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

João Paulo Oliveira Marum, H. Conrad Cunningham, and J. Adam Jones
Department of Computer and Information Science
University of Mississippi
jmarum@acm.org, hcc@cs.olemiss.edu, jadamj@acm.org

July 17, 2020

Abstract

This note is an addendum to the paper "Unified Library for Dependency-graph Reactivity on Web and Desktop User Interfaces" we presented at ACM SouthEast 2020. In addition to the comparisons reported in the full paper, we have also compared our approach to a pure .NET system and to a system built on the reactive library Rx.NET.

1 .NET and Rx.NET Tests

This note reports an extension of the research that we presented in our paper "Unified Library for Dependency-graph Reactivity on Web and Desktop User Interfaces" at the ACM SouthEast 2020 Conference [3]. In addition to the comparisons reported in the full paper, we have also compared our platform to:

1. a pure .NET system
2. a system that builds on the reactive library Rx.NET [4]

The first comparison sought to determine whether our platform is an improvement over the baseline .NET system. The second comparison enabled us to compare the performance of the two reactive libraries, Sodium [1] and Rx.NET, to each other as well as to our platform.

Reactive Extension for .NET (Rx.NET) is a library for developing asynchronous and event-based programs using observable collections and LINQ-style query operators to implement reactive programming for .NET applications on multi-platform systems. Malawski [2] argues that Rx.NET alleviates the side-effects of asynchronous execution in .NET systems. Rx.NET represents any data sequence from .NET as an observable stream. A *stream* is a theoretically infinite sequence of events, where each event is represented by the state of a variable after that event. It is defined as an *observable sequence* because each state can be evaluated (observed) by itself or inside of the sequence. An application can subscribe to these observable streams to receive asynchronous notifications as new data arrives. Rx.NET treats those streams as unbounded lists that can be iterated though, analyzed, and understood the same as any other object in .NET. It is important to note that dependencies between components (whenever component A raises an event, a sequence of actions happens and affects component B and others) in Rx.NET must be explicitly specified by the programmer.

Our tests emulate the behavior shown in the Figure 1. These tests were developed to show that our

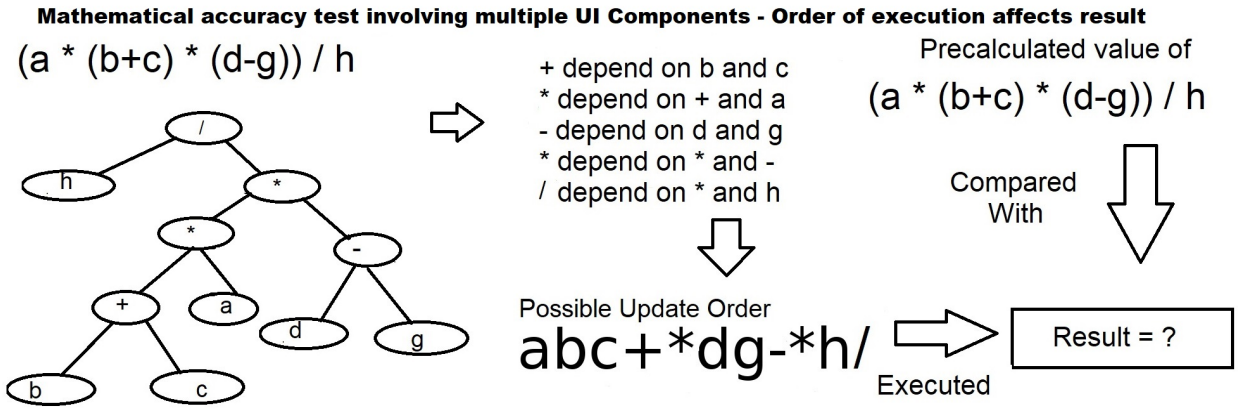

Figure 1: Diagram of the test behavior.

platform can correctly evaluate mathematical expressions when the values of the variables referenced are distributed among components of the user interface. In such a case, the order of execution can affect the final result. If our system can ensure that the accurate order will be respected in every cycle, than the accuracy of the entire operation can be ensured. Since operations in a user interface can often be decomposed into smaller interactions between components, if we can decrease the inaccuracies derived by the misordering of the executions, each one of these smaller interactions will be more accurate, thus making the entire operation more accurate.

For the pure .NET and Rx.NET tests, we used the same set of test scenarios that we used in the original paper to compare Sodium with our platform. These include:

1. A shopping list into which the prices of new items can be added, the sum of the prices updated, the sales tax (7% for Mississippi) calculated, and the total cost computed.

2. A calculator for geometric shapes that calculates and self-completes the values for the area, perimeter, and volume and supports conversions between U.S. and metric units (e.g. feet to and from meters).

3. A user form holding medical information.

Platform	Scenario 1	Scenario 2	Scenario 3
Rx.NET	27.58 ms	28.65 ms	29.12 ms
.NET	21.24 ms	20.45 ms	22.28 ms
Our Tool	51.24 ms	55.45 ms	58.28 ms

Table 1: Startup time for each test scenario

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

The implementations using our platform took an average of 0.30 seconds more to start up than the Rx.NET applications and 0.35 seconds more than the pure .NET application. 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 platform. Table 1 illustrates the average startup time for each implementation using both Rx.NET and our platform.

Against the Rx.NET implementation, our platform filled the entire form in an average of 30% of the time that the Rx.NET implementation took to do the same task. Figure 2 shows an average performance graph

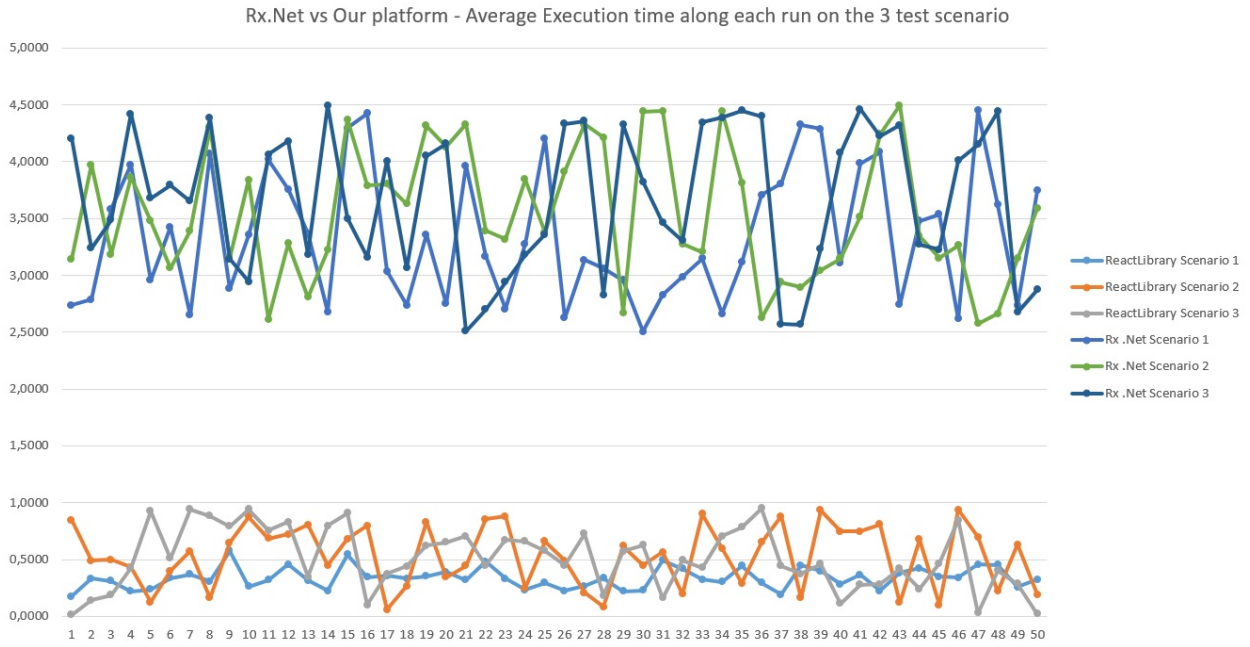

Figure 2: Average performance graph - All 3 Scenarios. Rx.NET vs Our platform.

over each one of the executions on the three scenarios.

Against the original .NET implementation, our platform filled the entire form in an average of 50% of the time that the .NET implementation took to do the same task. The Figure 3 below shows an average performance graph over each one of the executions on the three scenarios.

With respect to accuracy, our system outperformed all the other implementations. 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 a chain of falling dominos. If a later one falls before a previous one, the inaccurate result may be perceived by an observer.

Platform	Total Cycles	Total Errors	Avg Errors per Cycle	Latency in Cycles
Rx.NET	500	9	2	1
.NET	500	12	5	3
Our Tool	500	3	1	1

Table 2: Test Results for Scenario #1

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

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

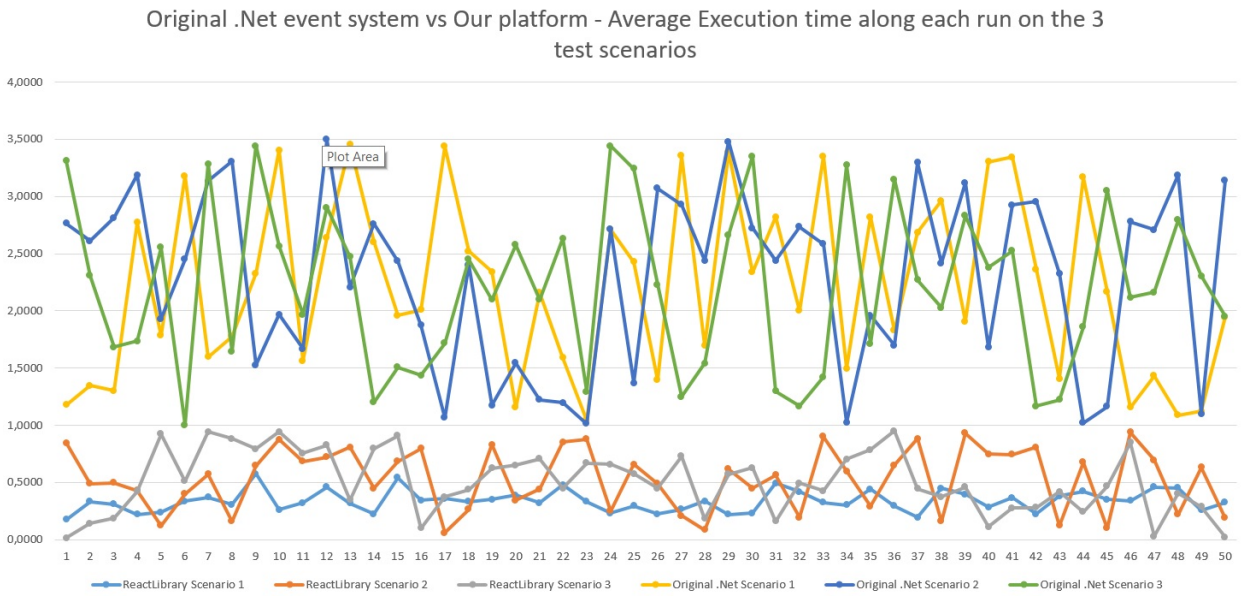

Figure 3: Average performance graph - All 3 Scenarios. Original .NET vs Our platform.

Platform	Total Cycles	Total Errors	Avg Errors per Cycle	Latency in Cycles
Rx.NET	500	8	2	1
.NET	500	20	1	5
Our Tool	500	2	1	1

Table 3: Test Results for Scenario #2

Platform	Total Cycles	Total Errors	Avg Errors per Cycle	Latency in Cycles
Rx.NET	500	10	2	1
.NET	500	16	1	4
Our Tool	500	3	1	1

Table 4: Test Results for Scenario #3

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

How can we explain the performance differences? The regular .NET implementations are based on asynchronous event-handling systems as described in Section 2 of the original paper. Because of the way asynchronous systems work, there is a time lag between one action and the next. The .NET event system relies solely on asynchronous calls handled by its event system. 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.

A disadvantage of Rx.NET relative to the performance of our platform is that Rx.NET represents each asynchronous call, or set of asynchronous calls, by self-contained occurrences in the system life span. When these reactive calls involve the chain execution of multiple controller event-handlers, Rx.NET treats reactively the internal mechanisms of each execution but between the executions, Rx. NET handles it as

the original .NET event system would treat them, through asynchronous callbacks.

References

[1] S. Blackheath and A. Jones. *Functional Reactive Programming*. Manning, Shelter Island, NY, 2016.

[2] K. Malawski. *Why Reactive?* O'Reilly Media, Sebastopol, CA, 2016.

[3] J. Marum, H. Cunningham, and J. Jones. Dependency graph-based reactivity for virtual environments. In *Proceedings of the ACM Southeast Conference (ACMSE 2020)*, Tampa, FL, USA, April 2020. ACM.

[4] reactivex.io. ReactiveX: An API for asynchronous programming with observable streams, 2020. URL `http://reactivex.io/`. Retrieved July 13, 2020.