Unified Library for Dependency-graph Reactivity on Web and Desktop User Interfaces

João Paulo Oliveira Marum
University of Mississippi
University, Mississippi, USA
jmarum@acm.org

H. Conrad Cunningham
University of Mississippi
University, Mississippi, USA
hcc@cs.olemiss.edu

J. Adam Jones
University of Mississippi
University, Mississippi, USA
jadamj@acm.org

ABSTRACT

In user interfaces on Web and desktop applications, the system must quickly respond to user inputs and accurately display the result. Current solutions for user interfaces often respond too slowly and display temporarily inaccurate or misleading states, resulting in low user satisfaction. To alleviate this problem, we develop a reactive programming approach that encodes the complex relationships among the user interface components in a dependency graph and then uses the graph to order the updates of the components without violating the dependency constraints. This enables more timely updates and more accurate visualizations, potentially providing users with a more satisfying experience. We evaluate our approach by comparing its performance with important alternative reactive libraries for user interfaces.

CCS CONCEPTS

• **Software and its engineering** → *Software libraries and repositories*; • **Human-centered computing** → *Graphical user interfaces*.

KEYWORDS

User interface, event-driven system, dependency graph, reactive programming

ACM Reference Format:
João Paulo Oliveira Marum, H. Conrad Cunningham, and J. Adam Jones. 2020. Unified Library for Dependency-graph Reactivity on Web and Desktop User Interfaces. In *2020 ACM Southeast Conference (ACMSE 2020), April 2–4, 2020, Tampa, FL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3374135.3385278

1 INTRODUCTION

Most contemporary software applications are reactive. That is, they engage in ongoing interactions with their environments [4, 14]. They respond to events, which may correspond to an interaction with the outside world (e.g. a mouse click) or with other components of the application (e.g. changes in the values of important data attributes).

In this paper, we consider both desktop and Web-based user interfaces (UIs). Foust et al. [9] argue that UIs are reactive applications that are normally implemented using imperative callbacks, which are then handled by an event-processing layer. Czaplicki and Chong [7] note that the user interface components are arranged in graph structures: for the Web, this graph is the Document Object Model (DOM) and for the .Net desktop, it is the `Designer` class. Unless the executions of the controls are explicitly scheduled by the program, they are scheduled in whatever order the controls happen to fit into the system. Normally, an event first causes its associated component to be executed, which may raise other events that can subsequently execute other components in the interface.

Foust [9] states that a graphical user interface (GUI) must be able to execute complex tasks in which one user interaction can initiate a chain of effects on other GUI components. It should be able to do so without introducing any inaccurate or misleading displays, even temporarily. The current approaches to the implementation of a GUI rely on asynchronous calls. These approaches enable the GUI to respond to a user at any time, but they make the management of the data dependencies among components difficult.

For example, consider the .Net framework. An event causes its associated GUI control to be scheduled for execution. The execution of this control may have an effect upon another control in a chain (e.g. the second control may use data that the first modifies). The first control does not directly call the second. Instead, the first control raises a new event, which will cause the execution of the second control at some later point. In a complex system, effects may appear slowly and in a different order than expected by the user. This may result in temporarily inaccurate and misleading displays.

In an attempt to alleviate this problem, developers sometimes employ reactive programming techniques. These can help because they make the data dependencies explicit and enforce them automatically at runtime, However, they still do not handle dependencies between controls.

In this research, we develop a reactive programming approach that addresses the problem described above. Our approach (described in Section 3) analyzes the complex relationships among the controls, encodes the dependencies between them in a dependency graph, and then uses the graph to order the updates of the components without violating the dependency constraints. We hypothesize that this approach can improve the performance of a set of updates that users conceptualize as occurring in a single chain and ensure a deterministic result. Whenever an event associated with some control A is fired in the user interface, our approach uses the graph to generate the list of other controls affected by control A. The system then executes these controls along with control A in the order defined by the list. Thus they appear as part of the same update of the display.

We evaluate our approach by comparing its performance to that of Sodium, a commonly used alternative reactive library for user interfaces. Section 4 describes our test methodology and Section 5 analyzes the results of our tests. Section 6 relates our research to previous work and Section 7 summarized this research and discuses possible future work. Before we look at the implementation, let's examine the problem in more detail in Section 2.

2 PROBLEM DEFINITION

Graphical user interfaces are critical components of many software products. Developers dedicate a large portion of development effort to their implementation. Given their prominence in software development and their role as mediators between users and computers, GUIs must be implemented correctly.

Bishop [1] defines a GUI as a hierarchical collection of user controls with each control containing its own position and attributes. Based upon this definition, we argue that in such a collection, some given user's interaction with one control may initiate a wave of changes that spreads incrementally across many other controls in the collection. In this paper, we call this situation a *chained execution*. For example, a selection of a radio button in a user form may activate or deactivate whole sections of the form, cause changes in default values, etc. These changes may, in turn, initiate their own waves of changes. These behaviors are mostly seen on portals and user forms, where certain controls are locked and unlocked based upon one or more answers.

In a typical implementation of a GUI, a user interacts with the GUI by raising an *event* (e.g. clicking the mouse while the cursor is positioned at a particular locus on the screen). If an event is associated with a particular GUI control, we call that control the *producer* of the event. Once an event is raised, it can be processed by event handlers associated with various *consumer* controls in the GUI. To associate some behavior with an event, a software developer must encode the desired behavior in an event handler.

In this approach, as stated by Bishop [1] and Silva [21], consumer controls are linked to events from the producer control. When the producer changes its state, it fires an event to let its consumers know about the change. Whenever there is an external interaction with some control A, then control A fires the event to notify the runtime system. The runtime system then asynchronously executes A's event handler whenever it is idle. If the execution of A's event handler affects another control B, then the control B fires a new event and then B's event handler is also asynchronously executed as soon as possible.

This is an asynchronous process that breaks the chained execution into several time-consuming steps. Usually, a chained execution requires the handling of several events, with one control executed per event-handling cycle. In a typical GUI, it may take several cycles for all the executions belonging in a chain of executions to propagate throughout the entire user interface. A user must wait for the entire sequence of steps to complete. The time may extend across more than one update of the display. What the developer intended to be a smooth and coherent experience may appear choppy and incoherent to the waiting user.

The event-handling approach described above is organized according to the well-known Observer design pattern described by Gamma et al. [10]. This structure can also affect the accuracy of the GUI and the predictability of the operations on the GUI. Salvaneschi [19, 20] attributes these shortcomings to the loose coupling of the GUI controls. Because events are handled asynchronously, the order and timing of change propagation is machine-dependent. This is especially problematic in situations where many events occur within a small time interval. The order in which events are processed may differ from the order in which they were generated. Listener-based asynchronous execution enables the system to keep responding to the user while one control is still executing. However, because it handles each execution independently from the others, it decreases the control over the execution, which complicates the handling of dependencies between controls. The existence of dependencies between controls means that the execution of one control's event may affect the outcome of another; therefore changing the order in which these events run may yield different outcomes.

The asynchronous nature of the Observer pattern guarantees neither the order in which events will be handled nor that the event will be handled at all. Events may occur in an order that does not respect the dependencies among the controls. This traditional approach thus can lead to misleading or inaccurate results.

Intuitively, when execution of some control A causes a change to its state, control A fires an event saying "here is my new state". Then some other control B with interest in and access to some portion of A's state can examine that portion and respond appropriately. This is basically how an event-driven system works. From our perspective, event-driven systems have two flaws. The first is that there may be a considerable time lag between A's state change and B's response. When a user perceives that the changes in A and B are linked, the time lag between may make the execution seem slow and choppy. The second flaw is that A's state may change a second time before B is able to examine the result of the first change. This can cause B to miss an update or retrieve data from A that is inconsistent with its other state. This can cause inaccurate or misleading displays, at least temporarily. Although the event-driven approach is appropriate in many circumstances, there are some situations in which executing all the updates as a single atomic chain is a more appropriate approach.

A functional reactive programming (FRP) library—such as Sodium [2], Reactive Extensions [13], ReactiveBanana [5], or Elm [7]—is an effective alternative to the use of the Observer pattern. According to Czaplicki [7], the FRP paradigm treats user events as discrete happenings on an infinite stream. Each event can be handled as it comes and the programmer can fully define the system's reaction to each event. There are no unexpected results. Because all data dependencies are explicit and are enforced on each event in the stream, the FRP paradigm is closer to being a solution to the problem described in this section than the traditional approach.

However, FRP does not fully solve the problem. Because each execution is self-contained, the FRP paradigm does not allow one control to impact the execution of another. Thus FRP does not well support user interfaces with dependencies between controls. In addition, the implementation of FRP libraries still relies on listeners. Each control's execution is regarded as a different point in time, a drop in the stream of events, with each drop handled internally as a regular event handler. So, for each control, the execution is still handled internally as an asynchronous event.

The primary goal of this research is to define a method for implementing a library for user interfaces that (a) supports dependencies among controls as described above, (b) avoids misleading or inaccurate visual presentations, and (c) performs competitively with other approaches. In the following section, we describe the implementation of our library.

3 IMPLEMENTATION

This section describes our reactive approach to implementing dynamic user interfaces. Each control within a page or form starts in some state and continuously interacts with the user and its environment (including other controls). Our approach analyzes the dynamic dependency relationships among the controls and builds a dependency graph. (For example, if a text box enables a button, then the button depends upon the text box.) This graph forms the basis for the reactive, dynamic user interface.

After the creation of the page, our approach creates the form's dependency graph by calling our `CreateGraph()` function on the controls in the UI. This function examines each control's properties, fields, and methods (its dynamic information, not its code) to construct a list of all other controls that this control affects. If a control in the form is intended to be reactive, it must implement `IUpdatable`, an interface that includes `getTarget()`, getter, and setter methods.

Then, the dependency graph is created based upon the analysis of the DOM, the hierarchical data structure that contains all the components available in the GUI. For every control we track the associated dependencies, using the dependency criteria defined for this application. The dependency graph is a directed acyclic graph (DAG), where the nodes represent the control's internal state and edges between nodes represent direct dependency relations between controls. Each node contains the control object copied from the original UI and the control's type.

Dependency is defined as a relation where a control `A`, through one of its methods, directly modifies one of the properties of another control `B` or even `B` itself. In such cases, B is dependent on A. Then, we argue that when `A` is executed, a subsequent execution of some control `B` is affected, then `B` must be executed so it can react to this change.

Algorithm 1 describes the function `CreateDGraph()` the process of building the dependency graph. This process executes only at the beginning of the GUI execution.

Algorithm 1 encodes the dependency relationships between all pairs of controls using a directed acyclic graph. Each node of this dependency graph represents exactly one of the reactive controls in the GUI. A node object contains a reference to the control object in the GUI and information about it such as its type and name or ID. If the control corresponding to some node (called the source) can affect the execution of some other node's control (called the destination), then the dependency graph includes a directed edge from the source node to the destination node. However, the algorithm does not allow the dependency graph to have cycles (which would correspond to an infinite update process). Function `createDGraph()`, which creates the dependency graph, is called only at the startup of the application, just after the GUI is built. In the case that a cycle is formed, the dependency is ignored by the system, and then this execution is handled as a regular event by the system.

ALGORITHM 1: Function CreateDGraph: Building the Dependency Graph from the DOM

```
if Form or Page is not empty and is IReactive then
    Q = empty queue;
    Tree = Document Object Model hierarchy;
    First = first control in the control list from the form or
      page;
    Enqueue the First in Q;
    while Q is not empty do
        P = Dequeue the next Control in Q;
        if P is not in the Dependency Graph then
            Insert P as a Node in the Graph;
        end
        while for each Control C in the list of targets of cont
          do
            if C is not empty and is IUpdatable then
                if C is not in the Dependency Graph then
                    Insert C as a Node in the Graph;
                end
                if Edge between C and cont does not exist
                  then
                    if Edge do not cause a cycle then
                        Create a Edge in the Graph between
                          source P and destination C;
                    end
                end
            end
        end
    end
end
```

Whenever a user interacts with a reactive control, the language's runtime system invokes that control's event handler as usual for event-driven systems. However, our approach modifies the event handler to call our function `UpdateGraph()` before executing the handler's other code. This function does a depth-first search (DFS) on the current dependency graph. If the function determines that any control in the UI structure has been modified, deleted, or inserted relative to the current dependency graph, then it updates the dependency graph accordingly. For this purpose, it compares every control in the DOM with the previous state stored in the dependency graph. Algorithm 2 shows the function `UpdateDGraph()` for the reanalysis of the graph, generation of the partial graph, and the execution of each object.

Algorithm 2 updates the dependency graph according to the type of changes made to the GUI:

New control inserted: The algorithm adds the new control as a new node in the dependency graph and encodes the new dependencies that arise from this insertion as new edges in the graph. The algorithm then recomputes the dependencies for all components that became dependent upon the new component.

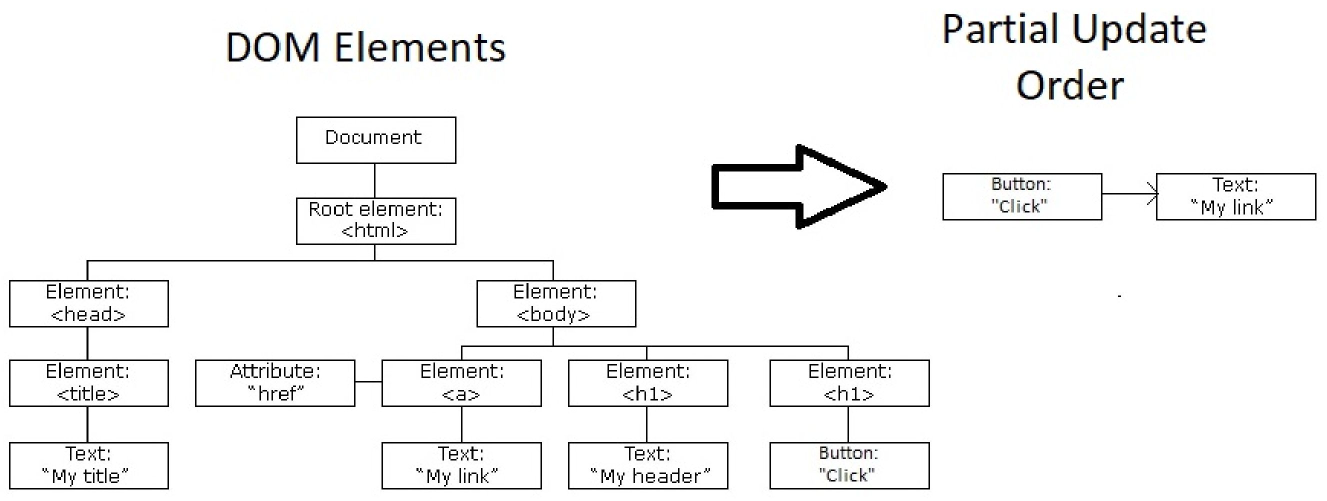

Figure 1: Diagram of the Partial Update Order Based on the Original User Interface

Control modified: The algorithm adds edges to or deletes edges from the dependency graph to reflect the new dependencies of the modified control.

Control deleted: The algorithm deletes the corresponding node and all its incoming and outgoing edges from the dependency graph. All the dependencies for all controls affected must be recomputed.

Once the dependency graph has been updated (if needed), the system traverses the graph to generate an update order. It begins with the reactive control that launches the event and considers all controls that are directly or indirectly affected by that control. (Figure 1 illustrates how our approach defines this execution order from the original structure.) The idea is to try to realize all the direct and indirect effects of the launching event within one cycle of the event-handling system. This "chain of execution" approach addresses the problem described in Section 2 by propagating the effects quickly through the GUI in an order that preserves the dependencies, thus decreasing the likelihood of inaccurate or misleading displays. The recomputation of the dependency graph does introduce some overhead, but, by only doing this once at the beginning of a related chain of executions, our approach seeks to minimize its impact on system performance. In many cases, the performance gain from executing a whole chain of controls at once should be greater than the overhead introduced by the computation of the dependency graph.

To enable the chain of execution behavior described above, the form or page must call the function `UpdateGraph()` as the first action in the event handler for every reactive control. Any form that is required to have this reactive behavior must implement the interface `IReactive` and provide an implementation of the `Update()` function. For each control that executes, its execution is redirected to `Update()`, this function identifies which control is being executed by its type and name. The `Update()` function then executes the reactive code respective to the control identified and after the execution is over, it returns to the `UpdateGraph()` execution so the next reactive execution can be handled.

We chose to develop our library using C# on the .Net framework. The primary reason for this choice is its support for interoperability; the same code using the same extensive library of GUI controls can be used in both Web and desktop GUI applications. This facilitates the testing approach described in Section 4. Another reason for the choice of the C#/.Net platform is its advanced object-oriented features and user-defined generic types, both of which promote code reuse in both the library and the testing framework. A third reason for the choice is the platform's metaprogramming and reflection facilities. These enable us to conveniently implement the library's analyses needed to build and update the dependency graph.

Along with our library, we developed a toolkit with several controls that extend the most popular controls from the .Net GUI framework. We developed reactive versions of `Button`, `TextBox`, `ListBox`, `ComboBox`, `Label`, and `RadioButton`. Each reactive control consists of a class that extends the original control and implements the `IUpdatable` interface. This interface includes getter, setter, and `getTarget()` methods, which must be implemented differently for each reactive control. The library also includes sets of controls for the Web and for the desktop, the interfaces `IReactive` and `IUpdatable`, the graph class, and the `dependencyAnalyzer` class encapsulating Algorithms 1 and 2.

The library's code is the same on both the Web and desktop platforms. However, there is a flag that indicates whether or not the library is being used on a Web-based system. Some small details of the implementation differ between the two platforms. For example, for the purposes of comparisons we use the attribute `name` for desktop controls and the attribute `id` for Web controls. We handle these differences by using conditional statements in the code.

ALGORITHM 2: Function UpdateDGraph: Reanalyze the DOM to Update the Dependency Graph and Create a Partial Update Queue

```
if Form or Page is not empty and is IReactive then
    Q = empty queue;
    Tree = Document Object Model hierarchy;
    First = reactive control that was executed;
    Enqueue the First in Q;
    while Q is not empty do
        Cont = Dequeue the first object in the queue;
        call C.getTarget() to update the target of controls;
        C1 = instance of Cont in the Graph, null if not found;
        if C1 is null then
            Insert Cont as a Node in the Dependency Graph;
            while for each target control P in Cont do
                if value of P is a control and is IUpdatable
                  and not null then
                    Insert P as a Node in the Graph;
                    if Edge between Cont and P do not cause a
                      cycle then
                        Create a Edge in the Graph from
                          Cont and P;
                    end
                end
            end
        end
        else
            Update the value of C1 in the Graph;
            while for each target control P in C1 do
                if value of P is control and is IUpdatable and
                  not null then
                    P1 = object equal to P in the Graph, null
                      if not found;
                    if P1 is null then
                        Insert P as a Node in the
                          Dependency Graph;
                        if Edge between C1 and P do not cause
                          a cycle then
                            Create a Edge from C1 and P;
                        end
                    end
                    else
                        if value of P is null or different from
                          P1 then
                            Update the value of P1 in the
                              Graph;
                        end
                    end
                end
            end
        end
        Breadth-First Search start with First to produce a
          valid partial Update Queue;
    end
end
```

Balancing the load between the code that is placed as a reactive code and the code that is placed as non-reactive code is the key aspect to maintaining the performance and accuracy on medium-to-large scale applications. In the following section, we describe our test methodology.

4 TEST METHODOLOGY

To evaluate our framework, we compare it to a similar environment that uses the Sodium library [2]. Sodium is a state-of-art Functional Reactive Programming (FRP) library implemented in several languages (e.g. C#, C++, Java, JavaScript, and Scala). It is based on the ideas promulgated by Elliot [8]. Sodium is a full FRP library providing functional combinators and abstractions like cells (which contains the value at any point of time) and streams (which are a sequence of events that can happen any time). Blackheath and Jones [2] argue that Sodium is a system with a strong semantics. By this they mean that the functions implemented in Sodium are based upon mathematical descriptions, delimited inputs and outputs, known internal mechanisms, and previously defined side effects.

Sodium provides a very good platform for FRP development. A particularly attractive feature is the ability to compose asynchronous streams using functional combinators. However, because Sodium's implementation is based on the Observer design pattern, we expect it to exhibit the shortcomings described in Section 2. Various researchers, such as Krouse [12], and Bregu et al. [3], have identified other shortcomings of Sodium. For example, in complex systems that integrate FRP and non-FRP code, FRP abstractions are prone to induce errors or high latency in non-FRP computations. This can lead to an unsafe state in applications, especially those running on the Java Virtual Machine or the .Net framework. Also, Sodium uses a large amount of memory to keep the underlying contextual information about the streams, especially because they are kept alive even when they stop to produce values.

For our comparisons, we chose to use self-completion forms and Web pages. We implemented each form using both Sodium and our library and then compared the performance of the two implementations. By a self-completion form, we mean a form in which the user supplies some initial information and then asks the system to populate dependent fields in the form from what has already been supplied. In each case, we constructed two different implementations: one on a Web page and one on the desktop.

For all of our tests, we implemented an example self-completion form that uses several reactive controls from our reactive library: `ReactiveButton`, `ReactiveTextBox`, `ReactiveComboBox`, `ReactiveListBox`, `ReactiveLabel`, and `ReactiveRadioButton`. We made the following reactive:

- the click events of the `Button` and `RadioButton` controls
- the `textChanged` properties of the `textBox` and `Label` controls
- the `selectChanged` properties of the `ComboBox` and `ListBox` controls
- the `Visible` and `Active` properties of all of the above controls

We scattered instances of these reactive controls across the example form (or page) and then linked them to each other. For example, when a button is clicked, it uses the value of a text box to populate two other text boxes with predefined values.

We use the .Net `StopWatch` class to measure the time spent filling a form. This class emulates the behavior of a real stopwatch, enabling the developer to start and stop it as needed. We start it at the first click on the form (the first modification in the cycle) and stop it as the last control is filled. The property `Elapsed` from the class `StopWatch` gives the amount of time in milliseconds spent from start to stop. The Microsoft documentation [17] of the class `StopWatch` claims that the default method for counting time is the timer ticks from the system timer. If the operating system or hardware supports a high-resolution performance counter, then the `Stopwatch` class uses that counter to measure the elapsed time.

We devised the following metrics to measure the accuracy of the result for each box inside the form:

- Measure the length of the latency between the user action (write something, click enter, press a mouse button) and the desired state being seen on the screen. For this application, we intend to measure the time needed to get the whole Web page into the desired state from the triggering action.
- Count the number of the errors detected for a sequence of complex interactions. By an error we mean a situation where one of the units is executed before one or more of its dependencies and that causes the units to use an out-of-date or missing value. This incorrect order either causes a total failure of the unit or a temporary mistaken state of the unit.
- Count the average number of errors on a test where errors are detected. To measure this, we plan to determine how many components are incorrectly ordered on a test where are errors detected. Then we count the total number of errors, count the total number of runs that reported an error, and then compute the average.

We ran all the tests on an Intel Core i5 5300U 2.3 GHz processor with 8 GB RAM and an Intel HD 5500 graphics card, running the Windows 10 64-bit operating system. We used Visual Studio 2019 for development with C# 8.0 and .Net framework version 4.8. We also used Sodium 2.0 for the Sodium tests.

5 RESULTS AND ANALYSIS

We conducted tests using three different test scenarios:

(1) A shopping list where new items (each item has a price that the user must input) are added, updating the sum of the prices, the sales tax (7% for Mississippi), and the total cost with the tax included.
(2) A calculator for geometric shapes. It calculates and self-completes the area, perimeter, and volume. It also supports conversions from U.S. to metric units and vice versa (e.g. from feet to meters and from meters to feet, etc.).
(3) A user form holding medical information.

We categorize our results in two ways: performance and accuracy.

Our system outperformed the Sodium system consistently across all test scenarios. The only downside of our system was the startup time. The implementations using our system took an average of 0.35 seconds more to start up than the Sodium applications. This can be explained by the overhead incurred by the creation of the dependency graph and the analysis of all the controls. This is the most time-consuming step in the execution of our library code. Table 1 illustrates the average startup time for each implementation using both Sodium and our library.

Table 1: Startup Time for Each Test Scenario

Platform	Scenario 1	Scenario 2	Scenario 3
Sodium	29.58 ms	30.65 ms	31.12 ms
Our Tool	51.24 ms	55.45 ms	58.28 ms

For each test, we calculated the time needed to execute the full self-completion routine. We started the stopwatch on the first button click and stopped it when the last control had been executed. We checked during the execution to make sure that no exceptions or errors were raised, because, in such a case, the execution would never reach the last control.

On average, for the three test scenarios, our implementation completed the form in 10% to 20% of the time that the Sodium implementation took for the same form. The average is taken for each of the 50 executions. On each execution, the form was opened and self-completed, then the information cleared from the form before the execution was repeated. Figure 2 shows the average performance graph in each of the executions for the three test scenarios in both implementations.

With respect to accuracy, our system outperformed the Sodium implementation. We measured accuracy by comparing the intended final state of the self-completion form (i.e. the state of each control) determined beforehand with the actual final state generated by the self-completion form.

This test case seeks to highlight the effect that execution order has on achieving a correct final display. Sodium does not consider the dependencies in scheduling updates. Our library seeks to guarantee that the dependencies between controls are not violated by the actual execution order—that only up-to-date and accurate information is used to fill in the form at all points during execution. This works works like dominoes falling. If a later one falls before a previous one, the inaccurate result may be perceived by an observer.

Table 2 shows the result of the first test scenario, which implements a shopping list user interface for both systems.

Table 2: Test Results for Scenario #1

Platform	Total Cycles	Total Errors	Avg Errors / Cycle	Latency in Cycles
Sodium	50	8	2	1
Our Tool	50	3	1	1

Table 3 shows the result of the second test scenario, which implements a geometric self-completion calculator user interface for both systems.

Table 4 shows the result of the third test scenario, which implements a metric converter user interface for both systems.

The Sodium-based implementation behaved similarly to our library. It yielded errors after a wave of updates in 15% of the tests.

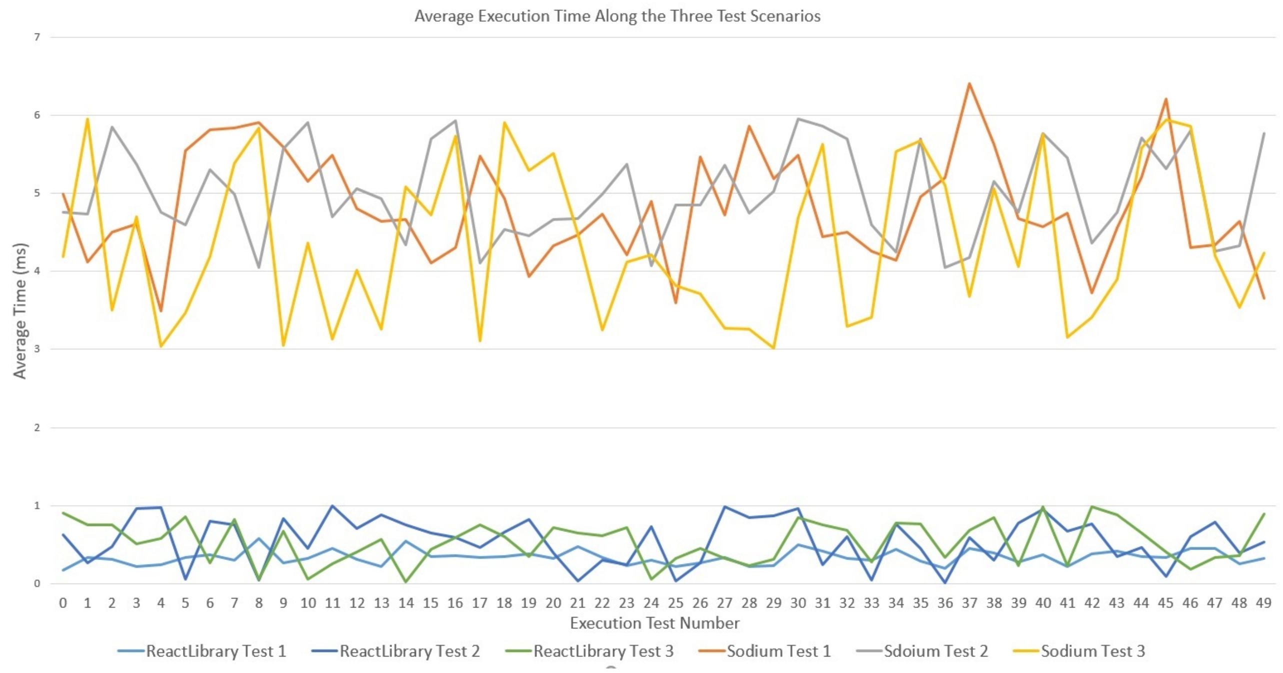

Figure 2: Average performance graph - All 3 Scenarios

Table 3: Test Results for Scenario #2

Platform	Total Cycles	Total Errors	Avg Errors / Cycle	Latency in Cycles
Sodium	50	6	1	1
Our Tool	50	2	1	1

Table 4: Test Results for Scenario #3

Platform	Total Cycles	Total Errors	Avg Errors / Cycle	Latency in Cycles
Sodium	50	7	1	1
Our Tool	50	3	1	1

Our implementation yielded errors in less then 10% of the tests. For both, it took one or two waves on average to restore the structure to a valid state as shown in Tables 2, 3, and 4.

With respect to performance, the graph depicted in Figure 2 shows that our framework performed the same task in 10% to 20% of the time in milliseconds that Sodium took. The execution time for our implementation was less than 1 second while Sodium's execution time was up to 6 seconds. Our implementation was substantially faster than the Sodium implementation. The performance of the three tests by our library are near the bottom of the graph. The tests performed by Sodium are near the top. The graphical distance between give the magnitude of the difference between both of the performances.

How can we explain the performance differences? Both the Sodium and regular .Net implementations are based on asynchronous event-handling systems as described in Section 2. Because of the way asynchronous systems work, there is a time lag between one action and the next. Although Sodium implements reactivity and significantly tames the problems of asynchronous calls, the way the system works behind the curtain limits its effectiveness for the kinds of applications this research addresses. Sodium connects events to values but not events to events. One stream does not connect directly with another; each event happens on a single control only. However, our system links an event directly with its dependent events, executing one directly after the other, avoiding a significant time lag between the event executions.

The tests described in this section demonstrate the effectiveness of our approach. Our library outperformed Sodium, a state-of-the-art reactive library. It alleviated the performance problems caused by the asynchronous nature of the event-handling approach used by .Net. Asynchronous calls are important for user interfaces because they enable the system to continue responding to the user while waiting for a result from a previous command. However, exclusive use of asynchronous calls means that the system has no mechanism for defining when a response will appear. Compared to Sodium-like systems, our library produces a faster and smoother experience for the class of problems it was designed to solve—applications with dependencies between events and the need to initiate a whole chain of executions as if it is a single execution. The tests also demonstrate that our library can give more accurate results than Sodium for a variety of user interface applications, while yielding small performance improvements.

6 DISCUSSION

Most papers focus primarily on how to build programs by generating and relating different parts of the source code. Czaplicki [6, 7], Foust et al. [9], Krishnaswami [11], Reynders et al. [18], and Salvaneschi et al. [19] present reactive implementations of GUIs. The difference between these approaches and ours is that our approach specifically builds a dependency graph and periodically updates it. This allows our approach to work well in dynamic environments with high unpredictability. Foust et al. [9] describes a reactive model that can be used to generate a GUI that satisfies the dataflow constraints (i.e. data dependencies between GUI components). This work addresses the same problem as our work but from the opposite direction. Czaplicki [6, 7] is developing Elm, a JavaScript-based language for creating dynamic GUIs and Web pages. However, Elm is evolving in a different direction, even though it was initially based on reactivity in general.

Our previous work [15, 16] presents a dependency-based, reactive approach to game development that targets Unity3D applications. That solution is similar to the one we present in this paper, but it is hosted in a different environment and addresses a particular problem encountered in game development for Unity3D.

The approach in this paper differentiates itself from the above in that it aims to solve what we consider the core of the problem: controlling the inherent delay and lack of control and inaccuracy resulting from the use of asynchronous execution in current user interface technologies. By restoring some synchronicity to the asynchronous executions, our approach increases the performance while decreasing the number of temporary inaccuracies.

7 CONCLUSIONS AND FUTURE WORK

The primary contributions of our research are as follows.

- We define a method for extracting the dependencies between the reactive controls in a UI and using them to generate an event-handling order that can adapt to changes in the UI's structure as the program executes.
- Our approach improves responsiveness and performance and results in a more accurate behavior. It should be possible to integrate our approach into similar systems that are based on similar built-in object hierarchies.
- Our framework has an advantage over other reactive libraries (e.g. Sodium) in that these frameworks do not work at a sufficiently low level. Internally, they still rely on asynchronous calls. This effectively makes a reactive system that only works on top of the asynchronous core, which means that whenever the execution spreads across multiple events, each of these events is handled asynchronously.

In future work, we plan both to consider additional testing scenarios and to add additional tests that include the unmodified .Net framework and several other reactive libraries (e.g. Rx.Net, React.js, and React.js + Redux). We also plan to identify and document the common threads running through this and our previous work [15, 16] and apply what we learn to other similar problems. Of course, each new application may have different concepts of dependency, different original structures, and different default execution orders. A version of our reactive library should enable these applications to respond expeditiously to interactions and adapt quickly to ever-changing environments resulting with similar gains in performance and accuracy.

ACKNOWLEDGMENTS

The first author's work was supported by CAPES, Coordination for Enhancement of Academic Level Individuals—Brazil. We also thank the anonymous referees for their suggestions, which helped us improve this paper.

REFERENCES

[1] J. Bishop and N. Horspool. 2004. Developing Principles of GUI Programming Using Views. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04)*. ACM, Norfolk, VA, USA, 373–377.
[2] S. Blackheath and A. Jones. 2016. *Functional Reactive Programming*. Manning, Shelter Island, NY.
[3] E. Bregu, N. Casamassima, D. Cantoni, L. Mottola, and K. Whitehouse. 2016. Reactive Control of Autonomous Drones. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16)*. ACM, Singapore, 207–219.
[4] K. Chandy and J. Misra. 1988. *Parallel Program Design: A Foundation*. Addison Wesley, Boston.
[5] G. Chupin and H. Nilsson. 2019. Functional Reactive Programming, Restated. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages (PPDP '19)*. ACM, Porto, Portugal, Article 7, 14 pages.
[6] E. Czaplicki. 2012. Elm: Concurrent FRP for Functional GUIs. Senior thesis, Harvard University, Cambridge, MA.
[7] E. Czaplicki and S. Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, Seattle, WA, USA, 411–422.
[8] C. Elliott. 2009. Push-Pull Functional Reactive Programming. In *Proceedings of the 2nd SIGPLAN Symposium on Haskell (Haskell '09)*. ACM, Edinburgh, Scotland, 25–36.
[9] G. Foust, J. Järvi, and S. Parent. 2015. Generating Reactive Programs for Graphical User Interfaces from Multi-way Dataflow Constraint Systems. *SIGPLAN Notices* 51, 3 (Oct 2015), 121–130.
[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Boston.
[11] N. Krishnaswami. 2012. Semantics for Graphical User Interfaces. In *Proceedings of the 8th SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. ACM, Philadelphia, PA, USA, 51–52.
[12] S. Krouse. 2018. Explicitly Comprehensible Functional Reactive Programming. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2018)*. ACM, Boston, MA, USA, 5.
[13] K. Malawski. 2016. *Why Reactive?* O'Reilly Media, Sebastopol, CA.
[14] Z. Manna and A. Pneulli. 1992. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, Berlin.
[15] J. Marum, J. Jones, and H. Cunningham. 2019. Towards a Reactive Game Engine. In *Proceedings of the 50th IEEE SouthEastCon*. IEEE, Huntsville, AL, USA, 8.
[16] J. Marum, J. Jones, and H. Cunningham. 2020. Dependency Graph-based Reactivity for Virtual Environments. In *Proceedings of the IEEE VR 2020 Workshop on Software Engineering and Architectures for Interactive Systems (SEARIS)*. IEEE, Atlanta, GA, USA, 8.
[17] Microsoft. 2019. .Net Framework Developer Documentation: StopWatch Class. https://docs.microsoft.com. Accessed Feb. 26, 2020.
[18] B. Reynders, D. Devriese, and F. Piessens. 2017. Experience Report: Functional Reactive Programming and the DOM. In *Companion to the First International Conference on the Art, Science and Engineering of Programming (Programming '17)*. ACM, Brussels, Belgium, Article 23, 6 pages.
[19] G. Salvaneschi, S. Amann, S. Proksch, and M. Mezini. 2014. An Empirical Study on Program Comprehension with Reactive Programming. In *Proceedings of the 2nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, Hong Kong, China, 564–575.
[20] G. Salvaneschi, A. Margara, and G. Tamburrelli. 2015. Reactive Programming: A Walkthrough. In *Proceedings of the 37th International Conference on Software Engineering: Volume 2 (ICSE '15)*. IEEE, Florence, Italy, 953–954.
[21] J. Silva, J. Saraiva, and J. Campos. 2009. A Generic Library for GUI Reasoning and Testing. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC '09)*. ACM, Honolulu, HI, USA, 121–128.