform. For example, React, a popular library for web and native user interfaces, implements its own event handling, thereby replicating functionality already provided by the browser [12]. This results in unnecessary bloat and prevents integrating this technology in legacy projects. To address these limitations, we want to leverage existing features of the web platform as a suitable universal building block that simplifies event management. To this end, this paper contributes to the area of engineering interactive computing systems by:

- Defining GUI events as an input/output operation (Section 3) that leverages the sophisticated, language-level concurrency features of the Async/Await paradigm, a standard for managing input/output-bound tasks [38] in modern programming languages[1] with StreamAsync (Section 4).
- Implementing StreamAsync, a library based on `async`/`await` for event management in GUIs of web applications (Section 5).
- Comparing our approach to five other practices (Section 6): synchronous programming, event listeners, event bus pattern, frame animation loops, functional-based event management frameworks, and reactive programming.

Section 2 summarizes work related to the area of event management in GUIs and positions our work with respect to it. Section 7 concludes this paper and presents some avenues to this work.

2 Related Work and Background

The related work includes prior work on input handling models [12], event languages [5] for GUI frameworks [9], and approaches to managing events in complex applications [7].

The concept of Event-Response Systems (ERS) [19] emerged with interactive computing environments in the eighties. Systems such as Smalltalk pioneered event-loop architectures where user actions were captured and processed asynchronously through an event queue and dispatcher. This separation of event detection from event handling laid the groundwork for subsequent GUI frameworks by establishing the canonical event loop as a core architectural element [31]. Hierarchical events [32] slightly reduced the complexity of even handling. [5] extend an event model for GUIs, introducing an event language that allows modular construction of complex event systems and supports a general-purpose runtime framework for UIs. Myers [31] defined a model for handling input devices in GUIs, encapsulating interactive behaviors into "Interactor" object types, proven convenient, efficient, and easy to learn in the Garnet toolkit. Chabou and Pierre [7] introduce an approach for handling events in complex applications, including GUIs, focusing on ease of understanding and suitability for interactive applications. Rajagopalan et al. [36] discuss optimizing event-based programs in GUIs using static optimization techniques, with experimental results demonstrating effectiveness. The Observer pattern provides a mechanism for observers to register interest in state changes and receive callbacks when events occur. Other patterns, such as Command and Mediator, have been applied to encapsulate user actions and coordinate complex interactions among UI components [4].

With the advent of web interfaces and hierarchical document structures, event propagation became an important research area. The Document Object Model (DOM) Events specification introduced bubbling and capturing phases, whereby events propagate through parent and child elements, allowing flexible interception and cancellation. The proliferation of mobile devices and touchscreens introduced new types of events and complexities in dispatch mechanisms: "Swipe", "Flick", "Pinch in/out" gestures require richer semantic interpretation than traditional click events.

Reactive programming [28] is now considered as a viable alternative to imperative event callbacks. Functional Reactive Programming (FRP) [10] treats events as streams of values that can be transformed and combined declaratively. Chen [9] introduces a formal modeling framework for Java GUI event handling using labeled transition systems to address nondeterminism and improve correctness. More recent contributions address event architecture for augmented paper interfaces, visual languages for end-user event specification, and optimization techniques for event-based programs. Liu et al. [25] present a visual language for UI event handling specification that allows end users to express simple and complex event handling mechanisms via visual specifications, incorporating event filtering, tool state querying, and action invocation, with evaluations of its effectiveness. Additional works cover event-flow models for GUI testing [18] and event-based profiling for refactoring, that represent diverse perspectives on event handling.

In sum, while there has been a significant effort towards improving even management in GUIs, developing this part still remains complex to achieve. We regret that existing event-handling approaches in GUIs still rely on callbacks or listener-based mechanisms, which fragment the program logic across multiple handlers and make complex interaction flows difficult to express and maintain. In contrast, Async/Await enables asynchronous event processing to be written in a sequential style, allowing developers to use conventional control-flow constructs while preserving the responsiveness required in GUIs, which is the goal of this paper.

[1] See https://en.wikipedia.org/wiki/Async/await

3 Preliminaries to Async/Await

To manage I/O operations in single-threaded JavaScript, developers traditionally used callbacks. However, nesting these for sequential operations leads to *callback hell* [13] (Listing 1), making code difficult to maintain.

```
jQuery.ajax({url:"userPreferenceSimpleMode",
  success(response) {
    if (response=="true") {
      jQuery.ajax({url:"getSimpleData",
        success(response) {
          display(JSON.parse(response));
        }})
    } else {
      jQuery.ajax({url:"getFullData",
        success(response) {
          display(JSON.parse(response));
        }})
}}})
```

Listing 1: Weaving callbacks and logic: callback hell.

Promises [34] improved this by flattening the structure (Listing 2), yet complex logic remains trapped in callback chains.

```
fetch("userPreferenceSimpleMode")
.then(response=> response.text())
.then(response=> {
if (response=="true") {
        return fetch("getSimpleData")
} else {
        return fetch("getFullData")
}})
.then(response=>response.json())
.then(response=> display(response))
```

Listing 2: Promises.

async/**await** [14] solves this by allowing asynchronous operations to be written in a synchronous-like style (Listing 3).

```
let response=await fetch("userPreferenceSimpleMode")
response=await response.text();
let data=await
↪  (response=="true"?fetch("getSimpleData"):fetch("getFullData"));
display(await data.json())
```

Listing 3: Async/Await.

As **await** changes the way instructions are orchestrated, it can only be used in a function prefixed by the **async** keyword. It is important to keep in mind that **async**/**await** is not multithreading. Multithreading is the ability of the operating system to concurrently run different threads of execution. **async**/**await** is a mechanism that orchestrates the execution order of instructions in a single thread . Each time **await** is met, the current execution of the **async** function is suspended, and control is given to another task, such as another awaiting function. An awaiting function can resume only when its awaiting instruction has been fully resolved. While this mechanism was designed for I/O operations, the idea behind this research is to reuse this mechanism with events management instead. In other words, if we see events as data added to a buffer, it makes sense to benefit from **async**/**await** for the IO operation of reading from this buffer in order to process each event.

4 Introduction to StreamAsync

We developed StreamAsync, a library that provides a function for streaming promises of specified events. The stream respects the `AsyncIterable`[30] interface of modern JavaScript, providing an `AsyncIterator` to the **for await** instruction [20]. This instruction consumes the promises from a stream, awaiting on each one of them, and using the resolved result as the value for the iteration variable.

```
<body>
    Counter example: <span id="counter"></span>
    <br>
    <button id="stop"></button><button id="start"></button>
</body>
```

```
(async () => {
    let counter = 0;
    while (true) {
        for await (const e of stream({ interval: 1000 }, { dom:
        ↪  '#stop', event: 'click', hint: 'stop' })) {
            if (e.hint == 'stop') break;
            document.getElementById("counter").innerText = counter++;
        }
        await stream({ dom: '#start', event: 'click' }).next();
    }
})()
```

Listing 4: StreamAsync example.

The example in Listing 4 illustrates a stopwatch, counting up each second. A stop button pauses the current count, and the start button resumes the chronometer. The main benefits of this paradigm to event management are:

(1) The `stream` (line #4) function manages many different types of events, unifying many different APIs. In this example, the first parameter, the `interval` event, relies on the `setInterval()` function which does not even trigger events, this function invokes callbacks instead. The second parameter is the usual `click` event on a button, relying on the `addEventListener()` function of the DOM (Document Object Model [41]) element.
(2) The stream function multiplexes many different sources of events together. As the user interface transitions between states, there is often a dependency on many different sources of events. In this example, clicking on the stop button pauses the chronometer and yields control to the play button. By reversing this thought process, we deduce that the chronometer ticks are dependent on two sources of events: the time interval ticks and the stop button.
(3) To simplify the identification of the event inside the loop, the stream function can decorate the triggered events with a `hint`. At line #4, the `click` event has its `hint` set to `stop`, and line #5 relies on this `hint`.
(4) The primary advantage of **async**/**await** is the ability to write asynchronous code as if it was synchronous. In lines #4-7, the code flow matches one-to-one with the behavior of the user interface. **while**(**true**) repeats the cycle of having a running/paused chronometer. **for await**(...) loops the behavior of incrementing the counter every second, and exiting (**break**) once the stop button is clicked. At line #8, **await** `stream(...).next()` extracts a single event of the stream; in other words, it waits until the play button is

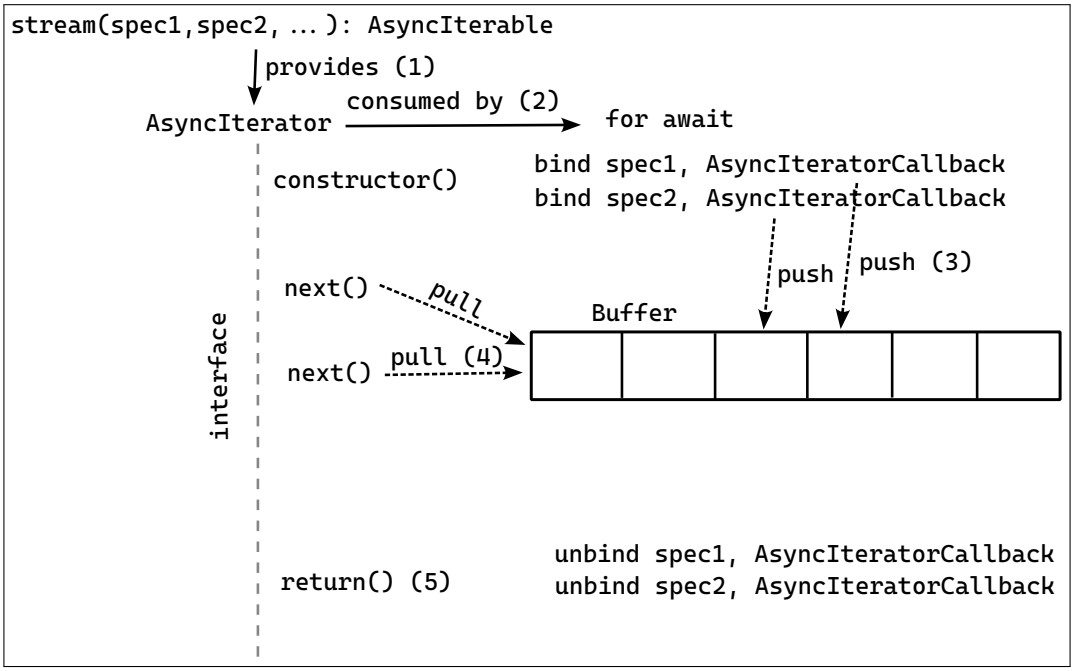

Figure 1: Life cycle of a stream. (1) The stream() function returns an `AsyncIterable` providing an `AsyncIterator` to `for await` (2). According to the stream specification, callbacks are bound to events pushing events to an internal buffer (3). `next()` consumes the events by pulling them from the buffer (4). At the end of life of the iterator, the callbacks are unbound (5).

clicked. The code is relying on the flow control operations (`while`, `for`, `break`, `if`) directly matching the UI behavior.

(5) Behind the curtains, `stream()` relies on APIs provided by the browser : `setInterval()`, `addEventListener()`, and so forth. An object is returned implementing the `AsyncIterable` interface. Each `for await` pulls an `AsyncIterator` from it. The `AsyncIterator` interface has a notification mechanism that can precisely pinpoint when the iterator is discarded, enabling releasing all allocated resources. In this example, the clean up calls `clearInterval()`, `removeEventListener()` and so forth. The user interface is not left with unneeded event listeners, reducing memory leaks and unnecessary computations. This is an implicit property of this paradigm, requiring no extra development effort.

StreamAsync supports all DOM events, but also DOM mutation events (attribute changes, add/remove nodes), different types of timer events (interval, animation frame, idle), and more. It is easily extensible to support any other type of events.

5 Implementation

Fig. 1 illustrates the life cycle of a stream:

(1) The `stream()` function returns an `AsyncIteratable` object whose purpose is to create `AsyncIterator` objects.

(2) Each `AsyncIterator` is consumed by a `for await` loop.

(3) At construction, the `AsyncIterator` object binds callbacks according to the specification given to the `stream()` function. An internal registry maps the different types of specification to binding and unbinding functions. This registry is user extensible. The callbacks push their events to an internal buffer of the `AsyncIterator`.

(4) The `next()` function of the `AsyncIterator` pulls an event from the buffer, and returns a promise. There is no guarantee that an event has been pushed to the buffer before a pull request, it may or not be the case. Consequently, there are two special cases to handle: (1) one or more events were already pushed before the pull request (Fig. 3): the pull request removes a single event from the buffer using First-In-First-Out order. (2) the pull request creates an entry in the buffer as an unresolved promise and extracts its `resolve()` function (Fig. 2). As the promise is unresolved, `for await` will block on this entry. These two cases are also handled by the callbacks when pushing entries to the buffer for new events: (1) there is an unresolved promise in the buffer (Fig. 2): use the extracted `resolve()` function to provide the event to the blocked `for await` loop, and remove this entry from the buffer. (2) all promises in the buffer are resolved (Fig. 3): create a promise auto-resolved to the event, and add it to the buffer.

(5) The `return()` function of the `AsyncIterator` is called when exiting the `for await` loop because of a `break`, a `return` or an exception. This function takes the opportunity to call all unbind functions of step (1), ensuring resources are freed.

The `AsyncIterable` returned by the `stream()` function also provides a `close()` function that closes all its currently running `AsyncIterators`. This is achieved by sending a `StreamCloseEvent` to all its `AsyncIterator` running, forcing all associated `for await` loops to eventually halt. The library is available at https://codeberg.

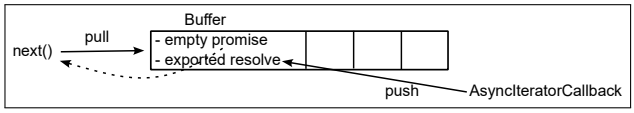

Figure 2: Internal buffer management for a pull-push.

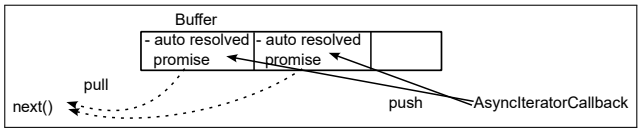

Figure 3: Internal buffer management for one or more push-pulls.

org/DonatienGrolaux/StreamAsync. It is open-sourced under the MIT license. The core library with support for DOM events is 33KB of source code (not minified), and the support for reactive programming is an extra 54KB (see section 6.6). There is no extra dependency, the library runs as is in the browser. A documentation project is available at https://codeberg.org/DonatienGrolaux/StreamAsyncDocumentation. While the library has been successfully used in small scale projects, including all examples of the documentation project, it remains a proof of concept and has not been battle tested in large scale applications.

6 Comparison to Other Event Management Paradigms

By leveraging the **async**/**await** mechanism, StreamAsync allows developers to write code that appears sequential while actually representing asynchronous behavior. This characteristic distinguishes it from most existing event management paradigms. Traditional approaches typically rely on fixed entry points for event handling, such as callbacks, which require complex interaction logic to be decomposed into multiple independent code fragments. Consequently, conventional control-flow constructs cannot be applied directly, and additional architectural structures are often necessary to regain a coherent representation of the GUI behavior.

StreamAsync helps in mitigating this *accidental complexity* [22], since asynchronous interactions can be expressed using familiar sequential control flow. Nevertheless, this approach does not eliminate this complexity entirely. Sophisticated behaviors may still require the coordination of multiple loops or the management of numerous event sources within these loops. Furthermore, although JavaScript execution remains single-threaded, the use of **await** interrupts the synchronous flow of execution, yielding control back to the main event loop. As a result, concurrency-like issues may arise, and developers must avoid assumptions about program state that could be invalidated during these suspension points. To better understand the benefits and limitations of StreamAsync, we perform a systematic comparison to other popular event management paradigms in this section. Each time, we reproduce the specifics of the paradigm using StreamAsync. This method informs us of weaknesses or even loss of capabilities of StreamAsync compared to other approaches.

6.1 Async/Await *vs.* Synchronous Communication

In traditional interactive terminal scripts, the user interface is considerably simpler and typically limited to synchronous read operations through which the user provides input. This interaction model is highly constrained, as program execution is halted while waiting for user input and the scope of interaction remains minimal. Nevertheless, from a programming perspective, such scripts are straightforward to implement with respect to user interaction. In these scripts, program instructions generally correspond directly to the sequence of interactions with the user. For example, placing a 'read' instruction inside a loop naturally results in repeated user input requests, thereby producing a cyclic interaction pattern. This direct correspondence between program structure and interaction flow simplifies reasoning about program behavior. The **async**/**await** mechanism is specifically designed to emulate this sequential interaction style in asynchronous environments. By allowing asynchronous operations to be expressed in a syntax resembling synchronous code, **async**/**await** preserves the intuitive structure found in terminal-based scripts. `StreamAsync` leverages this property to model event-driven interactions in GUIs while maintaining a programming style that closely resembles traditional sequential input handling.

6.2 Async/Await *vs.* Listeners

```
1 button.addEventListener("click", (e) => {
2         // do something
3     });
```

Listing 5: Event listener example.

```
1     for await (const e of stream({ dom: button, event: "click" })) {
2         // do the same
3     }
```

Listing 6: StreamAsync event listener correspondence.

The listener in Listing 5 is trivially rewritten in Listing 6. They look the same, however there is a difference between the two: **for await** is asynchronous while `addEventListener()` is synchronous. In many situations, there is no difference in behavior: eventually, a piece of code reacts to an event. However, addEventListener() takes advantage of its synchronous nature to provide control over the event propagation; should it bubble up to container components. Asynchronous management cannot retroactively control the event propagation, so we mitigate this issue at the event specification level, as shown in Listing 7.

```
1     for await (const e of stream({ dom: button, event: "click",
      ↪  stopPropagation:true, preventDefault:true })) {
2         // do the same
3     }
```

Listing 7: StreamAsync bubbling control.

This is a slight loss of functionality: our approach requires unconditional bubbling behavior while event listeners do not.

6.3 Async/Await *vs.* Event-bus Pattern

The Event-bus pattern [4] relies on a bus where all events are posted, and interested parties listen to, filtering for the events they are interested in. This pattern enables decoupling between emission and the reception of events [3]. This is useful in complex systems where it is beneficial to decouple emitters and receptors. Listing 8 shows StreamAsync version of a bus. It is structured as follows:

- Each bus maintains a list of callbacks, using `addListener()` and `removeListener()` functions.
- Each bus provides an `emit()` function to broadcast an event to all callbacks.

StreamAsync provides its own callback to the bus when instancing an `AsyncIterator`, so that `emit()` on the bus ends up pushing an event to the buffer of the `AsyncIterator`. Using this approach, the bus is shared by all emitters and receptors process the bus through their own **`for await`** loop, filtering for the events they are interested in. This is a direct correspondence to the event-bus.

```
let bus=stream.bus();
...
for await(const e of bus) {
        if (e meets some condition) {
                // do something
        }
}
...
bus.emit(someEvent);
bus.emit(someOtherEvent);
```

Listing 8: StreamAsync bus.

6.4 Async/Await *vs.* Frame Animation Loops

This pattern focuses on synchronizing animations with the refresh rate of a display. This is useful for animation or game development. Listing 9 shows StreamAsync support for a frame animation loop. The implementation relies on `window.requestAnimationFrame(callback)`. This function calls the provided callback once the browser is ready to draw the next frame, in sync with the display vertical refresh rate. As this function operates once, for a continuous animation, it must be called back after each animation step. With StreamAsync, we assume the drawing of a frame takes place in the main loop of the **`for await`** and it is only when awaiting the following event on the stream that `window.requestAnimationFrame()` is finally called again. This approach is a direct correspondence to using frame animation loops in a browser.

```
    for await (const e of stream({ animationFrame: true })) {
        ... // draw a frame
    }
```

Listing 9: StreamAsync frame animation loop.

6.5 Async/Await *vs.* Functional-based event management frameworks

Several events management frameworks rely on functional programming: RxJS [11], Bacon.js [33], most.js [6], or Kefir.js [26] to name a few. They rely on stream of events, like we do, but handle the streams using a functional programming (FP) paradigm instead of the imperative programming (IP) paradigm of StreamAsync. The FP approach relies on a number of standard functions that are piped together to achieve complex effects. Listing 10 illustrates this principle using RxJS.

```
fromEvent(document, 'click').pipe(scan(count => count + 1, 0),
↪  filter(n => n % 3 === 0)).subscribe(console.log);
```

Listing 10: RxJS example.

The breakdown of the flow is:

(1) `fromEvent(...)`: Creates the stream from DOM.
(2) `scan(...)`: The "accumulator" (like reduce for arrays) that keeps track of the total.
(3) `filter(...)`: Only allows values through that satisfy the condition.
(4) `subscribe(...)`: The trigger that finally executes the logic and prints the result.

Using `StreamAsync`, this example would be implemented by Listing 11.

```
let count=0;
for await (const e of stream({dom:document.body, event:"click"}) {
    count++;
    if (count % 3 == 0) console.log(count);
}
```

Listing 11: StreamAsync version of RxJS example.

It is beyond the scope of this paper to perform a comparison between FP and IP. Both are Turing-complete and each one can do what the other can. StreamAsync removes the barriers created by the functions of FP, which we argue may be beneficial in some scenario. There is a proposal to extend `AsyncIterator` to support FP, which should be to integrated into ECMAScript at some point in the future[15]. Once this proposal becomes an active standard, StreamAsync will support both FP and IP paradigms out of the box.

6.6 Async/Await *vs.* Reactive Programming

As popular UI frameworks for the web are currently based on the concept of reactive components, we show how a reactive component based system is possible with StreamAsync. Reactive web components are typically composed of three aspects glued together:

(1) A reactive data store that captures the inner state of a component;
(2) Reactive functions that affect computations over the reactive data;
(3) A reactive template that is a HTML fragment referencing the reactive data store and/or the reactive functions.

Upon changing the reactive data, the reactive functions affected are called implicitly and the templates are updated; eventually updating the UI. StreamAsync is not reactive programming, however we can achieve very similar effects. Contrarily to reactive functions, there is no implicit execution of code with StreamAsync, there is just regular flow control in a synchronous-like fashion. To mimic reactive components, we need a data store with observable mutations, which can feed a reactive template, and finally a reactive component.

6.6.1 Observable Data Store. A JavaScript `Proxy` is an object where access and mutations can be intercepted, hence observed. We extend this functionality to implement a generic observable data store, adding two important features: first, we recursively extend the Proxy so that embedded objects are observable too. In other words, our store can manage a JSON-like nested data structure, and observe any change, at any depth of this structure; second, we provide a way to specify which parts of the data structure needs to be observed. The specification supports paths through the data structure, wildcards (*) and recursive wildcards (**). The function `store.create()` creates a data store. A data store behaves like a pure JavaScript Object: set a key to a value, read a key, and delete a key. Listing 12 shows an example of these capabilities. The data store is independent to StreamAsync, but needed as a building block to reactive components. An integration is provided with StreamAsync to observe store changes: `store:mystore,key:property`.

```
let s=store.create();
store.addChangeListener(s, "**", (e) => { console.log("**: " +
↪ JSON.stringify(e.detail)) });
store.addChangeListener(s, "a.b", (e) => { console.log("a.b: " +
↪ JSON.stringify(e.detail)) });
store.addChangeListener(s, "a.c[*]", (e) => { console.log("a.c[*]: "
↪ + JSON.stringify(e.detail)) });
s.a = { a: "Hello", c:[] }; // internally, the object is turned into
↪ a store
s.a.c.push("World"); // triggers the "**" and "a.c[*]" listeners
```

Listing 12: Observable data store example.

6.6.2 Reactive Template. The second building block to reactive components is reactive templates. A `stream.reactive(?root,?store)` function extends StreamAsync by parsing a template `root` element, injecting data from a `store` and placing observers to react to future changes. The template mechanism supports {{key}} dynamic templates in text nodes, evaluated to the corresponding key value of the `store`. The template mechanism also supports HTML attributes and HTML properties dynamically tied to store values (Listing 13).

```
    <div>{{message}}</div> <!-- References the key "message" of the
    ↪ store -->
    <input data-prop="input|value=message"><br> <!-- on 'input'
    ↪ event, update the 'value' property -->
    <button onclick="store.of(this).message='Hello From
    ↪ Change!'">Change!</button><br> <!-- 'store.of(el)' returns
    ↪ the store that is in effect for this element-->
```

Listing 13: Reactive Template.

Internally, `stream.reactive()` relies on `stream()` to dynamically update the DOM. Once the DOM is parsed, `stream.reactive()` sets up loops that implement the reactivity, similar to Listing 14. Once the reactivity is set, there is no interpretation overhead, all updates implemented by direct DOM mutations. Common reactive systems like React or Vue rely on a virtual DOM mechanism. Their reaction is implemented by rendering the whole template with the updated data first into a virtual DOM, and then update the actual DOM to reflect this virtual DOM. This approach may be more efficient in some scenario where the rendering is delayed while updates occur, and only the final state is pushed to the actual DOM. For many common scenarios, it is faster to bypass the virtual DOM entirely, and just update the DOM directly. Svelte is a reactive framework that also relies on direct DOM mutations instead of virtual DOM, and is considered faster than other frameworks [35].

```
        (async()=>{
            for await (const e of stream({store:mystore,
            ↪ key:"message"}, stream.once)) {
                document.getElementById("message").
                innerText=mystore.message;
            }
        })();
```

Listing 14: Example of the internal updates managed by the reactive template of StreamAsync.

6.6.3 Reactive Component. The function `stream.component(spec)` creates a standard `WebComponent`[39]. `WebComponents` are quite complex to use. StreamAsync greatly simplifies the process. First, `stream.component()` relies on a simplified adapter object to a standard `WebComponent`. Second, the behavior of the `WebComponent` directly benefits from the advantages offered by `stream()`. A reactive `WebComponent` must adapt to different sources of events (changes to its DOM attributes, changes to its internal data store) handled by multiple entry points (constructor and methods). Using `StreamAsync`, all entry points can be redirected into a single one, using a single **`for await`** loop and usual flow control (Listing 15).

```
<body>
    <h1>my-compo</h1>
    <my-compo id="mine">
    </my-compo>
</body>
```

```
stream.component({
    name: "my-compo",
    template: '<input data-prop="input|value=message"><div> Here is
    ↪ what you typed: {{message}}</div>',
    async init() { // init is called during the WebComponent
    ↪ construction
        this.getStore().message = "";
        stream.reactive(this); // makes this component inner HTML
        ↪ reactive.
        for await (const e of stream({root:this, dom:"input",
        ↪ event:"dblclick"})) {
            this.getStore().message="Don't double click me!";
        }
    }
});
```

Listing 15: Example WebComponent using StreamAsync.

A store is associated to each StreamAsync component, corresponding to its reactive data part. The associated store may be a new one dedicated to the component, but we also provide a standardized way to change this behavior and have components share stores so they can work together. The component adapter provides a template, and `stream.reactive()` makes it reactive, extracting data from its associated store. The reactive template is the only part of this design that is implicitly reactive: the updates occur as the store is updated, without any extra intervention. The component itself is not implicitly reactive, there is no automatic function called in reaction to a data update. In a way, we may say that the component is explicitly reactive: the **`for await`** lists the exact events that trigger a reaction. We argue that this is an advantage to StreamAsync: the explicit execution path yields a clearer view of the

component behavior. Using this system, we implemented several standard components:

- <sa-reactive> makes its content reactive.
- <sa-store> creates a store that can be shared by other standard components (using their `from` attribute).
- <sa-template> can create and reuse templates.
- <sa-for> loops over an array from its store to repeat its content.
- <sa-if> displays its <sa-then> or <sa-else> child depending on a boolean from its store.
- <sa-switch> displays one of its <sa-case> child depending on a value from its store.

The implementation of these components is straightforward, thanks to `StreamAsync`. The components are still capable enough that they can map a tree data structure into a tree of components for example. They are also reactive to dynamic changes to their data stores as well as changes to their DOM attributes. Comparing this approach to Vue.js, we made two observations. First, we rely on different mechanisms to achieve reactivity, leveraging different aspects of the web platform: **`async`**/**`await`**, `WebComponents`. Vue.js, React and others tend to bypass the browser a lot more, reproducing some of its functionality in other ways: virtual DOM, custom event mechanism, and so forth. Therefore, StreamAsync is a lot simpler: the store module is around 1000 LoC (Lines of Code), the reactive DOM module is around 500 LoC, the component module is around 300 LoC, and the standard components all together require around 400 LoC. By focusing on leveraging existing features of the JavaScript platform, we provide standard `WebComponents` so that StreamAsync can provide a migration path to legacy code as it does not impose a complete technology switch. Complex events management can be progressively refactored and simplified thanks to `stream()`. When relevant, reactive components can be used side by side with legacy code. Second, the StreamAsync reactivity is explicit instead of being implicit. With StreamAsync, you cannot write a function depending on a reactive variable, and expect the function to be called on the fly when the variable changes. Instead, we have **`for await`** loops on explicit events. We see two benefits: first, we have complete control on when things happen. For example, we can easily decide to validate a text field after each keystroke or only after it loses its focus. Second, the code states its behavior explicitly. With popular reactive frameworks, a reactive function is called implicitly when deemed necessary. Hidden execution can make understanding the actual execution flow quite hard.

6.7 Intermediate conclusion

As this section demonstrated, StreamAsync is able to reproduce all other event management paradigms that we tested. Obviously, this study is limited by three factors. We base our reasoning on the core principles of each paradigm. There certainly exists implementations with special features that we did not take into account. The focus is on the core capabilities, without considering performance or memory consumption, among other possible criteria. While we have some empirical experience on small scale use of StreamAsync in real world applications, we lack experience for larger applications and problems that may emerge at scale.

The empirical experience gathered during the implementation of the StreamAsync version of reactive Web components, while limited, showed a real reduction in complexity over handling regular WebComponents. Since the **`async`**/**`await`** model can express other event-handling paradigms, there should be no loss of functionality over them. Moreover, its primary contribution lies in its ability to represent asynchronous event-driven interactions in a more structured and readable form. This capability constitutes a distinctive advantage compared to traditional event management techniques.

7 Conclusion and Future Work

This paper explored the use of **`async`**/**`await`** as a foundational paradigm for event management in GUIs. Existing GUI event-handling approaches, such as callbacks, listeners, event buses, functional streams, and reactive systems, typically require dedicated architectural constructs that fragment program logic across multiple entry points and abstractions. While these approaches enable powerful interaction patterns, they also introduce additional complexity and frequently reimplement mechanisms that already exist within the underlying execution platform.

To address these limitations, we proposed treating GUI events as input/output operations that can be consumed through the **`async`**/**`await`** model. Building on this idea, we developed StreamAsync, a library that exposes event sources as asynchronous streams compatible with the `AsyncIterable` interface. This design enables developers to express event-driven behaviors using familiar sequential control flow constructs such as loops, conditionals, and exception handling. As a result, interaction logic can be written in a style that closely mirrors the dynamic behavior of the user interface while maintaining the responsiveness required by asynchronous systems. Through conceptual comparisons with other paradigms, including synchronous scripts, event listeners, event-bus architectures, animation loops, functional streams, and reactive programming, we demonstrated that the **`async`**/**`await`** approach can express the functional capabilities of these models while offering improved clarity and reduced accidental complexity. Although this paradigm does not eliminate the inherent challenges of complex interactive systems, it provides a unified and expressive abstraction that leverages native platform features rather than introducing additional layers of architecture. Overall, our work suggests that **`async`**/**`await`** constitutes a compelling foundation for modern GUI event management.

Future work could explore static analysis and verification techniques for **`async`**/**`await`**. Because asynchronous suspension points may introduce concurrency, tools that help developers reason about program state, ordering, and potential race conditions would improve reliability in complex applications. While our current implementation focuses on web GUIs, the underlying principles could be generalized to other UI ecosystems such as desktop frameworks or mobile platforms. Investigating how the StreamAsync abstraction integrates with different runtime environments and event systems would help evaluate its generality as a cross-platform interaction model. Finally, while we showed how StreamAsync can express the functional capabilities of other event management approaches, and successfully used to approach to implement reactive components, we still lack insight and empirical data on its use at larger scale.

References

[1] Marco Blumendorf, Sebastian Feuerstack, and Sahin Albayrak. 2006. Event-based Synchronization of Model-Based Multimodal User Interfaces. In *Proceedings of the MoDELS'06 Workshop on Model Driven Development of Advanced User Interfaces Genova, Italy, October 2, 2006 (CEUR Workshop Proceedings, Vol. 214)*, Andreas Pleuß, Jan Van den Bergh, Heinrich Hußmann, Stefan Sauer, and Alexander Boedcher (Eds.). CEUR-WS.org. https://ceur-ws.org/Vol-214/paper9.pdf

[2] Oleksandra S. Bulgakova and Viacheslav V. Zosimov. 2019. Reactive Programming Paradigm for Development User Interfaces. *Control Systems and Computers* 5, 7 (2019), 62–69. doi:10.15407/csc.2019.05.062

[3] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. 2007. Past, Present, and Future Trends in Software Patterns. *IEEE Softw.* 24, 4 (2007), 31–37. doi:10.1109/MS.2007.115

[4] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. 2007. *Pattern-oriented software architecture, 4th Edition.* Wiley. https://www.worldcat.org/oclc/314792015

[5] N. V. Carlsen, N. J. Christensen, and H. A. Tucker. 1989. An event language for building user interface frameworks. In *Proceedings of the 2nd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology* (Williamsburg, Virginia, USA) *(UIST '89)*. Association for Computing Machinery, New York, NY, USA, 133–139. doi:10.1145/73660.73677

[6] Brian Cavalier and contributors. 2026. *most.js: Monadic streams for reactive programming.* https://github.com/mostjs/core Accessed: 2026-04-24.

[7] S. Chabou and S. Pierre. 1999. A new approach for handling events in complex applications. In *Engineering Solutions for the Next Millennium. 1999 IEEE Canadian Conference on Electrical and Computer Engineering (Cat. No.99TH8411)*, Vol. 1. 277–282 vol.1. doi:10.1109/CCECE.1999.807209

[8] Stéphane Chatty, Stéphane Sire, Jean-Luc Vinot, Patrick Lecoanet, Alexandre Lemort, and Christophe P. Mertz. 2004. Revisiting visual interface programming: creating GUI tools for designers and programmers. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology, Santa Fe, NM, USA, October 24-27, 2004*, Steven Feiner and James A. Landay (Eds.). ACM, 267–276. doi:10.1145/1029632.1029678

[9] Jessica Chen. 2002. Formal Modelling of Java GUI Event Handling. In *Formal Methods and Software Engineering*, Chris George and Huaikou Miao (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 359–370.

[10] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 411–422. doi:10.1145/2491956.2462161

[11] Paul Daniels and Luis Atencio. 2017. *RxJS in Action.* Simon and Schuster.

[12] Rémy El Sibaïe and Emmanuel Chailloux. 2016. Synchronous-reactive web programming. In *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems* (Amsterdam, Netherlands) *(REBLS 2016)*. Association for Computing Machinery, New York, NY, USA, 9–16. doi:10.1145/3001929.3001931

[13] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. 2015. Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript. In *Proceedings of 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Beijing, China). 1–10. doi:10.1109/ESEM.2015.7321196

[14] Raju Gandhi. 2019. *Seemingly Imperative with async and await.* Apress, Berkeley, CA, 211–222. doi:10.1007/978-1-4842-5394-6_14

[15] Kevin Gibbons, Gus Caplan, and TC39. 2026. Async Iterator Helpers Proposal. https://github.com/tc39/proposal-async-iterator-helpers. Accessed: 2026-04-24.

[16] Donatien Grolaux, Peter Van Roy, and Jean Vanderdonckt. 2004. Migratable User Interfaces: Beyond Migratory Interfaces. In *Proc. of 1st Annual International Conference on Mobile and Ubiquitous Systems, Networking and Services, 22-25 August 2004, Cambridge, MA, USA (MobiQuitous '04)*. IEEE Computer Society, 422–430. doi:10.1109/MOBIQ.2004.1331749

[17] Donatien Grolaux, Jean Vanderdonckt, and Peter Van Roy. 2005. Attach Me, Detach Me, Assemble Me Like You Work. In *Proc. of IFIP TC13 Int. Conf. on Human-Computer Interaction - INTERACT 2005, Rome, Italy, September 12-16, 2005 (Lecture Notes in Computer Science, Vol. 3585)*, Maria Francesca Costabile and Fabio Paternò (Eds.). Springer, 198–212. doi:10.1007/11555261_19

[18] David M. Hilbert and David F. Redmiles. 2000. Extracting usability information from user interface events. *ACM Comput. Surv.* 32, 4 (Dec. 2000), 384–421. doi:10.1145/371578.371593

[19] Ralph D. Hill. 1986. Event-Response Systems: A Technique for Specifying Multi-Threaded Dialogues. 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, 241–248. doi:10.1145/29933.275637

[20] Faraz K Kelhini. 2021. *Modern Asynchronous JavaScript.* The Pragmatic Programmers LLC.

[21] Adisheshu Reddy Kommera. 2020. The Power of Event-Driven Architecture: Enabling Real-Time Systems and Scalable Solutions. *Turkish Journal of Computer and Mathematics Education (TURCOMAT) ISSN* 11, 1 (2020), 1740–1751. doi:10.61841/turcomat.v11i1.14928

[22] Vinay Kulkarni, Souvik Barat, Tony Clark, and Balbir Barn. 2015. Toward overcoming accidental complexity in organisational decision-making. In *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems* (Ottawa, Ontario, Canada) *(MODELS '15)*. IEEE Press, 368–377.

[23] Shir Levkowitz. 2024. Theory and Method for Reactive Semantics in Application Development. In *Proceedings of the 2024 IEEE International Systems Conference (SysCon)* (Montreal, QC, Canada) *(SysCon '24)*. 1–3. doi:10.1109/SysCon61195.2024.10553547

[24] Geoffrey Litt, Nicholas Schiefer, Johannes Schickling, and Daniel Jackson. 2023. Riffle: Reactive Relational State for Local-First Applications. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 76, 16 pages. doi:10.1145/3586183.3606801

[25] Na Liu, John Hosking, and John Grundy. 2007. A visual language and environment for specifying user interface event handling in design tools. In *Proceedings of the Eight Australasian Conference on User Interface - Volume 64* (Ballarat, Victoria, Australia) *(AUIC '07)*. Australian Computer Society, Inc., AUS, 87–94.

[26] Roman Liutikov and contributors. 2026. *Kefir.js: A Fast and Light Reactive Programming Library for JavaScript.* https://kefirjs.github.io/kefir/ Accessed: 2026-04-24.

[27] Ashish Majmundar. 2017. User interface event orchestration. US Patent 9,720,655.

[28] Alessandro Margara and Guido Salvaneschi. 2018. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Transactions on Software Engineering* 44, 7 (2018), 689–711. doi:10.1109/TSE.2018.2833109

[29] Ben Moseley and Peter Marks. 2006. Out of the tar pit. *Software Practice Advancement (SPA)* 2006 (2006). https://moss.cs.iit.edu/cs100/papers/out-of-the-tar-pit.pdf

[30] MozDevNet. [n. d.]. Web. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/AsyncIterator

[31] Brad A. Myers. 1991. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology, UIST 1991, Hilton Head, South Carolina, USA, November 11-13, 1991*, James R. Rhyne (Ed.). ACM, 211–220. doi:10.1145/120782.120805

[32] Brad A. Myers and David S. Kosbie. 1996. Reusable Hierarchical Command Objects. In *Conference on Human Factors in Computing Systems: Common Ground, CHI '96, Vancouver, BC, Canada, April 13-18, 1996, Proceedings*, Bonnie A. Nardi, Gerrit C. van der Veer, and Michael J. Tauber (Eds.). ACM, 260–267. doi:10.1145/238386.238526

[33] Juha Paananen and contributors. 2026. *Bacon.js: Functional Reactive Programming library for JavaScript.* https://baconjs.github.io/ Accessed: 2026-04-24.

[34] Daniel Parker. 2015. *JavaScript with Promises: Managing Asynchronous Code.* O'Reilly Media, Inc.

[35] Fauzan Prasetyo Eka Putra, Hasbullah Hasbullah, Farhan Muslim, and Reni Paradina. 2025. Technical Performance Comparison of Modern Frontend Frameworks Study on Svelte, React, and Vue. *Brilliance: Research of Artificial Intelligence* 5, 1 (Jul. 2025), 355–364. doi:10.47709/brilliance.v5i1.6133

[36] Mohan Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, and Richard D. Schlichting. 2002. Profile-directed optimization of event-based programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 106–116. doi:10.1145/512529.512543

[37] Ovidiu-Andrei Schipor, Radu-Daniel Vatavu, and Jean Vanderdonckt. 2019. Euphoria: A Scalable, event-driven architecture for designing interactions across heterogeneous devices in smart environments. *Inf. Softw. Technol.* 109 (2019), 43–59. doi:10.1016/J.INFSOF.2019.01.006

[38] Ena Tominaga, Yoshitaka Arahori, and Katsuhiko Gondow. 2019. AwaitViz: a visualizer of JavaScript's async/await execution order. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (Limassol, Cyprus) *(SAC '19)*. Association for Computing Machinery, New York, NY, USA, 2515–2524. doi:10.1145/3297280.3297528

[39] Arie van Deursen and Ali Mesbah. 2017. Components of the Modern Web. In *Proceedings of the International Conference on Software Engineering.* doi:10.1145/3053555.3053556

[40] David F Wiley. 2020. Centralized selection context for user interface data binding and event handling. US Patent 10,725,969.

[41] Lauren Wood, Vidur Apparao, Laurence Cable, Mike Champion, Mark Davis, Joe Kesselman, Tom Pixley, Jonathan Robie, Peter Sharpe, and Chris Wilson. 2000. Document object model (DOM) level 2 specification. *World Wide Web Consortium. www.w3.org/TR/DOM-Level-2* (2000).

[42] Junya Xu, Jiaqi Yin, Huibiao Zhu, and Lili Xiao. 2021. Modeling and Verifying Producer-Consumer Communication in Kafka Using CSP. In *7th Conference on the Engineering of Computer Based Systems* (Novi Sad, Serbia) *(ECBS 2021)*. Association for Computing Machinery, New York, NY, USA, Article 9, 10 pages. doi:10.1145/3459960.3459961