1-1-2021

DEPENDENCY-BASED REACTIVE CHANGE PROPAGATION DESIGN PATTERN APPLIED TO ENVIRONMENTS WITH HIGH UNPREDICTABILITY

João Paulo Oliveira Marum
University of Mississippi

Recommended Citation

Oliveira Marum, João Paulo, "DEPENDENCY-BASED REACTIVE CHANGE PROPAGATION DESIGN PATTERN APPLIED TO ENVIRONMENTS WITH HIGH UNPREDICTABILITY" (2021). *Electronic Theses and Dissertations*. 2122.
https://egrove.olemiss.edu/etd/2122

DEPENDENCY-BASED REACTIVE CHANGE PROPAGATION DESIGN PATTERN APPLIED TO ENVIRONMENTS WITH HIGH UNPREDICTABILITY

A Dissertation
presented in partial fulfillment of requirements
for the degree of Doctor of Philosophy
in the School of Engineering
at The University of Mississippi

by

JOÃO PAULO OLIVEIRA MARUM

August 2021

ABSTRACT

Transitional turbulence is a period of chaotic or unreliable variation in the state of a software system that results from changes in the system's interconnected components. During these periods of instability, an external observer of the system's state may "see" erroneous results. This is a problem that can affect visual user interfaces such as those in virtual and augmented reality applications and desktop or Web GUIs. In this research, we study two different reactive applications developed in C# on .NET. We reduce the transitional turbulence by augmenting the base applications with a dependency-graph-based event scheduling approach. The first study investigates desktop and Web GUIs. The second study investigates virtual and augmented reality applications built on the Unity3D game engine. The two studies use similar approaches, but both are somewhat embedded in the details of their applications and implementation platforms.

In addition to presenting the two augmented applications, this dissertation characterizes the problem and its solution in a more general way. To do so, we use a design pattern to state the general problem-solution pair and enable it to be reused in similar contexts. We examine the two studies to identify their commonalities. We then unify the approaches by writing a new design pattern named Dynamically Coalescing Reactive Chains (DCRC). This dissertation both presents the new design pattern and records the systematic process we used to write it. To evaluate the design pattern and its usage, we apply it to the application in the first study as if we were approaching the application anew. The DCRC pattern facilitates the use of our approach for other applications and technologies and lays the foundation for further research on transitional turbulence and related software architecture issues.

DEDICATION

This dissertation is dedicated to my wife Selva, who loved me and came with me when this was just a crazy dream. I also dedicate this dissertation to my deceased Aunt Helena Marum, who told me to get out of the couch and pursue my dreams. Thank you for all the afternoons we shared. It is also dedicated to our forever baby Joy, that died in the middle of this journey. Your happiness, sweetness, and incredible intelligence helped me to get up every day when I was down.

ACKNOWLEDGEMENTS

First and most importantly, I thank God for the strength, wisdom, and for opening all the doors that took me to the end of this journey. I express my deepest appreciation to my two co-advisors, Dr. H. Conrad Cunningham and Dr. J. Adam Jones, and to my committee members, Dr. Philip J. Rhodes and Dr. Cesar Rego. I could not have financed my studies without the scholarship provided by CAPES, Coordination for Enhancement of Academic Level Individuals—Brazil and also the graduate assistantship provided by the Department of Computer and Information Science, represented by the department chair Dr. Dawn Wilkins. In addition, I must acknowledge the support of my family, my wife Selva Marum, our two sons Cesar and Caio, and my animals Lola, Joy, Talita and Athos. You made this dissertation possible with your everlasting love and support. In addition, I acknowledge my parents Abrão and Maria Bernadete and all of our family members and close friends from Brazil (honorable mentions to Airton, Rubens, and my Aunt Helena Marum) in the hope that you all know that I carry all of you with me throughout this amazing journey. I also acknowledge all the incognito people that somehow helped in the process of bringing to me the fuel that propelled me through all the nights spent for this dissertation, all the hard workers throughout this world who brought me the coffee. Without it there would be no dissertation. Lastly, I acknowledge the friendship and support from my fellow doctoral students and faculty members from the department and also each person I came in contact with throughout all these years.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

A visual user interface must respond expeditiously to user actions and display their effects accurately. This is especially important for virtual and augmented reality applications, but it is also important for desktop and Web applications. Each of these applications "interacts with its environment on an ongoing basis" (Chandy and Misra, 1988). It reacts to a stream of events, where an event may be a stimulus from the external environment (such as a user movement) or from the computational environment (such as a notification that some software component changes its state).

When the handling of an event affects the state of one component of a system, that component may affect several other components. Each of these may, in turn, affect others, and so forth as the effects ripple throughout the system. It may take several update cycles for the states of all components to be updated and the system to reach a stable state. This period of *transitional turbulence* (Lorenz, 1963)—or glitchiness (Cooper and Krishnamurthi, 2006)—can result in inconsistencies in the visible state if the display must render new frames before stability is reached. The user (at least temporarily) perceives the system as unreliable and inaccurate. As a result, programmers must take appropriate countermeasures to maintain an accurate external state. For example, they can ignore updates that occur in the wrong order or that use outdated inputs since these would result in incorrect observable states.

To alleviate this transitional turbulence problem, we developed a novel reactive (Bainomugisha et al., 2013) approach. The approach 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 use our approach to the transitional turbulence problem for two case studies. In Chapter 3, we address .NET-based dynamic user interface for both desktop and Web applications. In Chapter 4, we address virtual and augmented reality applications for the Unity3D game engine (which itself builds on the top of .NET). All applications in both case studies share a context that does not currently allow programmers to specify the update order for the units at runtime.

In Chapter 5, we analyze these two case studies, extract their common features, and systematically document our general approach. In the generalization, we focus on applications that are structured according to the implicit invocation architectural pattern (Shaw et al., 1995) and codify our general approach as a design pattern (Buschmann et al., 1996, 2007). The overall research question for this dissertation as follows:

> *Can we mitigate the transitional turbulence in an implicit invocation-based systems by applying a technology-independent formal description such as a design pattern created by highlighting common aspects of similar applications?*

To answer this overall research question, we consider more specific research questions for the research reported in each chapter. These include:

Chapter 3, the .NET dynamic user interface case study: *Can dependency graph-based execution reordering and self-adjusting state recomputation be used to augment the implicit invocation built-in event system resulting in reduced transitional turbulence execution inconsistencies and increased accuracy while maintaining performance in the chained execution of multiple control's event handlers specifically found in Web and desktop graphical user interfaces?*

Chapter 4, the Unity3D virtual and augmented reality case study: *Can dependency graph-based execution reordering and self-adjusting state recomputation be used to augment the implicit invocation game loop resulting in reduced transitional turbulence execution inconsistencies and increased accuracy while maintaining performance on graphical applications built on game engines?*

Chapter 5, the design pattern development: *How can we codify the transitional turbulence mitigation approach taken in Chapters 3 and 4 as a general, technology-independent design pattern?*

The two case studies address two different but related problems involving implicit invocation architectures and devise two similar solutions using different technologies. Both solutions work by augmenting the normal event processing mechanisms with new software mechanisms that use the dependency graph.

In the first case study (in Chapter 3), we aim to augment Web and desktop graphical user interfaces (GUIs) implemented in C#. The approach builds the dependency graph by analyzing the relationships among the GUI controls (Marum et al., 2020a). Many effects that had previously been spread across the rendering of multiple frames all now occur within a single frame. By conducting a set of experiments, we show that our approach improves performance and results in a more accurate behavior in comparison to implementations using the Sodium (Blackheath and Jones, 2016) and Rx.NET (reactivex.io, 2020) reactive programming libraries and the built-in .NET event system for user interfaces.

In the second case study (in Chapter 4), we aim to augment virtual and augmented reality applications implemented using the Unity3D game engine and C# (Marum et al., 2020c). Our augmentation, as described in Chapter 4, is similar to the one described above except that it addresses similar issues in Unity3D's existing object hierarchy (Unity Technologies, 2019). The approach builds the dependency graph by analyzing the relationships among the Unity3D game components. If Unity3D's object hierarchy changes, the approach recomputes the dependency graph. By reordering the events based on the dependencies, the approach removes many inconsistencies without degrading the performance of the system. By dynamically reacting to changes in the object hierarchy, the approach can smoothly handle relatively complex applications. By conducting a set of experiments, we show our approach performs better than both an unmodified Unity3D application and a similar application developed using the reactive library UniRx (Kawai, 2014).

Our general practical contribution is the understanding that the increased control of the update cycle improves the responsiveness of the update to changes in the structure and ensures a more predictable result. To answer the overall research question, we build a reactive framework around the dependency relationships among the objects in the user interface's structure. We define this general approach using a design pattern that we name Dynamically Coalescing Reactive Chains (DCRC). The DCRC pattern's practical objective is to guarantee the execution order of the components' update functions in applications based on the implicit invocation architecture pattern (Shaw et al., 1995) without degrading the performance while preserving the correctness and increasing the predictability of order-sensitive operations. To ensure that our final design pattern still fits into the original solution definition in the case studies, we reapply the design pattern to .NET GUI application as if we are trying to solve it anew.

The remainder of this dissertation is structured as follows. Chapter 2 defines background concepts that we use throughout this dissertation. Chapters 3 and 4 report on the research related to the case studies described above. Similarly, Chapter 5 reports on the research related to the design pattern development. Finally, Chapter 6 answers the overall research question, reviews the contributions of this research, and identifies possible future work.

CHAPTER 2

BACKGROUND

Before we describe the two case studies in Chapters 3 and 4, this chapter defines background concepts that we use throughout this dissertation. We start with reactive programming semantics. Understanding how reactive programming works is essential to the development of the two applications studied in Chapters 3 and 4 and to writing the design pattern description in Chapter 5.

2.1 Reactive Programming Semantics

Chandy and Misra (1988) and Manna and Pneulli (1992) define a reactive program (RP) as software that engages ongoing interactions with its environment. Reactive programming (RP) focuses on how software reacts to changes in a system's state. These changes can be caused by interactions among the software components or by an external actor such as a user or another software system. The developer must determine the chain of events caused by a particular change. Throughout the execution of the application, the program reacts to the changes as defined by the developer. Operating systems and embedded systems are examples of reactive programs that do not terminate.

Using a spreadsheet as an example, Westberg (2017) describes the RP approach as follows:

> When the value of a specific cell is changed, all the dependent cells to that cell are instantly updated as a function of that change. This type of model makes it possible to have sequences of dependent values and events. If object `C` is dependent on `B`, while `B` is dependent on `A`, then, if `A` changes, `B` is reevaluated, followed by `C`. Changing object `A` creates a chain of reactions.

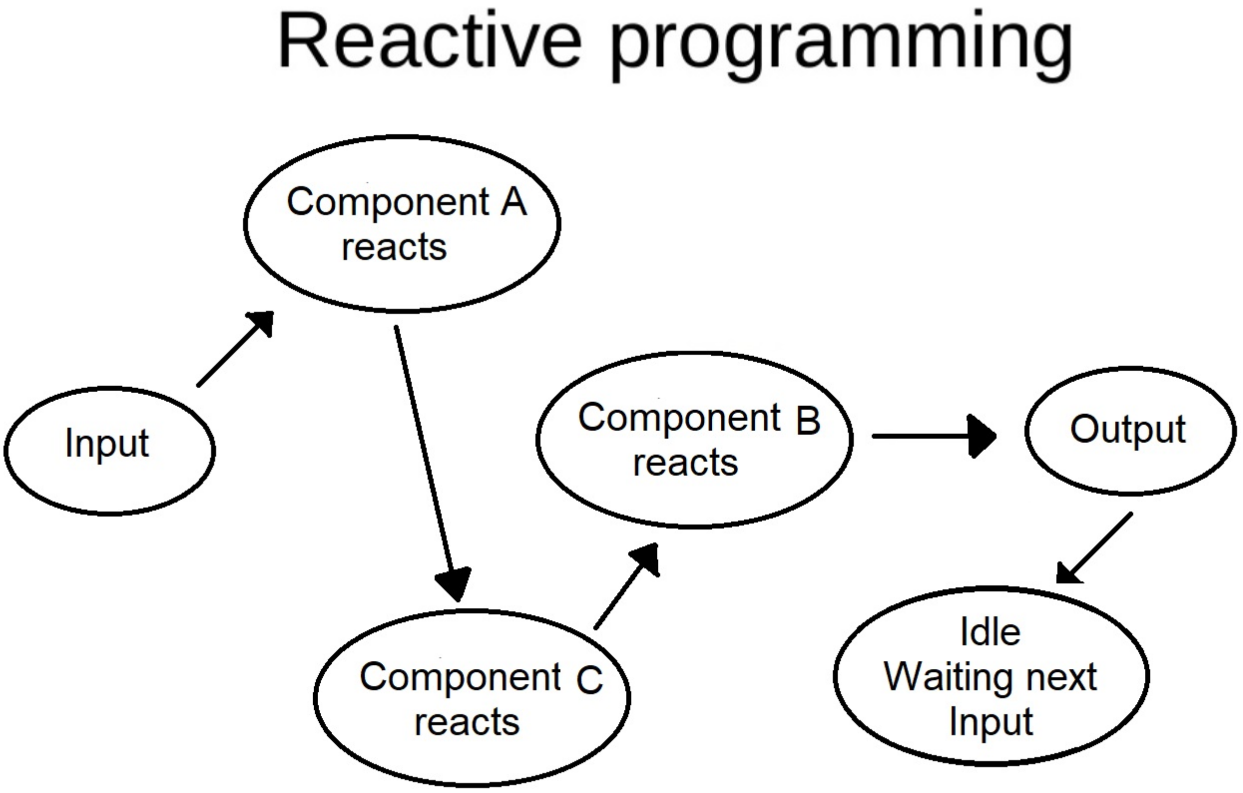
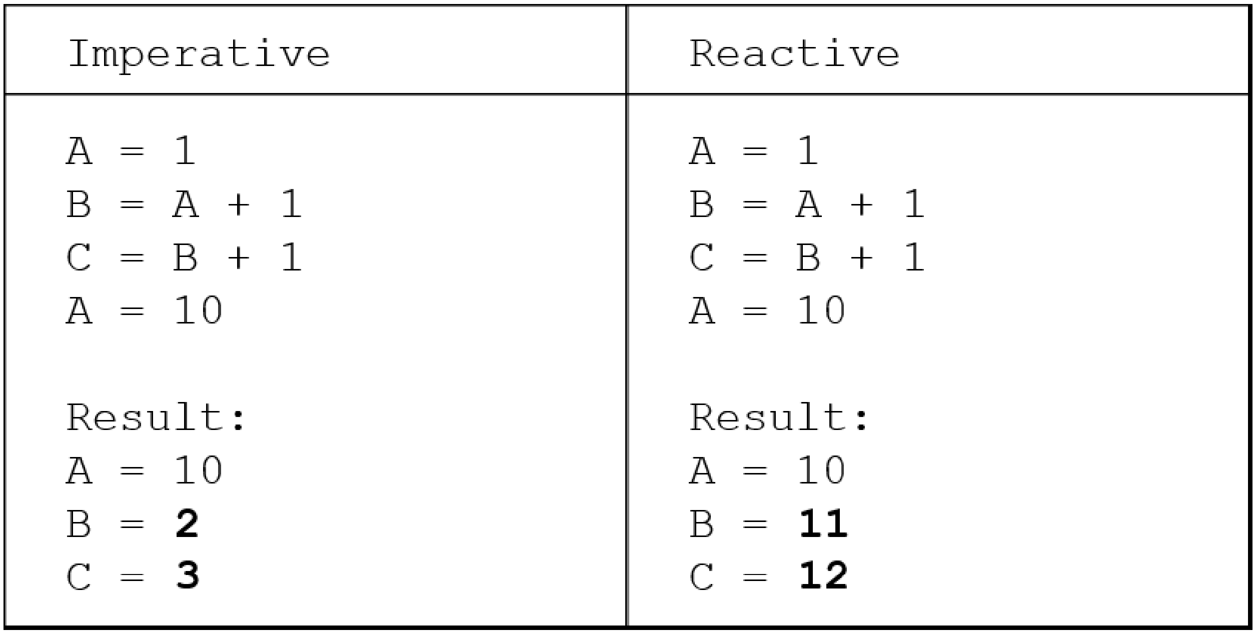


Figure 2.1. Illustration of the reactive behavior.

A change in the value of `A` causes `B` to be recomputed based on `A`'s new value. A change in the value of `B`, in turn, causes `C` to be recomputed because of the changes in the value of `B`. The changes thus ripple through the entire graph of dependencies. It uses a push-based or event-driven model of computation, where the environment rather than the program determines the speed at which the program interacts with the environment. Figure 2.1 illustrates this behavior.

The underlying model of spreadsheets stores the connections between the cells, so when one cell changes, the other cells that use the first cell's value to calculate their own results also update, and then the resulting reactions spread out until all the values change accordingly. This does not say that all reactive programming is as simple as a spreadsheet. For reactive programming, we aim to extract and store the underlying dependencies of complex software systems where those dependencies are not as simple or visual as in a



Imperative	Reactive

A = 1
B = A + 1
C = B + 1
A = 10
 	
Result:
A = 10
B = **2**
C = **3**
	A = 1
	B = A + 1
	C = B + 1
	A = 10
	
	Result:
	A = 10
	B = **11**
	C = **12**

Figure 2.2. Spreadsheet's auto-update functionality.

spreadsheet. Figure 2.2 shows how a reactive paradigm would recompute the calculations based upon a change in the inputs.

For example, if a state can be described with a finite set of variables and their values, then a transition gives a beginning state, some kind of stimulus from the environment (input), and ensuing changes to the environment (output), and the state (next state). An execution of a sequential program/process consists of an infinite sequence of states beginning with a valid initial state with adjacent states being associated with a tuple in the transition relation.

An execution of a set of sequential processes is often given as the nondeterministic interleaving of the sequences for the processes in the set. The execution of a transition in one process might change the state of another process. A problem with a reactive system is that some interleavings are undesirable. To remove undesirable interleavings, we have to schedule the transition executions appropriately. For example, in multithreaded programs, the lack of mutual exclusive access to critical sections would result in an undesirable interleaving. In such a case, the programmer uses synchronization mechanisms (e.g., locks, semaphores) to ensure mutual exclusion.

For the type of applications we target in this research, such as game applications or user interfaces, the rendering of the visual display is modeled as a separate process that runs periodically. The various interactions inside the game are one or more processes that run concurrently. “Inaccuracies” in the perceivable can result from undesirable interleaving of the transitions. For example, a virtual ball may penetrate a wall briefly instead of bouncing off the wall immediately. If the visual display runs during this interval, then the observer perceives this as an inaccurate simulation. To mitigate this issue, our approach schedule the transition executions (i.e., the changes to those components) to remove some of the undesirable interleaving. Because the visual display is not under the programmer’s control, it is not possible to remove all undesirable interleavings, but our approach aims to minimize their number.

One common way to enable reactiveness is to capture the data dependencies between the program’s components and execute the chain of functions necessary to propagate the changes based upon these dependencies (Foust et al., 2015; Lehmann et al., 2016). At the start of the application, the prototype creates a dependency graph. It then traverses this graph whenever a change occurs to any data item. It continues traversing the graph as long as changes are propagated from one node to others. When it stops, the graph remains stable until the next chain of reactions. Figure 2.3 shows how behavioral information is extracted from the systems described in Chapters 3 and 4, which are based on our earlier work (Marum et al., 2016, 2019, 2020a,b,c).

Van den Vonder et al. (2017) argue that the benefit of reactive programming becomes more evident in applications that are highly interactive with multiple user-interface events that can change data values. There are many examples of RP frameworks ranging from those designed to add reactivity to existing systems to purpose-built languages. For instance, Czaplicki and Chong (2013) describe a language which was initially reactive and evolved to a multi-purpose language and framework. Blackheath and Jones (2016) focus on the development process for a new reactive library called Sodium. Blom and Beckhaus (2008)

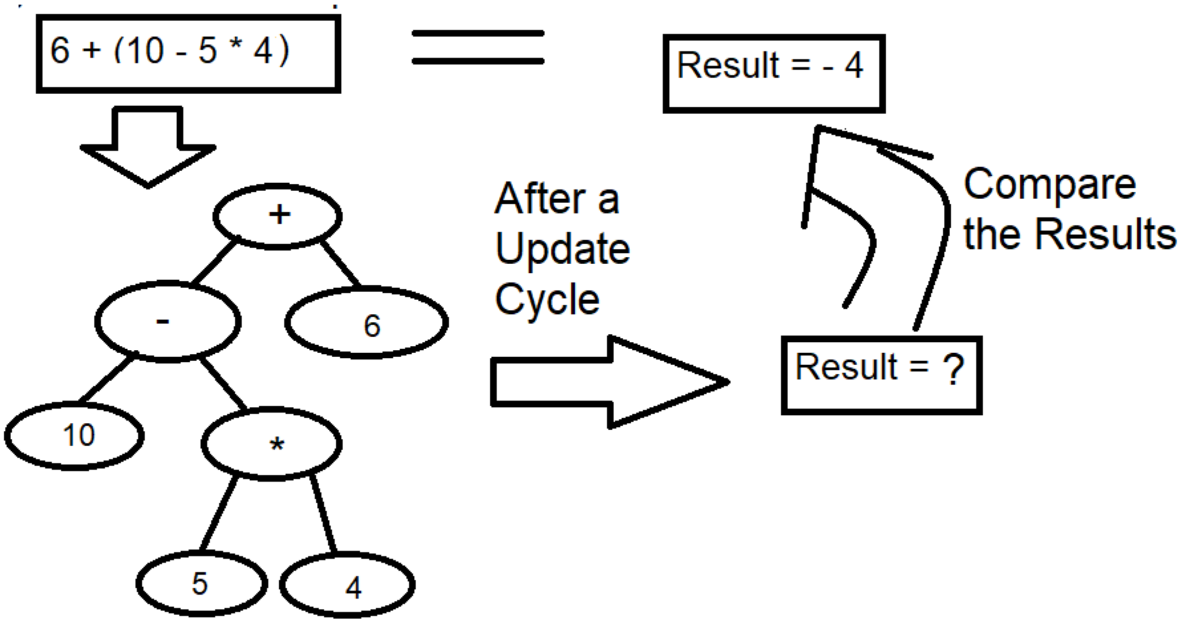


Figure 2.3. Extraction of dependencies from the original hierarchy.

examine a Haskell library for functional reactive game development. Their approach is based on a reactive programming layer that acts like a middleman between the reactive user interface and the non-reactive Virtual Environment (VE) manager. In the next section we describe multiple aspects of the framework we use for both of our implementations, focusing on the aspects that led us to choose it for our implementations

2.2 .NET Framework

Senthilvel and Qureshi (2017) describe Microsoft .NET (Microsoft, 2020b) as a managed execution environment made up of tools, programming languages, and libraries for building many different types of applications. There are implementations of .NET that allow .NET applications to execute on platforms such as Linux, macOS, Windows, iOS, and Android.

Microsoft (2020b) originally implemented the .NET Framework as a managed execution environment for its Windows operating system. This framework provides a variety of services for developing websites, services, desktop applications, etc. More recently, Microsoft expanded these services to other operation system platforms: .NET Core for Linux and macOS and Xamarin for Android and iOS devices. Both of those are open-source and based upon an older implementation of .NET for Unix-based operational systems called Mono.

The .NET Framework, which is also called the Common Language Infrastructure (CLI), is described by Nagel (2018) as consisting of the following components:

Common Language Runtime (CLR), which is described by the official .NET documentation (Microsoft, 2020b) as the execution engine that handles running apps. The CLR provides the services and runtime environment to the machine code. Senthilvel and Qureshi (2017) describe the Common Language Runtime (CLR) as the runtime execution environment. Any code that must run on .NET framework is executed under the control of the CLR. That is why it is often called managed code. Internally, the CLR loads any required functions or classes from the .NET Framework Class Library.

.NET Framework Class Library (FCL), which is described by the official .NET documentation (Microsoft, 2020b) as a class library that provides multiple system functionalities available to all technologies and languages in the .NET Framework. The FCL contains various classes, data types, and interfaces and a common event system that manages action notifications and handles subscription and ownership of these actions.

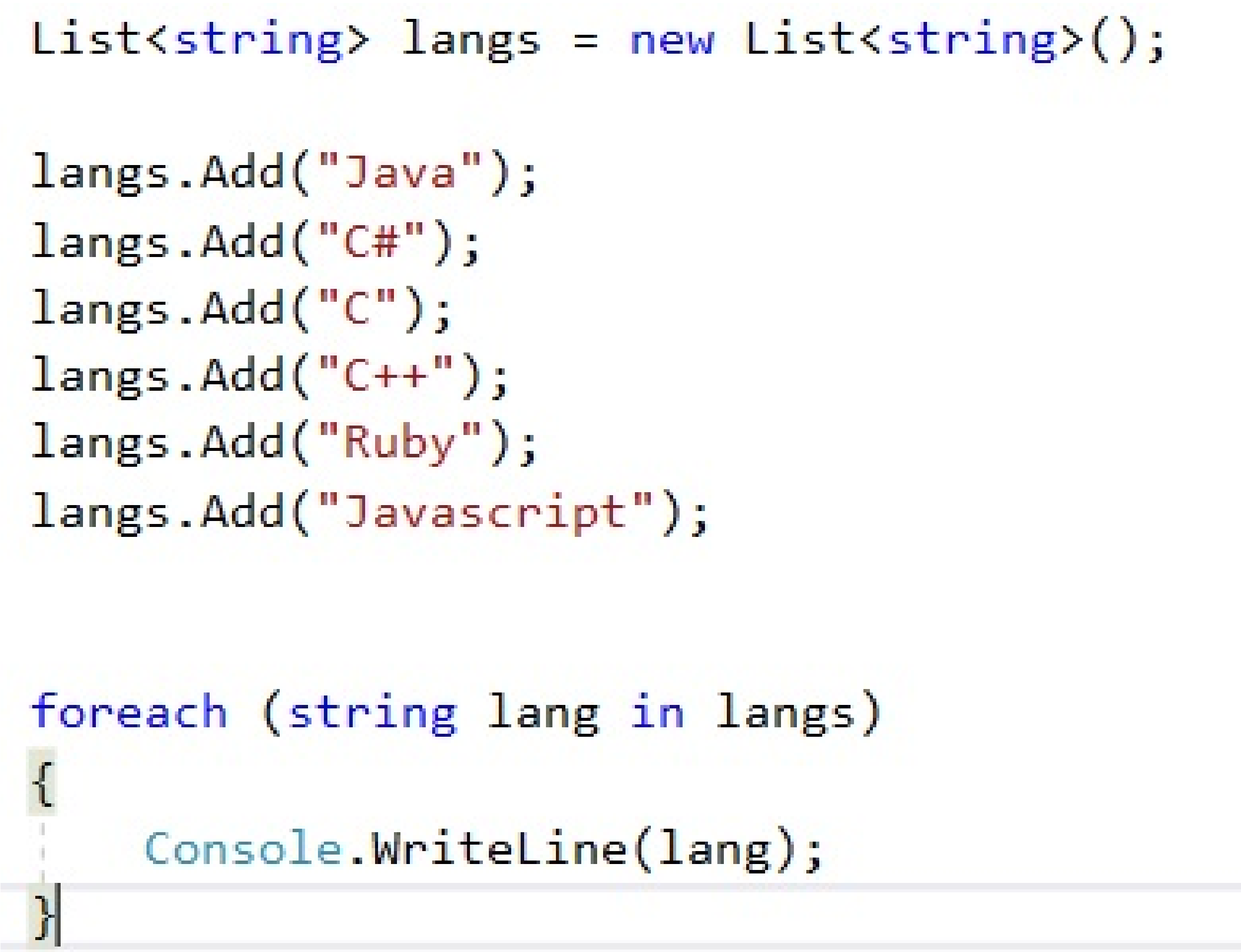
Common Language Specification (CLS), which is described by the official .NET documentation (Microsoft, 2020b) as the component responsible for converting the different .NET programming languages' lexical, syntactical rules and regulations into a CLR understandable format. It is a part of the compilation capabilities that unifies all .NET languages into a single standard. The CLS handles the lexical, syntactical, and semantic analyses and then translates the original source code into the respective version of the code in the Common Intermediate Language (CIL).

Common Type System (CTS), which is described by Price (2019) as the component responsible for understanding all the data type systems of .NET programming languages and converting them into a common format. All types in .NET follow a tight inheritance scheme. Therefore any developer-defined type inherits from `System.Object` or from one of its subtypes.

The components described above support the features that lead us to choose the .NET framework for the two case studies presented in Chapter 3 and 4. Having a common language specification and a common type system provides the uniform inherited type system upon which we build our reactive augmentation. It allows us to separate a subset of the components by defining another logic layer upon the existing classes in the .NET framework. This allows us to define a flexible type definition. That is, we can expand our search by accessing all classes that inherit from a given class, and then we can search for specific types among them to use specific capabilities. The Framework Class Library permitted us to tap into the .NET capabilities for efficiently iterating through large data structures, especially special iteration capabilities and functional combinators.

C# (pronounced "See Sharp") is the main language of the .NET Framework. The official C# documentation (Microsoft, 2020a) describes C# as a strongly typed, multiparadigm language in the C family of programming languages. It is primarily an object-oriented language, but it also has support for elements of the functional, event-oriented, and metaprogramming paradigms. In C#, everything is an object. Albahari and Johansen (2020) describe C# as a language that supports many unique programming abstractions such as delegates, properties, events, and attributes.

According to Price (2019), C# has a unified type system that includes all the common primitive types such as `int` and `double`, but even these types inherit from the primary `Object` type. The types that belong to the common type system share a set of common operations that allow data from these types to be stored, transported, and operated on in a standardized manner. For example, all classes inherit the `ToString()` method.



```
List<string> langs = new List<string>();

langs.Add("Java");
langs.Add("C#");
langs.Add("C");
langs.Add("C++");
langs.Add("Ruby");
langs.Add("Javascript");


foreach (string lang in langs)
{
    Console.WriteLine(lang);
}
```

Figure 2.4. An implementation of a `foreach` iterating through a list in C#.

According to Price (2019) and Albahari and Johansen (2020), C# has the following characteristics that we relate to our research:

For Each: C# provides an additional flow control statement, `foreach`. Albahari and Johansen (2020) define `foreach` as a built-in loop construct that iterates over all items in array or collection without requiring an explicit specification of the indices. Figure 2.4 demonstrates the use of a collection (a list) that holds an unknown and theoretically unlimited amount of data. It uses `foreach` iteration to easily iterate through the collection.

Generics: Microsoft (2020a) defines a *generic* as a language feature that enables classes and methods that work with a set of types. To use a generic, a developer needs to use a placeholder (e.g., `<T>`) on the methods or classes. Microsoft (2020b) explains that .NET can infer the appropriate type at runtime. Generic data types enable a function to be called without knowing the specific data type being handled. Thus, the same function can be used for a wide variety of data types.

Figure 2.5 demonstrates the use of generic to establish a functionality that works for any data type.

Collections: Price (2019) defines collections in a library of several built-in data structures (e.g., lists, queues, hash tables) that are available in the .NET languages.

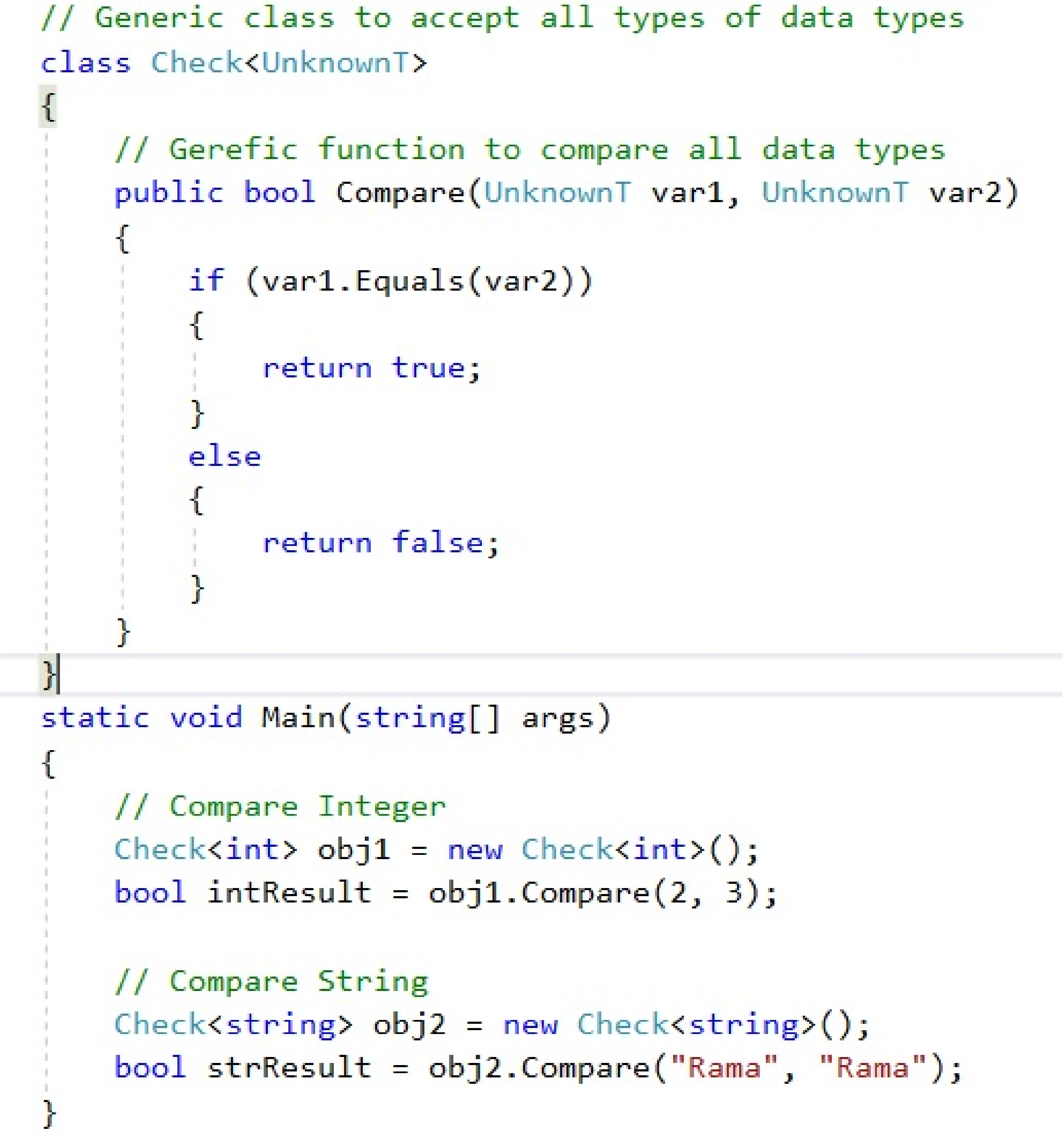
Since event handling is an important aspect of our research, we examine .NET's event-handling capabilities in some detail in the next section.

2.3 Event System

An *event* is a message or signal that represents an action or occurrence. An event is triggered whenever the system needs to do something to respond to that phenomenon. From a programming standpoint, an event is a message sent by a component that was acted upon to indicate some occurrence that must be dealt with. Events are an effective mechanism for interprocess communication because they signal state changes, which may be valuable to the reaction of that event in the system.

Event-handling mechanisms can take many forms across different languages and implementations; for object-based systems an event is often an object whose properties contain any contextual information needed to process the desired occurrence. Nagel (2018) explains that the .NET Framework, more specifically C#, models event handling by using two abstractions: event-handlers and delegates. Figure 2.6 illustrates a code sample in .NET that handles events.

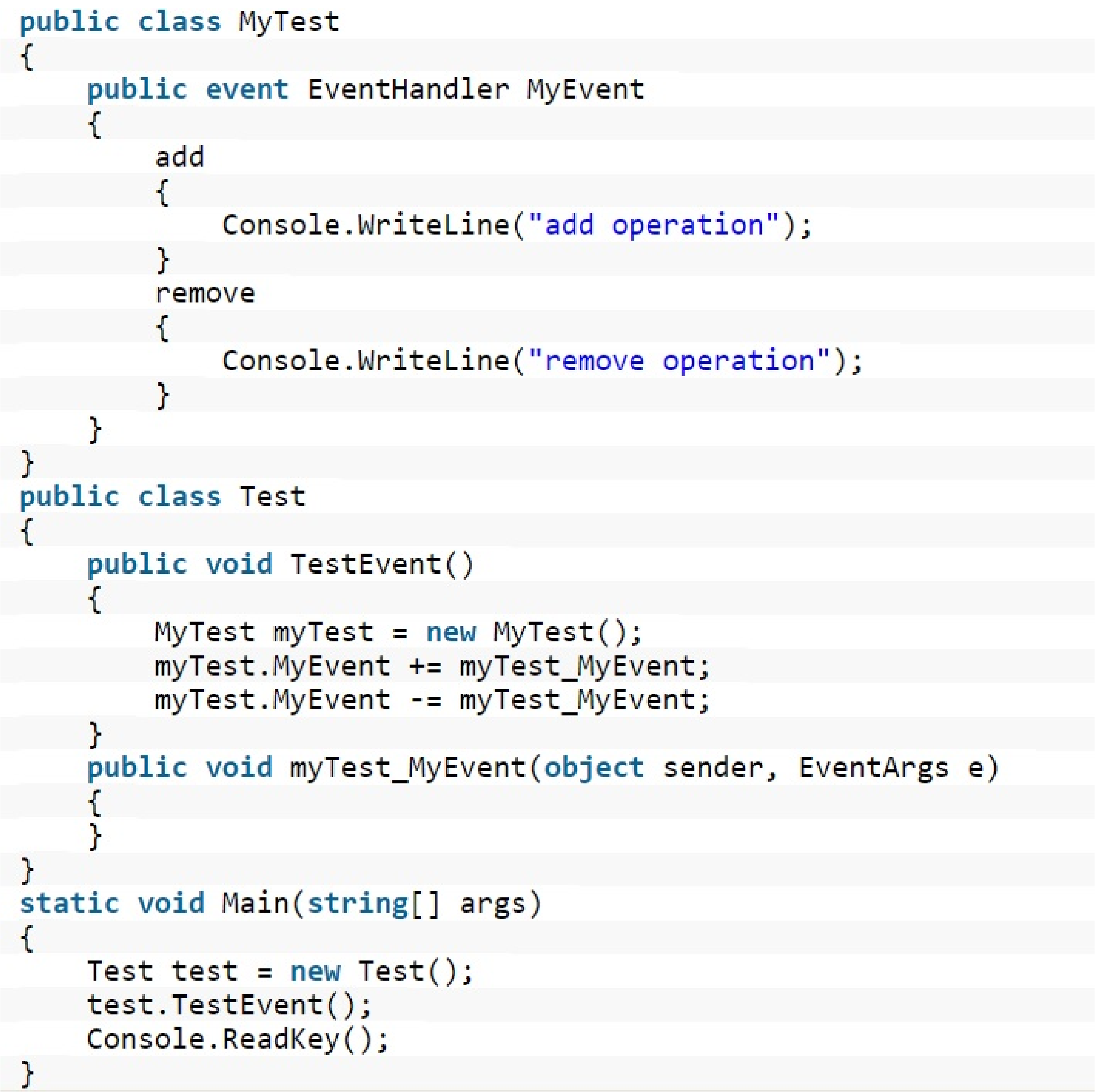
Based on the definition from Microsoft (2020a), a *delegate* is a type that represents



```
// Generic class to accept all types of data types
class Check<UnknownT>
{
    // Gerefic function to compare all data types
    public bool Compare(UnknownT var1, UnknownT var2)
    {
        if (var1.Equals(var2))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
static void Main(string[] args)
{
    // Compare Integer
    Check<int> obj1 = new Check<int>();
    bool intResult = obj1.Compare(2, 3);

    // Compare String
    Check<string> obj2 = new Check<string>();
    bool strResult = obj2.Compare("Rama", "Rama");
}
```

Figure 2.5. An implementation of a generic data type and function in C#.



```
public class MyTest
{
    public event EventHandler MyEvent
    {
        add
        {
            Console.WriteLine("add operation");
        }
        remove
        {
            Console.WriteLine("remove operation");
        }
    }
}
public class Test
{
    public void TestEvent()
    {
        MyTest myTest = new MyTest();
        myTest.MyEvent += myTest_MyEvent;
        myTest.MyEvent -= myTest_MyEvent;
    }
    public void myTest_MyEvent(object sender, EventArgs e)
    {
    }
}
static void Main(string[] args)
{
    Test test = new Test();
    test.TestEvent();
    Console.ReadKey();
}
```

Figure 2.6. Handling an event in C#.

or holds a reference to a method, provided that the parameter list and the return type are similar and the method is accessible. Price (2019) defines a delegate as a way of treating a method as a first-class object. Whenever a delegate references a method, C# treats the method as a class attribute, returning a reference to that method.

Whenever a delegate references a method, no behavioral information needs to be known about that method. The only information required is its signature. As Price (2019) notes, the delegate can point to any method whose arguments and return values match the ones described in the method of the delegate signature. Albahari and Johansen (2020) argues that the primary reason to use a delegate is to add function polymorphism to the code, which is something that a developer cannot achieve by just using interfaces and abstract classes. Delegates allow functions to be passed to other functions so these other functions can invoke the first ones directly instead of duplicating the same code in multiple places.

Figure 2.7 shows the declaration of a delegate called `MyDelegate`, which has a `void` return type and a `String` parameter. The target method can be attached to the delegate by assigning a method directly to the object of delegate type.

Microsoft (2020a) explains that there are three steps for working with a delegate:

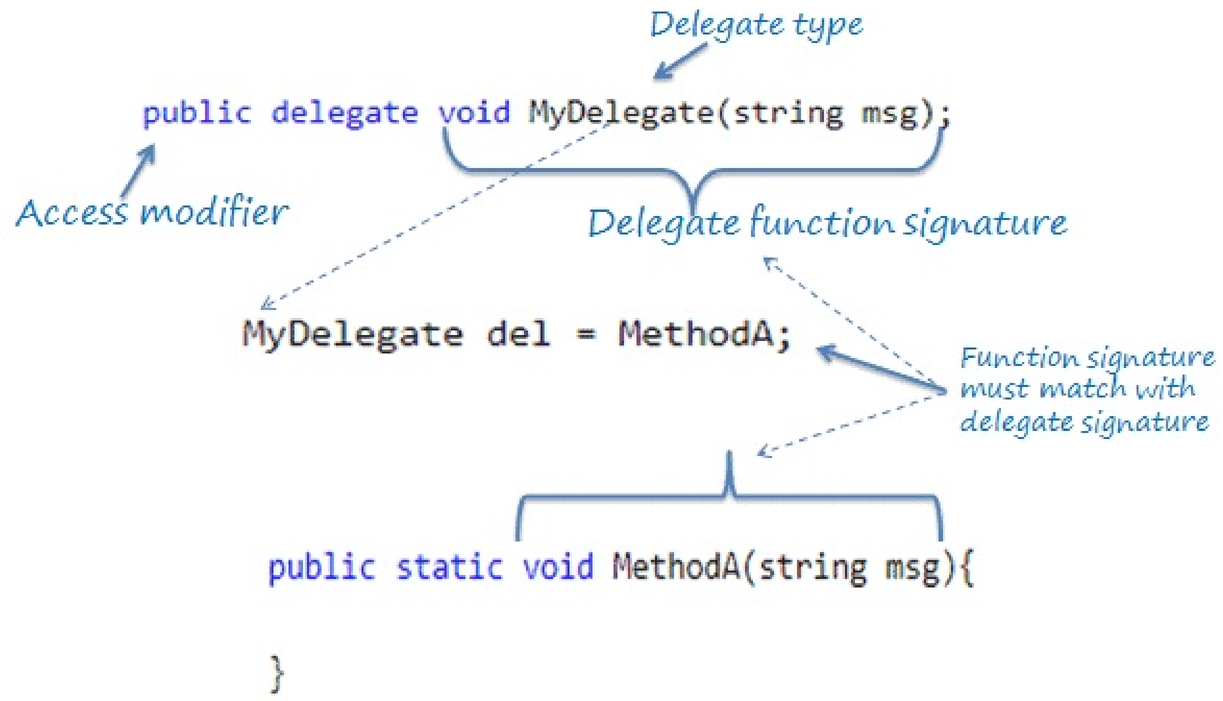
1. declare the delegate
2. set a target method (whose signature must match the delegate's signature)
3. call the delegate (which indirectly invokes the method attached to it)

A delegate is declared using the following syntax:

```
[accessModifier] delegate [returnType] [delegateName]([parameters])
```

Figure 2.7 highlights the method `methodA()` being attached to the delegate `MyDelegate`.

An `Event` object in C# is defined by Microsoft (2020a) as a special kind of delegate designed to facilitate event-driven programming. Delegates are very flexible. They can be



```
public delegate void MyDelegate(string msg);

MyDelegate del = MethodA;

public static void MethodA(string msg){

}
```

Delegate type
Access modifier
Delegate function signature
Function signature must match with delegate signature

Figure 2.7. A `delegate` implementation in C#.

passed around and can reference any method that matches its signature. However, this also raises a problem: delegates can easily have their properties overridden, and that leads to errors where references to a changed delegate become invalid. That means a developer must not use delegates as public properties. To avoid the above problem but still work with delegates, Price (2019) points out that C# uses `Events`, which defines a wrapper around the delegate. Figure 2.8 shows the relationship between `Event` and `Delegate`.

The `Event` type resolves the problem of exposing a delegate outside of a class by defining wrappers around `Delegate`. Albahari and Johansen (2020) state that a developer must never raise an event unless at least one other method is added to the event. In other words, the delegate attached to the event must not be equal to null. So, once an event is created and attached to a delegate, it can neither be set to `Null` nor overridden by using the keyword `new`. It can only have event-handling methods added or removed.

The signature of an event handler contains the method that is invoked in case of a notification of an event. Price (2019) explains that this event handler signature has two

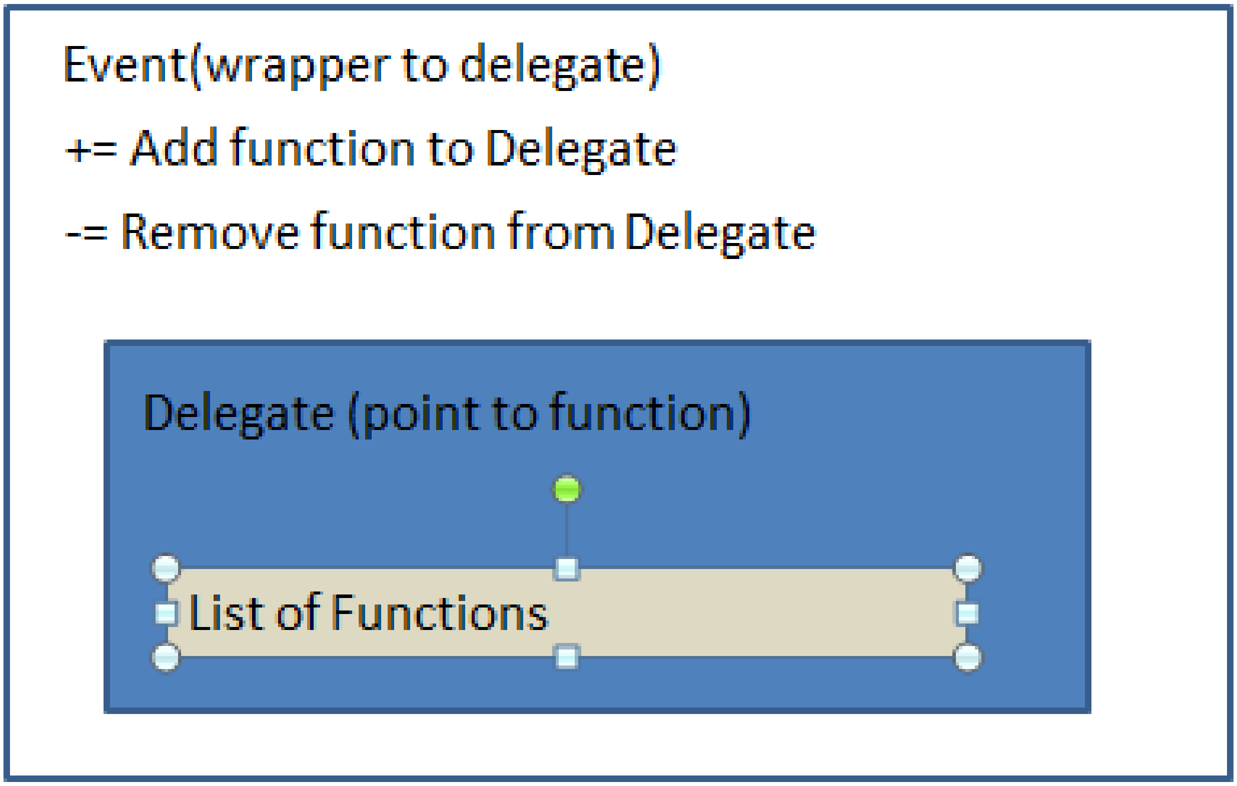

Event(wrapper to delegate)
+= Add function to Delegate
-= Remove function from Delegate
Delegate (point to function)
List of Functions

Figure 2.8. The relationship between `event` and `delegate`.

parameters. The first parameter is the event data, which is a collection of data regarding the event notified. The second parameter is the sender object, which refers to the control that generated the event. The method bound to the event object is an action notification that originates from an interaction with a control from an external or internal actor instead of a simple method call. The event object is primarily used for signaling a message or notification passing. In the next section, we describe the general aspects of reflection and metadata and how .NET makes this information available.

2.4 Reflection and Metadata

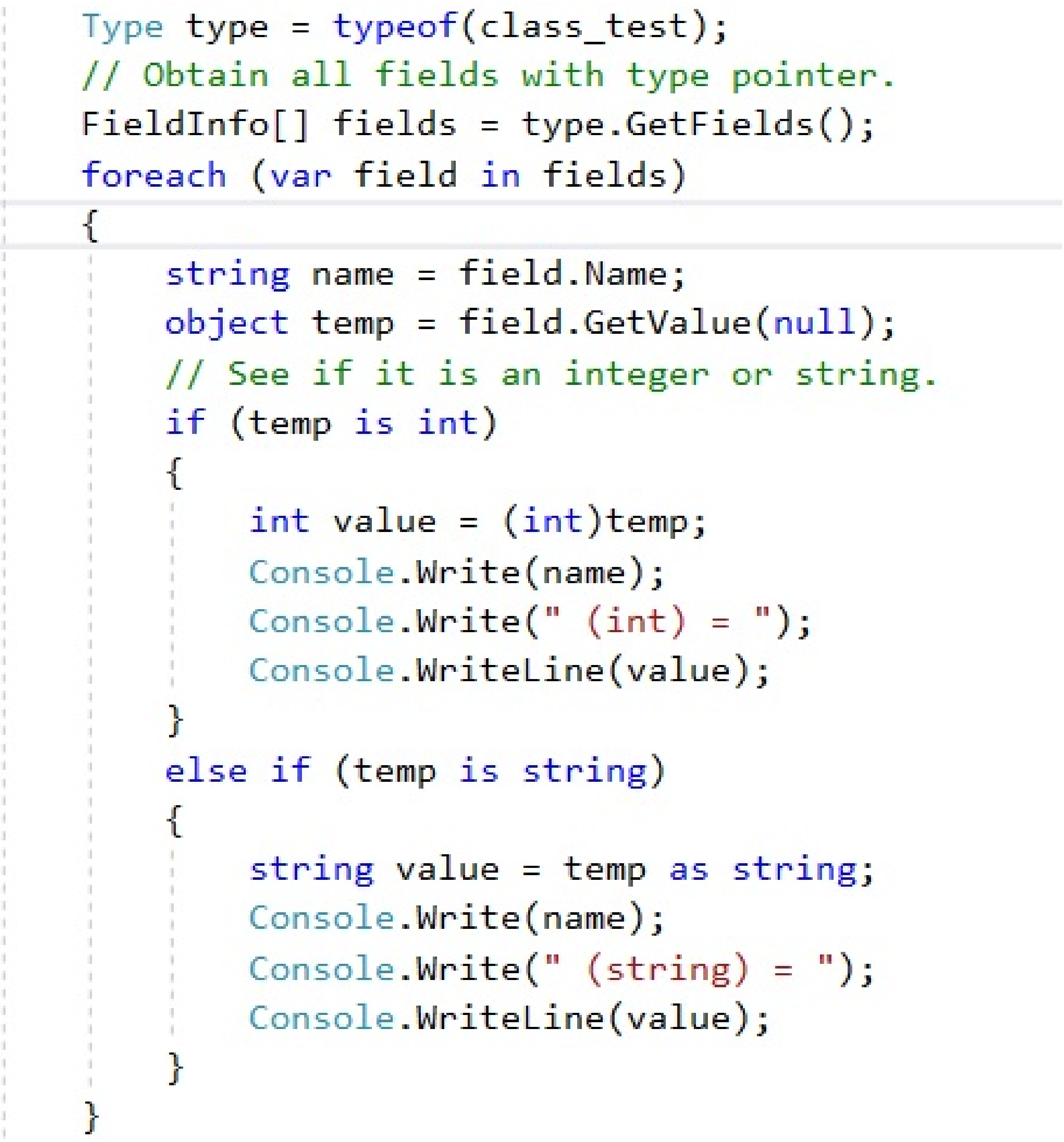
In general terms, Pontes et al. (2019) define *reflection* as a programming capability that allows a running program to have access to information about the structure of the objects and classes from which the program is constructed. This includes information about the methods and data attributes within the individual classes. Li et al. (2019) define reflection as the ability to examine a program and possibly change its structure and behavior at runtime. It is a feature present in many languages, like Python, C#, and Java. For .NET, Hamilton (2003) explains that reflection is the process of runtime type discovery. It enables a program to dynamically inspect an instance of a given type, analyzing and invoking its members at runtime, without having compile-time knowledge of its existence.

The information required to make such analyses is commonly called *metadata*. Microsoft (2020b) defines metadata as the structural information about the components present in a program. This metadata includes information about the way components are loaded, how memory is laid out, how methods are invoked, how classes are structured (information about the names and types of methods and attributes, and any connection between these components), and how types are declared. Figure 2.9 shows how a program can access information about an object at runtime, including all its fields.

Rodriguez and Swierstra (2015) describe the `Type` class as the main component of all reflection operations. A `Type` object represents a type inside the system. The `Type` class enables a program to access the metadata. It provides methods for obtaining information about a type declaration, such as the constructors, methods, fields, properties, and events of a class, as well as the module and the assembly in which the class is deployed.

For Rodriguez and Swierstra (2015), .NET applications store metadata in assembly files. Assemblies are composed of multiple modules and each module contains the structural information of a single class and the structural details of all the methods and attributes within that class. An *assembly* is a versioned binary file that contains all information that the .NET framework needs to execute the file. Assemblies are composed of multiple modules. Each module contains the structural information for a single class and the structural details of all the methods and attributes within that class.

Damyanov and Holmes (2004) explain that a program can use reflection to access and modify an object's class information, including the values of its attributes. Using reflection, it can invoke an object's methods and create objects and attributes dynamically at runtime.



```
Type type = typeof(class_test);
// Obtain all fields with type pointer.
FieldInfo[] fields = type.GetFields();
foreach (var field in fields)
{
    string name = field.Name;
    object temp = field.GetValue(null);
    // See if it is an integer or string.
    if (temp is int)
    {
        int value = (int)temp;
        Console.Write(name);
        Console.Write(" (int) = ");
        Console.WriteLine(value);
    }
    else if (temp is string)
    {
        string value = temp as string;
        Console.Write(name);
        Console.Write(" (string) = ");
        Console.WriteLine(value);
    }
}
```

Figure 2.9. Using `Type` class to access information about a given object.

For Pontes et al. (2019), these capabilities mean that managed code can actually examine other managed code, including itself, to determine information about that code.

Rodriguez and Swierstra (2015) explain that reflection uses the information present in the metadata to access the components' structure programmatically. Whenever a class needs to use information from a class compiled in a different assembly, Rodriguez and Swierstra (2015) highlight that the compiler must describe the location of the assembly containing that class, so the CLR can link the assembly to the final executable file. The CLR automatically links the classes present in the .NET class library to the executable file.

In the next section, we consider the implicit invocation model. This model describes the general structure and behavior of the applications of interest in this dissertation research.

2.5 Implicit Invocation Model

Complex systems are typically comprised of many components. Garlan and Shaw (1993) find that the fundamental issue in the development of these systems is the mechanism for integrating the components. Traditionally, Shaw and Garlan (1996) note that these systems provide a collection of functions. Components interact with each other by explicitly invoking each other's functions. They call this mechanism for integration *explicit invocation*. In explicit invocation, a component directly accesses and invokes functions on other components of the system. However, Shaw and Garlan (1996) note that there is an alternative called *implicit invocation*. The idea behind implicit invocation, as stated by Garlan and Shaw (1993), is that instead of invoking a function on another component directly, a component can post an event notification to the other components interested in knowing about the event. Components can register themselves to be notified when the event notification is posted. Each component can associate the invocation of one of its own functions in response to the event notification. When the event notification is posted, the system lets each component execute the function associated with the event. Thus, an event notification implicitly or indirectly causes the invocation of functions in all the registered components.

The implicit invocation model assumes the existence of a loosely coupled collection of components, each of which carries out some process to which other components may need to react.

Garlan et al. (1998) state that these components (processes, actors, programs, structures, or objects depending on the technology applied) can post a notification to a central manager that some event of interest has occurred locally. The central manager can then notify other interested components of what has occurred. These other components have no knowledge of what components are in the system and may only communicate with the other components through asynchronous event notifications. Figure 2.10 shows the overall functioning of the implicit invocation model.

Garlan and Khersonsky (2000) observe that a component posting an event notification does not know if any other component is interested, neither do they know in what order the events from different components will be received by the interested parties. As a result, implicit invocation should scale well as the number of events and components increases.

According to Garlan and Shaw (1993), an implicit invocation system has three key concepts:

Component: A logically independent entity that can be the poster (post event notification to a manager) or the listener (registered in the manager as interested in being notified that certain event has occurred). The components in an implicit invocation system may all be a part of one software system or may be spread across several software systems.

Event: An occurrence of possible interest or importance within a system. The notification of an event must be posted to the manager. The event must contain the identifier for its source component to enable other interested parties (components) to register to receive the event notification. The event may contain data.

Manager: A medium that enables the components to register to receive the events of their

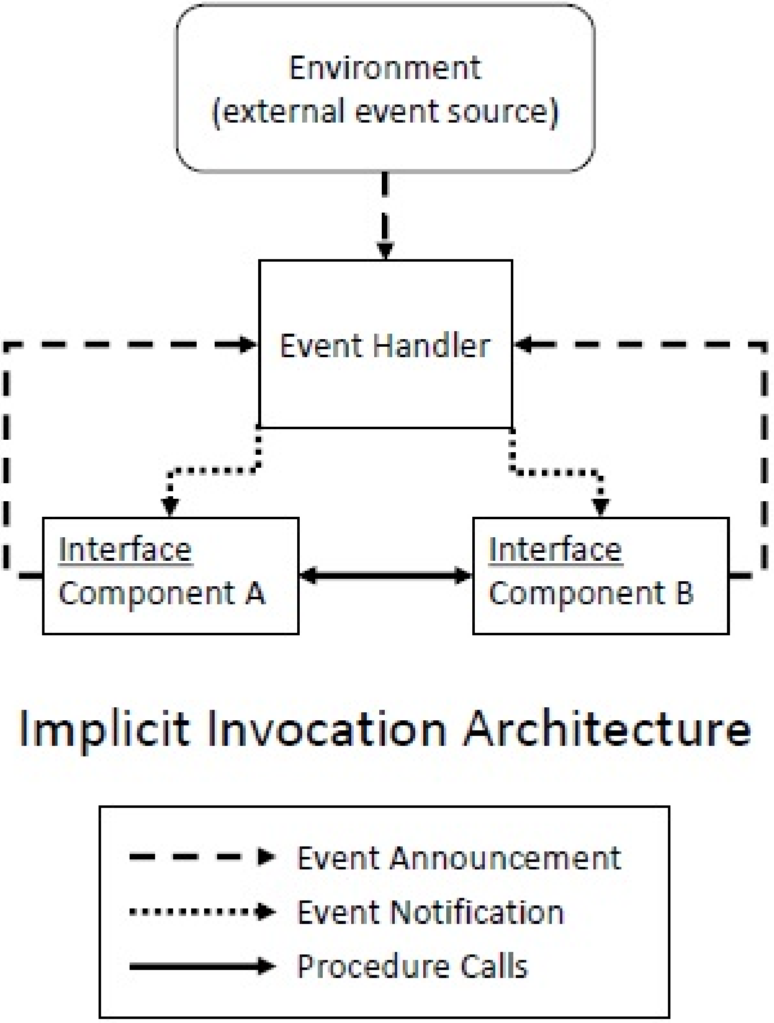


Figure 2.10. The implicit invocation architecture model.

choice. When one of the components posts an event, the manager dispatches the event to the registered parties by invoking the function on each party associated with this event.

In addition to the asynchronous nature and the scalability of implicit invocation, it has other possible benefits (Garlan and Shaw, 1993; Shaw, 1996):

- Implicit invocation provides strong support for reuse. A new component can be introduced into the implicit invocation system simply by connecting it to the manager and registering it for the events of that system.

- Implicit invocation eases refactoring. Components may be replaced by other components without affecting the interfaces of other components in the system.

In contrast to implicit invocation's loose coupling, systems based on explicit invocation are tightly coupled. A change to one component or function likely requires that all other components that use that component or function must also be changed.

The primary disadvantage of implicit invocation is that components relinquish control over the computation performed by the system (Shaw and Garlan, 1996). When a component announces an event, it has no idea what other components will respond to the event nor can it rely on the order in which listener functions are invoked. This creates concern related to the correctness of the event ordering. Since the event notification itself has no logical meaning, the meaning of an event notification depends on the listeners' functions that will be invoked. This makes reasoning about the correctness of a system more difficult than for explicit invocation systems.

2.6 Dependency Graph

A dependency graph is a graphical representation of a set of nodes that depicts a transitive dependency relation on the set (Féray et al., 2018). In a transitive relation, if `A` is related to `B` and `B` is related to `C`, then `A` is also related to `C`.

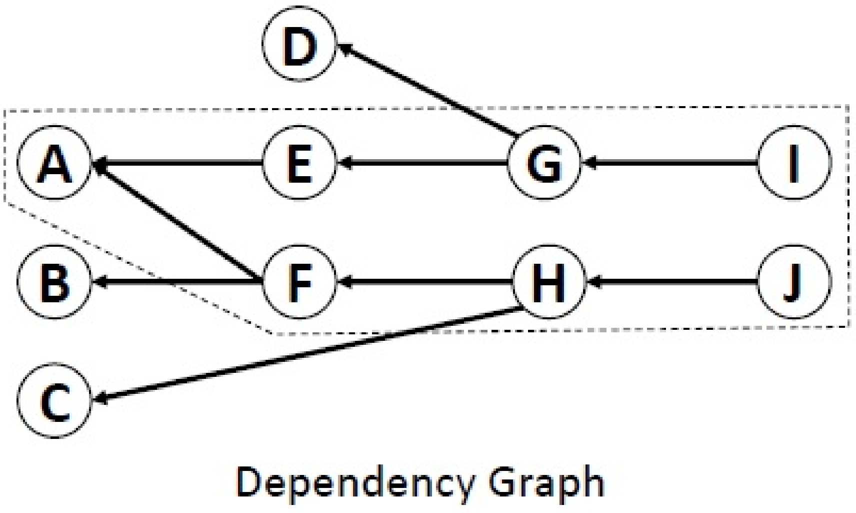


Figure 2.11. The dependency graph and the relation between nodes.

In this dissertation research, a node in the dependency graph represents some kind of computational entity. If some node B depends on a node A, then the computation at A can somehow affect the computation at B. For example, node A may change, create, or modify a value that B uses in its computation. In such a case, the computation at A likely should precede the computation at B. Thus, from an acyclic dependency graph, it is possible to derive a sequence for the computations (i.e., an evaluation order) in which every node of the graph is visited and no node is visited before any node it depends on. Figure 2.11 shows a dependency graph, showing the dependency relationships between the nodes.

In computer science, we can define a dependency graph as a directed acyclic graph that represents a given component collection and shows how the components relate to each other using directed links called dependencies (Azero, 2013). In most variations of dependency graphs, the nodes consist of programmable units (e.g., classes, structs, objects, code files, etc.). The graph encodes the relationships between any two components. Suppose we have two components `A` and `B`. If `B` depends on `A`, then `A` must be evaluated first and any change to `A` will require a change to `B`.

The dependency graph must be always acyclic (Azero, 2013) since the presence of cycles of dependencies (also called circular dependencies) leads to a situation in which no valid evaluation order exists. By topologically sorting the nodes in the dependency graph, we can infer a valid evaluation order. Most topological sorting algorithms are capable of detecting cycles in their inputs (Franciscus et al., 2019). However, it may be desirable to perform cycle detection separately from topological sorting to provide appropriate handling for the detected cycles.

Figure 2.12 shows the use of a dependency analysis to build a dependency graph. Each edge between the nodes in the graph is defined as a dependency relationship between the computational components between them. The successful evaluation of this dependency graph will yield an evaluation order that will allow all the components to execute without violating any of the dependency constraints represented in the dependency graph.

For a given dependency graph, there can be more than one correct evaluation order. Computationally speaking, as stated by Azero (2013), an evaluation order is a topological order that uses the dependencies as constraints. Thus, any algorithm that derives a correct topological order from a dependency graph derives a correct evaluation order.

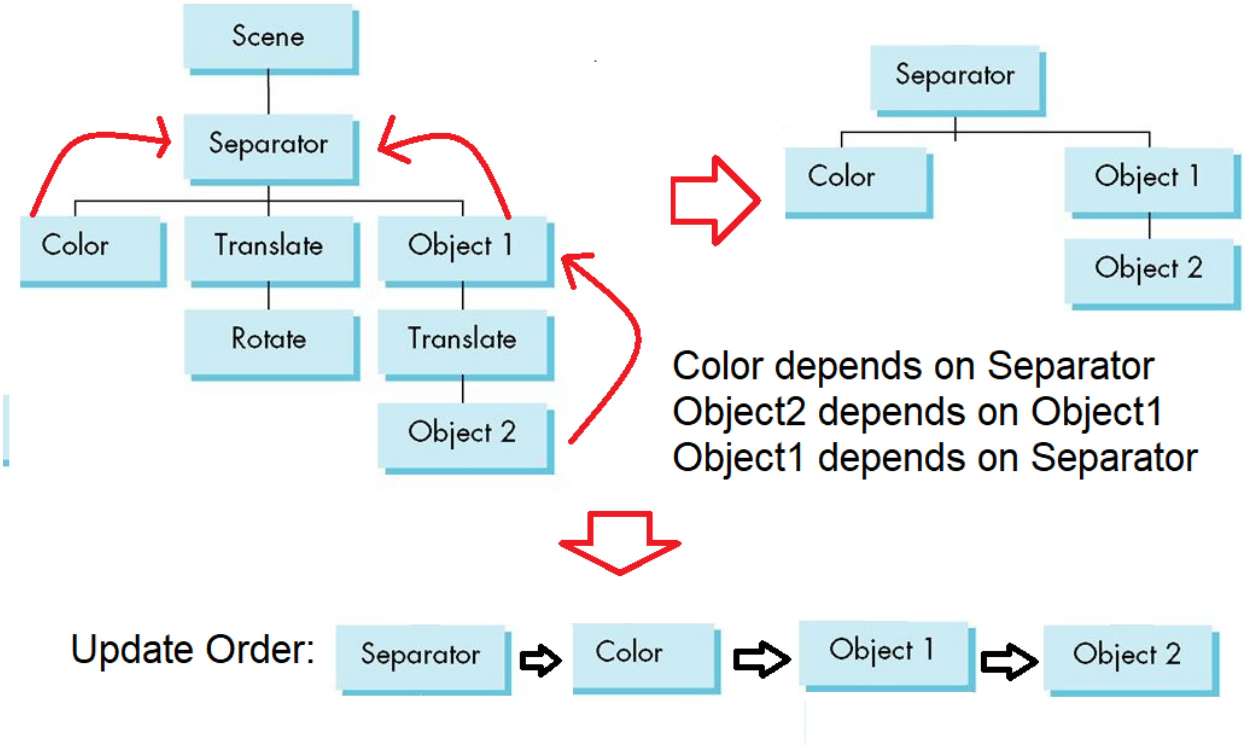


Figure 2.12. Mathematical dependency graph used to infer evaluation order (Marum et al., 2019).

CHAPTER 3

DEPENDENCY GRAPH-BASED REACTIVE AUGMENTATION OF WEB AND DESKTOP USER INTERFACES

3.1 Introduction

In this chapter, we consider both desktop and Web-based user interfaces (UI). Foust et al. (2015) argue that UIs are reactive applications normally implemented using imperative callbacks that are then sent to a programming layer to handle the event processing. Czaplicki and Chong (2013) define that the user interface components are arranged in graph structures: For a Web page, 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 application's execution. 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 et al. (2015) state 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 the data that the first modifies).

The first control does not directly call the second. Instead, the first control raises a new event, which causes the execution of the second control at some later point.

In a complex application, the 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 3.4) analyzes the complex relationships among the controls, encodes the dependencies between them in a dependency graph, and then topologically sorts the graph to extract an evaluation order for the event handlers of the components that does not violate the dependency constraints. This augmentation improves the performance by coalescing the set of updates so that users conceptualize it as occurring in a single chain. It ensures 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 user interface 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.

Along the path to achieving this, our research aims to determine whether dependency graph-based execution reordering and self-adjusting state recomputation can augment the GUI's built-in event system to mitigate the execution inconsistencies due to transitional turbulence and increase the accuracy without degrading the performance significantly. Preliminary versions of this research are described in Marum et al. (2020a) and Marum et al. (2020b).

We evaluate our approach by comparing its performance against the performance of original .NET, Rx.NET (Malawski, 2016), and Sodium (Blackheath and Jones, 2016) implementations. Rx.NET and Sodium are commonly used alternative reactive libraries for user interfaces. Before we look at the design of our augmentation and the implementations, let us examine the technologies present in the .NET framework in Section 3.2 to build user interfaces: especifically the `WindowsForms` and ASP.NET. Then in Section 3.3 we examine the aspects of the problem in more detail.

3.2 Background

`WindowsForms` is a set of managed libraries for the .NET Framework that simplifies common application tasks related to the UI (Brown, 2006). It is the User Interface (UI) manager. It allows the developer to create visual client applications that display information, request input from users, and communicate with remote computers over a network (Microsoft, 2020b).

In the `WindowsForms` architecture, a form is a class that contains a list of controls, along with their descriptive information (Greene and Stellman, 2013). Controls can be nested inside of other controls. Normally, the parent control owns and defines certain characteristics of its child controls, such as their locations and colors. Given the definition of the UI, .NET creates the respective executable form at runtime. This UI can display information to the user and let the user interact with the system. A `WindowsForms` application's behavior is built by programming each control's response to user actions and interactions in general, such as mouse clicks or key presses (Albahari and Johansen, 2020). Each control is a discrete UI element that displays data or accepts data input. The `WindowsForms` architecture features a library of commonly used UI controls that can be added to forms (Nagel, 2018): text boxes, buttons, drop-down boxes, radio buttons, and others. However, developers can extend the behavior of an existing control or create their own custom controls. They can create new classes that inherit from the specific control they want to extend or they can create a class that inherits directly from the top-level `UserControl` class to create new types of controls.

When a user does something to the form or one of its controls, the action generates an event (Price, 2019). The application reacts to these events by using user-defined event-handlers to process the events as they occur. Events are notifications that can be generated by a user action (such as clicking the mouse or pressing a key), by the program code, or by the system. Event-driven applications execute code in response to an event. Each form and control exposes a predefined set of events that a developer can use to program its behavior (Nagel, 2018). If one of these events occurs and there is code in the associated event handler, that code is invoked. The types of events raised by an object vary, but many types are common to most controls. For example, most objects can handle a `Click` event. If a user clicks in a control inside the form, the event handler of that control in the form code is executed.

Forms in the `WindowsForms` architecture are defined using partial classes. A partial class is a feature of C# that spreads the definition of a class across multiple class files. Then, during compilation, these files are integrated to form a single class. The benefit of using a partial class is to separate logically different aspects of the class into separate groupings. In real-world programming, there are several situations where splitting the definition of a class is desirable. By declaring a class partial, a developer can separate the different functionalities of that class into multiple source files that will be combined into a single class definition at compile time as a way to extend the functionality of the class. Each source file adds a distinct part of the overall class definition. Since it is a single class, the source files can reference each other's methods and attributes. However, all parts combined must form a valid class definition, so two parts cannot define the same method with the same signature. The same constraint holds for properties, constructors, and so forth. The merge occurs at the class level; two methods with the same name are considered overloads and are never merged into a larger method.

This separation, in the case of the `WindowsForms` form, keeps automatically generated code from the logic file where the developer implements the UI behavior. It keeps this automated code and the developer code safe from unwanted modifications (Microsoft, 2020a).

However, this separation does not make any difference to the C# compiler, as it treats all these partial classes as a single entity at the time of compilation and compiles them into a single type in the Microsoft Intermediate Language (MSIL).

The most relevant partial class example is the file `file.cs` that is used to build a form in `WindowsForms` applications (Albahari and Johansen, 2020). .NET formats the UI into two separate files, the `file.Designer.cs` designer class file and the `file.cs` logic class file. The designer file has code related to the implementation of the controls and the appearance of the UI and the logic file contains the event handlers and any logic related functions and variables. Microsoft .NET's main Integrated Development Environment (IDE), called Microsoft Visual Studio, contains a native form designer system with a toolbox containing all the common controls used in UIs. It has an option to add third-party control libraries and the developer's own controls (Microsoft, 2020b). The developer can drag a control from the toolbox to the form, position it in the desired place, and modify its properties. As a control is dragged, a related object with the same type as the control dragged is instantiated on the `file.Designer.cs` page and its properties are automatically modified in the same way the developer changed the control's properties in the form designer.

The `file.Designer.cs` contains the auto-generated code that is added every time a new control is dragged from the toolbox to the form. So, this file contains the code that implements the controls, changes their features, positions the controls in the form, and handles the form's overall appearance (Microsoft, 2020b). Meanwhile, the `file.cs` file handles the behavioral code, i.e., event-handling methods, connections to other classes, and any other logic-related methods required. The event handler is a method that is bound to an event. When the event is raised, the code within the event handler is executed. In the definition of an event handler, each event handler normally provides two parameters that allow the developer to handle the event properly: the first is the sender which is a reference to the control that produced the event (that is necessary since the same event in multiple controls can be handled by the same method) and the second is a reference to an array of event-related data (information such as the location of the mouse for mouse events or data being transferred in drag-and-drop events).

3.3 Problem Definition

Graphical user interfaces are critical components of many software products. Developers dedicate a large portion of development effort to the design and implementation of GUIs. Given their prominence in software development and their role as mediators between users and computers, GUIs must be designed and implemented correctly. Bishop and Horspool (2004) define 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 chapter, we call this situation a *transitional turbulence* (or what Cooper and Krishnamurthi (2006) call glitchiness).

Transitional turbulence or turbulence of transition (Lorenz, 1963) is a term applied to fluids in which a fluid's overall state and its surface remain troubled after an external force is applied to it. To translate the term into computational terminology, transitional turbulence is a period of chaotic or unreliable variation in the state of a software system that leads to instability in its execution. It can result from changes to the system's interconnected components and lead to an external presentation that does not accurately represent the system's expected behavior and inconsistencies in the visible state if the display must render new frames before stability is reached. Suppose a certain interaction affects a control within an interconnected cluster of controls. This generates a "ripple effect" where all controls around the initial control are affected, and then all controls around those controls, and so on until the effects propagate through a large portion of the system. In this particular situation, transitional turbulence can be defined as the *ripple effect* caused by the *chained execution* of multiple interconnected events. 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.

In this typical implementation of a GUI, a user interacts with the GUI by triggering 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 into the built-in event-handling mechanism provided by the language. This asynchronous event-handling architecture is called *implicit invocation* (Shaw, 1996).

In an implicit invocation architecture, each control responds to events in which it is "interested". A response to an event may result in the control changing its state and triggering new events that notify other controls of the state change. Thus, one control responding to one event may trigger chains of events affecting several other controls in the GUI. In complex cases, these event chains may be long; reaching a stable state may require the processing of many events. The propagation of events is done by an event-handling layer of the system, not by the controls themselves. Therefore, from the perspective of an application developer, the order in which events are handled is non-deterministic.

Although the GUI's controls are loosely coupled from a communication perspective, an implementation usually arranges them into some hierarchical data structure. For example, the controls within a Web-based GUI are organized by the Document Object Model (DOM) within a browser. Similarly, the controls within a C# desktop GUI are organized by a separate class named `Designer`; this class abstracts the UI visual representation and contains a hierarchical set of controls. The display system uses these data structures when it periodically renders the GUI onto the screen. This is where transitional turbulence can arise due to the loose coupling of the GUI controls. The controls are frequently interacting with each other, the nature of these interactions can be unpredictable, and the ramifications of an interaction can affect multiple objects. This interaction generates a “ripple effect” where all objects around the initial object are affected, and then all objects around those objects, and so on until the effects propagate through a larger portion of the system. The processing of a long chain of events may span several cycles of the display system. A control may be rendered with a state that is inconsistent with the states of other controls. This may result in displays that are temporarily inaccurate or misleading from the perspective of a human user.

In this implicit invocation style, an action is performed on a given control, this control notifies the system about an interaction that occurred on it. The system calls the interested control, prompting this control to answer this call by invoking specific functions that are declared within it to react to these specific changes. Once a function has finished executing, the system goes back to idle and the execution is returned to the manager. This is an asynchronous process that breaks the ripple effect into several time-consuming ripples. The event-handling approach described above is organized according to the well-known Observer design pattern (Gamma et al., 1995).

The shortcomings of this structure affect the accuracy of the GUI and the predictability of the operations on the GUI. Usually, the effect of transitional turbulence spreads through 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 to 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 intends to be a smooth and coherent experience may appear choppy and incoherent to the waiting user.

Because the events are handled asynchronously, the order and timing of change propagation is machine-dependent (Salvaneschi et al., 2014, 2015). 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 Observer pattern does not guarantee the order in which events will be handled. 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. The asynchronicity of the implicit invocation yields many benefits in the development of user interfaces, but the implicit invocation also imposes some liabilities on the system as well. The absence of control of the order in which events are received and the responses to those events occur in the implicit invocation event system causes issues relative to the correctness. An example of that would be two components A and B which are executed in this order: A executes, and then later on the same cycle, B executes as well. When B affects A, the execution of A happens before the execution of B in this cycle, which means that A will not execute a second time and the modification made in A by B will not be identified and reacted by A until the next update cycle.

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 states. 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. Functional reactive programming (FRP) libraries —such as Sodium (Blackheath and Jones, 2016), Reactive Extensions (Malawski, 2016), ReactiveBanana (Chupin and Nilsson, 2019), and Elm (Czaplicki and Chong, 2013)—are effective alternatives to the use of the Observer pattern. According to Czaplicki and Chong (2013), 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 poorly supports 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. Therefore, for each control, the execution is still handled internally as an asynchronous event.

In the path to achieving this, our research aims to answer the following specific Research Question:

> *Can dependency graph-based execution reordering and self-adjusting state recomputation be used to augment the implicit invocation built-in event system resulting in reduced transitional turbulence execution inconsistencies and increased accuracy while maintaining performance in the chained execution of multiple control's event handlers specifically found in Web and desktop graphical user interfaces?*

To answer this question satisfactorily, we pursue a few other secondary research questions.

A. *Can we augment the built-in event system of .NET to improve the execution control, system responsiveness and accuracy in the chained execution of multiple control's event handlers?*

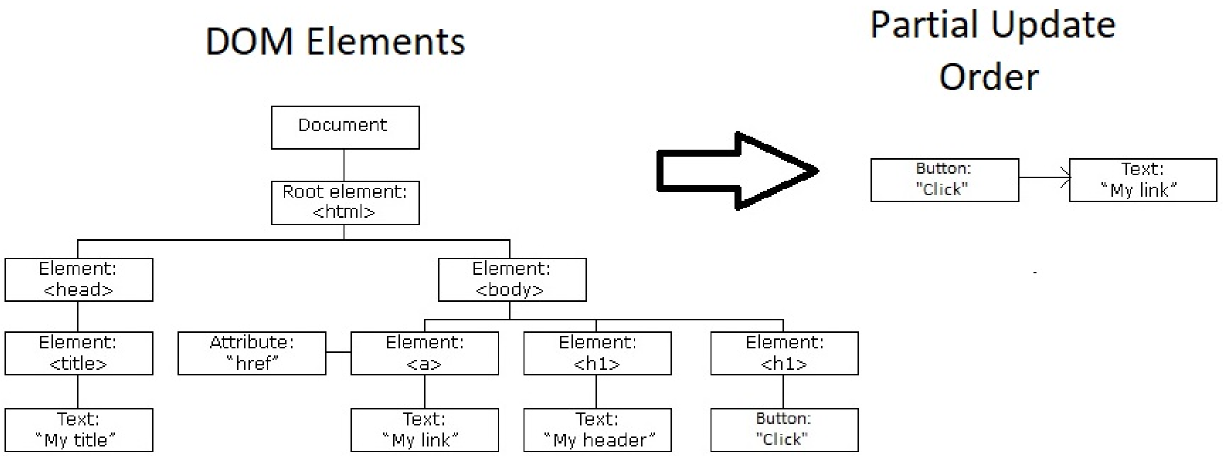


Figure 3.1. Partial update order based on the original user interface.

B. *Can the dynamic extraction of the dependencies among controls and the creation of a dependency graph represent the relevant constraints to the invocation order of the control's event handlers?*

C. *Can the regular checking for changes in the control's state on the user interface's hierarchy enable prompt rearrangement of both the dependencies and the execution order?*

D. *Can the implementation work without degrading the performance of the original application?*

Figure 3.1 depicts how our approach uses the original hierarchy to extract dependencies and define a partial update order.

In the following section, we describe the implementation aspects of a proof-of-concept system that can answer the research questions posed above.

3.4 Design and Implementation

This section describes our reactive approach to designing and implementing dynamic user interfaces (UI). 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

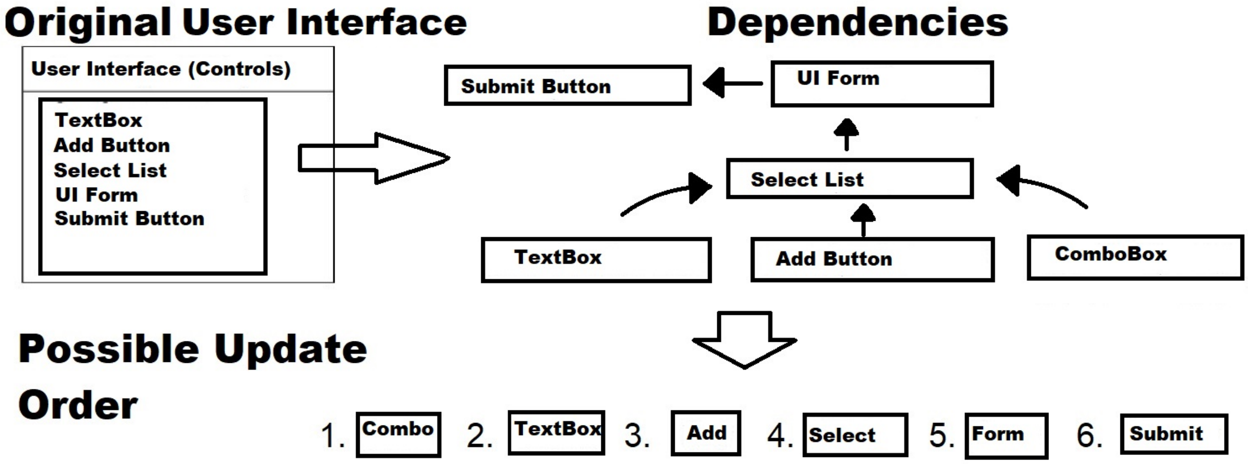


Figure 3.2. Dependency graph analysis and update order creation.

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. Figure 3.2 depicts the process of analysis starting from the UI hierarchy to determine the update order.

After the creation of the page, our approach creates the form's dependency graph by calling our `CreateGraph()` function on each of 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 the `getTarget()`, getter, and setter methods.

The dependency graph is created based upon the analysis of the UI hierarchical data structure that contains all 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. Each node of this dependency graph represents exactly one of the reactive controls in the GUI. Algorithm 1 encodes the dependency relationships between all pairs of controls using a directed acyclic graph.

Algorithm 1 Function CreateDGraph: Building the dependency graph from the DOM Marum et al. (2020a).

```
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
```

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 the built-in event mechanism of the application.

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. We consider three cases in which the component is modified. We implement different protocols on each as listed below:

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.

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 dependencies for all controls affected must be recomputed.

Algorithm 2 shows the first part of the `UpdateGraph()` that updates the dependency graph according to the type of changes made to the GUI.

Algorithm 2 Function UpdateDGraph: Reanalyze the DOM to update the dependency graph and create a partial update queue - part 1 (Marum et al., 2020a).

```
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 each control;
        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 Dependency Graph;
                    if Edge between Cont and P do not cause a cycle then
                        Create a Edge in the Graph between Cont and P;

                    end
                end
            end
        end
    end
end
```

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 3.1 illustrates how our approach defines a coalescing event chain from the original structure.) The idea is to try to realize the direct and indirect effects of the launching event within one cycle of the event-handling system.

Algorithm 3 shows the second part of the `UpdateGraph()` that updates the dependency graph according to the type of changes made to the GUI.

Algorithm 3 Function UpdateDGraph: Reanalyze the DOM to update the dependency graph and create a partial update queue - part 2 (Marum et al., 2020a).

```
if continues... then
    while continues... do
        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 in the Graph between 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
```

This “chained execution” approach addresses the *transitional turbulence* problem described in Section 3.3 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 the original application 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 develop our augmentation using C# with 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 experimental approach described in Section 3.5.3. 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 augmentation and experimental implementation. A third reason for the choice is the platform's metaprogramming and reflection facilities. These enable us to conveniently implement the augmentation's analyses needed to build and update the dependency graph.

Along with our augmentation, we develop a library with several controls that extend the most popular controls from the .NET GUI framework. We develop 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 the getter, setter, and `getTarget()` functions, which must be implemented differently for each reactive control. The augmentation 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, 2, and 3.

The augmentation's code is the same on both the Web and desktop platforms. However, there is a flag that indicates whether or not the augmentation 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 comparison, 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.

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

3.5 Test Methodology

To evaluate our augmentation, we compare it to other similar environments built using other technologies:

- Original .NET system;
- Sodium library (Blackheath and Jones, 2016);
- Reactive library Rx.NET (Malawski, 2016);

We start by measuring the overall improvement of our approach against the original .NET itself to determine whether our platform achieves better results against the baseline.

In the next subsections, we establish the characteristics of two similar reactive libraries, Sodium (Blackheath and Jones, 2016) and Rx.NET (Malawski, 2016). Their performances are compared to each other as well as to our platform.

3.5.1 Sodium

The Sodium library (Blackheath and Jones, 2016) 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 Elliott (2009). 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 (2016) argue that Sodium is a system with 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 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 3.3. Various researchers, such as Krouse (2018) and Bregu et al. (2016), 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 into non-FRP computations. This can lead to an unsafe state in applications, especially those running on the Java Virtual Machine (JVM) or the .NET framework. Furthermore, 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.

By comparing against Sodium, we address a comparison between our approach and a mainstream reactive library for .NET. The comparison addresses the capacity of our approach to adapt to changes in the user interface, recompute the dependencies, infer a new evaluation order, and finally coalesces all the events in such an order that each event that depends upon the execution of another component will always execute after it.

3.5.2 Rx.NET

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 multiplatform systems. Malawski (2016) 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 a 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.

The comparison of our approach against Rx.NET is a comparison with another well-known current implementation of a reactive programming library. The goal is to measure the performance on both startup and after each UI interaction and to determine the accuracy after each interaction against the same metrics as we do with Sodium.

In the next section, we explain our experimentation setup and the results we collected from this experimentation.

3.5.3 Experimentation Setup

For our comparisons, we use self-completion forms and Web pages. We implement each form using the original .NET, Sodium, Rx.NET, and our augmentation and then compare the performance of all four 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 the 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.

We conducted tests using three different test scenarios:

1. A shopping list that takes a list of the prices of items, adds the prices, and applies the tax value;

2. A calculator for geometric forms. 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.

For each of these tests, we implement the forms using several reactive controls from our reactive augmentation: `ReactiveButton`, `ReactiveTextBox`, `ReactiveComboBox`, `Reactive ListBox`, `ReactiveLabel`, and `ReactiveRadioButton`. Then, we configured the controls to react upon certain reactive interactions. We scattered instances of these reactive controls across the example form (or page) and then linked them to each other. For example, when one of the buttons is clicked, it uses the value of a given textbox to populate other controls with predefined values. The list below itemizes the interactions between the controls that we programmed in each one of the experimentation examples:

- 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 above controls

Each of these controls has a similar version that we programmed into the library. We categorize our results in two ways: performance and accuracy.

During each test run, we use the .NET `StopWatch` class to measure the time spent starting the application and 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 (Microsoft, 2019) of the class `StopWatch` claims that the default function 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. This measurement is linked with secondary research question `D`.

Besides the performance, we devise the following two metrics and a statistic to evaluate the accuracy of the result for each box inside the form:

Latency: 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 measure the time needed to get the whole Web page into the desired state from the triggering action. This test is concerned with answering the secondary question `C` from Section 3.1.

Errors: Count the number of the errors detected for a sequence of complex interactions. This test is concerned with answering both primary research question and secondary research question `A` from Section 3.1. 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.

Average Errors: Count the average number of errors on a test where errors are detected. This test is concerned with answering both the primary research question and the secondary research question `B` from Section 3.1. To measure this, we first plan to determine how many components are incorrectly ordered on a test where are errors detected. Second, we count the total number of errors, count the total number of runs that reported an error, and then compute the average.

Our tests emulate the behavior shown in Figure 3.3. These tests were developed to

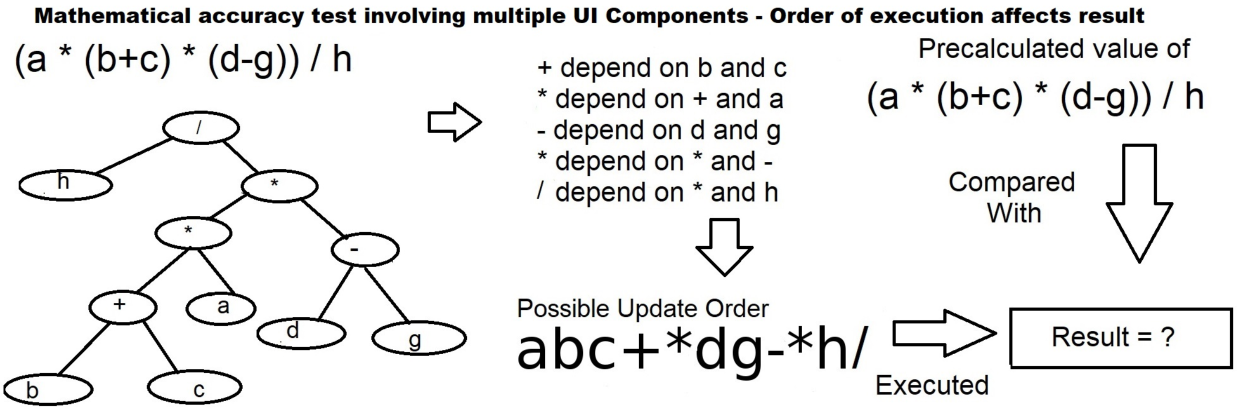


Figure 3.3. Test scenario expected behavior.

show that our 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, then 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, then each one of these smaller interactions will be more accurate, thus making the entire operation more accurate. The idea is that, as it happens in mathematical computations, the order of execution affects the final result. If our system guarantees that the accurate order will be respected in every cycle, than the accuracy of the entire operation can be guaranteed. Since operations in a user interface can often be broken into smaller interactions between components, if we can guarantee each one of these smaller interactions are accurate, then the entire operations will have its overall accuracy increased.

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 Rx.NET 4.4.1 and Sodium 2.0 for the Sodium tests.

3.6 Results and Analysis

The implementations using our system took an average of 0.30 seconds more to start up than the Sodium and Rx.NET applications and 0.35 seconds more than the original .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 augmentation code. Table 3.1 illustrates the average startup time for each implementation using our augmentation, original .NET system, Rx.NET, and Sodium.

Platform	**Scenario 1**	**Scenario 2**	**Scenario 3**
.NET	21.24	20.45	22.28
Sodium	29.58	30.65	31.12
Rx.NET	27.58	28.65	29.12
Our Platform	51.24	55.45	58.28

Table 3.1. Startup time in milliseconds for each test scenario

For each test, we calculated the time needed to execute the full self-completion routine. We started the stopwatch with 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 20% to 10% of the time that the Sodium implementation took on the same form. The average is taken over the 50 executions. On each execution, the form was opened and self-completed, then the information was cleared from the form before the execution was repeated.

Figure 3.4 shows the average performance graph in each of the executions for the three test scenarios in both implementations.

Against the Rx.NET implementation, our platform performed the fill on the entire form in an average of 30% of the time that the Rx.NET implementation took to do the same task. And finally, against the original .NET implementation, our platform performed the fill on the entire form in an average of 50% of the time that the .NET implementation took to

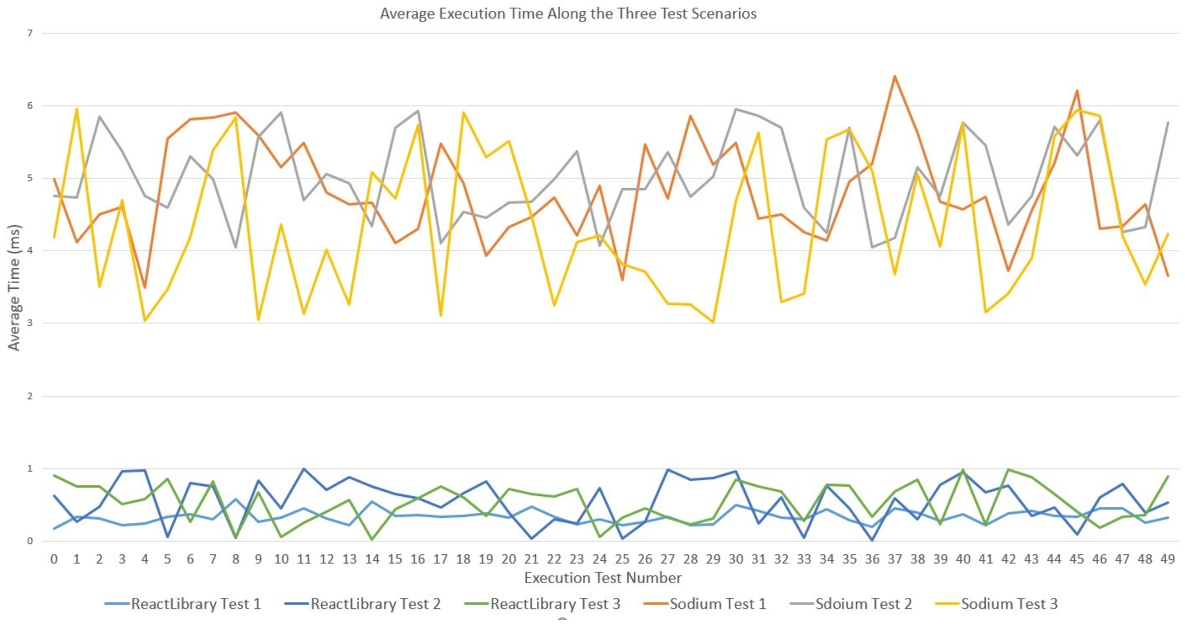


Figure 3.4. Average performance graph - All 3 scenarios. Sodium vs. our platform.

do the same task.

With respect to accuracy, our system outperforms all the other implementations. We measured the 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.

Figure 3.5 shows an average performance graph for our platform against Rx .NET over each one of the executions for the three scenarios.

This test case seeks to highlight the effect that the execution order has on achieving a correct final display. The inconsistencies observed in the original .NET implementation tests show that an original .NET implementation displays the issues that we are considering here.

Neither Sodium nor Rx.NET fully addressed the problem here and display many inconsistencies because they do not consider the dependencies in scheduling updates. Our augmentation 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

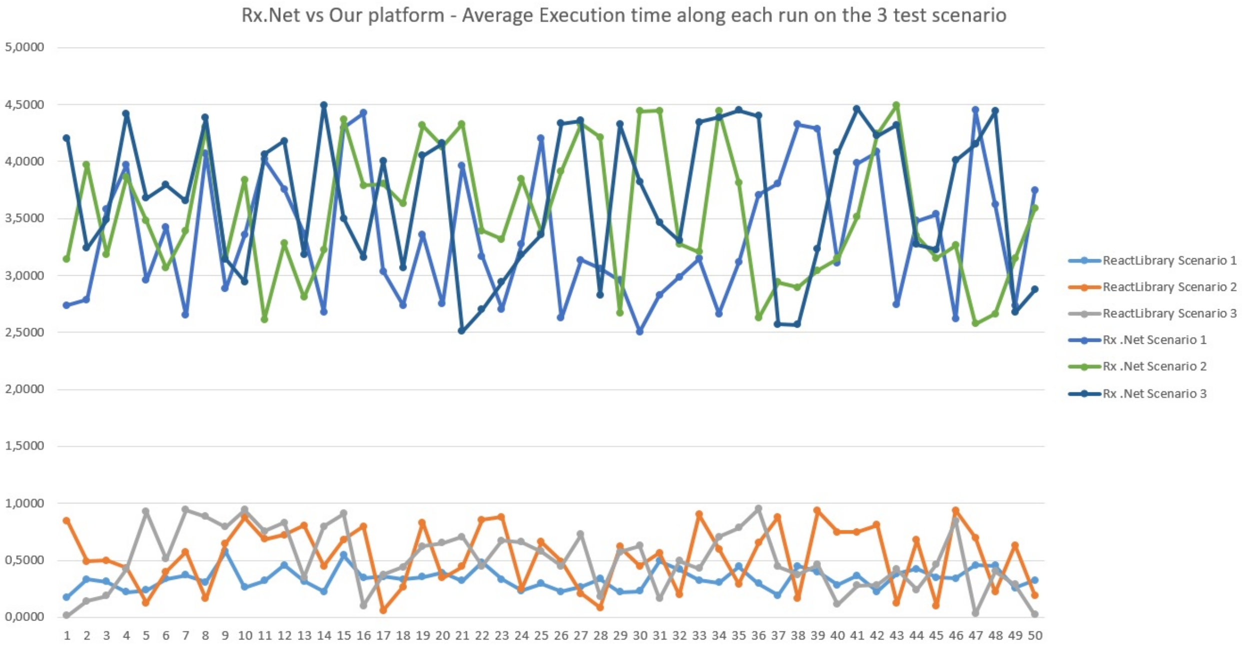


Figure 3.5. Average performance graph - All 3 scenarios. Rx.NET vs. our platform.

the form at all points during execution. This works like a chain of falling dominoes. If a later one falls before a previous one, the inaccurate result may be perceived by an observer.

Figure 3.6 shows an average performance graph for our platform against original .NET system over each one of the executions for the three scenarios.

Table 3.2 shows the result of the first test scenario, which implements a shopping list user interface for all four systems.

Platform	**Total Cycles**	**Total Errors**	**Avg. Errors per Cycle**	**Latency in Cycles**
.NET	500	12	5	3
Sodium	500	8	2	1
Rx.NET	500	9	2	1
Our Platform	500	3	1	1

Table 3.2. Test results for scenario #1

The original .NET implementation yielded errors in almost 25% of the cycles. Both the Rx.NET and the Sodium-based implementation had a worse performance but not by a large margin if compared to our augmentation. They yielded errors after a wave of updates

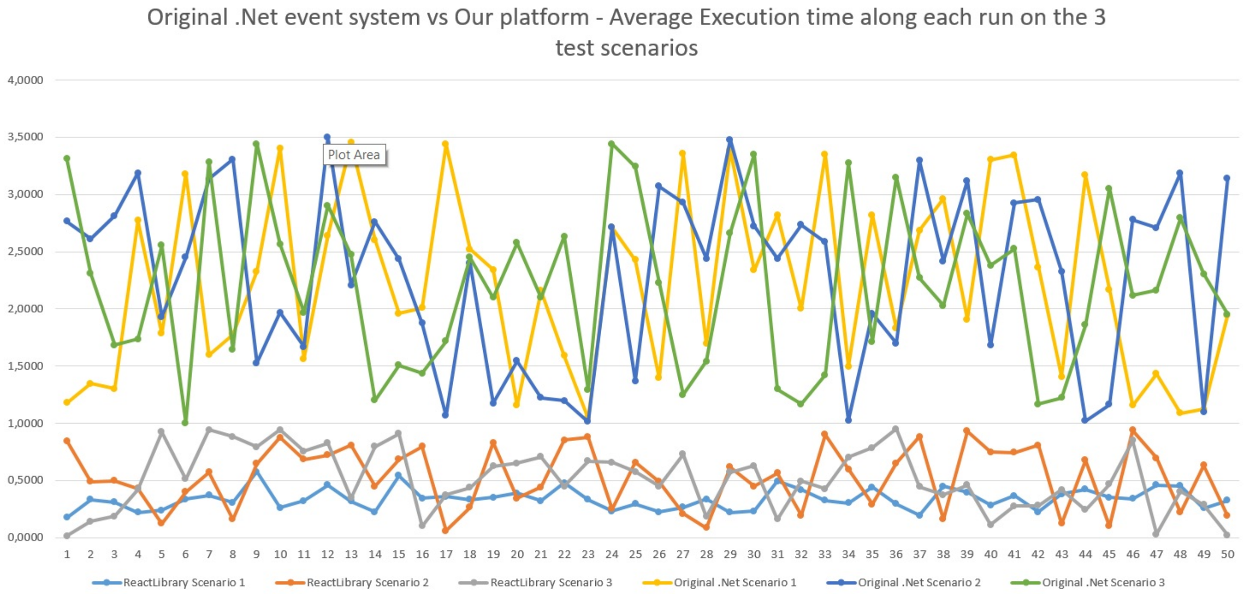


Figure 3.6. Average performance graph - All 3 scenarios. Original .NET vs. our platform.

in 12% to 15% of the tests. Our implementation yielded errors in less than 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 3.2, 3.3, and 3.4.

Table 3.3 shows the results of the second test scenario, which implements a geometric self-completion calculator user interface for all four systems.

Platform	**Total Cycles**	**Total Errors**	**Avg. Errors per Cycle**	**Latency in Cycles**
.NET	500	20	1	5
Sodium	500	6	1	1
Rx.NET	500	8	2	1
Our Platform	500	2	1	1

Table 3.3. Test results for scenario #2

Table 3.4 shows the results of the third test scenario, which implements a metric converter user interface for all four systems.

With respect to performance, the graph depicted in Figure 3.4 shows that our augmentation performed the same task in 10% to 20% of the time in milliseconds that original .NET, Rx.NET, and Sodium took. The execution time for our implementation was less than

Platform	**Total Cycles**	**Total Errors**	**Avg. Errors per Cycle**	**Latency in Cycles**
.NET	500	16	1	4
Sodium	500	7	1	1
Rx.NET	500	10	2	1
Our Platform	500	3	1	1

Table 3.4. Test results for scenario #3

1 second while other three implementations' execution time was up to 6 seconds. Our implementation was substantially faster than the other implementations. The performance of the three tests by our augmentation are near the bottom of the graph. The tests performed by Sodium, Rx.NET, and the original .NET are near the top. The graphical distances between the graph lines give the magnitude of the difference between both performances.

How can we explain the performance differences? Sodium, Rx.NET, and original .NET implementations are based on asynchronous event-handling systems as described in Section 4.3. 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 between events. One stream does not connect directly to 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 reactively treats 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.

The tests described in this section demonstrate the effectiveness of our approach. Our augmentation outperformed Sodium and Rx.NET, both state-of-the-art reactive libraries. It alleviated the performance problems caused by the asynchronous nature of the event-handling approach used by original .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 augmentation 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 augmentation can give more accurate results than Sodium for a variety of user interface applications, while yielding small performance improvements.

3.7 Discussion

Most papers focus primarily on how to build programs by generating and relating different parts of the source code. Czaplicki (2012), Czaplicki and Chong (2013), Foust et al. (2015), Krishnaswami (2012), Reynders et al. (2017), and Salvaneschi et al. (2014) 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. (2015) describe 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 (2012) and Czaplicki and Chong (2013) describe the development of 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.

The approach in this chapter 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.

3.8 Conclusion

In this chapter, we describe the development of a reactive programming (Bainomugisha et al., 2013) approach that analyzes the complex relationships among the GUI controls, encodes these dependencies in a dependency graph, and then uses the graph to rearrange the updates in an order consistent with the dependency constraints. It builds the graph when the GUI starts up and then rebuilds it whenever it detects that the dependencies might have changed. The approach thus coalesces the processing of a chain of what may be several events in the original system into a single, large-grained event that updates the states of many controls at once. Although our approach does not totally eliminate the transitional turbulence that can cause inaccurate or misleading displays, it does potentially decrease the number of inaccuracies as well as increase the performance of the system. The primary contributions of our research in this chapter are as follows.

A. As defined in section 3.1, our primary research question involved the reduction of transitional turbulence execution inconsistencies and increased accuracy while maintaining performance. Our results indicate that, on average, our augmentation do maintain performance while mitigating transitional turbulence and actually achieve better performance when compared with other reactive implementations and against original .NET as well. Our augmentation requires less total time and exhibits fewer errors, at the cost of a modest increase in startup time compared to all three alternatives. Each application developed with the prototype augmentation required approximately twice as much time to start up as the corresponding original .NET application required. However, it was able to complete the entire chain of form updates in a small fraction of the time the corresponding original .NET application required. In addition, it exhibited significantly fewer visual inaccuracies than the corresponding original .NET application exhibited. Based upon the results of our experiments, we concluded that our approach improves performance and results in a more accurate behavior. This result also answers the secondary research question `D` which is also related with maintaining the performance.

B. Secondary question `B` was related to extracting dependencies and define an event-handling order. In the section 3.4, we describe the mechanism that analyze each UI control and copy it to the dependency graph. In this process we do not modify the original UI, instead we mirror the controls from the original UI highlighting the dependency relationships between the controls to build a dependency graph. The execution order inferred from the dependency graph in our approach relies upon the mathematical concept of evaluation order that consists in a order in which each node of the graph is visited only once and all nodes are visited after the nodes in which they depend on. The final evaluation order must not violate any of the dependencies expressed in the graph. Our experiments shows that the execution of the UI is functionally similar to the non-reactive approach that must manually invoke each UI control sequentially, which means that the dependency relationships detected and extracted from the UI is a valid representation of the *depends-on* relationships we intend to capture. That proves we are successful in defining a function to extract the dependencies and use them as constraints to generate an event-handling order based upon the interconnection between reactive controls during the execution cycle.

C. Secondary question `C` was related to the adaptiveness and prompt reconfiguration of the dependency graph whenever it was necessary to match changes in the original user interface hierarchy. The design of our approach rely on the fact that by copying the current version of the UI into the dependency graph we are keeping a history of the previous iteration safely stored. This means that when the next update happens we can compare the current version of the UI against the previous version to detect any difference between the two states. That allow the system to immediately react to any change detected by rearranging the dependency graph to match the current state. The tests described both in Section 3.5 involved modifications in the tree that modified both the state of a control, and also changes in the structure of the UI. Those tests were created with the purpose of measuring our system's capacity of self-adjustment and prompt reconfiguration when a change is detected. Our experiments shows that our augmentation improves responsiveness and enables prompt rearrangement of both the dependency graph and the execution order even when there are structural changes in the hierarchy. This results in an application that can self-adjust to redefine the dependencies as they change to maintain the accuracy and performance benefits observed.

D. Secondary question `A` was related to maintaining accuracy and responsiveness when dealing with the chained execution of multiple controls. Our augmentation performed better than other reactive libraries (e.g., Sodium), demonstrating that our approach improve the .NET event system providing better execution control and responsiveness while maintaining performance and even increasing it. By augmenting the event system we maintain native code working while targeting only the subset of the controls that the user chooses to.

CHAPTER 4

DEPENDENCY GRAPH-BASED REACTIVE AUGMENTATION OF GAME ENGINE APPLICATIONS

4.1 Introduction

Virtual reality (VR) and augmented reality (AR) applications, collectively referred to as virtual environments (VEs), are *reactive* in nature. They respond to events, which may correspond to an interaction with the outside world (e.g., a user's movement) or with other components of the application (e.g., changes in the values of important data attributes). VR/AR applications apply a variant of the implicit invocation model (similar to the approach discussed in Chapter 2). These applications work similarly to animations, using a game loop. A *game loop* is a time abstraction in which the states of all components in the game are updated on each cycle. The states are redrawn onto the screen. Otherwise, a game is an implicit invocation system. Each component relinquishes control to a central game manager. Whenever the game loop starts to update the overall state of the game, the manager notifies each component to update its own state (as a response to this game loop). After each cycle, the game renders its image on the screen.

As game engines benefit from implicit invocation, they also suffer from implicit invocation's liabilities. The absence of control over the order in which events and their corresponding responses occur may cause seemingly incorrect behaviours. In such applications, whenever an interaction is performed, the system may not display the desired result immediately and thus seem to behave erratically.

Since a VE is a reactive system, its behavior is modeled by the interactions between the components and external actors and the interactions between components. An interaction with one component generates a “ripple effect” where all components around the initial component are affected, and then all components around those components, and so on until the effects propagate through a portion of the system. This is called *transitional turbulence* as defined in Chapter 3. Transitional turbulence can result in inconsistencies in the visible state if the display must render new frames before stability is reached.

In this chapter, we examine an augmentation of the implicit invocation model present in the game engine by introducing some execution control and partially removing the instability corresponding to the transitional turbulence. Our augmentation reorders the execution of events based upon the transitive dependency relationship (as defined in Chapter 3). The augmentation uses the dependency graph to infer the invocation order of the update function of all components. This approach results in the entire “transitional turbulence” often being completed within one update cycle.

In this chapter, we focus our attention on applications running on the Unity3D game engine, which is a popular platform for low-cost VE applications. During our tests on Unity3D, we found that Unity3D does not provide a way to control the order of execution. As noted in Chapter 2, another disadvantage of using the implicit invocation pattern is the loss of control over the order in which components execute.

This degree of control and accuracy guarantee is essential for high accuracy systems. In such systems, simulated interactions must occur in the same order as the interactions would in a corresponding real-world situation. If they do not, then the simulation fails to be realistic. An example is a chain of dominoes falling. When the first falls, the second falls only when the weight of the first domino causes it to fall, and then the third falls similarly until the last one falls. If one of the dominoes does not perform its fall correctly, the entire chain is compromised. In doing this, we seek to remove the transitional turbulence mentioned above.

We are not the first to propose creating reactive programming frameworks for virtual environments. There have been many solutions proposed for this problem in the past, but most of these utilize specialized or purpose-built development environments (Blom and

Beckhaus, 2008; Kawai, 2014; Westberg, 2017; Jankovic, 2000; Stefan et al., 2018; Wiebusch and Latoschik, 2014; Lange et al., 2016). In this work, we propose an alternative approach that attacks a common flaw present in most modern, widely used development environments, such as Unity3D. Furthermore, we propose designing this approach in such a way as to be intuitive for developers who are already familiar with these environments. The approach that we chose utilizes the development environment's native object hierarchy to introduce more control into the implicit invocation model internally used by game engines to handle the interaction between components by defining reactive dependency relationships between objects and inferring an execution order from them in a virtual environment.

On the path to achieving this, our research aims to answer the following primary Research Question:

> *Can dependency graph-based execution reordering and self-adjusting state recomputation be used to augment the implicit invocation game loop resulting in reduced transitional turbulence execution inconsistencies and increased accuracy while maintaining performance on graphical applications built on game engines?*

Our primary means for answering this question is to evaluate the improvement in the accuracy and predictability of simulations with mathematical and logical constraints. To test it appropriately, we develop an automated testing platform that includes game object hierarchies with the purpose of evaluating mathematical expressions and programmatically determining their values. In the remainder of this chapter, we analyze Unity3D architecture in Section 4.2, then we define our problem in Section 4.3. In Section 4.4, we show the details about the design and implementation of a proof of concept tool. In Section 4.5, we compare our solution's performance with the results achieved by the same solutions built using the default Unity3D event system and also using UniRx (Kawai, 2014), an existing, reactive library for Unity3D. In Section 4.6 we discuss related work on reactive programming, especially the ones related with games. Section 4.7 summarizes the results of this work. In the next section, we explore Unity3D architecture.

4.2 Unity3D Architecture

A Virtual Environment (VE) is defined by Furness and Barfield (1995) as "any technology that induces targeted behavior in an organism by using artificial sensory stimulation, while the organism has little or no awareness of the interference". The perceived environment could be a captured "real" world just as well as a completely synthetic world. LaValle (2017) devised a more general concept in which the perceived environment need not seem "virtual" but can also be an augmented, or engineered, perception of reality.

The development of VEs is the final product assembled from several different developed products into a coherent application as defined by Cowan and Kapralos (2014). A game engine is a collection of different tools, utilities, and interfaces that aid the development of the various tasks that make up a VE on a single integrated application. Unity3D (Unity Technologies, 2019) is a game engine that uses a hierarchical, component-oriented programming approach to organize these components into an application. Unity3D's native language is C#. Development of applications using Unity3D often takes advantage of many .NET framework characteristics, as explained by Hocking (2015), including the .NET common type system, class library, and other resources from the .NET platform.

The Unity3D architecture is based upon the idea that a `Scene` is a hierarchical collection of `GameObjects` (Seligmann, 2018). Figure 4.1 highlights the relationship between game objects and components in a scene.

`GameObjects` are placeholders without logical behavior and components must be attached to the `GameObject` to implement a specific behavior into that `GameObject` (Unity Technologies, 2019). `GameObjects` are defined by Baron (2019) as the building blocks for scenes in Unity3D. Unity Technologies (2019) notes that the `GameObject` class provides a collection of methods that allows the developer to find the methods, makes connections and sends messages between `GameObjects`, and add or remove components attached to the `GameObject`.

The behavior of a `GameObject` depends upon its attached components. Each com-

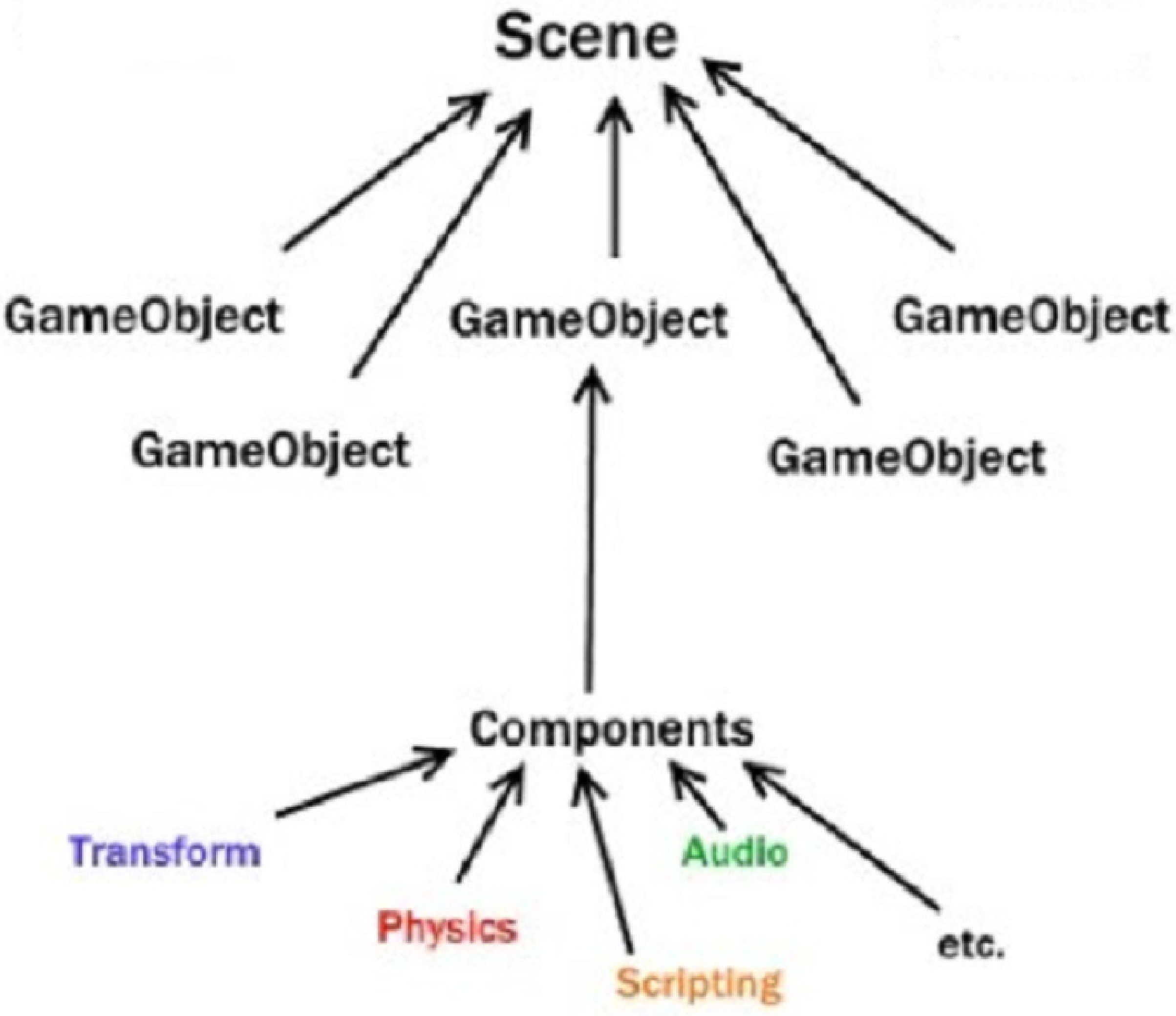


Figure 4.1. Unity3D scene hierarchy (Marum et al., 2019).

ponent's functionality can be augmented by attaching the script that describes that desired functionality. A script is a code file. It describes the characteristics of a component and defines functions the component can perform and how the component reacts to update and start events. A script is similar to a class in that it defines the behavior of an object in an object-oriented program.

Unity Technologies (2019) defines `Script` as a class that inherits from the built-in class `MonoBehaviour` and contains a predefined set of methods including the game event handlers `Start()` and `Update()`. Each time a script is attached to a `GameObject`, Unity3D

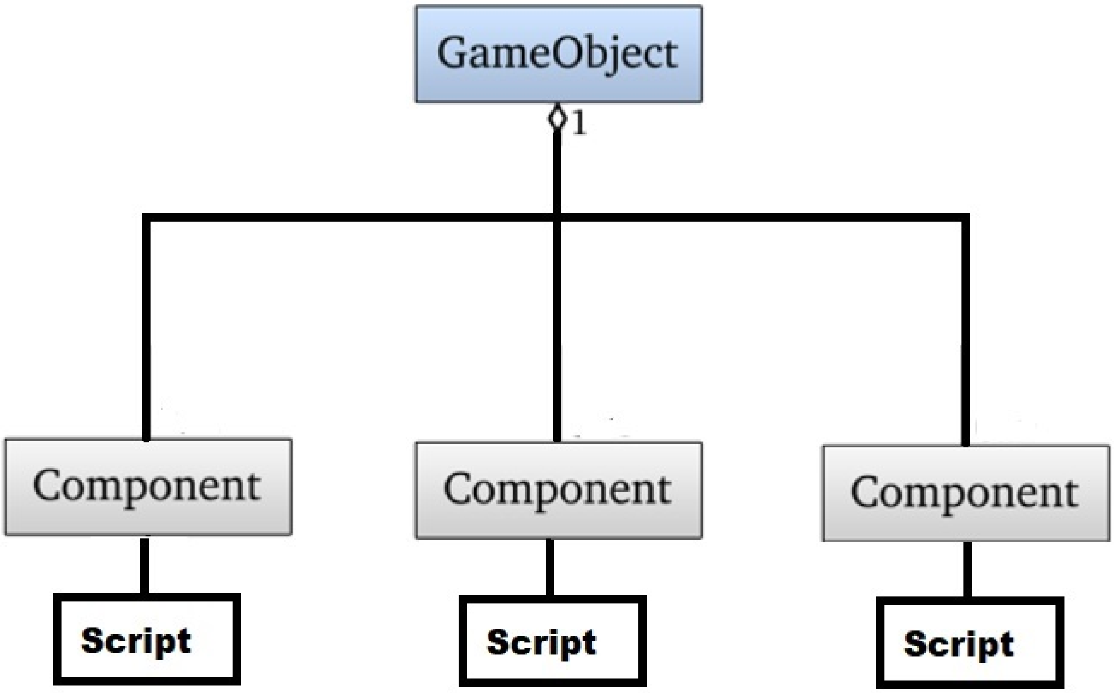


Figure 4.2. Unity3D relationship of multiple scripts being implemented in multiple components and attached to a single `GameObject`.

creates a new instance of the component defined by that class. Each game object-component pair is unique since only one instance of a distinct component can be attached to a specific game object. Baron (2019) explains that the same script can be attached to multiple `GameObjects`, implementing a specific component with different characteristics in each one of them. Each `GameObject` can have multiple different components (each one from a different script), enabling different capabilities to the same `GameObject`. New abilities can be attached or removed dynamically even during runtime, giving programmers the ability to switch functionalities to a particular `GameObject` given specific circumstances. Figure 4.2 shows the relationship between `GameObjects`, components, and scripts.

Scripts allow the developer to create a component from scratch, handling game events, creating and modifying other components' and its own properties over time, and responding to user input in any way necessary (Baron, 2019). The `MonoBehaviour` class is the base class from which every Unity3D script derives, by default. When a C# script is created

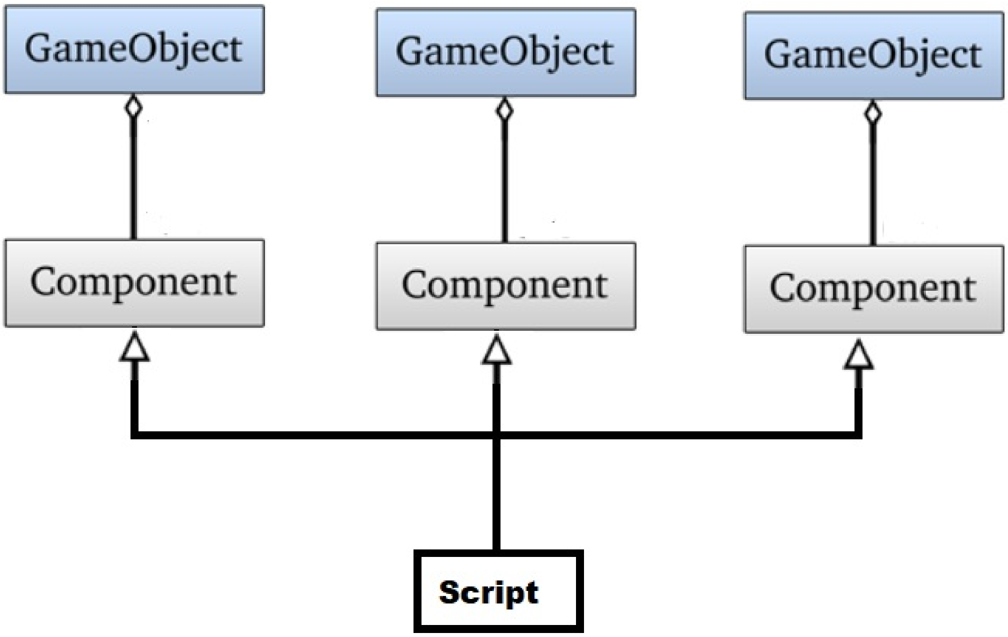


Figure 4.3. Unity3D relationship of one script being implemented as component and attached to multiple `GameObjects`.

from Unity3D's project window, it automatically inherits from `MonoBehaviour` and provides the event handlers and other information that is fundamental for Unity3D to handle the script (such as ID, name, and a pointer to the parent `GameObject`). The `MonoBehaviour` class provides the framework that allows the developer to attach its script to a `GameObject` in the editor, and it provides access to a large collection of event messages, which allows the developer to execute its code based on what is currently happening in the project. Figure 4.3 shows the relationship between scripts and their instances on each `GameObject`.

4.2.1 Unity3D Game Loop

A Unity3D application does not behave in a typical manner for interactive software. Instead, it behaves like an animation, which means, after a given time span, a new frame is generated reflecting the current state of the application. A key concept in game programming is the game loop, described by Gregory (2018) as a sequence of application processes that updates the state of any applicable object in the game and then renders the frame. The game loop is implemented as an abstraction of the CPU's clock cycle by implementing a ticker (which is normally based upon the relative time difference from the last game loop) so the code in every frame executes in a consistent and independent manner (Sherrod, 2006). Since Unity3D is a closed-source, proprietary game engine, the specifics of the core game loop and its rendering pipeline are abstracted, so Unity3D allows the developer to focus on what to do whenever the actual update process starts, instead of the actual time span (Gregory, 2018).

The `Update()` function is the function inside each script that contains the code that is executed in each game loop in Unity3D (Unity Technologies, 2019). The `Update()` function of every script is called before the frame is rendered and once per frame. It is the main workhorse function for frame updates. The update mechanism of Unity3D has every object expose an `Update()` function that is indirectly invoked in every frame (Baron, 2019). That means the game loop is not aware of the content of the `Update()` function of each component. The game loop does not even call the `Update()` directly. It just notifies each component that it is the component's time to update and the component invokes its own `Update()` method, so that every component that has a `Update()` method will be invoked in every game loop. Thus, the developer encapsulates the reaction to the updating process of the component's relevant information through a single interface.

Any component that is a valid Unity3D script must inherit from the `MonoBehaviour` class, which provides multiple event handling methods for updating, starting, and destroying processes (Unity Technologies, 2019). In the script code of each component, the developer implements the behaviors that are needed in each of the instances created for this component. Since `MonoBehaviour` is the common inherited type for all components, Unity3D dynamically maintains a list of all entities that are components. Throughout the game loop, a game manager initiates the game loop, during this game loop, the manager notifies each component in the scene that inherits `MonoBehaviour`, allowing each component to call its own `Update()` method. This update mechanism is a modified version of the implicit invocation architecture

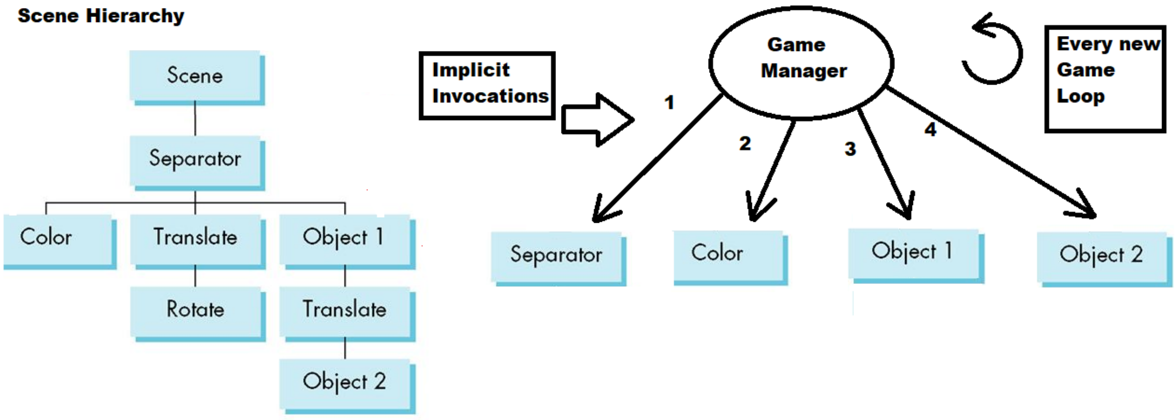


Figure 4.4. Execution style of Unity3D based on the scene hierarchy (Marum et al., 2019).

model. Figure 4.4 illustrates how this process works.

In each iteration of the game loop, a game behaves as an implicit invocation system. The components are controlled by a central game manager that notifies each component to update its own state. Then the game updates its rendered image back to the user. So, this is the difference between the game loop abstraction and the regular implicit invocation pattern. Even though there is still an implicit mechanism that is outside the control of the user, the game manager is guaranteed to call each component once per iteration of the game loop.

4.2.2 Tests on Unity3D Execution Order

One of the details hidden inside the proprietary implementation of the Unity3D game manager is how Unity3D organizes the notification and invocation of each update method since there is no defined order of execution for objects within the game's hierarchy. The only comment about this issue in the Unity3D manual (Unity Technologies, 2019) is: "By default, the `Awake()`, `OnEnable()`, and `Update()` functions of different scripts are called in the order the scripts are loaded (which is arbitrary). However, it is possible to modify this order using the Script Execution Order settings." The Unity3D manual does not describe in what order the `Update()` functions are executed in a given group of scripts. This is an important issue as each component may change the value of other components and affect the overall reactivity of the environment.

We performed experiments to assess the execution order of the updates in Unity3D. Each experiment involved a system with several objects, each of which inserted a unique number into a list each time it updated. The application maintains in memory only one copy of the list accessed by all objects from that class. The updated test involves programming the update function of every object to insert a number (a unique index for every object in the scene) into the global list. Since the order in which objects and components are added seems to be the order used to arrange the updates, we searched for what type of modification to the game tree causes a modification in the update order. From one test scenario to another, the only thing that changed was the order in which the `GameObjects` and their components were inserted or modified in the tree.

We performed the following sequence of tests:

1. Create the `GameObjects` level by level from the game tree, inserting them from the root toward the leaves.

2. Alter the creation order from the first test setup by exchanging the positions of `GameObjects` from different levels of the tree.

3. Alter the creation order from the first test setup by exchanging the positions of `GameObjects` within the same level of the tree.

4. Rearrange the creation order for the whole tree used in the first test setup but keep components assigned to the same `GameObjects`.

5. Detach some of the components from their original `GameObjects` in the first test setup and reattach them to other `GameObjects` in different levels of the tree.

In all of the tests, Unity3D consistently updated in the same fashion. This order is independent of the hierarchical position of the game object as the first test appeared to demonstrate, updating in an order starting from the leaves of the tree and finishing with the root. The updates are done by going from the newest objects inserted or modified to the oldest. Dragging objects around within the hierarchy has no effect on this order. The only test that produced a different execution order is the fifth test, where we remove a component from the root of the tree and insert it beneath one of the other components (one of the branch nodes). This change may occur as a result of a change to the project's generated metadata. Because of this change, both the root game object (where the component is taken from) and the branch game object (where the component is inserted) move to the beginning of the update order. Moreover, it is important to note that such modifications are done when the code is not running. Any modification during runtime (rearranging objects, attaching, reattaching, or detaching components) has no effect on the metadata and thus has no effect on the Unity3D execution order.

4.3 Transitional Turbulence in Unity3D Applications

In this research, we start with an unmodified Unity3D-built graphical application (meaning any application built using Unity3D) as the basic application to be augmented. We argue that this application does not achieve an accurate result due to transitional turbulence. As we consider the term in this dissertation, transitional turbulence is a period of chaotic or unreliable variation in the state of a software system that leads to instability in its execution. We argue that the current Unity3D application implements timely updates on the application state using the game loop. During each iteration of the game loop, every component is implicitly invoked and called to execute its own `Update()` event. The order in which these components are invoked is outside the developer's control and produces a temporarily undesirable state due to the fact that a component uses outdated information from other components, which is especially problematic when one interaction in a single component causes a chain of reactions affecting several other components. Such situations happen often during execution, since these components are interconnected and the execution of each component may affect the characteristics and executions of other components.

The current model of execution for Unity3D is prone to result in inaccuracies during the execution of applications due to transitional turbulence created by a lack of control over the order of execution of such components' update-handling functions. Scripts and the components instantiated based on them in Unity3D are not executed as they would in the traditional concept of a program. Instead, Unity3D implements an implicit invocation-like scheme. During the game loop iteration, it passes control to each component by notifying it to invoke specific functions that are declared within each component. Unity3D does not define an explicit order in which these components are called. The implicit order cannot be changed during runtime.

This research, initially presented in Marum et al. (2019) and Marum et al. (2020c), aims to answer the primary Research Question, as defined in the Section 4.1. To answer this question satisfactorily, we pursue several secondary research questions:

A. Is the corresponding dependency graph built by repurposing the original hierarchy using a dependency analysis a valid representation of the depends-on relationship between components in the scene hierarchy?

B. Is an evaluation order inferred from the dependency graph above a valid constrained execution order in which no dependent component executes before any component it depends on?

C. Does the timely reanalysis of the original scene and the rebuilding of the dependency graph assure a prompt reaction to changes in the overall state to the components and changes to the scene hierarchy as well resulting in an application that can react quickly to unpredictable changes in the application's state and structure?

D. Does a non-locking augmentation of the Unity3D event system, which keeps the remaining parts of the application unchanged while maintaining the benefits, results in building an augmented Unity3D application that is functionally equivalent to the same application built using the unmodified Unity3D?

Based on these questions, we built and tested a proof-of-concept augmentation to determine if it increases the accuracy of the state rendered on the screen and the Unity3D application's overall performance without increasing the time to an unacceptable level. We also plan to integrate our approach with third-party libraries and the Unity3D/.NET libraries, so it must maintain their benefits without interfering with their functioning. Our approach should outperform the Unity3D built-in event system and other reactive libraries used in game engines, demonstrating superior performance and accuracy in both static and dynamic environments.

4.4 Design and Implementation

This section describes the design and implementation of our reactive dependency graph-based component augmentation for game engines. It operates by augmenting the game engine's implicit invocation model by capturing the associations between components and reordering their execution to match their evaluation order. We based this augmentation on a proof-of-concept design of a dependency graph reactive component described in our previous work (Marum et al., 2019, 2020c).

We developed a reactive augmentation to mitigate transitional turbulence problems. Our augmentation is built as a component to be added to any Unity3D application. We call it the `DependencyManager`. To ensure that the dependency graph will analyze the entire tree, this component must be attached to the root of the game scene tree. This is the component that builds and manages the dependency graph, determines the evaluation order, and handles the event's execution order mentioned above. Since the root of the tree has access to all its children, this placement allows the `DependencyManager` to detect changes within all the "reactive" components that implement `IUpdatable`.

We seek to limit our manipulations to only a subgroup of the components and their relationships. We focus on those that can reduce most of the transitional turbulence. We do not manipulate those that cannot be accessed directly or that represent an expensive computation. Generally speaking, we include the component relationships arising from the application's custom code and exclude those from .NET, Unity3D, or other supporting frameworks. We designate the components that the developer wants to be considered in the dependency analysis by requiring that they implement the interface `IUpdatable`. (An interface is a special class that can contain only method signatures. The implementation of the method's body is the responsibility of each class that implements the interface.) Our approach iterates through the list of `GameObjects` to gather the components that meet the criteria and the other components they modify to determine the dependency relationships and construct the graph. One of our goals is to operate with already established VE systems and technologies such as Unity3D.

Our augmentation works through the following steps:

1. When the application starts, traverse the object hierarchy to build the dependency graph. Our approach iterates through the game scene hierarchy. This is Unity3D's built-in data structure that contains the `GameObjects`. This structure is used to divide the objects into subgroups, parents and children. This hierarchy organizes the `GameObjects` and makes it easier to move around this structure. The loop extracts the dependency relationships between the components and adds each component as a node of the graph and each dependency relationship as a directed edge. However, it omits any edge that would create a cycle in the graph.

2. After the entire dependency graph has been built, topologically sort the graph to determine an evaluation order that satisfies the dependencies. This evaluation order identifies a valid execution order for the updates.

3. On every update cycle, traverse the original game hierarchy to determine whether there are any changes in the state of any component or in the dependencies between components

and then update the dependency graph accordingly and determine a new evaluation order.

4. Wrap the chain of events defined by the evaluation order from the dependency graph into a single large-grained event to be executed as an `Update` method within the Unity3D game loop.

5. Ensure that building the dependency graph initially and rebuilding it on each update cycle are lightweight. That is, they should execute efficiently and require minimal modification of existing applications.

4.4.1 Building the Dependency Graph

In augmentation step 1, the algorithm starts by analyzing the dependency relationships between the components in the original game using the established criteria for dependency. We create the dependency graph by accessing each component, identifying its type information, gathering all its fields dynamically, and examining each field to determine which other components it refers to. The resulting dependency graph is a digraph formed by placing each one of the components gathered in the step above as a node and adding an edge from one node to another if there is a depends-on relationship recognized between the corresponding components of the edge. We use the evaluation order determined from the dependency graph to rearrange the execution order of those components throughout the updating game loop. Our approach uses a non-locking approach that works as part of Unity3D's game loop by coalescing the execution of all the dependency-related components in the game scene in an order that satisfies the dependency graph's constraints into a single large-grained Unity3D `Update()` method.

We define a dependency relationship between two components as a relationship where a component `A` contains the value of one of its properties or fields fully or partially defined by the value of another component `B` or one of `B`'s property or fields. A *property* is a C# capability that defines built-in get and set functions that allow the developer to encapsulate a specific field. A *field* is an object-oriented term related to an attribute encapsulated inside of an object. The algorithm used for building the graph is shown in Algorithm 4.

Algorithm 4 Evaluating the scene and building the dependency graph (Marum et al., 2020c, 2019, 2016).

```
Q = empty queue;
Tree = a subset of all the game objects;
Root = root of the game tree;
Enqueue the root in Q;
while Q is not empty do
    comp = Dequeue the next object in Q;
    Enqueue in Q each child object of comp;
    while for each Component C attached to game object comp do
        if C is not in Unity3D or .Net type then
            Insert C as a Node in the Graph;
            while for each Field or Property P in the Component C do
                if value of P is a component that implements IUpdatable and P is not null
                then
                    Insert P as a Node in the Graph;
                    if Edge between C and P does not exist and do not cause a cycle then
                        Create a Edge in the Graph between source C and destination P;

                    end
                end
            end
        end
    end
    Breadth-First Search to produce a valid update queue
end
```

A dependency can also exist if the update function of component `B` alters the value of component `A` itself, one of the `A`'s properties or one of the `A`'s fields. In these cases, the system records an edge going from `A` to `B`. We have established, as a condition for the system to work, that these connecting fields and properties must be explicitly defined or referenced. This condition can be relaxed in the future by adding code to verify which components and values are being modified by an event.

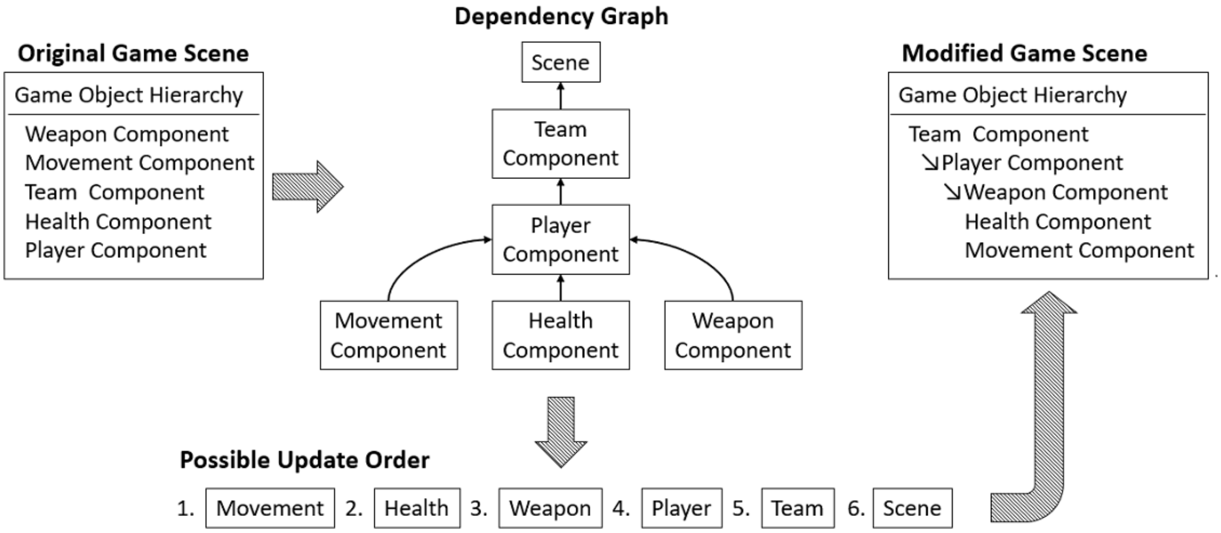


Figure 4.5. Alteration of the scene graph using the dependencies and establishment of a modified execution order (Marum et al., 2019, 2020c).

4.4.2 Determining Evaluation Order

Augmentation step 2 establishes an execution order for the `Update` methods. It determines the order from the dependency graph using a *topological sort* based on a Breadth-First Search (BFS). A *topological sort* of a directed acyclic graph (DAG) is a linear ordering of nodes such that for every directed edge from a node `A` to a node `B`, `A` comes before `B` in the ordering. In our approach, this ordering is defined by the dependency relationship between the components.

Figure 4.5 shows the process of building the dependency graph and the update order from the game tree.

In the resulting DAG, each node contains the component object copied from the scene graph, its parent `GameObject` information, and the component's type. This is important for the identification and equality comparison since each pair of `GameObject`-component is unique. Each connection is encoded as a directed edge. The source node of each edge is the component that contains the value used by the other component, and the destination is the component that depends upon the other—the component that uses the value from the source node. As it is a condition for building correctly a DAG, the algorithm makes sure that it does not create a circular dependency. When a cycle is detected, that dependency is omitted from the dependency graph, but the execution will still happen as an event in the Unity3D game loop.

4.4.3 Updating the Dependency Graph

Augmentation step 3 concerns the reanalysis of the dependency graph during the update cycle. Once for every update cycle the algorithm reanalyzes the graph using a similar BFS to step 1. During this cycle, the framework determines whether any component in the scene graph has been modified, deleted, or inserted relative to the current state of the dependency graph. If it detects that the component architecture has changed, the solution then reconstructs the dependency graph to reflect the new architecture. Our approach considers three distinct cases:

New component added to the scene: The algorithm adds the new component to the graph and determines which components that it depends upon. The algorithm then recomputes the dependencies for all components that became dependent upon the new component.

A component modified in the scene: If some of the component's properties are changed, the algorithm must update the dependency graph around that component appropriately. The modified component may now be dependent upon different components and different components may now be dependent upon the modified component.

A component deleted from the scene: All edges coming from or going to the deleted component must be deleted from the dependency graph. The dependencies for all components that depended upon the deleted component must be recomputed.

4.4.4 Calling the Update Function

Augmentation step 4 defines the way our framework calls each `Update()` function.This process is described in Algorithm 5.

Algorithm 5 Reanalyze the scene graph to perform any needed updates to the dependency graph—part 1 (Marum et al., 2016, 2019, 2020c).

```
Q = empty queue;
Tree = Subset of the game objects;
Root = root of the game tree;
Enqueue Root on Q;
while Q is not empty do
    Comp = Dequeue the first object in the queue;
    C1 = same instance of Comp previously stored in the Graph, null if none is found;
    if C1 is null then
        Insert Comp as a Node in the Dependency Graph;
        while for each unchecked Field or Property P in Comp do
            if value of P is a component that implements IUpdatable and P is not null then
                Insert P as a Node in the Dependency Graph;
                if Edge between C1 and P do not cause a cycle then
                    Create an Edge in the Dependency Graph between Node Comp and Node
                    P;

                end
            end
        end
    end
end
```

Our framework requires that a script in the scene must implement the interface `IUpdatable` for it to be considered during our augmentation. This interface specifies a single function `ReactiveUpdate`. This function is called by our augmentation instead of the default `Update` function from the Unity3D event system. All code in the script that will be handled reactively must be executed in this function.

This is done so that any behavior that must be handled without reactivity can update in the default way without sacrificing performance or dealing with the internal mechanisms of the Unity3D framework. From the same standpoint, Unity3D internal classes, the .NET prototype classes, and other scripts that will not be reactive are ignored by our framework. They are not triggered by changes and do not trigger changes in other scripts. Thus, the mechanisms implementing our approach should be implemented carefully so that the solution can augment specific areas of the application without interfering in the remaining parts.

When a change occurs in any component, in the next game loop, all update notification methods through the entire dependency chain are wrapped into a single large-grained event where all these methods will be executed in the order inferred from the dependencies. The reactive components shall be updated as a single block during the subsequent update cycles. We implement it with wrapper classes for the components and a library implementing the algorithms for constructing/reconstructing the dependency graph and using it to coalesce chains into "large-grained" events. This process is described in Algorithm 6.

Algorithm 6 Reanalyze the scene graph to perform any needed updates to the dependency graph—part 2 (Marum et al., 2016, 2019, 2020c).

```
while continues... do
    else
        Update the value of C1 in the Graph;
        while for each unchecked Field or Property P in C1 do
            if value of P is a component that implements IUpdatable and P is not null then
                P1 = object that is equal to P in the Dependency Graph, null if none is found;
                if P1 is null then
                    Insert P as a Node in the Dependency Graph;
                    if Edge between C1 and P does not exist or does not cause a cycle then
                        Create an Edge in the Dependency Graph between Node C1 and Node
                        P;
                    end
                end
                else
                    if value of P is null or different from P1 then
                        Update the value of P in the Graph;
                    end
                end
            end
        end
    end
    Breadth-First Search to produce a valid Update Queue; while for each Component c in
    the Update Queue do
        execute Update Function;
    end
end
```

Another important characteristic is that the current state of the application is self-contained, which means there are neither dependencies across states nor dependencies across update cycles. That is a crucial characteristic of our augmentation since the accuracy and predictability of each update cycle is the core issue of this work.

The implementation of our augmentation in the game engine, specifically in Unity3D, results in a reduction of transitional turbulence without degrading the performance of the application and the scheduling the execution of each component without violating the dependency constraints.

4.4.5 Ensuring Lightweight Mechanisms

Augmentation step 5 concerns how each `Update` function call is wrapped up as a single function call. In our augmentation, when a change occurs in any component, in the next game loop, all the update notification methods through the entire dependency chain are wrapped into a single large-grained event where all these methods will be executed in the order inferred from the dependencies. The reactive components shall be updated as a single block during the subsequent update cycles. Another important characteristic is that the current state of the system is self-contained, which means there are neither dependencies across states nor dependencies across update cycles. That is a crucial characteristic of the system since the accuracy and predictability of each update cycle is the core issue of this work. The implementation of our augmentation in the game engine, specifically Unity3D, results in a reduction of transitional turbulence without degrading the performance of the system and in an execution schedule for each component that does not violate the dependency constraints.

4.4.6 Comparing for Equality

One of the biggest issues encountered in the development phase was the Unity3D system's inability to determine the equality of references to the same object in different data structures. In our approach, the comparison of multiple instances of the same object is a key feature needed to trace changes in objects and spread them throughout the dependency graph. This is also important when checking if objects were added or deleted and if dependencies were added, changed, or deleted. When comparing using `object.equals()`, there were no positive answers, even though the comparisons were made between exactly the same objects. The same was observed when comparing tags and references (using `ReferenceEquals()`). The use of `Find()`, `FindByTag()`, and `FindByName()` requires removing many false positives and leads back to the problem of finding a positive using one of the techniques mentioned above. As such, we defined an indirect definition of equality comparison between objects that focuses on the characteristics of the objects. If two components have the same type and name and they belong to game objects that are equal (which means they also have the same properties), then they must be the same. This approach relies on the fact that no two objects of the same component type and with the same properties (name, tag, and position) can be attached to the same game object. This set of information represents, for our purposes, a unique identification for each component. These are required to allow us to define the equality of objects effectively.

4.5 Test Methodology

For purposes of comparison, we designed, built, and tested similar environments using three different system combinations:

- Unity3D using its own event system
- Unity3D with UniRx (Unity3D Reactive Extension) (reactivex.io, 2020)
- Unity3D with our reactive augmentation

The comparison against test scenarios built using only the Unity3D without any reactivity was made to measure our approach against Unity3D itself and its own game loop. We use it to measure the time increases for both startup time and in the game loops. We wanted to measure the time taken to build the dependency graph and to update the graph and determine the evaluation order. We use both a small-size game hierarchy with no graphical interface and a middle-sized game hierarchy with a graphical interface. We compare our design against both applications built using the unmodified Unity3D and Unity3D with a third-party reactive programming library. We sought to determine whether our augmentation improves performance, and if so, by how much.

By comparing against UniRx, we are comparing our approach against one of the current prominent implementations of reactive programming. We claim that the augmentation we perform in this work overcomes the issue that no other reactive programming implementation succeeded in doing: the loss of accuracy in the graphical application due to nondeterminism of the game loop's internal implicit invocation mechanism. Even though some of those reactive programming libraries overcome the issues related to streams of events between components and overcome the issues by ascertaining what happens when a specific action happens on the application, those libraries do not handle situations where the interaction that happens between the user and the application is unpredictable or where an event is caused by a casual interaction between objects. If the reactive system cannot programmatically predict the event and define the reaction to it, then the system handles it non-reactively, as it would if built only using the Unity3D built-in event system. Since we made such a claim in the problem definition, we must address the comparison between our approach and the mainstream reactive library made for game engines, specifically for Unity3D. The comparison addresses the two points we consistently make throughout this work, the capacity of our approach to adapt to changes in the scene, recomputing the dependencies, and inferring a new evaluation order; the capacity of our system of coalescing all events in such an order that each event that depends upon the execution of another component will always execute after it.

The computer that we used for testing was a Dell Latitude E5550 laptop with a Intel Core i5-5300 2.3 GHz processor with 8 Gb RAM and Intel HD Graphics 5500. We ran the Windows 10 64-bit operating system and used Visual Studio 2017 with C# 8.0 and Unity3D

2019 3.0.

We chose to record two metrics and two statistics based on these metrics to compare performance between the three applications. They are based on three measurements: latency, errors, and visible errors. When referring to an *error*, we specifically mean a situation where one of the game components is executed before one or more of its dependencies, thus causing inadvertent use of out-of-date or missing values. This incorrect order causes a failure of the component or a temporarily erroneous state of the component. *Latency* means the number of loops (or the time is taken) between the beginning of the test and when the expected system state has been reached. A *visible error* is the temporarily incorrect state of one or more of the components that are visible in the rendered imagery of the game. Since many game loops can execute per frame of rendered video, we considered an error to be visible if it persists for enough loops to outlast a single rendered frame. This does not mean that an observer will necessarily be able to see the error in question, but instead that if some visible element of the environment relied on this component, the resulting error could potentially be visible to the observer. As such, this can be thought of as a lower bound or minimal criterion for a visible error to occur in the environment. For the first scenario, no visual errors were recorded, since the expression tree simulation had no visual component to be verified.

Two test scenarios were designed and built:

Scenario #1 includes insertion, deletion, and modification of components. It uses the example of an expression tree calculator that was presented previously in Marum et al. (2019) and Marum et al. (2020c).

Scenario #2 demonstrates how the update order affects common interactions among multiple objects in a target shooting example.

For both scenarios, we also collected data concerned with the additional time and processing overhead at startup and during each game loop iteration. In particular, we recorded the extra costs incurred by our approach to construct and update the dependency graph. These time increases should be kept small in proportion to the accuracy gain and performance improvement. We measure the time spent for each system tested by using the `StopWatch` class from the .NET framework. This class emulates the behavior of a regular stopwatch, giving us the ability to start and stop it as needed. We start it at the beginning of the `start()` and `Update()` and stop it at their end. The property `Elapsed` from the class `StopWatch` gives the amount of time in milliseconds spent from start to stop. 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. Our tests do not reveal any significant performance degradation either in the update time or start-up time. That happens because only the objects that can trigger reactivity are considered and only when their values change. On average, the time consumed for the system on the updates remains relatively small in comparison with Unity3D alone. In the future, we plan to develop tests to measure more rigorously the above time increase utilizing the test methodology described in Jones et al. (2019).

Performance-wise, our framework showed that the time spent in the game loop is, on average, similar to the performance achieved by UniRx or Unity3D alone. Thus, the use of our framework did not indicate a significant increase in the time spent in each game loop iteration. Creating the dependency graph and performing the topological sort initially took, on average, 198 milliseconds. When the dependency graph needs to be reconstructed, our updating process inside the game loop took up to 100 milliseconds to redo the analysis and sort. These statistics are directly connected to secondary research question #4 and demonstrate that the time overhead produces by building and updating the dependency graph on both startups and in each game loop is reasonable enough to not influence negatively the development and usage of the system. That means that the algorithm described in the previous section builds a dependency graph and traverses it completely in an amount of time very close to the time performed by Unity3D alone between each frame.

Next, we show the implementation and results of the two scenarios implemented using the three combinations. These scenarios were executed through a set of automated tests using UniRx, Unity3D, and our reactive framework.

4.5.1 Scenario #1: Implementation and Results

Scenario #1 includes insertion, deletion, and modification of components by using the example of an expression tree calculator that was presented previously in Marum et al. (2019) and Marum et al. (2020c). It is generally quite difficult to determine whether a series of objects updates in an expected way in a virtual environment without degrading the system performance by storing state data in memory or writing to a file. Instead, our approach is to design a game environment where objects within the scene behave as mathematical elements (operands and operators) within an expression tree. This allowed us to build test cases where the value computed by the expression tree is known a priori. Any error in the execution order would then be detectable in the final state of the system simply by comparing the tree's computed values with the expected values. This serves as a very sensitive method for detecting errors in execution without introducing additional overhead costs. The test scenario included insertion, deletion, and modification of components.

Scenario #1 was built for the purpose of creating a nongraphical, pure mathematical version of the game hierarchy. Along with the tests, we hoped to be able to isolate the interaction of our game hierarchy while still working inside the Unity3D game loop. We used this test to measure the ability of our approach to reduce the transitional turbulence and to propagate the effects to all its directly or indirectly dependent components without the delays and non-determinism introduced by the normal event-handling system as if all were part of the processing of one large-grained event. Also, we hoped to ascertain that our approach will recognize whenever any of the components' overall state has changed or when the component architecture has changed (e.g., the addition, modification, or deletion of any component). It must detect that the component architecture has changed, then reconstruct the dependency graph to reflect the new architecture. Performing the test involved:

- Randomly generating binary trees that represent mathematical expressions. The tree is either a leaf or an internal node. If the tree is a leaf, then the algorithm randomly selects some integer value. If the tree is an internal node, then the algorithm randomly selects an arithmetic operator chosen from addition, subtraction, multiplication, division, and exponentiation.

- Randomly placing integer values in the leaf nodes and binary operators in the internal nodes of each tree. The "current" value of an internal node can be computed by performing its operation on the "current" values of its two children. We require that the root node be an operator to eliminate trivial cases.

- Computing the "current" value of the tree by computing the "current" value of the root. For the "current" value of the tree to be the correct value of the expression, the values of all nodes must be computed in the correct order. That is, the value of both subtrees of an internal node must be computed before the value of the internal node.

This algorithm generates mathematical expressions such as the following:

- $5 + (4 * 9) + 3 - 5^2$

- $32 * (7 + 9) - 12 - 16$

- $50 - (12 + 16)/8 - (12 - 9)$

The result of an expression tree `A` is a calculation between its child nodes `A1` and `A2`, which means that in order to obtain the result of `A` correctly, both `A1` and `A2` must be available and correct. `A1` similarly depends on its two children `A11` and `A12`. So the calculation will obtain the correct final result only if the updates that trigger the calculations respect the following order: `A11` → `A12` → `A1` → `A2` → `A`. Any execution order in which `A` executes

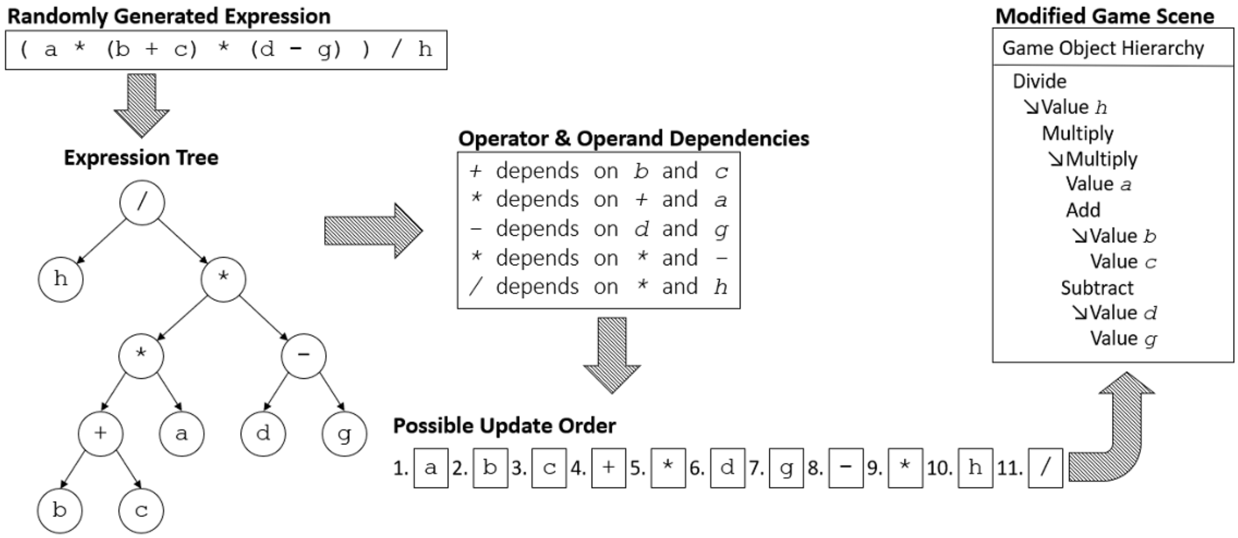


Figure 4.6. Mathematical expression tree generation used in scenario #1 for testing our framework (Marum et al., 2019).

before its children (A1 and A2) would produce a wrong result since it used an outdated or nonexistent value. The diagram in Figure 4.6 depicts the test process.

The tree is first embedded in a game hierarchy, then the "current" value of the tree is calculated, then this value is compared to the expected (i.e., correct) value of the expression. To determine the adaptability, the test is repeated with several variations of the original tree.

In our tests, after the end of the update cycle, the value of each internal node is compared with the expected value computed beforehand. At some random time during the tests, we introduce changes to the scene graph by inserting new nodes in random locations, deleting random nodes, or modifying the value of a node. In the next update cycle, the test compares the result of the expression with the new expected value. We also keep testing between modifications to ensure that the result remains stable.

To measure the accuracy of each testing platform, we determined whether it reached a accurate state at the end of each game loop iteration, despite having to handle unpredictable modifications inserted in the original tree across the runtime execution. In this test, which appears in Marum et al. (2019) and Marum et al. (2020c), our framework performed better tb

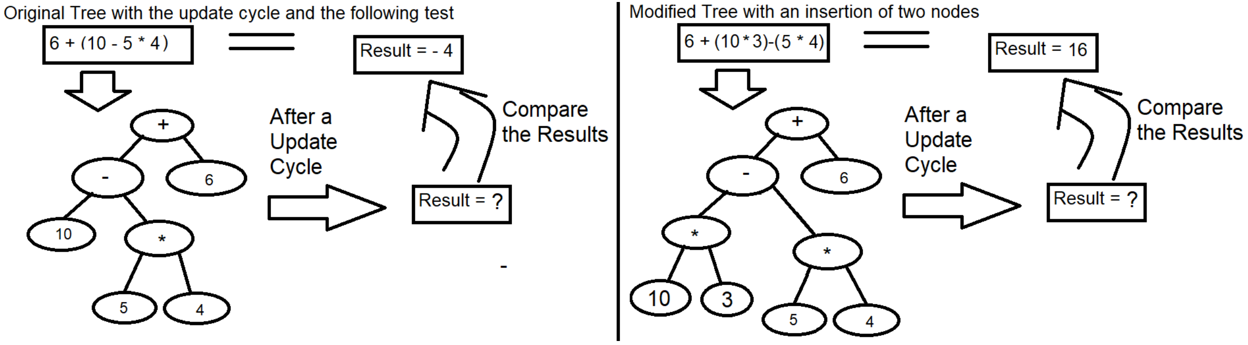


Figure 4.7. Mathematical expression tree generation with or without modifications on the original tree (Marum, 2017).

than UniRx and Unity3D with default functionality. Figure 4.7 shows the behavior of our test with and without modifications to the original tree.

We recorded the measurements and took the average based upon the total number of game loop iterations. The average results were recorded for each platform in each scenario:

- The latency (number of game loop iterations) needed to get the game into the expected state once set into action. This measurement is directly linked with our secondary research question C that claims that analyzing the game hierarchy in each game loop, rebuilding the dependencies, and updating an evaluation order dynamically during each game loop iteration ensures that the system can expeditiously adapt to changes in the components and in the hierarchy structure as well.

- The number of errors detected for a sequence of interactions was expressed as the average number of update method executions that contained at least one error calculated across all update cycles in each run, which is directly connected with research question A

- The average number of errors in a single test where errors were detected. This is how many components' update method executions were incorrectly ordered in a test where there were errors detected. We count the total number of errors, the total number of runs that reported an error, and finally compute the average. This measurement is

Platform	**Total Game Loops**	**Total Errors**	**Avg. Errors per Loop**	**Visible Errors**	**Latency in Loops**
Unity3D	100	95	5	0	5
UniRX	100	15	2	0	5
Our Platform	100	15	2	0	1

Table 4.1. Test results for scenario #1

Platform	**Total Game Loops**	**Total Errors**	**Avg. Errors per Loop**	**Visible Errors**	**Latency in Loops**
Unity3D	100	100	5	0	7
UniRX	100	80	5	0	20
Our Platform	100	20	3	0	1

Table 4.2. Test sesults for scenario #1 with modification during execution

linked to research question `B`.

- The average number of errors causing observable inaccuracies. This means the average number of cycles where one or more components were in a temporarily incorrect state. This statistic is connected with the primary research question and research question `D`.

Table 4.1 shows the result of the first test, with the expression tree running for 100 user cycles and no modifications inserted.

Table 4.2 shows the results of the second test, with the expression tree running for 100 user cycles (which means cycles initiated by direct user interaction) and a modification inserted in the expression tree (game objects modified, added, or deleted).

4.5.2 Scenario #2: Implementation and Results

Scenario #2 is designed to demonstrate how the update order affects common interactions among multiple objects in a target shooting example. Scenario #2 was used to see how our system acts in an environment that could be easily extended to have VR or AR capabilities. This test environment used both developer-made components and off-the-shelf Unity3D assets. This was a point of particular interest because practical VE development commonly utilizes off-the-shelf assets, libraries, or packages. We generate a scene where the first-person player can move, collide with objects and walls, and shoot targets. This scenario was adapted from one of the examples included in the Standard Assets package downloaded from the Unity3D asset store. We apply the same methodology from the first scenario mentioned above. We change components between game objects during the test execution. For example, a wall that was not a shooting target at a random moment may change by adding the shooting target component that causes it to react to shots, thus becoming a shooting target, and vice versa. Future research should be conducted to definitively expand this research by using the same component in a VR/AR system that uses devices or libraries from third parties.

Scenario #2 was developed to assess how our system behaves in a more realistic scenario. It is a graphical application with a camera and player movement. It thus uses many built-in Unity3D classes. We assess the capacity of our augmentation to handle the components related to physics including ballistics, collision, and *ray-casting*. *Ray-casting* is a technique to handle target identification and light effects by creating a target vector from the tip of the source towards any specific physical body that the "projectile" hits across a mathematically generated trajectory. We developed two special classes of components in the scene, the `Shootable` and `Collidable` objects. A player shooting a `Shootable` object or colliding with a `Collidable` object initiates a reaction on those objects. The desired behavior is that the reaction happens as soon as the collision or hit occurs, with any other secondary reactions also occurring. The whole chain of reactions starts as a single flow, as it should happen in the real world.

This test is concerned with our approach's ability to adapt any existing application by selecting a set of mechanisms appropriate for that application's design and implementation and by applying the dependency analysis so that it maintains a balance between the time loss and the accuracy/performance gain. To determine that, we assess the capacity of our approach to produce graphical results with fewer inaccurate states rendered on the screen.

We record the measurements and compute the average based upon the total number of game loop iterations. We record the average results for a given number of update cycles with interactions shown in Table 4.3 and Table 4.4 for each platform in each scenario.

Tables 4.3 and 4.4 show the results for the second test scenario. We run the shooting/collision simulation for 100 user cycles (i.e., cycles initiated by a direct user interaction). The first shows the results when no modifications are inserted and the second when modifications are inserted (i.e., game objects are modified, added, or deleted).

Platform	**Total Game Loops**	**Total Errors**	**Avg. Errors per Loop**	**Visible Errors**	**Latency in Loops**
Unity3D	100	91	10	40	15
UniRX	100	16	10	15	10
Our Platform	100	12	5	0	1

Table 4.3. Test results for scenario #2

Platform	**Total Game Loops**	**Total Errors**	**Avg. Errors per Loop**	**Visible Errors**	**Latency in Loops**
Unity3D	100	96	20	40	20
UniRX	100	75	30	45	30
Our Platform	100	15	15	0	1

Table 4.4. Test results for scenario #2 with modification during execution

4.6 Results and Analysis

As can be seen for the Scenario #1 in Table 4.1 and for the Scenario #2 in Table 4.3, especially the numbers concerning the UniRx implementation, we observed inaccuracies in only 15% of the tests using the scenario with no changes. However, Table 4.2 for Scenario #1 and Table 4.4 for Scenario #2 show the inaccuracy rate increased to 80% when we introduced changes randomly throughout the test. These episodes of inaccuracy continue to occur for several cycles after a modification of the game tree. During the unstable period, UniRx produced several errors, cataloged in one of two possible categories:

- The system entered into an error state with a null or a type-related exception. (The system expected an object of a certain type and found an object of another type or found a null pointer.)
- The system ignored the existence of the new or modified node.

In the tests where modifications were introduced at runtime, UniRx took up to 30 cycles to recover from the error and reach a stable state. Additionally, in the tests where there was an error, the game loop iteration containing an error had a high number of incorrectly ordered executions per update cycle. This happens because any interaction with one of the modified/inserted/deleted components was not correctly handled, resulting in an erroneous state. Although UniRx is designed to handle input streams in a reactive way, it does not react properly to changes in the objects themselves. This significantly limits the dynamics of virtual environments where components may be arbitrarily inserted or removed, such as multiuser experiences. Consequently, UniRx can only handle such situations if they are predictable and properly handled by the programmer.

In the measurements concerning the default Unity3D event system, Table 4.1 for Scenario #1 and Table 4.3 for Scenario #2 demonstrate that episodes of error happened in 95% of the cases in the scenario with no changes to the scene graph. Several update cycles were necessary before the scene graph was up-to-date. This can be explained by the fact that the Unity3D event system uses an arbitrary order for updates as defined in the Unity Manual (Unity Technologies, 2019). When a component executes, it uses the available values, without knowing if the dependencies were updated accordingly. That is why a single chain of reactions takes time to spread through the scene graph. The number of game loops needed to reach a desirable result in Unity3D is invariably connected to the complexity of the simulations and how many components are involved in that loop. This behavior can also be observed in the number of errors per game loop.

The default Unity3D event system behaves similarly when changes are introduced to the game hierarchy. It took several game loops to stabilize the components' states as shown in Table 4.2 and Table 4.4. The system also produces errors in 95% of the cases. For our framework, Table 4.1 and Table 4.3 show that episodes of error occur in only 15% of the cases. Table 4.2 and Table 4.4 shows that these results differed very little for situations where changes were introduced to the system. Compared to the other systems, our framework detected the changes and reached a stable state more quickly than the others.

One potential reason for the poor performance of UniRx is that it assumes that the structure of the scene graph will remain unchanged from one frame to the next. When the system makes an asynchronous operation, UniRx expects to find the same game structure that was there when the request was sent. When the structure changes, UniRx considers it to be a situation that it cannot handle and triggers a runtime exception. In the development of our framework, we intended to handle cases where the game tree may change during runtime supporting a more general class of asynchronous operations. Our framework solves this problem, handling environments with changing properties. It also circumvents the Unity3D game loop by coalescing all the dependent update events into a single large-grained event that executes all these updates in the evaluated order but does not satisfy all the issues pertinent to the development of a reactive system.

The tests described in this section demonstrate the effectiveness of our approach. Our framework mitigates the effects of the transitional turbulence caused by interactions among multiple components. We intended to augment the current Unity3D game loop and coalesce the component's update methods into a single event in the game loop. By doing that, our framework produced a smoother experience when compared with the existing alternatives. Because this approach was developed to solve this class of problem, it effectively propagates the chain of executions through the scene graph. It also proves to be able to adapt to a significant range of changes in the structure of the tree or in the objects themselves.

4.7 Conclusion

In this research, we have addressed the instability resulting from the transitional turbulence that occurs in virtual environments. We have demonstrated this by exploiting Unity3D's existing object hierarchy.

Initially, and whenever the hierarchy changes thereafter, our approach extracts the intercomponent dependencies and generates a new event-handling order that satisfies these constraints. Our approach groups a chain of reactions corresponding to some external action into a large-grained reactive event that can be performed in one update cycle, that is, within the execution of a single Unity3D event. This approach seems to have the benefits of locking, non-locking, and wait-free-based approaches without dealing with concurrency issues.

To evaluate the effectiveness of our approach, we used a test scenario built around an expression evaluator to demonstrate how the update order of the game objects affects the interactions among the game objects. We represented the expression as a game tree with each operator at an internal node with its operands as its sub-trees. The values are at the leaf nodes. For the correct value of the expression to be calculated, the operand subtrees must be evaluated before the corresponding operator. To determine the effects of reconfiguration of Unity3D's game tree, we ran (a) tests that kept the expression tree stable throughout the run and (b) tests that randomly introduced changes in the tree's structure during the test run. For each test run, we measured the startup time, latency, and total errors during the run and computed the average startup time, latency, errors per cycle, errors that resulted in a visibly inaccurate state, and number of miscalculations that occurred in a test that failed. Our experiments indicate that, on average, our approach exhibited a shorter latency and fewer errors, at the cost of a modest increase in startup time compared to the other two alternatives. Our approach improves responsiveness and performance to changes and results in more accurate mathematical and visual behavior.

Our primary research question, presented at the Section 4.1 establishes that we intend to reduce the transitional turbulence, mitigate the execution inconsistencies among component's `Update()` functions, especially between those components that have a *depends-on* relationship and increased accuracy while maintaining performance on graphical applications built on game engines. To achieve that, we designed and implemented a dependency graph-based execution reordering and self-adjusting state recomputation to augment the Unity3D built-in game loop. The overall performance of our approach in the tests described in Section 4.5.1 and 4.5.2 is superior against the unmodified Unity3D and UniRx in all metrics measured. The performance decrease is not perceptible to the external observer. These information means that our approach is successful in mitigating most of the transitional turbulence while maintaining performance and semantic equivalence with the unmodified Unity3D application.

Our secondary research question `A` is related to whether the dependencies obtained being a valid representation of the dependency relationship between components. In Section 4.4, we describe the mechanism that analyzes each component and copy it to the dependency graph. In this process, we do not modify the original hierarchy, instead we mirror the components from the original hierarchy highlighting the dependency relationships between the components to build a dependency graph. The execution order inferred from the dependency graph in our approach relies upon the mathematical concept of evaluation order that consists in an order in which each node of the graph is visited only once and all nodes are visited after the nodes in which they depend on. The final evaluation order must not violate any of the dependencies expressed in the graph. The mathematical expression evaluator test that was described in Section 4.5.1 was developed with the goal of creating a test that depicts these *depends-on* relationships between the components as mathematical relationships between operators and operands. In the majority of executions, our approach evaluated the expression correctly, inferring correctly both the dependency relationship and the evaluation order. We demonstrate that the dependencies extracted from the original hierarchy are a valid representation of the depends-on relationship.

Since the secondary research question `B` is related to the validation of the evaluation order inferred from the dependency graph that we built, the same tests and results can also be used to advocate that our evaluation order extracted from the dependency graph is demonstrated to be an order in which no dependent component executes before any component it depends on. In the experiments mentioned above, the evaluation order inferred from the dependency graph is proved to be in the majority of times an order that satisfies the set of constraints or dependencies used to build the graph.

The secondary research question C is related to the prompt reconfiguration of the dependency graph and the adjustment of the evaluation order in the face of a state change of any component or a structural change. The design of our approach rely on the fact that by copying the current version of the hierarchy into the dependency graph, we are keeping a history of the previous iteration safely stored. This means that when the next update happens, we can compare the current version of the hierarchy against the previous version to detect any difference between the two states. That allows the system to immediately react to any change detected by rearranging the dependency graph to match the current state. The tests described in both Sections 4.5.1 and 4.5.2 related that involved random modifications in the tree were created with the purpose of measuring our system's capacity of self-adjustment and prompt reconfiguration when a change is detected. Our results have shown that our augmentation can assure a prompt reaction to changes in the overall state to the components resulting in an application that can react quickly to unpredictable changes in the application's state and structure.

The secondary research question D is related to keeping the remaining parts of the application unchanged while maintaining the benefits. This augmentation is designed with the intention that it can be used in conjunction with other third-party libraries and assets. Our tests illustrate that this is possible in principle using the Unity Standard Assets. We expect similar compatibility to exist with a wide variety of applications, including libraries for adding additional simulation functions. So, our augmentation results in building an augmented Unity3D application that is functionally equivalent to the same application built using the unmodified Unity3D.

Our results also show that our augmentation preserves the proper functioning of the original update mechanism of those systems. We seek to decrease the extra costs incurred by the solution during the startup process making it small in proportion to the potential accuracy and performance gains, so we balance the extra costs incurred in check and modify the dependency graph low to avoid worsening the overall effect on the performance based upon the fact that there is a low likelihood of a dramatic change in the dependency graph from one cycle to another.

CHAPTER 5

DYNAMICALLY COALESCING REACTIVE CHAINS DESIGN PATTERN

5.1 Introduction

Chapters 3 and 4 have addressed two different, but related problems and devised two similar solutions using different technologies. Both solutions work by augmenting the normal event-processing mechanisms used in the applications. But what is the essence of our approach? What kind of problems does it solve? To what kinds of technologies can it be applied?

In this chapter, we examine our research from Chapters 3 and 4 and reveal a general, technology-independent approach that we expect to be useful to developers of similar applications. The primary contribution of this chapter is the codification of this general approach as a design pattern we name Dynamically Coalescing Reactive Chains (DCRC). We also intend to accomplish the careful documentation of the systematic process we used to develop this pattern. This provides an example that other pattern writers may find useful.

As we analyze the previous solutions and extract the abstract features into a more formal description, we intend to answer the following research question:

> *How can we codify the transitional turbulence mitigation approach taken in Chapters 3 and 4 as a general, technology-independent design pattern?*

We systematically step through the process for developing the pattern. We explore the scope of the pattern in Section 5.3, review the two previous solutions in more detail in Section 5.4, develop and refine the pattern element by element in Section 5.5, and present the full pattern in Appendix A. In Section 5.6, we reflect on the pattern-writing process and discuss how the pattern may evolve in the future. In Section 5.7, we demonstrate the efficacy of the pattern by showing how it can be applied to a real-world example based on the .NET-based GUI application described in Chapter 3. Section 5.8 answers the research question posed above and summarizes this chapter's contributions. Before we write the DCRC pattern let's look more closely at what we mean by a software pattern in Section 5.2

5.2 Software Patterns

A *software design pattern* is defined in the classic "Gang of Four" patterns book as a "general and reusable solution to a set of problems with common characteristics within a given context" (Gamma et al., 1995). A pattern is not invented; it is distilled from practical experience (Buschmann et al., 1996). Patterns codify "best practices" for software architecture and design (Meszaros and Doble, 1998). Patterns are written and published to document these best practices and enable others to apply them in their own work.

The "Siemens" book (Buschmann et al., 1996) on patterns groups software patterns into three categories: architectural patterns, design patterns, and idioms.

An *architectural pattern*—sometimes called an *architectural style* (Garlan and Shaw, 1993)—is a high-level, language-independent abstraction that guides the design of the system-wide structure. Examples include the PIPES AND FILTERS pattern (Buschmann et al., 1996; Garlan and Shaw, 1993; Shaw, 1996) that is a fundamental approach to structuring programs on Unix-like systems and the MODEL-VIEW-CONTROLLER pattern (Buschmann et al., 1996; Goldberg and Robson, 1983) that is a "best practice" for structuring Web and other user interface applications.

A *design pattern* is a mid-level, (mostly) language-independent abstraction that guides the design of a subsystem. The classic patterns in the "Gang of Four" patterns book (Gamma et al., 1995) all fall into this category. Although not language-specific, these classic patterns focus primarily on design issues that arise in the use of statically typed, single-dispatch, object-oriented languages without first-class functions. Some of the patterns are less meaningful for other languages.

An *idiom* is a low-level, language-specific abstraction that guides some aspects of both design and implementation. For example, consider the classic SINGLETON design pattern (Gamma et al., 1995), which guarantees that exactly one object from a class exists. Although this is a general abstraction, its implementation varies significantly from one language to another, leading to a related idiom for each language (Buschmann et al., 1996).

Among several existing approaches for describing patterns, we choose the approach described by Wellhausen and Fiesser (2011) because of its simplicity. They describe "a creative, iterative process" that is suitable for our needs to refine and evolve our pattern description. In their approach to writing patterns, the following elements must be present in the given order:

Pattern Name gives an evocative name for the pattern.

Context describes the circumstances in which the problem occurs.

Problem describes the specific problem to be solved.

Forces describe why the problem is difficult to solve, giving different considerations that must be balanced to solve the problem.

Solution describes how the solution to the problem works at an appropriate level of detail.

Consequences describe what happens when a software designer applies the pattern. It gives both the possible *benefits* and possible *liabilities* of using the pattern.

All of the above elements except Consequences are also included in the MANDATORY ELEMENTS PRESENT pattern from Meszaros and Doble's *A Pattern Language for Pattern Writing* (Meszaros and Doble, 1998).

In this chapter, we follow the steps below adapted from the pattern "writer's path" described by Wellhausen and Fiesser (2011). In the various steps, we apply patterns from the pattern languages of Meszaros and Doble (1998) and Harrison (1999, 2006) to refine our new pattern.

1. *Explore the new pattern's rationale and scope.* Write short notes to answer questions such as: Why should we write a new pattern at all? What is included in and excluded from its scope? What concrete examples do we have to examine? State a crisp definition for the scope.

2. *Examine existing solutions.* Read the notes from the previous step. Discuss the solutions with others. Collect a list of possible names. Briefly summarize the general solution, focusing on its essence. List any clever ideas identified in the solutions for later consideration, even if not essential to the solution.

3. *Describe the problem that leads to the solution.* Strive to get this in one sentence. Be careful to separate the problem from its solution. Make sure the solution actually solves the problem.

4. *Consider the consequences of the solution, both its benefits and its liabilities.* Think about what happens if the solution is applied and what happens if it is not.

5. *Identify the forces that make the problem difficult to solve.* The forces usually conflict, pushing in different directions. Consider what differentiates the chosen solution from other possible solutions to the problem to help identify the different forces at work. Give each force a meaningful name.

6. *Match each force with the corresponding consequence.* A force makes the problem difficult to solve. How the solution resolves this difficulty leads to the corresponding consequence. Each force must be resolved and may have both benefits and liabilities.

7. *Describe the context in which the problem exists.* The problem might not exist outside this context. The context cannot be changed by the solution.

8. *Choose a pattern name.* A good name should evoke the core idea of the solution. It should be easy to remember.

9. *Reexamine and rewrite the six pattern elements.* Use the Context to describe the background and assumptions. Focus on devising a short, crisp Problem description. Put what makes the Problem difficult in the Forces. Make sure the Solution solves the Problem and balances the Forces. Link the Forces with the Consequences.

10. *Put the pattern elements in the standard order.* Restate the Solution and Consequences appropriately to match the other elements. Write the pattern so that it flows smoothly from Context to Consequences.

11. *Get feedback from experts in the technical area and pattern writing.* After a period of time, reexamine and rewrite the pattern description. Be patient.

5.3 Exploring Rationale and Scope

In writing the Dynamically Coalescing Reactive Chains (DCRC) design pattern, we first explore the new pattern's rationale and scope, that is, its context. By analyzing the case studies in Chapters 3 and 4, we observe that both exhibit the characteristics of an implicit invocation architecture.

The Implicit Invocation (II) pattern is an architectural pattern (Garlan and Shaw, 1993; Qian et al., 2010; Shaw, 1996) that can describe a wide range of systems, including systems like those in Chapters 3 and 4. Using typical software architecture terminology, the *system model* can be defined as a graph with software components at the nodes and connectors along the edges (Shaw, 1996; Shaw et al., 1995). The *components* are high-level computational and data storage entities and the *connectors* are the interactions among the components. In addition, there is a *control structure* that governs how the system executes.

Figure 5.1 depicts the implicit invocation pattern. According to Shaw (1996), the Implicit Invocation system consists of a "loosely coupled collection" of "independent

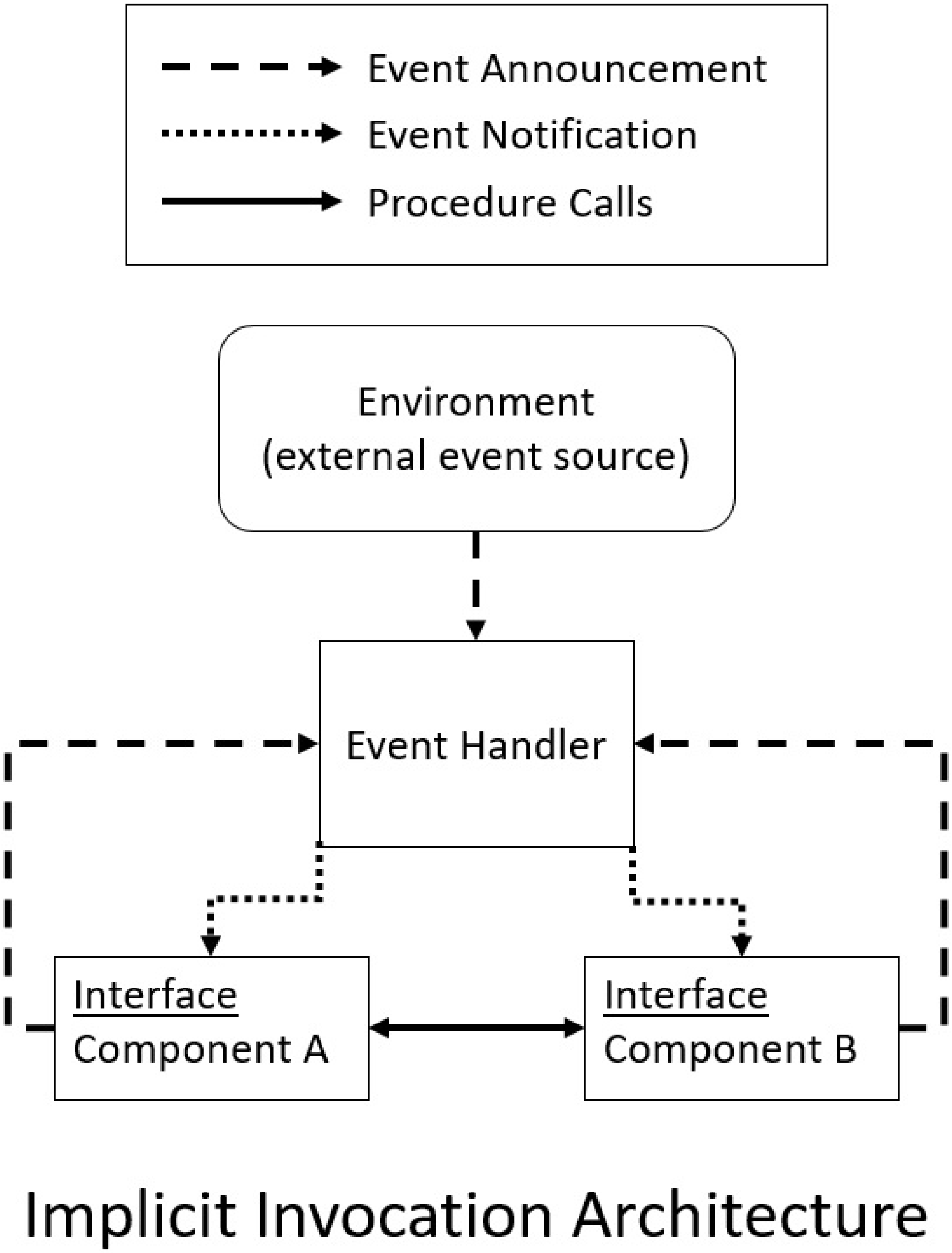


Figure 5.1. Unmodified representation of the implicit invocation architecture model.

reactive processes"—or perhaps a better term for our purposes is "modules" (Garlan and Shaw, 1993). The components are these modules, which can "signal significant events without knowing the recipients of the signals". The connectors are the implicit (or automatic) invocations of procedures in the modules' interfaces "that have registered interest in events". The control structure is "decentralized" so that the individual components are unaware of the recipients of their signals.

An implementation of the Implicit Invocation pattern usually requires some kind of "event handler that registers components' interest in receiving events and notifies them that events have been raised" (Shaw, 1996). When a component registers interest in an event, it associates a procedure with that event.

When a component registers interest in an event, it associates a procedure with that event. To notify the component that the event has been raised, the event handler implicitly invokes the associated procedure (Garlan and Shaw, 1993). We assume the event handler is nondeterministic but fair. That is, once an event is signalled by a component, all the listeners' associated procedures will eventually be invoked, but there is no guarantee in what order the events will be handled.

An implicit invocation system has advantages and disadvantages (Garlan and Shaw, 1993; Shaw, 1996). Among the advantages are support for software reuse and dynamic reconfiguration. Among the disadvantages are the nondeterminism of processing order and the difficulty in reasoning about correctness.

There are, of course, many different variations of the implicit invocation concept, such as the Publisher-Subscriber (Buschmann et al., 1996; Eugster et al., 2003) and Observer (Gamma et al., 1995) design patterns. In this dissertation, we use the term Implicit Invocation, which seems to describe the general operation of the event-driven programming mechanisms we used in the previous solutions and many other similar systems.

Consider the case studies in Chapters 3 and 4:

- For Web and desktop graphical user interface applications like those in Chapter 3, we

> Context
>
> We have an application constructed according to the Implicit Invocation architectural pattern (Shaw, 1996), assuming nondeterministic but fair handling of events.

Figure 5.2. Context element draft.

layer our solution to the transitional turbulence problem on top of the .NET framework's built-in event system. That is, we "augment" the built-in event-handling system with new software mechanisms.

- For virtual and augmented reality applications like those in Chapter 4, we layer our solution on top of the Unity3D game engine's (Unity Technologies, 2019) built-in event-handling system.

In both case case studies, the built-in event-handling systems follow the implicit invocation architectural pattern as described above. In both case studies, we also observe transitional turbulence as described in Chapters 3 and 4

Thus, to define the Context for our new pattern, we focus on applications built around the implicit invocation pattern. The first draft of our new pattern's Context element is shown in Figure 5.2. As we continue to write the pattern, we identify other assumptions about the Context in which the pattern is relevant.

In writing our new pattern, we also constrain it in the following ways:

- As suggested by the Clear Target Audience (Meszaros and Doble, 1998) and Consistent-"Who" (Harrison, 2006) patterns, we focus our attention on developers who are working within a software architecture described by the Implicit Invocation pattern. We do not assume any particular programming language or platform in the general description.

- As suggested by the Terminology Tailored to Audience and Understood Notations patterns (Meszaros and Doble, 1998), we use terminology, concepts, and notations that should be familiar to the identified target audience. We also relate the terminology we use in the pattern description to that we use in the Implicit Invocation architectural pattern description.

- As suggested by the Dead Weasels pattern (Harrison, 2006), we seek to identify any “weasel words”—words that “imply meaning but have no real substance” or that are too ambiguous or imprecise to guide the reader in applying the pattern effectively. We try to replace a “weasel word with a phrase or paragraph that is more specific”.

5.4 Examining Existing Solutions

In step 2 on the writer’s path, we examine existing solutions with the primary objective of unifying them. We seek to describe the general pattern that emerges from two related case studies:

- Dynamic Web and desktop graphical user interfaces (GUIs) designed and implemented with C# on the .NET platform (from Chapter 3).
- Dynamic virtual reality (VR) and augmented reality (AR) applications designed and implemented in the Unity3D game engine using C# (from Chapter 4).

There is, of course, a wealth of other research on reactive programming languages and systems (Bainomugisha et al., 2013; Cooper and Krishnamurthi, 2006; Czaplicki and Chong, 2013; Drechsler et al., 2014; Elliott, 2009; Meyerovich et al., 2009; Reynders et al., 2017) we could profitably examine. However, in this chapter we focus our attention on solutions that follow the Implicit Invocation architectural pattern and that work by augmenting the normal event-processing mechanisms of the technologies they are built upon. The solutions from Chapters 3 and 4 satisfy these criteria. The new design pattern seeks to document how a developer should analyze an existing application, design appropriate new mechanisms to reduce transitional turbulence, and implement the mechanisms in a modified application.

5.4.1 Dynamic GUI Application

Chapter 3 explores the issue of transitional turbulence occurring in Web and desktop graphical user interfaces (GUIs) implemented with C# on the .NET platform.

In this environment, a GUI consists of a loosely coupled collection of controls (i.e., the components in the Implicit Invocation architectural pattern). Each control responds to events in which it is "interested". A response to an event may result in the control changing its state and triggering new events that notify other controls of the state change. Thus one control responding to one event may trigger chains of events affecting several other controls in the GUI. In complex cases, these event chains may be long; reaching a stable state may require the processing of many events. The propagation of events is done by an event-handling layer of the system, not by the controls themselves. So, from the perspective of an application developer, the order in which events are handled is nondeterministic.

Although a GUI's controls are loosely coupled from a communication perspective, an implementation usually arranges them into some hierarchical data structure. For example, the controls within a Web-based GUI are organized by the Document Object Model (DOM) within a browser. Similarly, the controls within a C# desktop GUI are organized by a separate class named `Designer`; this class abstracts the UI visual representation and contains a hierarchical set of controls. The display system uses these data structures when it periodically renders the GUI onto the screen.

This is where transitional turbulence can arise. The processing of a long chain of events may span across several cycles of the display system. A control may be rendered with a state that is inconsistent with the states of other controls. This may result in displays that are temporarily inaccurate or misleading from the perspective of a human user.

To combat this problem in Chapter 3 we developed a reactive programming (Bainomugisha et al., 2013) approach that analyzes the complex relationships among the GUI controls, encodes these dependencies in a dependency graph, and then uses the graph to rearrange the updates in an order consistent with the dependency constraints. It builds the

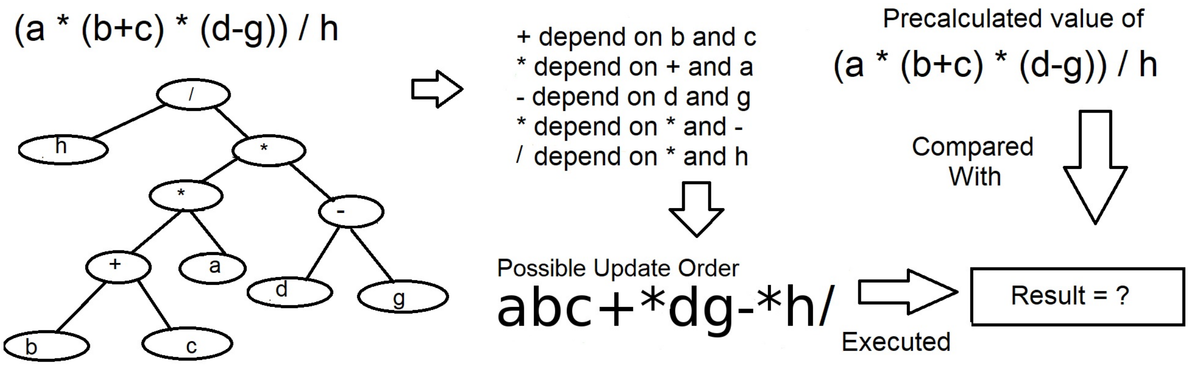


Figure 5.3. Constructing dependency graph and determining execution order.

graph when the GUI starts up and then rebuilds it whenever it detects that the dependencies might have changed. The approach thus coalesces the processing of a chain of what may be several events in the unmodified system into a single, large-grained event that updates the states of many controls at once. Although the approach does not totally eliminate the transitional turbulence that can cause inaccurate or misleading displays, it does potentially decrease the number of inaccuracies as well as increase the performance of the system.

To evaluate their approach, we developed a prototype library and used it to conduct several experiments. The experiments involved both desktop and Web versions of three different forms that self-complete (i.e. compute the values in some fields from values supplied in other fields). We executed a test 50 times on each form and measured the startup time, the total time, and the total number of inaccuracies. We compared our approach with approaches that used the Sodium (Blackheath and Jones, 2016) and Rx.NET (reactivex.io, 2020) reactive programming libraries and the builtin .NET GUI library. Figure 5.3 shows, in general, how our approach augments this GUI application's event-handling mechanisms.

Our experiments indicate that, on average, our approach requires less total time and exhibits fewer errors, at the cost of a modest increase in startup time compared to all three alternatives. Each application developed with the prototype library required approximately twice as much time to start up as the corresponding unmodified .NET application required. However, it was able to complete the entire chain of form updates in a small fraction of the time the corresponding unmodified .NET application required. In addition, it exhibited significantly fewer visual inaccuracies than the corresponding unmodified .NET application exhibited. Based upon the results of their experiments, we concluded that our approach improves performance and results in a more accurate behavior.

Our approach in Chapter 3 seems to work well for the kind of problems envisioned and the technologies used in this solution. Given how the approach works, it seems reasonable that it will work in similar situations. But what is the essence of their solution and how can we characterize the problem that it solves? In the next section, we examine another existing solution to see what we can learn.

5.4.2 Dynamic VR Applications

Chapter 4 extended and systematized our research from a preliminary study (Marum et al., 2019). This preliminary work also motivated the solution we examined in the previous section.

The solution in Chapter 4 sought to remove the instability corresponding to the transitional turbulence occurring in virtual reality (VR) and augmented reality (AR) applications implemented with C# in the Unity3D game engine (Unity Technologies, 2019). (We chose Unity3D because it is a popular platform for VR/AR application development.) These applications are inherently *reactive* and *nondeterministic* with respect to how and when the internal mechanism will execute such events and eventually deliver the resulting state.

Whenever multiple game objects in the simulated scene interact with one another, it may take several cycles for the VR/AR application to update the states of all components and reach stability. As we discuss for the first existing solution, this is called transitional turbulence or, sometimes, the "ripple effect". Transitional turbulence can result in inconsistencies in what is displayed for the user, which may lead to inconsistent and misleading states within the VR/AR application, making the entire application seem unreliable and unpredictable. Our approach focuses on reordering the execution of events so that the "ripple effect" can often be resolved within one update cycle.

Much of the nondeterminism is due to the unpredictable nature of the user interactions, but some of it is due to the lack of the application developer's control over some aspects of the execution, especially those aspects affecting the order in which events and the responses to those events occur in the system. The removal of this type of nondeterminism yields a more accurate system.

This study shows that Unity3D does not provide a mechanism for controlling the order of execution. Chapter 4 argues that the ability to change the execution order of components—and, consequently, to enable the correct ordering of the components' executions in a scene graph—is key to achieving highly accurate systems. To be perceived as accurate, simulated interactions must occur in the same order as the interactions would in the corresponding real-world situation. If they do not, then the simulation does not seem realistic to the user. Consider a domino chain. When the first domino falls, the second should fall when the weight of the first domino causes it to fall. The third falls similarly, and so forth throughout the chain. If any one of these falls is shown incorrectly, the whole simulated sequence is likely to be perceived as unrealistic.

As with the Dynamic GUI solution we examined above, we developed a reactive programming (Bainomugisha et al., 2013) approach to mitigate the transitional turbulence problems. This approach analyzes the complex relationships among the game objects present in the scene hierarchy, encodes these dependencies in a dependency graph, and then uses the graph to rearrange the updates in an order consistent with the dependency constraints. It builds the graph when the application starts up and then rebuilds it whenever it detects that the dependencies might have changed. The approach thus coalesces the processing of a chain of what may be several events in the unmodified system into a single, large-grained event that updates the states of many controls at once. Although the approach does not totally eliminate the transitional turbulence that can cause inaccurate or misleading displays, it does potentially decrease the number of inaccuracies as well as increase the system's performance.

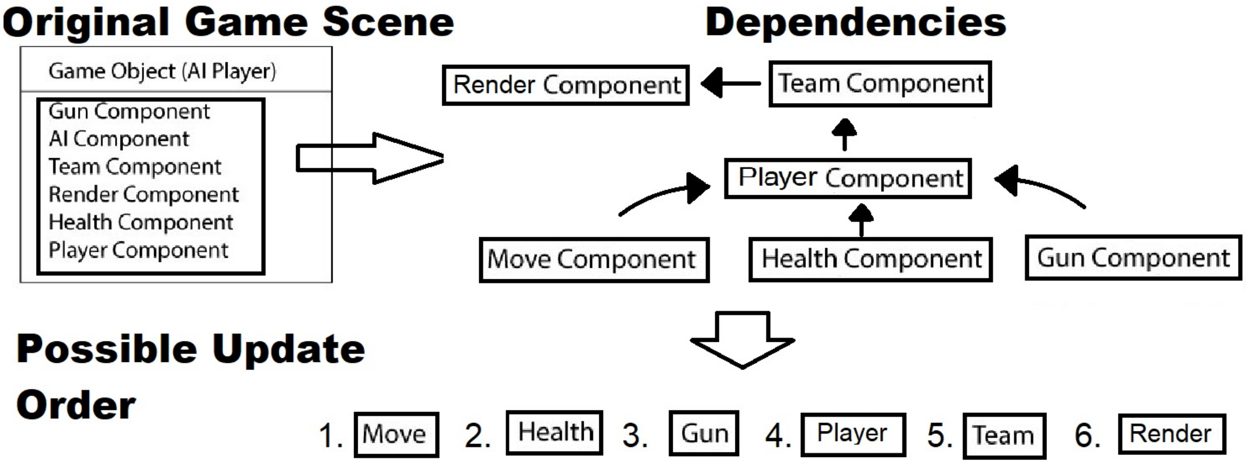


Figure 5.4. Dependency test using the expression evaluator.

To evaluate our approach, we developed a prototype library and used it to conduct several experiments. The experiments involved a three-way comparison among applications on Unity3D using our approach, the default Unity3D event system, and UniRx, the Reactive Extensions library for the Unity3D platform (Kawai, 2014). Figure 5.4 also shows, in general, how our approach augments this VR/AR application's event-handling mechanisms.

To evaluate the effectiveness of our approach, we used a test scenario built around an expression evaluator to demonstrate how the update order of the game objects affects the interactions among the game objects. we represented the expression as a game tree with an operator at each internal node with its operands as its subtrees. The values are at the leaf nodes. For the correct value of the expression to be calculated, the operand subtrees must be evaluated before the corresponding operator. To determine the effects of reconfiguration of Unity3D's game tree, we ran (a) tests that kept the expression tree stable throughout the run and (b) tests that randomly introduced changes in the tree's structure during the test run. For each test run, we measured the startup time, latency, and total errors during the run and computed the average startup time, latency, errors per cycle, errors that resulted in a visibly inaccurate state, and number of miscalculations that occurred in a test that failed. Our experiments indicate that, on average, our approach exhibited a shorter latency and fewer errors, at the cost of a modest increase in startup time compared the other two alternatives. We concluded that our approach improves performance and results in more accurate behavior as described in Chapter 4.

Thus, our approach also seems to work well for the kind of problems envisioned and the technologies used in the second solution. Our task is now to identify the commonalities of the two specific solutions and codify a general, technology-independent approach as a new design pattern.

5.4.3 Generalizing the Solutions

In Chapter 3 and 4, we examine each of the systems above in detail. Both solutions develop a reactive programming approach that encodes the complex relationships among the components of a specialized Implicit Invocation system 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.

In the "What"-Solutions pattern, Harrison (2006) suggests writing the core idea of a solution in a one- or two-sentence summary. This summary will form a prominent part of the full description of the new pattern's Solution element.

The draft summary shown in Figure 5.5 captures these observations. Consider the GUI solution from Chapter 4. This solution augments .NET's default event-handling mechanism by building a dependency graph and using it to schedule the updates of the controls. If the update of some control `A` modifies some other control `B`, then `B` depends on `A`. At startup, the augmentation first analyzes the entire GUI to identify all the depends-on relationships and then builds a dependency graph that records these relationships as directed edges. During the execution of the GUI, whenever a user interacts with some control `X`, the augmentation immediately updates all other controls that directly or indirectly depend upon `X`. The unrelated controls in the GUI are not affected. This approach to execution matches how we expect GUI's to operate.

> Solution
>
> A solution encodes the complex relationships among the application's components in a dependency graph and then uses the graph to order the updates of the components without violating the dependency constraints. The goal is to reorder the updates of the components so that the new order reduces transitional turbulence without degrading the performance of the system.

Figure 5.5. Solution element summary.

In our study of these solutions, we tentatively identify Dynamically Coalescing Reactive Chains as the name of this pattern.

5.5 Writing the Pattern

In this section, we continue along the pattern writer's path, traversing steps 3-10 from Section 5.2. We formulate the new pattern's Problem, Forces, and Consequences elements, refine the Context and Solution elements, and choose a Pattern Name.

5.5.1 Describing the Problem

In step 3 on the pattern writer's path, we describe the problem that leads to the solution. The core of a pattern is the pairing of a Problem with the corresponding Solution. However, Harrison (2006) observes that often "the problem and solution are basically restatements of one another" during the early phases of writing a pattern. To help differentiate these, the "Why"-Problems pattern (Harrison, 2006) suggests that pattern writers ask themselves "how the world would be worse" if the new pattern is not used. Of course, the pattern writers can make "the world" as specific as it needs to be by how they define the Context.

In the solutions we examine in Section 5.4, the core issue addressed by the existing solutions (from Chapters 3 and 4) is reducing the transitional turbulence. Transitional turbulence can result in an external presentation that does not accurately represent the system's expected behavior. This leads us to the initial statement of the Problem element for the new pattern in Figure 5.6.

Problem

We want to eliminate or reduce the length of the periods of transitional turbulence during which the external presentation does not accurately reflect the state of the application. We need to do this without sacrificing performance. The goal is to better satisfy observers' expectations by increasing the accuracy of the external presentation.

Figure 5.6. Problem element draft.

Consider how the problem can be specifically observed. In the GUI study in Chapter 3, when a user enters data in the form, it may reconfigure itself. If the interconnections among controls are complex then it may take several display cycles for all the changes to propagate throughout the form. During this period, the form may show invalid options or may redraw itself while the user is entering data. It is understandable that both situations would be frustrating to the user.

Although the Context and Solution must be refined further, the proposed Solution seems to solve the stated Problem in the given Context. The Problem (Figure 5.6) specifies *what* must be done. The Solution (Figure 5.5) proposes *how* that can be accomplished. The Context (Figure 5.2) describes the environment in which the Problem and its Solution exist.

5.5.2 Considering the Consequences

In step 4 on the pattern writers path, we consider the consequences of the solution. The Solution to a Problem has both *benefits* and *liabilities* (Wellhausen and Fiesser, 2011). Figure 5.7 shows the Consequences we identify for the new pattern.

To identify benefits, we consider what happens if the Solution is applied? And what happens if it is not applied? To identify liabilities, we consider what its drawbacks are? Let us consider the experimental results from the two studies we examine in Section 5.4

In the dynamic GUI application in Chapter 3, the result shows that our approach requires less time to reach a stable state (i.e., has shorter latency) and exhibited fewer errors (i.e., inaccuracies) at the cost of a modest increase in startup time. In the dynamic VR application in Chapter 3, our approach also reduces the latency for reaching a stable state

Consequences

Benefits:

- A solution dynamically adapts to changes in an application's component architecture during normal operation.
- A solution coalesces sets of dependent internal events into "large-grained events" such that the handling of a large-grained event causes the same overall state change as the corresponding set. This can decrease latency and increase accuracy (i.e. decrease the number of errors).
- An application can be readily adapted to use the mechanisms implementing the solution.

Liabilities:

- Changes to an application's component architecture during normal operation can increase latency and decrease accuracy.
- An implementation of a solution often causes additional processing overhead at startup and shutdown of the application.
- An implementation of a solution often causes additional processing overhead during normal operation, especially when the component architecture changes.
- An application must be adapted to use the mechanisms implementing the solution. Modifying the application often complicates its design, implementation, testing, or use.

Figure 5.7. Consequences element draft.

and exhibits fewer errors at the cost of a modest increase in startup time. Both applications adapt appropriately to the runtime reconfiguration of the user interfaces' structures. Neither application seems to require significant changes to the application's design or implementation. In both studies, our approach eliminates or reduces transitional turbulence, which results in more accurate behaviors.

Some of the phrases used in the Consequences and in other elements may need explanation. By "changes to an application's component architecture", we mean the addition, deletion, or modification of components and events within the same architecture constructed according to the Implicit Invocation pattern. By "mechanisms implementing the solution", we mean some combination of libraries, frameworks, tools, and design and program-

Forces

Runtime Reconfiguration: We want to adapt to changes in an application's component architecture during normal operation.

Transitional Turbulence Reduction: We want to decrease the transitional turbulence in the application's execution to better satisfy the observers' expectations.

Startup Cost Inflation: We want to avoid adding significant startup or shutdown costs.

Operational Overhead Creep: We want to avoid adding significant processing overhead during the application's normal operation.

Code Cluttering: We want to avoid significantly complicating the application's design, implementation, testing, and use.

Figure 5.8. Forces element draft.

ming techniques used to implement the solution.

5.5.3 Identifying the Forces

In step 5 on the pattern writer's path, we identify the forces. The Forces are aspects of the stated problem and its context that make selecting and devising a solution difficult. Figure 5.8 shows the Forces we identify for this Problem. Following the suggestion of the Visible Forces pattern (Meszaros and Doble, 1998), we assign each force in the new pattern a meaningful name and display the set of forces as a list.

The forces concern issues such as runtime changes in the application's component architecture (Runtime Reconfiguration), the core issue of Transitional Turbulence Reduction, increasing runtime overhead (Startup Cost Inflation and Operational Overhead Creep), and increasing implementation complexity (Code Cluttering).

To characterize transitional turbulence, we use two measures (as in Chapters 3 and 4): latency and error (inaccuracy) counts. *Latency* is the period of "time" (perhaps measured in update cycles) that it takes for all components to reach a stable state following some stimulus (such as the processing of an external event). An *error* (or *inaccuracy*) is an inconsistency in the externally "visible" state of the application that can occur during such a period of instability. To reduce transitional turbulence means, in general, that we need to reduce both of these measures. (We also refer to decreasing the number of errors/inaccuracies as increasing accuracy.) Following the suggestion of the Forces Hint at Solution pattern (Harrison, 2006), we order them from Problem-oriented issues toward Solution-oriented issues.

5.5.4 Matching Forces with Consequences

In step 6 on the pattern writer's path, we match each force with the corresponding consequence. Figure 5.9 shows how we map the Forces to the benefits and liabilities in the new pattern.

A force makes the problem difficult to solve. How the solution resolves this difficulty leads to the corresponding consequence. Each force must be resolved. In most cases, the mapping from forces to consequences should be one-to-one (Wellhausen and Fiesser, 2011) (i.e., each consequence should be matched to exactly one force). However, a force may be mapped to both a benefit and a liability. In Figure 5.9 note that the Code Cluttering and Runtime Reconfiguration forces each match with both a benefit and a liability. Also note that each consequence matches with exactly one force.

As first-time pattern writers, we found it necessary during this step to revisit the previous two steps and refactor both the Forces and the Consequences to ensure that we had the primary issues covered in compatible ways. The steps from Section 5.2 are not rigid; instead they are soft guidelines to help writers think through the issues. However, it is still useful for us to "write the documentation" (this chapter) as if "we had followed the ideal process" (Parnas and Clements, 1986).

5.5.5 Describing the Context

In step 7 on the pattern writer's path, we describe the context in which the problem exists. The context defines "aspects and requirements that are so important that the problem may not exist outside the context but that are, at the same time, not modified by the solution" (Wellhausen and Fiesser, 2011). The context "imposes constraints on the solution"

Matching Forces

Runtime Reconfiguration: We want to adapt to changes in an application's component architecture during normal operation.

- Benefit: A solution dynamically adapts to changes in an application's component architecture during normal operation.
- Liability: Changes to an application's component architecture during runtime can increase latency and decrease accuracy.

Transitional Turbulence Reduction: We want to decrease the transitional turbulence in the application's execution to better satisfy the observers' expectations.

- Benefit: A solution coalesces sets of dependent internal events into large-grained events such that the handling of a large-grained event causes the same overall state change as the corresponding set. This can decrease latency and increase accuracy (i.e. decrease the number of errors).

Startup Cost Inflation: We want to avoid adding significant startup or shutdown costs.

- Liability: An implementation of a solution often causes additional processing overhead at startup and shutdown of the application.

Operational Overhead Creep: We want to avoid adding significant processing overhead during the application's normal operation.

- Liability: An implementation of a solution often causes additional processing overhead during normal operation, especially when the component changes.

Code Cluttering: We want to avoid significantly complicating the application's design, implementation, testing, or use.

- Benefit: An application can be readily adapted to use the mechanisms implementing the solution.
- Liability: An application must be adapted to use the mechanisms implementing the solution. Modifying the application often complicates its design, implementation, testing, or use.

Figure 5.9. Matching forces with consequences.

(Meszaros and Doble, 1998).

The basic Context (from Figure 5.2) is that we have an application constructed according to the Implicit Invocation architectural pattern. Beyond that, we examine what additional assumptions that the two existing solutions make about contexts in which they execute. By doing so, we refine the Context to that shown in Figure 5.10.

The existing solutions assume that components can be added, removed, or modified during normal operation and the components are organized into a hierarchical data structure that can also change as the application executes. They also assume that the collective state of the components is sampled periodically (e.g., by a display system) and presented externally and, as a result, the application can exhibit transitional turbulence. In addition, the applications assume that the components encapsulate their states behind interfaces and that the implementation environment or the application itself allows a program to extract metadata about the components and their interfaces.

5.5.6 Choosing the Pattern Name

In step 8 on the pattern writer's path, we choose a pattern name. We can use several patterns from Meszaros and Doble (1998) to guide us in this task. The Evocative Name pattern suggests choosing a name that evokes an image that conveys "the essence of the pattern solution to the target audience". The name should be memorable and suitable for addition to the technical vocabulary of software developers.

The Noun Phrase Name pattern suggests naming the pattern for the result it creates. The Meaningful Metaphor Name pattern further suggests choosing a name based on a metaphor that is familiar to the target audience.

We adopt the name Dynamically Coalescing Reactive Chains because it seems to best meet the criteria given by Meszaros and Doble (1998). For convenience, we sometimes use the acronym DCRC. We show this choice in Figure 5.11.

Context

We have an application that has the following characteristics:

- The application is constructed according to the IMPLICIT INVOCATION architectural pattern (Shaw, 1996), assuming nondeterministic but fair handling of events.
- The application's component architecture may change during normal operation. The application organizes the components into a hierarchical structure. This structure may change dynamically during the application's normal operation as a result of external stimuli or the actions of components.
- The application presents some aspects of its state that can be observed from outside the system periodically. The timing of this presentation is not under the control of the application.
- Because of the asynchronous nature of the application's operation, the externally observable presentation may exhibit periods of *transitional turbulence*. By transitional turbulence, we mean a period of chaotic or unreliable variation in the application's state that can result from one or more changes to the application's interconnected components. It can result in an externally observable state that does not accurately represent the expected result.
- Each component is an information-hiding module (Parnas, 1972) with a well-defined interface (Britton et al., 1981). The only way to change or access its state explicitly is by calling one of its accessor or mutator procedures (e.g. properties in some object-oriented languages).
- The application supports *reflection* capabilities. That is, application-level code can examine the application's features (such as its components, events, event handlers, and hierarchical structure) at runtime and extract metadata (such as names, types, and the type signatures of the procedures in component interfaces).

Figure 5.10. Context element revision.

Pattern Name

DYNAMICALLY COALESCING REACTIVE CHAINS

Figure 5.11. Pattern name element draft.

5.5.7 Rewriting the Pattern Elements

In step 9 on the pattern writer's path, we reexamine and rewrite the six pattern elements. At this point in our process, the primary element that needs attention is the Solution, including how it relates to the Problem and the Forces. We need to provide sufficient detail for the reader to use the pattern effectively to design and implement a concrete solution. However, we want to keep the new pattern technology-independent and do not want to overwhelm the reader with arcane details of particular implementations and implementation technologies.

5.5.7.1 Solution-writing Guidelines

Several of Harrison's pattern-writing patterns (Harrison, 1999, 2006) give us guidance on how to refine the Solution:

- The Big Picture pattern (Harrison, 1999) suggests that the Problem and Solution should "by themselves" convey the key idea—"the big picture"—of the new pattern.
- The Matching Problem to Solution pattern (Harrison, 1999) suggests that the Solution should solve the "whole" Problem "but not more".
- The Convincing Solution pattern (Harrison, 1999) suggests that pattern writers seek to make the Solution "compelling". Often this means making it "narrower and deeper".
- As we discussed in Section 5.4, the "What"-Solutions pattern (Harrison, 2006) suggests writing the core idea of the Solution in a one- or two-sentence summary placed at the beginning of the Solution description. The "How"-Process pattern (Harrison, 2006) suggests extending the summary with more detail about "what to do, how to do it, and why to do it that way," including providing any appropriate illustrations. In particular, it should describe how the Solution balances the Forces and identify any Forces that are not considered.

- The Forces Hint at Solution pattern (Harrison, 2006) suggests that the Forces should guide the reader from the Problem to the Solution.

Because of our goal of keeping the new pattern technology-independent, we found satisfying Harrison's Convincing Solution and Matching Problem to Solution patterns (Harrison, 1999) challenging. The latter required us to tweak the statement of the Context to include subtle assumptions the Solution makes about the environment.

5.5.7.2 Refined Solution

Summary (from Figure 5.5). A solution encodes the complex relationships among the application's components in a dependency graph and then uses the graph to order the updates of the components without violating the dependency constraints. The goal is to reorder the updates of the components so that the new order reduces transitional turbulence without degrading the performance of the system.

As we note in Section 5.4, our solutions rely on multiple aspects. To apply the pattern, a developer must first understand the concept of dependency and then build a graph that encodes the dependencies in an application. A design that applies the pattern must augment the existing system with various mechanisms. Below we propose the mechanisms that must be defined along with the pattern.

Definitions. What do we mean by a "dependency graph" in this Context?

- If the execution of a component C can directly affect a subsequent execution of a component D in any way, then D *depends on* C.

  For example, C might trigger an event for which D listens, change aspects of its state that D accesses, directly call one of D's mutator procedures, or create or modify component D.

- A *dependency graph* is a digraph formed by placing the components at the nodes and

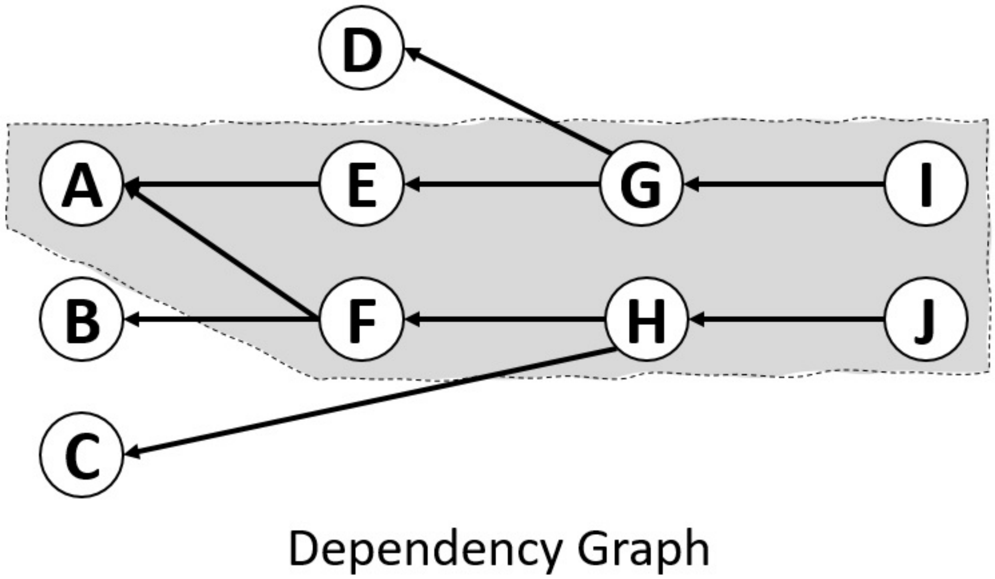


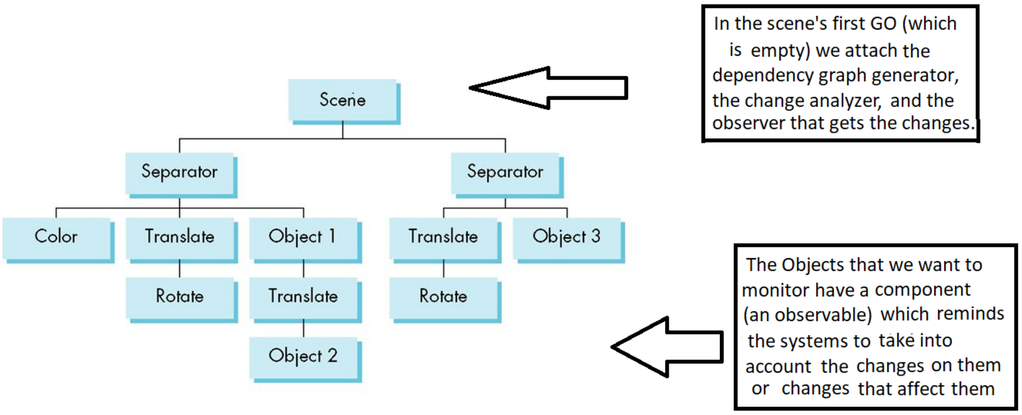
Figure 5.12. Theoretical evaluation order overwritten on top of the dependency graph.

adding an edge from one node to another if the corresponding components have a *depends-on* relationship.

Figure 5.12 shows a dependency graph as the highlighted area on the top of the existing component graph.

For example, the GUI augmentation in Chapter 3, we define dependency as a relationship in which one component is related to another if the execution of the first leads (directly or indirectly) to the execution of the second and the order in which they execute affects the final result of the execution. Each directed edge denotes a dependency relationship in which the execution of source control leads to the execution destination control.

To apply the DCRC pattern, we are primarily interested in recording the dependencies related to the implicit invocations—between components that listen for an event and those that trigger the event. Of course, being able to record other kinds of dependencies may also be helpful.



In the scene's first GO (which is empty) we attach the dependency graph generator, the change analyzer, and the observer that gets the changes.

The Objects that we want to monitor have a component (an observable) which reminds the systems to take into account the changes on them or changes that affect them

Any change on an observable object will prompt a verification of all of its dependents. It does not matter if these changes were prompted by another object or by an input. If an observable change affects another observable object or multiple observable changes are prompted at the same time by a change, they will remain in a stack until all changes are propagated. That is why no circular dependencies or coupled dependencies are allowed. The final dependency graph generated must be DAG (directed acyclic graph). How dependencies will be covered and how to develop this routine in every frame still needs to be discussed. Also regardless of what kind of environment the algorithm should be tested, the end game is to build a framework.

Figure 5.13. Modifications that will occur in the original hierarchy.

Augmenting the application. To apply the DCRC pattern to an application that satisfies the Context, we augment the application with appropriate *software mechanisms.* Some of these mechanisms depicted in Figure 5.13.

These mechanisms may include some combination of libraries, frameworks, tools, and design and programming techniques. The various mechanisms should be *lightweight.* That is, they should execute efficiently and should not require extensive modifications of the existing application. The "software mechanisms" needed and the meaning of "lightweight" depend upon the application's specific implementation technologies and performance requirements.

In the GUI augmentation in Chapter 3, we cite the mechanisms provided by the .NET. .NET provides lightweight capabilities for iterating through the controls of a form, tracking and executing the controls, and detecting when a user has interacted with a control.

For applications that satisfy the Context, we can augment the application's event-handling mechanisms to solve the Problem. The solution involves three processes: *analyzing* the application to identify how to add the mechanisms, *developing* the mechanisms, and *incorporating* the mechanisms into the application's operation.

1. In the **augmentation analysis process**, the solution's developer must:

    (a) Examine the hierarchical structure to identify how a program can iterate through the components (i.e., accessing each component exactly once).

    (b) Examine the design and implementation of the components and the features of the implementation language to identify how a program can extract the dependency relationships among the components at runtime.

    This may involve use of the components' existing features or the implementation language's reflection capabilities. If sufficient capabilities do not exist, we can design lightweight modifications that implement sufficient application-specific capabilities.

    (c) Examine the components and events to determine which component relationships to include in the dependency graph and which to exclude. To reduce transitional turbulence, the augmented application program can manipulate the components and relationships included but cannot manipulate those excluded.

    Generally speaking, we include the component relationships arising from the application's custom code (which we can modify if needed) and exclude those in the supporting framework (which we cannot modify). We may also want to exclude any component relationship if that relationship represents an expensive computation or arbitrary delay.

2. As it is illustrated in Figure 5.14, in the **augmentation development process**, the solution developer must:

    (a) Design and implement a lightweight runtime mechanism that enables the program to differentiate between the components that are to be included in the dependency

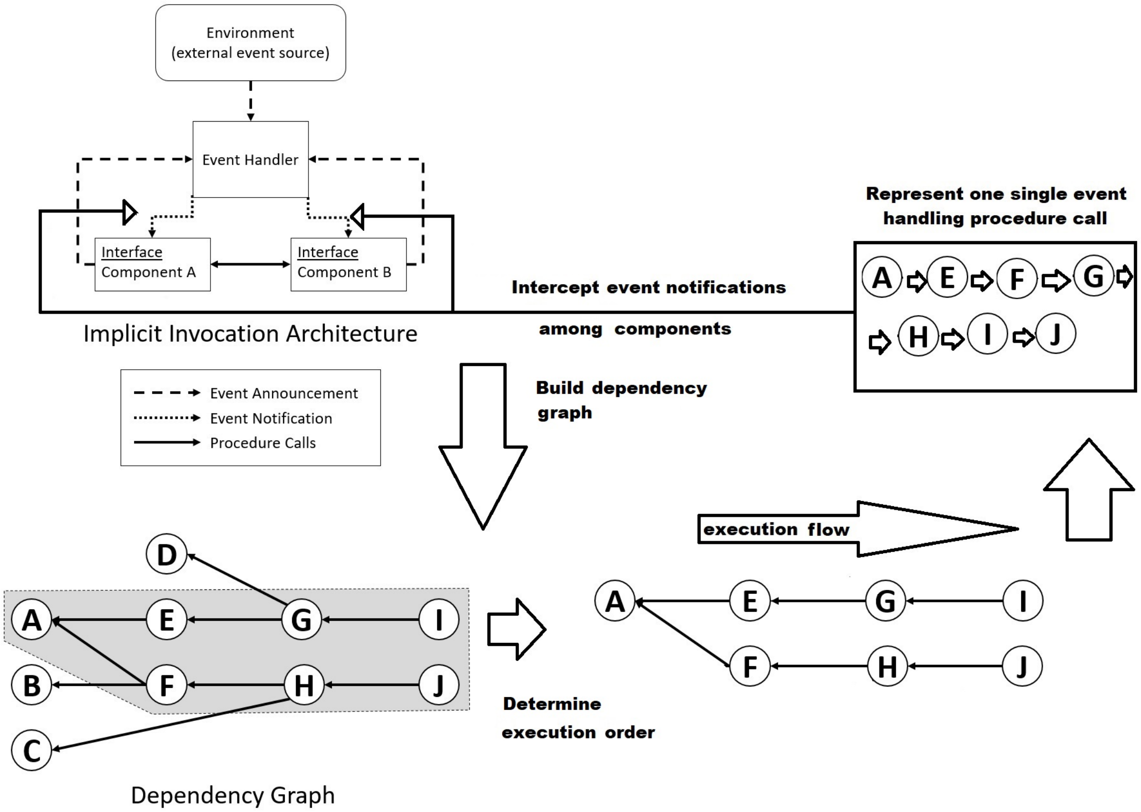


Figure 5.14. Augmentation from the implicit invocation to the dependency graph.

graph and those that are not.

This may involve features already present in the application (e.g., types, value of some property, metadata) or may involve modifying the application to add appropriate features. For example, in an object-oriented system in which the components are objects, we could modify the included components to implement a “marker interface” that can can be checked by reflection.

The developer should establish a criterion to determine what to include in the dependency graph and what to exclude. In general, this criterion can be defined as a function to be called by the dependency graph building procedure. It must return a boolean value `true` if its argument should be inserted in the dependency graph and otherwise return `false`.

(b) Design and implement a lightweight runtime mechanism that enables the program to detect whether the component architecture or the dependencies among the components have changed since the previous check (or since the beginning of operation).

In this Context, we assume that a change to the hierarchical structure holding the components likely means a change to the component architecture.

(c) Design and implement lightweight mechanisms to construct the dependency graph initially and to reconstruct it when needed.

To build a dependency graph, the program can traverse the hierarchical structure (e.g., do a breadth-first traversal of the Document Object Model), placing each component at a node and adding edges to other nodes according to the *depends-on* relationships between components. However, it must prune the graph appropriately to remove any cycles.

3. In the **augmentation incorporation process**, the solution developer must modify the application's operation in the following ways:

(a) The application must construct the dependency graph at or before startup. (The left side of Figure 5.14 illustrates this augmentation.)

(b) When some component C included in the dependency graph signals an event E, the application must *intercept* E and directly call the procedures associated with event E on all listening components as recorded in the dependency graph. Then it must recursively apply the process to all events signalled by the listening components. It continues this as long as there are dependencies indicated in the graph (which cannot have cycles). This process dynamically coalesces the processing of chains of events into what is processed as one "large-grained" event. (The upper half of Figure 5.14 illustrates this augmentation.) The meaning of "intercept" depends upon the application's specific implementation technologies.

(c) After the processing of each "large-grained" event in the previous step, the application must check whether the application's component architecture has changed (e.g., the addition, modification, or deletion of any component in the hierarchical structure) or the dependencies among components have changed. If so, then it must update the dependency graph appropriately to reflect the new component architecture.

Balancing the forces. In the Solution described above, we handle all the identified Forces. How do we balance the various Forces to achieve this Solution?

- *Transitional Turbulence Reduction.*

  For a state change in any component, the augmented application must propagate the effects to all its directly or indirectly dependent components without the delays and non-determinism introduced by the normal event-handling system—as if all were part of the processing of one large-grained event. This can decrease latency and increase accuracy (i.e., decrease the number of errors).

- *Runtime Reconfiguration.*

  Frequently during normal operation of the application, the augmented application checks whether the its component architecture has changed. If it detects a change, it then reconstructs the dependency graph to reflect the new architecture. The extra costs incurred in reconstructing the dependency graph must not itself worsen the solution's overall effect on the latency and accuracy.

  Changes to an application's component architecture during normal operation can increase latency and decrease accuracy. However, a good solution must dynamically adapt to such changes and seek to mitigate the effects on latency and accuracy.

- *Startup Cost Inflation.*

When applying the pattern, developers should seek to keep the cost of initially constructing the dependency graph low. The developers should carefully select the components to include in the analysis and use efficient methods for determining dependency relationships and constructing the graph.

The augmented application likely incurs additional processing overhead at startup and shutdown. In particular, the extra costs for constructing the initial dependency graph should be small in proportion to the potential accuracy and performance gain in an application that runs sufficiently long.

- *Operational Overhead Creep.*

  The augmented application likely incurs additional processing overhead during normal operation, especially when the component architecture changes. In particular, the extra costs for checking for changes in the component architecture and reconstructing the dependency graph should be small in proportion to the potential accuracy and performance gain in an application that runs sufficiently long. In cases in which the component architecture changes infrequently, the augmented application should incur minimal costs.

- *Code Cluttering.*

  To implement a solution, the developer must augment the existing application by incorporating a set of software mechanisms as described above. Unfortunately, modifying the application often complicates its design, implementation, testing, and use.

  However, in a good design and implementation of the solution's new software mechanisms, it should be possible to readily augment the existing solution. Thus the new software mechanisms should be designed, implemented, and documented carefully so that the solution can work well with typical application designs.

  For example, for a typical GUI application it should be possible to implement the solution approach as a software framework with wrapper classes for the controls and a library implementing the algorithms for constructing/reconstructing the dependency graph and using it to coalesce chains into "large-grained" events.

5.5.8 Putting the Elements in Standard Order

In step 10 on the pattern writer's path, we put the pattern elements in the standard order. We seek to organize the DCRC pattern according to Meszaros and Doble's SINGLE-PASS READABLE pattern (Meszaros and Doble, 1998). That is, we seek to write the pattern so that it flows smoothly from Context to Consequences, capable of being read sequentially and understood in one pass.

We give the full pattern description in Appendix A, including the refined Solution element from Section 5.5.7. We also restructure the Consequences component from Figure 5.7 to show the mapping from the Forces as shown in Figure 5.9.

5.6 Evolving the Pattern

The final step on the pattern writer's path is to get feedback from experts in the technical area and pattern writing. The patterns community (The Hillside Group, 2020) often uses a process called *shepherding* to assist pattern writers (Harrison, 1999, 2006). This is a "process in which a pattern author receives feedback from another, experienced pattern author" (Wellhausen and Fiesser, 2011). It is an iterative process in which the experienced writer—the "shepherd"—gives feedback to the pattern's author—the "sheep". Harrison's THREE ITERATIONS pattern (Harrison, 1999) suggests that approximately three rounds of feedback and revision are required. Often this coaching is associated with a conference such as Pattern Languages of Programs (PLoP) (The Hillside Group, 2020).

This DCRC pattern is based on the work reported in Chapters 3 and 4. It is being written primarily by the author of this dissertation in collaboration with his two dissertation advisors and another collaborator. Two members of the team have expertise in software architecture and two in the application areas and technologies underlying the existing solutions we examined in Section 5.4. None have previous experience writing software patterns, but most have some experience using patterns. One member from each group is taking the lead in writing the pattern itself, with the other two serving as reviewers. We organized our work using the writer's path outlined in Section 5.2, which is adapted from the helpful Wellhausen and Fiesser (2011) tutorial, and applied patterns from the published pattern languages for pattern writing (Harrison, 1999, 2006; Meszaros and Doble, 1998). In this pattern, we record our steps along the path as well as give the full DCRC pattern.

As Buschmann et al. (2007) notes, "Just as useful software evolves over time as it matures, pattern descriptions evolve over time as they mature." Software patterns are, in some sense, always works in progress that can incorporate "deeper experience gained when applying patterns in new and interesting ways". In particular, they may be refined to form part of a *pattern language*—"a network of interrelated patterns that defines a process for resolving software development problems systematically." A pattern language combines a *vocabulary*—a set of evocatively named patterns—with a *grammar*—the rules for combining individual patterns into valid sequences in which they can be applied.

In this pattern description, we seek to specify a design pattern with a relatively broad context. It would have been an easier task for us to specify an idiom (or idioms) for the narrower context of C#.NET and Unity3D by drawing on our understanding of the work in Chapters 3 and 4. As our work on the pattern evolves, we plan to evaluate how well the general pattern can be specialized to specific technologies, which may lead us to define related idioms. However, it may also be that future evolution will lead us in the direction of a network of patterns that we will seek to weave into a pattern language covering variations of the event-driven, Implicit Invocation architecture (Shaw, 1996) and implementation platforms.

In the future, we also plan is to evaluate the generality of the DCRC pattern by replicating one or more of our previous case studies using different technologies (e.g., replicating the GUI case study from Chapter 3 using Java and JavaFX (Chin et al., 2019; OpenJFX

Project, 2020)). We can also examine other reactive programming approaches (Bainomugisha et al., 2013; Cooper and Krishnamurthi, 2006; Czaplicki and Chong, 2013; Drechsler et al., 2014; Elliott, 2009; Meyerovich et al., 2009; Reynders et al., 2017) to see what new ideas can be incorporated into a future revision of the DCRC pattern.

As we describe in Section 5.3, we built our context around the Implicit Invocation architectural pattern as specified by Shaw in 1996 (Shaw, 1996). One attraction to this classic presentation is that it is concise and described using consistent software architecture concepts. Another attraction is that this architecture has been the focus of formal modeling research (e.g., using Z notation (Garlan and Notkin, 1991), process algebra and trace semantics (Garlan et al., 1998), and model checking (Garlan et al., 2003)). Our primary focus has been on defining a pragmatic design pattern useful to practitioners as well as to researchers. To date, we have not given attention to devising a formal specification. However, our ongoing work to evolve the pattern's Solution could benefit from a better formal understanding of the operation of the kinds of event-driven, Implicit Invocation systems we focus on.

5.7 Applying the Pattern

In this section, we demonstrate the efficacy of the Dynamically Coalescing Reactive Chains (DCRC) pattern by applying it to a realistic example. For convenience, we choose to reexamine the .NET-based dynamic GUI application case study from Chapter 3. In the future we plan to replicate this work using other technologies—such as Java and JavaFX—as discussed in Section 5.6. In this section, we consider the DCRC pattern element by element and evaluate its applicability to the dynamic GUI application.

5.7.1 The Context

The DCRC pattern's Context element lists six characteristics (shown in italics below). We consider each Context characteristic with respect to the dynamic GUI application.

- *The application is constructed according to the* Implicit Invocation *architectural* *pattern, assuming nondeterministic but fair handling of events.*

A .NET GUI consists of a loosely coupled collection of controls. The controls execute asynchronously with respect to each other. A control can interact externally (e.g., with a user) and internally (e.g., with other controls). The controls interact via events, which are scheduled in a nondeterministic manner. Thus, a .NET GUI exhibits the characteristics of an implicit invocation architecture as defined in Section 5.3.

- *The application's component architecture may change during normal operation. The application organizes the components into a hierarchical structure. This structure may change dynamically during the application's normal operation as a result of external stimuli or the actions of components.*

  In a .NET GUI, the "components" are the controls, each of which is represented by an object. The GUI arranges the objects representing the controls into a hierarchical data structure internally (e.g., the DOM in a Web application). This data structure forms the "component architecture". It may change as the result of some action from outside the GUI or by execution of the GUI's controls themselves.

- *The application presents some aspects of its state that can be observed from outside the system periodically. The timing of this presentation is not under the control of the application.*

  The display system operates independently from a .NET GUI. It renders the GUI onto the screen periodically. To do so, it accesses the GUI's data structures (e.g., the DOM).

- *Because of the asynchronous nature of the application's operation, the externally observable presentation may exhibit periods of transitional turbulence.*

  As we note above, a .NET GUI's controls execute asynchronously and communicate via an event-handling mechanism. Because of the fine-grained nature of the events, it may be necessary to process many events to propagate the changes at one control to all other controls. However, the display system operates independently from the GUI and directly accesses the GUI's data structures. Thus, a .NET GUI can exhibit transitional turbulence as defined in this dissertation.

- *Each component is an information-hiding module with a well-defined interface. The only way to change or access its state explicitly is by calling one of its accessor or mutator procedures (e.g., properties in some object-oriented languages).*

  Each control in a .NET GUI is an object that instantiates a class from the `Control` class hierarchy. This object implements its class's interface and encapsulates (i.e., hides) all its attributes. Thus, the only way for another object to access or alter a control's internal state is to call a method in its interface. Some of the control's methods are associated with the operation of the event-handling system. Therefore, a control is an information-hiding module (Parnas, 1972) with a well-defined interface (Britton et al., 1981). The only way to access a control's state is by calling its methods.

- *The application supports reflection capabilities. That is, application-level code can examine the application's features (such as its components, events, event handlers, and hierarchical structure) at runtime and extract metadata (such as names, types, and the type signatures of the procedures in component interfaces).*

  The .NET framework's primary programming language is C#. C# is an object-oriented language in which everything is an object. The language's extensive *reflection* facilities enable a program to examine its objects at runtime and extract metadata about their features (e.g., the names, types, and values of attributes, the names and type signatures of methods, the types of objects, and the classes and interfaces extended by classes).

The dynamic .NET GUI applications we consider have all the above characteristics. The related .NET and C# features are sufficient to implement the software mechanisms (e.g., the dependency graph) needed to design and implement the Solution described below.

5.7.2 The Problem

The DCRC pattern's Problem element states the problem description as follows: *We want to eliminate or reduce the length of the periods of transitional turbulence during which the external presentation does not accurately reflect the state of the application. We need to do this without sacrificing performance. The goal is to better satisfy observers' expectations by increasing the accuracy of the external presentation.*

As we note in the discussion of the Context, the .NET GUIs we consider can exhibit transitional turbulence as defined in this dissertation. This can cause the GUI display to inaccurately reflect the state of the application for periods of time. In some circumstances, we may want to eliminate or reduce the length of these periods without sacrificing performance. Thus, the DCRC pattern addresses a problem that is relevant for .NET-based dyanmic GUI applications.

5.7.3 The Forces

The DCRC pattern's Force element identifies five forces (shown in italics below) that must be balanced appropriately to solve the Problem. We consider the dynamic GUI application with respect to each force.

Transitional Turbulence Reduction**:** *We want to decrease the transitional turbulence in the application's execution to better satisfy the observers' expectations.*

Decreasing transitional turbulence is the primary motivation for attempting to solve the Problem in this Context. In an implicit invocation architecture like .NET GUIs, this likely requires a solution that optimizes the event processing.

The Transitional Turbulence Reduction force is in conflict with all the other forces: Runtime Reconfiguration, Startup Cost Inflation, Operational Overhead Creep, and Code Cluttering. They represent factors that make achieving transitional turbulence reduction difficult.

Runtime Reconfiguration**:** *We want to adapt to changes in an application's component architecture during normal operation.*

In .NET GUI applications, the structure of the GUI itself can change during execution. The Context requires that we handle this situation in any Solution to the Problem. However, this situation may complicate any solution that optimizes the event processing based on the GUI's structure. For example, if the solution builds and uses a dependency graph of the controls, then changes in the GUI's structure invalidates the graph. This requires that the dependency graph be updated whenever the GUI's structure changes, which likely makes the code more complex and degrades performance.

The Runtime Reconfiguration force is in conflict with the Transitional Turbulence Reduction, Overhead Cost Creep, and Code Cluttering forces.

Startup Cost Inflation**:** *We want to avoid adding significant startup or shutdown costs.*

For .NET GUI applications, any Solution to the Problem likely requires that the GUI be analyzed and modified before normal operation begins. Both steps require that new software mechanisms (i.e., code) be developed and executed. In addition, the modified GUI likely has more complex code and increased execution time. For example, the analysis may construct a dependency graph of the controls and the modification may augment the GUI to use the dependency graph to optimize the event processing.

If the analysis and modification can be done statically, then they can be done in a preprocessing phase and will thus have limited impacts on the startup and shutdown of the GUI's execution.

If the analysis and modification must be done dynamically, then they must done at runtime and can thus have more significant impacts on the startup and shutdown of the GUI's execution. For the Startup Cost Inflation force, we want to keep these costs small.

Because of the requirement to support dynamic changes to the GUI (as discussed above for the Runtime Reconfiguration force), the solution presented in Chapter 3 did the analysis and modification completely at runtime. Some of the initial analysis and modification could have been done in a preprocessing step, but that would have required the mechanisms to be implemented in two completely different ways.

The Startup Cost Inflation force is in conflict with the Transitional Turbulence Reduction and Code Cluttering forces.

Operational Overhead Creep**:** *We want to avoid adding significant processing overhead during the application's normal operation.*

As we note in the discussion of the Startup Cost Inflation force, the modified .NET GUI has more overhead and more complex code for the event processing. The costs of supporting Runtime Reconfiguration also adds processing overhead and code complexity. For the Operational Overhead Creep force, we want to keep the execution costs of event processing small.

The Overhead Cost Creep force is in conflict with the Transitional Turbulence Reduction, Runtime Reconfiguration, and Code Cluttering forces.

Code Cluttering**:** *We want to avoid significantly complicating the application's design, implementation, testing, and use.*

For .NET GUIs, all the mechanisms we introduce in the discussion of the other forces above increase the complexity of the GUI program. This increases the cost to design, implement, test, and maintain the application.

For the Code Cluttering force, we want any modifications of the GUI programs to be simple and to be supported by libraries and/or tools. We also want the modifications to the event processing to work on top of the standard .NET event-processing mechanisms. The solution presented in Chapter 3 designed and implemented a library to handle most of the additional processing needed; it works on top of the standard .NET event handler.

The Code Cluttering force is in conflict with the Transitional Turbulence Reduction, Runtime Reconfiguration, Startup Cost Inflation, and Operational Overhead Creep forces.

The dynamic .NET GUI applications we consider exhibit all the above forces. Any Solution must balance the forces to solve the Problem acceptably.

5.7.4 The Solution

For applications that satisfy the Context, the DCRC pattern's Solution element describes how to augment the application's event-handling mechanisms to solve the Problem. The Solution involves three *augmentation processes* that are involved in building the needed software mechanisms:

1. *analyzing the application to identify how to add the mechanisms*
2. *developing the mechanisms*
3. *incorporating the mechanisms into the application's operation*

Here we focus on these processes and related aspects of the Solution element, which are shown below in italics. We consider the dynamic GUI application with respect to each item.

1. *In the* ***augmentation analysis process****, the solution's developer must:*

   (a) *Examine the hierarchical structure to identify how a program can iterate through the components (i.e., accessing each component exactly once).*

   A .NET-based GUI provides a hierarchical collection of its controls. For Web development using C#, this collection holds the Document Object Model (DOM). For desktop development using C#, the `Designer` class holds a collection of controls as objects of class `Control` class or one of its subclasses. .NET also supports the iteration construct `foreach` that enables programs to conveniently iterate through collections of objects such as the collection of controls. Thus, by using `foreach` on the collection of controls, a C# program can examine each control in the structure.

(b) *Examine the design and implementation of the components and the features of the implementation language to identify how a program can extract the dependency relationships among the components at runtime.*

In the .NET-based GUI case study, we develop the GUI using C#, which provides extensive reflection facilities. The `Type` class enables a C# program to examine any of its own objects and extract metadata about their features, including the names and type signatures of its methods and the names, types, and values of its attributes. By examining the objects for the controls, the program can determine the dependencies.

Suppose `A` and `B` are two controls in the GUI. If one of control `A`'s attributes holds a reference to control `B` or one of `A`'s methods has a formal parameter of type `B`, then control `B` depends on control `A`. (Our assumption for unmodified .NET GUIs is that the only communication between controls is through the event system.)

(c) *Examine the components and events to determine which component relationships to include in the dependency graph and which to exclude. To reduce transitional turbulence, the augmented application program can manipulate the components and relationships included but cannot manipulate those excluded.*

To augment a .NET GUI application, the developers must decide which controls to include as a part of the event-processing optimization and which to exclude. For example, the developers should exclude any control that they cannot modify—such as a .NET or system control or a control in a third-party library. Similarly, they will normally want to exclude any control that takes a long time to execute. The other controls—which we call "reactive" controls—should be included in the event-processing optimizations; these controls and their interrelationships must be included in the dependency graph.

The C#/.NET `interface` is a class-like construct. It declares a set of (abstract) public methods but does not define the concrete bodies of the methods. An `interface` defines a C#/.NET type that is visible to the reflection facilities. However, an `interface` itself cannot be instantiated to create an object. An `interface` can be implemented by any number of classes. A class that implements an `interface` must provide concrete definitions for all its methods. A class can extend just one parent class, but it can implement any number of `interface`s.

An `interface` is a good way to enable C#/.NET to distinguish reactive controls from other controls present in the GUI. We define an `interface` named `iReactive`. This `interface` defines a special event handler method for the reactive controls; a class that implements `iReactive` must define an appropriate method body. The class for any reactive control must implement `iReactive`. If an existing control needs to be made reactive, it can be "wrapped" by a class that implements `iReactive`. When the augmented application builds the dependency graph, it needs to include all controls that implement `iReactive` and exclude any that do not.

2. *In the **augmentation development process**, the solution developer must*:

   (a) *Design and implement a lightweight runtime mechanism that enables the program to differentiate between the components that are to be included in the dependency graph and those that are not.*

   As we discussed above, all control objects implement the `iReactive` interface. Using the reflection facilities, a C#/.NET program can easily check to see if an object implements this interface. If an object does, it must be included in the dependency graph; it it does not, it should be excluded from the dependency graph.

(b) *Design and implement a lightweight runtime mechanism that enables the program to detect whether the component architecture or the dependencies among the components have changed since the previous check (or since the beginning of operation).*

To determine whether the GUI changes, the augmented .NET GUI application stores a “snapshot” of the GUI’s structure at the beginning of an update cycle (as defined below). The snapshot consists of the dependency graph with each node holding a reference to its associated control object. At the end of the update cycle, the augmented application checks whether the GUI structure has changed since the beginning of the cycle. In particular, it must detect GUI changes that add new dependencies or modify existing dependencies or add new dependencies.

To determine if there are changes in the dependencies, the augmented application examines each reactive control in the GUI. If that control did not appear in the previous snapshot, then the dependency graph is no longer valid. (To compare two control objects, a C#/.NET program checks whether they have the same name and type.) If that control did appear in the previous snapshot and any of its dependencies have changed, then the dependency graph is no longer valid. To check whether a control’s dependencies have changed, the augmented application checks the control’s attributes and methods with the reflection package as discussed above. If any control appears in the previous snapshot, but not in the current GUI, then the dependency graph is no longer valid. If nothing has changed from the previous snapshot, then the dependency graph remains valid. If the dependency graph is no longer valid, then it needs to be rebuilt.

(c) *Design and implement lightweight mechanisms to construct the dependency graph initially and to reconstruct it when needed.*

As discussed above, a .NET GUI consists of a hierarchical collection of control objects. A C# program iterates through this collection. It examines each control using the C#/.NET reflection facilities. If the current control is one that should be included in the dependency graph, which means it implements the `iReactive interface`, then the augmented application inserts a new node into the dependency graph for the control. For each other control that the current control depends on, then the augmented applications inserts an appropriate edge into the dependency graph going from the other control to the current control signifying that the other control depends on current control.

The process is essentially the same for the initial construction of the dependency graph and for its reconstruction because of a change in the GUI's structure. The reconstruction is a slightly different in that it only iterates through the controls referenced by the previous dependency graph (not all the controls in the GUI).

3. *In the **augmentation incorporation process**, the solution developer must modify the application's operation in the following ways*:

   (a) *The application must construct the dependency graph at or before startup.*

   In C#/.NET GUI, we implement a GUI as an instance of the `Form` class or one of its subclasses. This class has an event-handler method `form_start(). Form` executes this method during its instantiation—after it instantiates all its controls and before it renders the form to the user. This is where we incorporate the construction of the dependency graph into the event-handling system.

   To make the GUI reactive, we declare the interface `iUpdatable` to designate a reactive form. This interface requires that the `Form` subclass implementing it overrides and defines the `form_start()` method. Using the techniques discussed above, the method must analyze the GUI and construct the initial dependency graph as an object in the `Form` subclass.

   (b) *When some component C included in the dependency graph signals an event E, the* *application must intercept E and directly call the procedures associated with event E on all listening components as recorded in the dependency graph. Then it must recursively apply the process to all events signalled by the listening components. It continues this as long as there are dependencies indicated in the graph (which cannot have cycles). This process dynamically coalesces the processing of chains of events into what is processed as one "large-grained" event.*

As discussed above, we use the interface `iReactive` to designate reactive controls. We declare this interface to include the event-handler method `reactiveUpdate`. Any `Control` subclass that implements the interface must provide appropriate behavior (which can be the code that usually would go on the built-in event-handler method `Update()`). The reactive control class must also override the built-in `Update()` method so that it calls the `reactiveUpdate` method. The augmented GUI application thus executes this method instead of the control's standard event handler.

If a reactive control responds to an external (e.g., user interaction) event, then the built-in event handler method `Update()` must detect the external event and redirect its handling to the augmented event-handling method `reactiveUpdate`. The augmented event-handling mechanism constructs a sequence of the control updates triggered by the initial external event based on the constraints in the dependency graph and then invokes the sequence of `reactiveUpdate` methods of each control explicitly. This process thus propagates the effects of one external event throughout the GUI. From the standpoint of the built-in event-handling system, this whole sequence of updates executes as one "large-grained" event. We call the period from the receipt of the external event until completion of the sequence one *update cycle*.

(c) *After the processing of each "large-grained" event in the previous step, the application must check whether the application's component architecture has changed* *(e.g., the addition, modification, or deletion of any component in the hierarchical structure) or the dependencies among components have changed. If so, then it must update the dependency graph appropriately to reflect the new component architecture.*

At the end of an update cycle, the augmented application checks whether the GUI structure has changed since the beginning of the cycle (as previously described). If it has changed, it must rebuild the dependency graph (as previously described). As mentioned in the previous step, the built-in event handler of the control which detected the event calls the augmented event-handling mechanism (that is the first and only thing the built-in event handler must do). The routine to update the dependency graph according with the modifications in the GUI is executed as part of the augmented event-handling, right after inferring the execution order from the dependency graph and call the `reactiveUpdate()` methods of each control sequentially as a single large-grained event.

5.7.5 The Consequences

The DCRC pattern's Consequences element lists the consequences of applying the pattern—both benefits and liabilities (shown in italics below). The Consequences element maps each force to a benefit and/or a liability. We consider the dynamic GUI application with respect to each benefit and liability.

BENEFITS

Transitional Turbulence Reduction**:** *A solution coalesces sets of dependent internal events into "large-grained" events such that the handling of a large-grained event causes the same overall state change as the corresponding set. This can decrease latency and increase accuracy (i.e., decrease the number of errors).*

The .NET GUI application optimizes the event processing by combining the state changes associated with sequences of related events into larger units. By doing so, it seeks to mitigate the effects of transitional turbulence.

***Runtime Reconfiguration*:** *A solution dynamically adapts to changes in an application's component architecture during normal operation.*

The .NET GUI application adapts to runtime changes in the structure of the GUI. It seeks to preserve the benefits of the event-processing optimizations that mitigate the effects of transitional turbulence.

***Code Cluttering*:** *An application can be readily adapted to use the mechanisms implementing the solution.*

The .NET GUI application augments the standard .NET event processing system but does not replace it. The solution presented in Chapter 3 uses a library and the reflection facilities to optimize event processing.

LIABILITIES

***Runtime Reconfiguration*:** *Changes to an application's component architecture during normal operation can increase latency and decrease accuracy.*

The .NET GUI application adapts to runtime changes in the structure of the GUI. These changes may, in themselves, degrade the event-processing performance. In addition, they may degrade the effectiveness of optimizations that are based on the GUI's structure. The application may need to undertake a costly reanalysis of the GUI structure to incorporate different optimizations. A solution should minimize the cost of adapting to runtime structural changes.

***Startup Cost Inflation*:** *An implementation of a solution often causes additional processing overhead at startup and shutdown of the application.*

As we note in the discussion of the Startup Cost Inflation force, any Solution to the transitional turbulence reduction Problem for .NET GUIs likely requires that the GUI be analyzed and modified before normal operation begins. This can be a costly operation, particularly if performed at runtime. A solution should minimize this startup overhead.

***Operational Overhead Creep*:** *An implementation of a solution often causes additional processing overhead during normal operation, especially when the component architecture changes.*

As we note in the discussion of the Operational Overhead Creep force, any Solution to the transitional turbulence reduction Problem for .NET GUIs likely adds overhead to the normal processing of events. This overhead may be especially significant when the solution must adapt to changes in the GUI's structure. A solution should minimize this operational overhead.

***Code Cluttering*:** *An application must be adapted to use the mechanisms implementing the solution. Modifying the application often complicates its design, implementation, testing, or use.*

As we note in the discussion of the Code Cluttering force, any Solution to the transitional turbulence reduction Problem for .NET GUIs likely makes the GUI programs more complex and, hence, more costly to design, implement, test, and maintain. The added software mechanisms should be kept lightweight.

5.7.6 Summary

In this section, we have shown that the Dynamically Coalescing Reactive Chains pattern is applicable to .NET-based GUI applications and that the pattern can guide the development of a solution to the transitional turbulence reduction problem. Of course, as we discuss in Section 5.6, additional research can help us refine the pattern description.

5.8 Conclusion

Transitional turbulence is a period of chaotic or unreliable variation in the state of a software system that results from changes in the system's interconnected components. During these periods of instability, an external observer of the system's state may "see" erroneous results. This is a problem that can affect visual user interfaces such as those in virtual and augmented reality applications and in desktop or Web GUIs.

Guided by the development of the applications in Chapters 3 and 4, we formulated the Dynamically Coalescing Reactive Chains design pattern to answer the research question:

> *How can we codify the transitional turbulence mitigation approach taken in Chapters 3 and 4 as a general, technology-independent design pattern?*

We seek to write this pattern to document our approach and enable others to apply it in their own work.

To answer the research question, we pursued a research approach with three phases:

1. We explored the conceptual background on design patterns and their development and adopted the pattern writer's path documented by Wellhausen and Fiesser (2011) to discover the elements of the new pattern in a stepwise fashion. We applied the pattern-writing patterns of Meszaros and Doble (1998) and Harrison (1999, 2006) to refine the new pattern.

2. We analyzed the applications presented in Chapters 3 and 4 to identify and write suitable descriptions for the new pattern's Context, Problem, Forces, and Consequences elements. We then documented a technology-independent Solution that balances the Forces to solve the Problem and establish the desired Consequences. We selected the Name Dynamically Coalescing Reactive Chains (DCRC) for the new pattern.

3. To demonstrate the effectiveness of the DCRC pattern for mitigating transitional turbulence, we applied the pattern to a .NET-based dynamic GUI application (similar to that in the Chapter 3 case study). We showed that the pattern is applicable to this application and can guide us to derive a realistic solution (similar to the one in Chapter 3).

This chapter both presents the design pattern and records the systematic process we used to write it. It lays the foundation for further research on transitional turbulence and related software architecture issues. We found that this work required that we think deeply about the applications and our process for developing them. As a result, we were able to refine the original applications themselves.

Overall, we are pleased with the results of the research reported in Chapters 3, 4, and 5. The first two chapters reported how we developed applications on different technologies that successfully mitigated transitional turbulence. The third chapter captured this knowledge in a more technology-independent way—a way that can guide others to apply the "same" approach and be successful in mitigating transitional turbulence in other implicit invocation applications.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Introduction

In this chapter, we summarize the accomplishments of the research reported in this dissertation and explore possible future research.

6.2 Conclusion

In Chapter 1, we define the following overall research question:

Can we mitigate the transitional turbulence in an implicit invocation-based systems by applying a technology-independent formal description such as a design pattern created by highlighting common aspects of similar applications?

To answer the overall research question, we organized our research in three parts:

- In Chapter 3 and Marum et al. (2020a), we investigated the transitional turbulence problem for dynamic, .NET-based GUI applications for the desktop and Web. We designed, implemented, and tested an augmented application that successfully mitigated transitional turbulence in this context. It does so by optimizing the .NET event-handling mechanisms. Our experimental tests indicated that our augmented applications performed better than the unmodified .NET applications and similar applications implemented using the reactive libraries Rx.NET (Malawski, 2016) and Sodium (Blackheath and Jones, 2016).

  This research result is novel, significant, and potentially useful by itself. It yielded a publication (Marum et al., 2020a). It also contributed to answering the overall research question.

- In Chapter 4 and Marum et al. (2019, 2020c), we investigated the transitional turbulence problem for virtual and augmented reality applications using the Unity3D game engine (Unity Technologies, 2019). We chose this environment because Unity3D is a low-cost platform commonly used in virtual and augmented reality research. We designed, implemented, and tested an augmented application that successfully mitigated transitional turbulence in this context. It does so by optimizing the Unity3D event-handling mechanisms. Our experimental tests indicated that our augmented applications performed better than the unmodified Unity3D applications and similar applications implemented using the reactive library UniRx (Kawai, 2014).

  As above, this research result is novel, significant, and potentially useful by itself. It yielded two publications (Marum et al., 2019, 2020c). It also contributed to answering the overall research question.

- Encouraged by the success of the previous two parts of this research, in Chapter 5, we sought to unify and generalize the approach. We focused on a type of system characterized by the implicit invocation architectural pattern (Shaw, 1996), which the applications above exhibited.

  We decided to document the general approach using a design pattern (Buschmann et al., 1996), a well-accepted technique in the software architecture community. We organized our work using a writer's path adapted from the helpful Wellhausen and Fiesser (2011) tutorial and applied patterns from the published pattern languages for pattern writing (Harrison, 1999, 2006; Meszaros and Doble, 1998). We recorded our steps along the path as well as presenting the new pattern Dynamically Coalescing Reactive Chains (DCRC). To demonstrate the effectiveness of the pattern for mitigating transitional turbulence, we applied the pattern to a .NET-based dynamic GUI application (similar to that in the Chapter 3 case study). We showed that the pattern is applicable to this application and can guide us to derive a realistic solution (similar to the one in Chapter 3).

This research result is also novel and significant. The systematic recording of our development process is also interesting and potentially useful for others developing design patterns. We are confident that future work will show that the pattern is useful for a wide range of problems. Combined with the two previous results, this part shows that the answer to the research question is “Yes”.

Overall, we are pleased with the results of the research implementations reported in Chapters 3 and 4 and the design pattern description in Chapter 5. The first two chapters reported how we developed applications on different technologies that successfully mitigated transitional turbulence. In both, we reported improvements in performance and accuracy. The third chapter captured this knowledge in a more general, technology-independent manner using a design pattern. The pattern can guide others to apply the same approach to different but related applications and technologies. We believe that the result can be similarly successful in mitigating transitional turbulence in other implicit invocation applications. We showed that the pattern is applicable to an implicit invocation-based application and that it can guide us to derive a realistic solution to a given problem.

6.3 Future Work

In this dissertation research project, we have investigated the problem of transitional turbulence and related issues affecting user interface applications. We devised a novel approach to mitigate transitional turbulence and incorporated it into two case studies. We also generalized the approach and documented it as a design pattern. However, like most research efforts, new questions arise from the process of answering the old ones. In this section, we identify several issues that we, or others, can explore in future research.

Although we have published papers on the research reported in Chapter 3 (Marum et al., 2020a) and Chapter 4 (Marum et al., 2019, 2020c), we have not yet published the results of the research reported in Chapter 5. In the future months, we expect to write papers based on the design pattern research and submit them for publication. The possible topics and venues include:

- the DCRC pattern itself (e.g., in a journal or at a premier patterns conference such as *Pattern Languages of Programming* (PLoP)).
- reflections on the process of developing the DCRC pattern (e.g., in a journal such as *IET Software* or *The Art, Science, and Engineering of Programming* (<*Programming*>))

We note that PLoP, the premier venue in the software patterns community, and similar conferences around the world (e.g., EuroPLoP, AsianPLop) have unusual formats. As we discussed in Section 5.6, they are based on the practice of *shepherding* (Harrison, 1999, 2006). Accepted pattern submissions are assigned to an experienced pattern writer—called a shepherd—who gives feedback to the pattern author and works with the author to refine the pattern description iteratively. Once sufficiently developed, the pattern description is published by the conference.

In Chapter 5 we defined the Dynamically Coalescing Reactive Chains (DCRC) design pattern. We wrote the pattern to be relatively independent of any programming platform or specific application. However, to keep the dissertation research within a reasonable scope, we limited our practical development work to .NET-based GUIs and Unity3D VR/AR applications. The Unity3D game engine itself runs on top of .NET, so the range of base technologies and kinds of applications explored so far are small. Thus, in the future, other related applications and platforms should be explored. This should enable the pattern to be refined. It may also enable us to explore the limitations of our approach; that is, where does the approach fail to mitigate transitional turbulence?

We have identified the following related application areas to explore:

- interactive data visualization applications
- game applications involving nonplayer characters and robots

- task scheduling applications

We have also identified the following related technologies to explore:

- the Java language with a user interface package such as JavaFX (Chin et al., 2019)
- the Python 3 language with a user interface package such as PyQt6 (Riverbank Computing, 2021)
- the CryEngine Entity-Component-System (ECS) game engine (Berns et al., 2019; CryTek, 2021; Raffaillac and Huot, 2019) using the Lua and C++ languages.

Depending on the outcomes of the research above, we can also revisit the design pattern itself. As we noted in Section 5.6, we can seek to evolve the DCRC pattern into a whole pattern language (Buschmann et al., 2007), define more specific idioms for various technologies or application areas, or develop a formal model for the architectures and processes associated with the DCRC pattern.

BIBLIOGRAPHY

BIBLIOGRAPHY

Albahari, J., and E. Johansen (2020), *C# 8.0 in a Nutshell: The Definitive Guide*, O'Reilly Media, USA.

Azero, P. (2013), Dependency graphs and their application to software engineering, *Jalasoft Techzone 2013*, retrieved October 19, 2020.

Bainomugisha, E., A. L. Carreton, T. van Cutsem, S. Mostinckx, and W. de Meuter (2013), A survey on reactive programming, *ACM Computing Surveys*, 45(4), doi:10.1145/2501654.2501666.

Baron, D. (2019), *Hands-On Game Development Patterns with Unity 2019*, Packt Publishing, Birmingham, UK.

Berns, C., G. Chin, J. Savitz, J. Kiesling, and F. Martin (2019), Myr: A web-based platform for teaching coding using vr, in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, p. 77–83, Association for Computing Machinery, New York, NY, USA, doi:10.1145/3287324.3287482.

Bishop, J., 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, pp. 373–377, ACM, Norfolk, VA, USA.

Blackheath, S., and A. Jones (2016), *Functional Reactive Programming*, Manning, Shelter Island, NY.

Blom, K., and S. Beckhaus (2008), On the creation of dynamic, interactive virtual environments, in *Proceedings of the IEEE VR 2008 Workshop (SEARIS): Software Engineering and Architectures for Interactive Systems*, Alexandria, VA, USA, retrieved January 14, 2020.

Bregu, E., 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, pp. 207–219, ACM, Singapore.

Britton, K. H., R. A. Parker, and D. L. Parnas (1981), A procedure for designing abstract interfaces for device interface modules, in *Proceedings of the 5th International Conference on Software Engineering*, pp. 195–204, IEEE, San Diego, CA, USA.

Brown, E. (2006), *Windows Forms in Action*, Manning Publication, Greenwich, CT.

Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996), *Pattern-Oriented Software Architecture: A System of Patterns*, vol. 1, Wiley, Chichester, UK.

Buschmann, F., K. Henney, and D. Schmidt (2007), *Pattern-Oriented Software Architecture, On Patterns and Pattern Languages*, vol. 5, Wiley, Chichester, UK.

Chandy, K., and J. Misra (1988), *Parallel Program Design: A Foundation*, Addison Wesley, Boston, MA, USA.

Chin, S., J. Vos, and J. Weaver (2019), *The Definitive Guide to Modern Java Clients with JavaFX*, Apress, New York, NY, USA.

Chupin, G., 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.

Cooper, G. H., and S. Krishnamurthi (2006), Embedding dynamic dataflow in a call-by-value language, in *Programming Languages and Systems, 15th European Symposium on Programming*, edited by P. Sestoff, ESOP 2006, pp. 294–308, Springer, Vienna, Austria, doi:10.1007/11693024_20.

Cowan, B., and B. Kapralos (2014), A survey of frameworks and game engines for serious game development, in *2014 IEEE 14th International Conference on Advanced Learning Technologies*, ICALT, pp. 662–664, Athens, Greece.

CryTek (2021), CryEngine game engine, retrieved June 22, 2021.

Czaplicki, E. (2012), Elm: Concurrent FRP for functional GUIs, *Senior thesis*, Harvard University, Cambridge, MA, USA.

Czaplicki, E., 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, pp. 411–422, ACM, Seattle, WA, USA.

Damyanov, I., and N. Holmes (2004), Metadata driven code generation using .NET framework, in *Proceedings of the 5th International Conference on Computer Systems and Technologies*, CompSysTech '04, p. 1–6, Association for Computing Machinery, New York, NY, USA, doi:10.1145/1050330.1050387.

Drechsler, J., G. Salvaneschi, R. Mogk, and M. Mezini (2014), Distributed REScala: An update algorithm for distributed reactive programming, in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, p. 361–376, ACM, Portland, Oregon, USA, doi:10.1145/2660193.2660240.

Elliott, C. (2009), Push-pull functional reactive programming, in *Proceedings of the 2nd SIGPLAN Symposium on Haskell*, Haskell '09, pp. 25–36, ACM, Edinburgh, Scotland.

Eugster, P., P. Felber, R. Guerraoui, and A. Kermarrec (2003), The many faces of publish/subscribe, *ACM Computing Surveys*, 35(2), 114–131.

Féray, V., et al. (2018), Weighted dependency graphs, *Electronic Journal of Probability*, 23.

Foust, G., J. Järvi, and S. Parent (2015), Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems, *SIGPLAN Notices*, 51(3), 121–130.

Franciscus, N., X. Ren, and B. Stantic (2019), Dependency graph for short text extraction and summarization, *Journal of Information and Telecommunication*, 3(4), 413–429, doi: 10.1080/24751839.2019.1598771.

Furness, T., and W. Barfield (1995), *Virtual Environments and Advanced Interface Design*, Oxford Publications, Oxford, UK.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Boston, MA, USA.

Garlan, D., and S. Khersonsky (2000), Model checking implicit-invocation systems, in *Tenth International Workshop on Software Specification and Design*, IWSSD, pp. 23–30, IEEE, San Diego, CA, USA, doi:10.1109/IWSSD.2000.891123.

Garlan, D., and D. Notkin (1991), Formalizing design spaces: Implicit invocation mechanisms, in *Proceedings of the International Symposium of VDM Europe*, pp. 31–44, Springer, Berlin.

Garlan, D., and M. Shaw (1993), An introduction to software architecture, in *Advances in Software Engineering and Knowledge Engineering*, pp. 1–39, World Scientific, Singapore.

Garlan, D., S. Jha, D. Notkin, and J. Dingel (1998), Reasoning about implicit invocation, in *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '98/FSE-6, pp. 209—221, ACM, Lake Buena Vista, FL, USA.

Garlan, D., S. Khersonsky, and J. S. Kim (2003), Model checking publish-subscribe systems, in *International SPIN Workshop on Model Checking of Software*, pp. 166–180, Springer, Portland, OR, USA.

Goldberg, A., and D. Robson (1983), *Smalltalk-80 The Language and Its Implementation*, Addison Wesley Longman, Boston, MA, USA.

Greene, J., and A. Stellman (2013), *Head First in C#*, O'Reilly Media, Sebastopol, CA.

Gregory, J. (2018), *Game engine architecture*, Taylor and Francis Ltd.

Hamilton, J. (2003), Language integration in the common language runtime, *SIGPLAN Not.*, 38(2), 19–28, doi:10.1145/772970.772973.

Harrison, N. (1999), The language of shepherding: A pattern language for shepherds and sheep, in *Pattern Languages of Program Design*, vol. 4, edited by N. Harrison, B. Foote, and H. Rohnert, pp. 507–530, Addison Wesley, Boston, MA, USA.

Harrison, N. (2006), Advanced pattern writing: Patterns for experienced writers, in *Pattern Languages of Program Design*, vol. 5, edited by D. Manolescu, M. Voelter, and J. Noble, chap. 16, pp. 433–451, Addison Wesley, Boston, MA, USA.

Hocking, J. (2015), *Unity in Action*, Manning, Shelter Island, NY.

Jankovic, L. (2000), Games development in VRML, *Virtual Reality*, 5(4), 195–203.

Jones, J., E. Luckett, T. Key, and N. Newsome (2019), Latency measurement in head-mounted virtual environments, in *Proceedings of IEEE SouthEastCon 2019*, Huntsville, AL, USA.

Kawai, Y. (2014), UniRx: Reactive extensions for Unity3D, retrieved Feb. 9, 2020.

Krishnaswami, N. (2012), Semantics for graphical user interfaces, in *Proceedings of the 8th SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pp. 51–52, ACM, Philadelphia, PA, USA.

Krouse, S. (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.

Lange, P., R. Weller, and G. Zachmann (2016), Wait-free hash maps in the Entity-Component-System pattern for realtime interactive systems, in *2016 IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pp. 1–8, IEEE.

LaValle, S. (2017), *Virtual Reality*, Cambridge University Press, Cambridge, UK.

Lehmann, S., T. Felgentreff, J. Lincke, P. Rein, and R. Hirschfeld (2016), Reactive object queries: Consistent views in object-oriented languages, in *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pp. 23–28, ACM, M&#225;laga, Spain.

Li, Y., T. Tan, and J. Xue (2019), Understanding and analyzing Java reflection, *ACM Trans. Softw. Eng. Methodol.*, 28(2), doi:10.1145/3295739.

Lorenz, E. N. (1963), Deterministic nonperiodic flow, *Journal of the Atmospheric Sciences*, 20(2), 130–141, doi:10.1175/1520-0469(1963)020⟨0130:DNF⟩2.0.CO;2.

Malawski, K. (2016), *Why Reactive?*, O'Reilly Media, Sebastopol, CA.

Manna, Z., and A. Pneulli (1992), *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer, Berlin.

Marum, J. (2017), Survey of functional reactive programming approaches to virtual environment applications on the .Net platform, *Masters project report*, University of Mississippi, retrieved January 14, 2020.

Marum, J., J. Jones, and H. Cunningham (2016), Functional reactive augmented reality: Proof of concept using an extended augmented desktop with swipe interaction, in *Proceedings of the 26th International Conference on Artificial Reality and Telexistence and the 21st Eurographics Symposium on Virtual Environments: Posters and Demos*, ICAT-EGVE '16, pp. 13–14, Eurographics Association, Little Rock, AR, USA.

Marum, J., J. Jones, and H. Cunningham (2019), Towards a reactive game engine, in *Proceedings of the 50th IEEE SouthEastCon*, IEEE, Huntsville, AL, USA.

Marum, J., H. Cunningham, and J. Jones (2020a), Unified library for dependency graph reactivity on web and desktop user interfaces, in *Proceedings of the ACM Southeast Conference (ACMSE 2020)*, ACM, Tampa, FL, USA.

Marum, J., H. Cunningham, and J. Jones (2020b), Unified library for dependency graph reactivity on web and desktop user interfaces: Addendum, *Tech. rep.*, University of Mississippi, Department of Computer and Information Science, Oxford, MS, USA, retrieved July 29, 2020.

Marum, J., J. Jones, and H. Cunningham (2020c), 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.

Meszaros, G., and J. Doble (1998), A pattern language for pattern writing, in *Pattern Languages of Program Design 3*, pp. 529–574, Addison Wesley, Boston, MA, USA.

Meyerovich, L. A., A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi (2009), Flapjax: A programming language for Ajax applications, in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pp. 1–20, ACM, Orlando, Florida, USA, doi:10.1145/1640089.1640091.

Microsoft (2019), *.Net Framework Developer Documentation: StopWatch Class*, retrieved Feb. 26, 2020.

Microsoft (2020a), C# documentation, retrieved October 12, 2020.

Microsoft (2020b), .net documentation, retrieved October 12, 2020.

Nagel, C. (2018), *Professional C# 7 and .Net Core 2.0*, Wrox, USA.

OpenJFX Project (2020), JavaFX, retrieved July 22, 2020.

Parnas, D. L. (1972), On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, 15(12), 1053–1058, doi:10.1145/361598.361623.

Parnas, D. L., and P. C. Clements (1986), A rational design process: How and why to fake it, *IEEE Transactions on Software Engineering*, SE-12(2), 251–257, doi:10.1109/TSE.1986.6312940.

Pontes, F., R. Gheyi, S. Souto, A. Garcia, and M. Ribeiro (2019), Java Reflection API: Revealing the dark side of the mirror, in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, doi:10.1145/3338906.3338946.

Price, M. (2019), *C# 8.0 and .Net Core 3.0*, Packt Publishing, UK.

Qian, K., X. Fu, L. Tao, C. Xu, and J. Diaz-Herrera (2010), Implicit asynchronous communication software architecture, in *Software Architecture and Design Illuminated*, chap. 8, pp. 177–198, Jones & Bartlett Learning, Burlington, MA, USA.

Raffaillac, T., and S. Huot (2019), Polyphony: Programming interfaces and interactions with the entity-component-system model, *Proc. ACM Hum.-Comput. Interact.*, 3(EICS), doi:10.1145/3331150.

reactivex.io (2020), ReactiveX: An API for asynchronous programming with observable streams, retrieved July 13, 2020.

Reynders, B., 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, pp. 23:1–23:6, ACM, Brussels, Belgium.

Riverbank Computing (2021), PyQt6 reference guide, retrieved June 22, 2021.

Rodriguez, E., and W. Swierstra (2015), Datatype generic programming in f#, in *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, p. 23–32, Association for Computing Machinery, New York, NY, USA, doi:10.1145/2808098.2808101.

Salvaneschi, G., 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, pp. 564–575, ACM, Hong Kong, China.

Salvaneschi, G., A. Margara, and G. Tamburrelli (2015), Reactive programming: A walkthrough, in *Proceedings of the 37th International Conference on Software Engineering: Volume 2*, ICSE '15, pp. 953–954, IEEE, Florence, Italy.

Seligmann, R. (2018), Creating a mobile VR interactive tour guide, *Bachelor's thesis*, Haaga-Helia University of Applied Sciences, Helsinki, Finland, retrieved Feb. 9, 2020.

Senthilvel, O., G.; Khan, and H. Qureshi (2017), *Enterprise Application Architecture with .NET Core*, Packt Publishing, USA.

Shaw, M. (1996), Some patterns for software architectures, in *Pattern Languages of Program Design*, vol. 2, edited by J. Vlissides, J. Coplien, and N. Kerth, pp. 255–269, Addison Wesley, Boston, MA, USA.

Shaw, M., and D. Garlan (1996), *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Englewood Cliffs, NJ, USA.

Shaw, M., R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik (1995), Abstractions for software architecture and tools to support them, *IEEE Transactions on Software Engineering*, 21(4), 314–335.

Sherrod, A. (2006), *Ultimate 3D Game Engine Design & Architecture*, Charles River Media, Inc., Rockland, MA.

Stefan, L., S. Hermon, and M. Faka (2018), Prototyping 3D virtual learning environments with X3D-based content and visualization tools, *BRAIN. Broad Research in Artificial Intelligence and Neuroscience.*

The Hillside Group (2020), Website, retrieved July 22, 2020.

Unity Technologies (2019), Unity user manual, retrieved January 14, 2020.

Van den Vonder, S., F. Myter, J. De Koster, and W. De Meuter (2017), Enriching the internet by acting and reacting, in *Companion to the First International Conference on the Art, Science and Engineering of Programming*, Programming '17, pp. 24:1–24:6, ACM, Brussels, Belgium.

Wellhausen, T., and A. Fiesser (2011), How to write a pattern? a rough guide for first-time pattern authors, in *Proceedings of the 16th European Conference on Pattern Languages of Programs*, EuroPLOP '11, ACM, Irsee, Germany.

Westberg, J. (2017), UniRx and Unity3D 5: Working with C# and object-oriented reactive programming, *Bachelor's thesis*, Uppsala University, Uppsala, Sweden, retrieved November 23, 2018.

Wiebusch, D., and M. Latoschik (2014), A uniform semantic-based access model for realtime interactive systems, in *2014 IEEE 7th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pp. 51–58, IEEE.

APPENDICES

APPENDIX A

FINAL VERSION OF DCRC DESIGN PATTERN

A.1 Pattern Name

Dynamically Coalescing Reactive Chains

A.2 Context

We have an application constructed according to the Implicit Invocation architectural pattern (Shaw, 1996), assuming non-deterministic but fair handling of events. Furthermore, it has the following characteristics:

- The application's component architecture may change during normal operation.
- The application organizes the components into a hierarchical structure. This structure may change dynamically during the application's normal operation as a result of external stimuli or the actions of components.
- The application presents some aspects of its state that can be observed from outside the system periodically. The timing of this presentation is not under the control of the application.

- Because of the asynchronous nature of the application's operation, the externally observable presentation may exhibit periods of *transitional turbulence*. By transitional turbulence, we mean a period of chaotic or unreliable variation in the application's state that can result from one or more changes to the application's interconnected components. It can result in an externally observable state that does not accurately represent the expected result.

- Each component is an information-hiding module with a well-defined interface. The only way to change or access its state explicitly is by calling one of its accessor or mutator procedures (e.g., properties in some object-oriented languages).

- The application supports *reflection* capabilities. That is, application-level code can examine the application's features (such as its components, events, event handlers, and hierarchical structure) at runtime and extract metadata (such as names, types, and the type signatures of the procedures in component interfaces).

Figure A.1 depicts the implicit invocation architecture, which describes the basic nature of applications to which this pattern can be applied.

A.3 Problem

We want to eliminate or reduce the length of the periods of transitional turbulence during which the external presentation does not accurately reflect the state of the application. We need to do this without sacrificing performance. The goal is to better satisfy observers' expectations by increasing the accuracy of the external presentation.

A.4 Forces

Runtime Reconfiguration: We want to adapt to changes in an application's component architecture during normal operation.

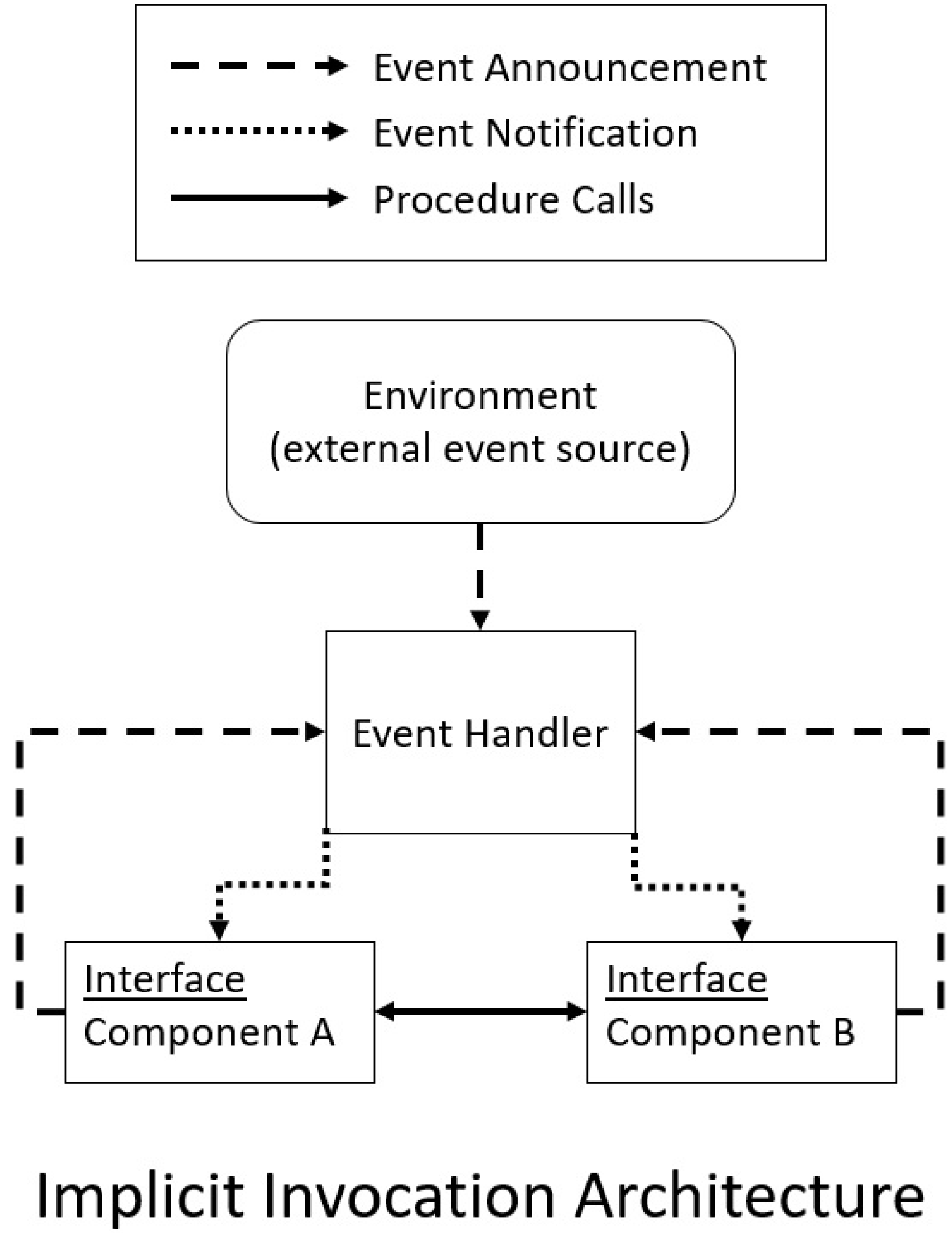


Figure A.1. Abstract definition of a implicit invocation-based system.

Transitional Turbulence Reduction: We want to decrease the transitional turbulence in the application's execution to better satisfy the observers' expectations.

Startup Cost Inflation: We want to avoid adding significant startup or shutdown costs.

Operational Overhead Creep: We want to avoid adding significant processing overhead during the application's normal operation.

Code Cluttering: We want to avoid significantly complicating the application's design, implementation, testing, and use.

A.5 Solution

A.5.1 Summary

A solution encodes the complex relationships among the application's components in a dependency graph and then uses the graph to order the updates of the components without violating the dependency constraints. The goal is to reorder the updates of the components so that the new order reduces transitional turbulence without degrading the performance of the system.

A.5.2 Definitions

What do we mean by a "dependency graph" in this Context?

- If the execution of a component C can directly affect a subsequent execution of a component D in any way, then D *depends on* C.

  For example, C might trigger an event for which D listens, change aspects of its state that D accesses, directly call one of D's mutator procedures, or create or modify component D.

- A *dependency graph* is a digraph formed by placing the components at the nodes and adding an edge from one node to another if the corresponding components have a

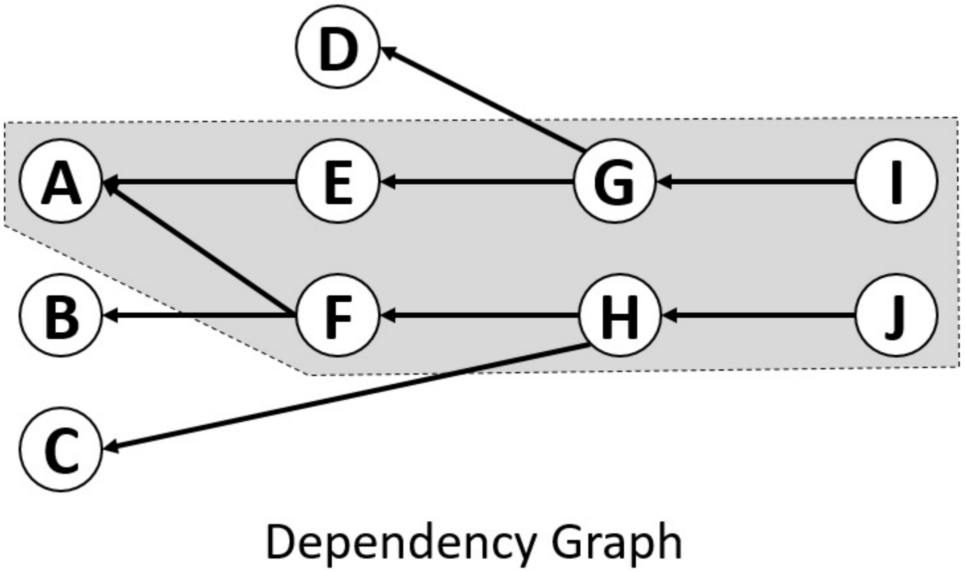


Figure A.2. The dependency graph.

depends-on relationship. Figure A.2 shows a dependency graph and the evaluation order as it is inferred from the dependency relationships between the nodes.

To apply the DCRC pattern, we are primarily interested in recording the dependencies related to the implicit invocations—between components that listen for an event and those that trigger the event. Of course, being able to record other kinds of dependencies may also be helpful. Figure A.2 shows a simple dependency graph, the grey area depicts a theoretical dependency relation between multiple objects.

A.5.3 Augmenting the Application

To apply the DCRC pattern to an application that satisfies the Context, we augment the application with appropriate *software mechanisms*.

These mechanisms may include some combination of libraries, frameworks, tools, and design and programming techniques. The various mechanisms should be *lightweight*. That is, they should execute efficiently and should not require extensive modifications of the existing

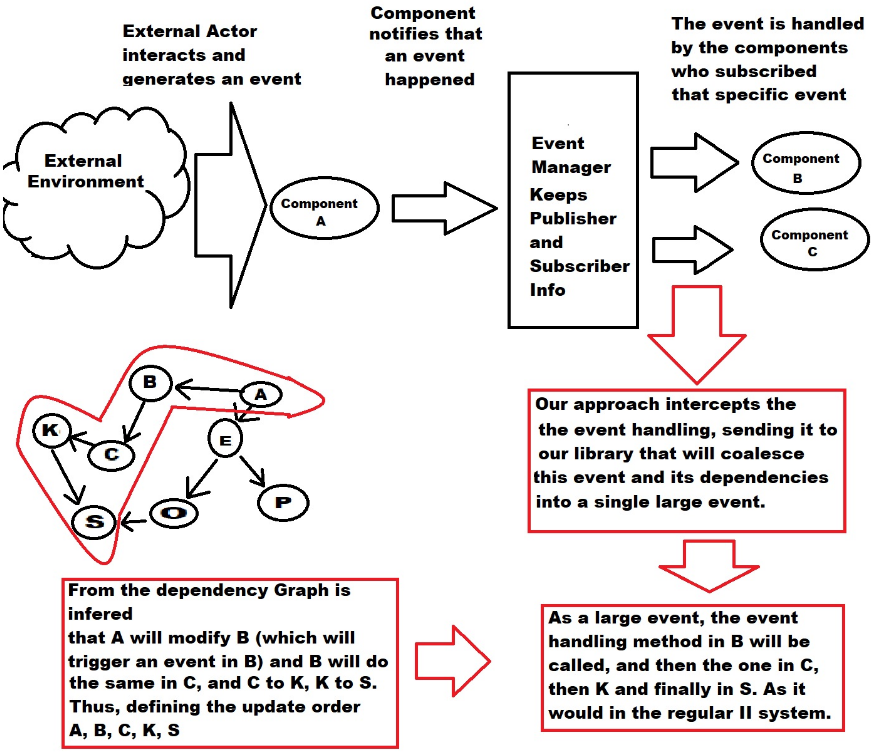

External Actor interacts and generates an event
Component notifies that an event happened
The event is handled by the components who subscribed that specific event
External Environment
Component A
Event Manager Keeps Publisher and Subscriber Info
Component B
Component C
B
A
K
C
E
P
S
O
Our approach intercepts the the event handling, sending it to our library that will coalesce this event and its dependencies into a single large event.
From the dependency Graph is infered that A will modify B (which will trigger an event in B) and B will do the same in C, and C to K, K to S. Thus, defining the update order A, B, C, K, S
As a large event, the event handling method in B will be called, and then the one in C, then K and finally in S. As it would in the regular II system.

Figure A.3. The characteristics of the abstracted augmentation.

application. The "software mechanisms" needed and the meaning of "lightweight" depend upon the application's specific implementation technologies and performance requirements. For applications that satisfy the Context, Figure A.3 shows how, by using this pattern, we can augment the application's event-handling mechanisms to solve the Problem.

The solution involves three processes: *analyzing* the application to identify how to add the mechanisms, *developing* the mechanisms, and *incorporating* the mechanisms into the application's operation.

1. In the **augmentation analysis process**, the solution's developer must:

   (a) Examine the hierarchical structure to identify how a program can iterate through the components (i.e., accessing each component exactly once).

   (b) Examine the design and implementation of the components and the features of the implementation language to identify how a program can extract the dependency relationships among the components at runtime.

This may involve use of the components' existing features or the implementation language's reflection capabilities. If sufficient capabilities do not exist, we can design lightweight modifications that implement sufficient application-specific capabilities.

(c) Examine the components and events to determine which component relationships to include in the dependency graph and which to exclude. To reduce transitional turbulence, the augmented application program can manipulate the components and relationships included but cannot manipulate those excluded.

Generally speaking, we include the component relationships arising from the application's custom code (which we can modify if needed) and exclude those in the supporting framework (which we cannot modify). We may also want to exclude any component relationship if that relationship represents an expensive computation or arbitrary delay.

2. In the **augmentation development process**, the solution developer must:

(a) Design and implement a lightweight runtime mechanism that enables the program to differentiate between the components that are to be included in the dependency graph and those that are not.

This may involve features already present in the application (e.g., types, value of some property, metadata) or may involve modifying the application to add appropriate features. For example, in an object-oriented system in which the components are objects, we could modify the included components to implement a "marker interface" that can can be checked by reflection.

(b) Design and implement a lightweight runtime mechanism that enables the program to detect whether the component architecture or the dependencies among the

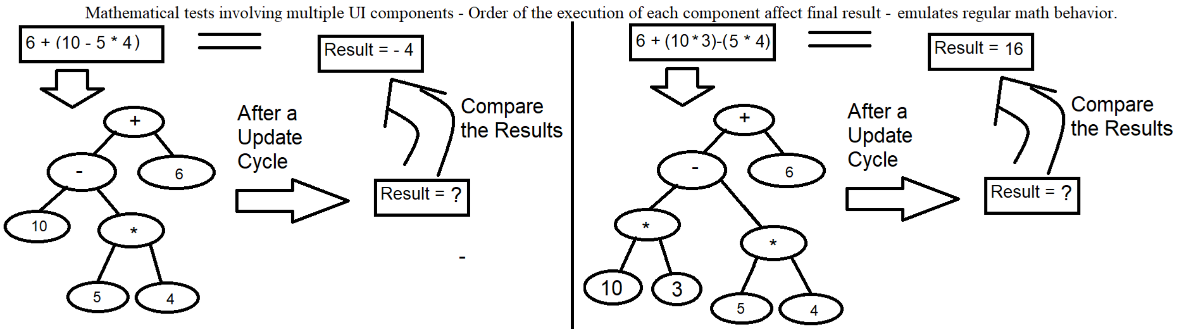


Figure A.4. Runtime reconfiguration capability of our augmentation.

components have changed since the previous check (or since the beginning of operation).

In this Context, we assume that a change to the hierarchical structure holding the components likely means a change to the component architecture. The figure A.4 depicts how the final result of the overall execution of all components still yield the accurate result even after a modification.

(c) Design and implement lightweight mechanisms to construct the dependency graph initially and to reconstruct it when needed.

To build a dependency graph, the program can traverse the hierarchical structure (e.g., do a breadth-first traversal of the Document Object Model), placing each component at a node and adding edges to other nodes according to the *depends-on* relationships between components. However, it must prune the graph appropriately to remove any cycles.

3. In the **augmentation incorporation process**, the solution developer must modify the application's operation in the following ways:

(a) The application must construct the dependency graph at or before startup. (The left side of Figure A.3 illustrates this augmentation.)

(b) When some component C included in the dependency graph signals an event E, the application must *intercept* E and directly call the procedures associated with event E on all listening components as recorded in the dependency graph. Then it must recursively apply the process to all events signalled by the listening components. It continues this as long as there are dependencies indicated in the graph (which cannot have cycles). This process dynamically coalesces the processing of chains of events into what is processed as one "large-grained" event. (The upper half of Figure A.3 illustrates this augmentation.)

The meaning of "intercept" depends upon the application's specific implementation technologies.

(c) After the processing of each "large-grained" event in the previous step, the application must check whether the application's component architecture has changed (e.g., the addition, modification, or deletion of any component in the hierarchical structure) or the dependencies among components have changed. If so, then it must update the dependency graph appropriately to reflect the new component architecture.

A.5.4 Balancing the Forces

In the Solution described above, we handle all the identified Forces. How do we balance the various Forces to achieve this Solution?

- *Transitional Turbulence Reduction.*

  For a state change in any component, the augmented application must propagate the effects to all its directly or indirectly dependent components without the delays and nondeterminism introduced by the normal event-handling system—as if all were part of the processing of one large-grained event. This can decrease latency and increase accuracy (i.e., decrease the number of errors).

- *Runtime Reconfiguration.*

Frequently during the normal operation of the application, the augmented application checks whether its component architecture has changed. If it detects a change, it then reconstructs the dependency graph to reflect the new architecture. The extra costs incurred in reconstructing the dependency graph must not itself worsen the solution's overall effect on the latency and accuracy.

Changes to an application's component architecture during normal operation can increase latency and decrease accuracy. However, a good solution must dynamically adapt to such changes and seek to mitigate the effects on latency and accuracy.

- *Startup Cost Inflation.*

  When applying the pattern, developers should seek to keep the cost of initially constructing the dependency graph low. The developers should carefully select the components to include in the analysis and use efficient methods for determining dependency relationships and constructing the graph.

  The augmented application likely incurs additional processing overhead at startup and shutdown. In particular, the extra costs for constructing the initial dependency graph should be small in proportion to the potential accuracy and performance gain in an application that runs sufficiently long.

- *Operational Overhead Creep.*

  The augmented application likely incurs additional processing overhead during normal operation, especially when the component architecture changes. In particular, the extra costs for checking for changes in the component architecture and reconstructing the dependency graph should be small in proportion to the potential accuracy and performance gain in an application that runs sufficiently long. In cases in which the component architecture changes infrequently, the augmented application should incur minimal costs.

- *Code Cluttering.*

  To implement a solution, the developer must augment the existing application by incorporating a set of software mechanisms as described above. Unfortunately, modifying the application often complicates its design, implementation, testing and use.

  However, in a good design and implementation of the solution's new software mechanisms, it should be possible to readily augment the existing solution. Thus, the new software mechanisms should be designed, implemented, and documented carefully so that the solution can work well with typical application designs.

  For example, for a typical GUI application, it should be possible to implement the solution approach as a software framework with wrapper classes for the controls and a library implementing the algorithms for constructing/reconstructing the dependency graph and using it to coalesce chains into "large-grained" events.

A.6 Consequences

A.6.1 Benefits

- *Runtime Configuration:* A solution dynamically adapts to changes in an application's component architecture during normal operation.

- *Transitional Turbulence Reduction:* A solution coalesces sets of dependent internal events into "large-grained" events such that the handling of a large-grained event causes the same overall state change as the corresponding set. This can decrease latency and increase accuracy (i.e., decrease the number of errors).

- *Code Cluttering:* An application can be readily adapted to use the mechanisms implementing the solution.

A.6.2 Liabilities

- *Runtime Reconfiguration:* Changes to an application's component architecture during normal operation can increase latency and decrease accuracy.

- *Startup Cost Inflation:* An implementation of a solution often causes additional processing overhead at startup and shutdown of the application.

- *Operational Overhead Creep:* An implementation of a solution often causes additional processing overhead during normal operation, especially when the component architecture changes.

- *Code Cluttering:* An application must be adapted to use the mechanisms implementing the solution. Modifying the application often complicates its design, implementation, testing, and use.

VITA

João Paulo Oliveira Marum, a.k.a. JP, was born in Sorocaba (Brazil). He earned his Bachelor of Engineering (B.E.) degree in computer engineering from the Sorocaba College of Engineering. He moved to Oxford, Mississippi, USA in 2014 to enroll in graduate school at the University of Mississippi (Ole Miss). He earned an M.S. in computer science in 2017 and a Ph.D. in computer science in 2021. His research is focused on using multiparadigm programming to solve accuracy issues in user interactive systems, especially in Virtual and Augmented Reality applications. He is a professional member of the Association of Computing Machinery (ACM), Institute of Electrical and Electronics Engineers (IEEE), IEEE Computer Science Society, and Order of the Engineer. For five years, he was a graduate instructor at the University of Mississippi, teaching programming languages for major and nonmajor students. He was also a researcher at the Hi5 (High FIdelity Virtual Environments) laboratory at the University of Mississippi. He published articles in the proceedings of ICAT–EGVE (Eurotronics–Virtual Environments), IEEE SouthEastCon, ACM SouthEast, and IEEE VR (the most prominent conference in the area of Virtual Reality). In 2021, he was given the Outstanding Doctoral Student Award by the School of Engineering in the University of Mississippi.