algorithms

Article

Following the Writer's Path to the Dynamically Coalescing Reactive Chains Design Pattern

João Paulo Oliveira Marum[1,*], H. Conrad Cunningham [2,*], J. Adam Jones [3] and Yi Liu [4]

[1] Department of Electrical Engineering and Computer Science, Syracuse University, 3-127 CST, Syracuse, NY 13244, USA
[2] Department of Computer and Information Science, University of Mississippi, 201 Weir Hall, University, MS 38677, USA
[3] Department of Computer Science and Engineering, Mississippi State University, 143 Rice Hall, Mississippi State, MS 39762, USA; jadamj@acm.org
[4] Department of Computer and Information Science, University of Massachusetts Dartmouth, 302E Dion Building, Dartmouth, MA 02747, USA; yliu11@umassd.edu
[*] Correspondence: jomarum@syr.edu (J.P.O.M.); hcc@cs.olemiss.edu (H.C.C.); Tel.: +1-315-443-1135 (J.P.O.M.)

Abstract: Two recent studies addressed the problem of reducing transitional turbulence in applications developed in C# on .NET. The first study investigated this problem in desktop and Web GUI applications and the second in virtual and augmented reality applications using the Unity3D game engine. The studies used similar solution approaches, but both were somewhat embedded in the details of their applications and implementation platforms. This paper examines these two families of applications and seeks to extract the common aspects of their problem definitions and solution approaches and codify the problem-solution pair as a new software design pattern. To do so, the paper adopts Wellhausen and Fiesser's writer's path methodology and follows it systematically to discover and write the pattern, recording the reasoning at each step. To evaluate the pattern, the paper applies it to an arbitrary C#/.NET GUI application. The resulting design pattern is named DYNAMICALLY COALESCING REACTIVE CHAINS (DCRC). It enables the approach to transitional turbulence reduction to be reused across a range of related applications, languages, and user interface technologies. The detailed example of the writer's path can assist future pattern writers in navigating through the complications and subtleties of the pattern-writing process.

Keywords: design pattern; writer's path; event-based architecture; implicit invocation; reactive programming; transitional turbulence; dependency graph

check for updates

Citation: Oliveira Marum, J.P.; Cunningham, H.C.; Jones, J.A.; Liu, Y. Following the Writer's Path to the Dynamically Coalescing Reactive Chains Design Pattern. *Algorithms* **2024**, *17*, 56. https://doi.org/10.3390/a17020056

Academic Editors: Simone Fontana, Silvia Corchs and Aurora Saibene

Received: 22 October 2023
Revised: 17 January 2024
Accepted: 22 January 2024
Published: 25 January 2024

1. Introduction

A visual user interface must respond quickly 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" [1]. 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 visual user interface, that component may cause events that affect several other components (i.e., *update* the other components). Each of these components may, in turn, cause events that affect additional components, and so forth, as the component updates ripple throughout the user interface. We call the processing of all the component updates resulting from some initial event an *update cycle*. When an update cycle is completed, the user interface can potentially enter a *stable state* with no updates pending. For convenience, we define *latency* as the period of “time” it takes for all components of a system to reach a stable state after some stimulus (such as processing an external event).

The display system operates independently of the event handling system. Therefore, it may take several cycles of the display system for the states of all components to be updated and the user interface to reach a stable state. This period of *transitional turbulence* [2] (or glitchiness [3,4]) can result in displays in which the visible states of the components do not meet the users’ expectations of the user interface’s behavior. Due to these inconsistent displays, users can (at least temporarily) perceive the system to be unreliable and inaccurate. For convenience, we often refer to the occurrence of a visible inconsistency on a display as an *error*.

If some component `C` causes an event that directly affects some other component `D`, then `D` *depends on* `C`. To alleviate the transitional turbulence problem, Marum et al. [5,6] developed a reactive programming approach [7] that encodes these dependency relationships between components in a *dependency graph* and then uses the graph to rearrange the processing of the updates from an update cycle in any order consistent with the dependency relationships. This enables the processing of all the events from one update cycle as if the update cycle were a single large-grained event that updates all the components. This enables faster updates and more accurate visualizations, potentially providing users with a more satisfying experience.

For Web and desktop graphical user interfaces (GUIs) implemented with C#, the approach builds the dependency graph by analyzing the relationships between the components of the GUI (i.e., its graphical controls such as radio buttons) [6,8]. Many effects that had previously been spread across multiple display cycles now occur within a single cycle. By conducting a set of experiments, the Marum et al. case study [6,8] shows that the approach can perform better (i.e., decrease latency) and exhibit more accurate behavior (i.e., display fewer errors) than similar applications using the standard C#/.NET GUI and the Sodium [9] and Rx.NET [10] reactive programming libraries.

For virtual and augmented reality applications implemented using the Unity3D game engine and C# [11], the approach is similar, except that it takes advantage of Unity3D’s existing object hierarchy [5]. 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 eliminates many of the inconsistent displays 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, Marum et al. [5] shows that the approach can also perform better and exhibit more accurate behavior than both an unmodified Unity3D application and a similar application developed using the reactive library UniRx [12].

The Marum et al. case studies [5,6] address two different but related problems and devise two similar solutions using different user interface technologies. Both solutions work by augmenting the normal event processing mechanisms used in the applications. In this research, we examine the two case studies and seek to isolate the essence of the solution approach so that it can potentially be applied by others to similar problems using similar programming languages and user interface technologies. To do so, we address the following research questions:

(RQ1) Can we codify the solution approach as a new software design pattern [13,14]?
(RQ2) Can we follow Wellhausen and Fiesser’s writer’s path [15] to write the new pattern step by step?

Section 2 describes the writer’s path and the other methods that we use. Sections 3–12 then record how we use these methods systematically to write the desired new software design pattern, which we name DYNAMICALLY COALESCING REACTIVE CHAINS (DCRC). Appendix B shows the complete DCRC pattern. Section 13 then demonstrates the technical feasibility and efficacy of the DCRC pattern by applying it to an arbitrary C#/.NET GUI application. Section 14 discusses the evolution of the pattern and related and future work.

2. Writing 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" [14]. A pattern is not invented; it is distilled from practical experience [13]. Patterns codify "best practices" for software architecture and design [16]. Patterns are written and published to document these best practices and enable others to apply them in their own work.

The "Siemens" book [13] groups software patterns into three categories:

- An *architectural pattern*—also called an *architectural style* [17,18]—is a high-level, language-independent abstraction that guides the design of the system-wide structure.
- A *design pattern* is a mid-level, (mostly) language-independent abstraction that guides the design of a subsystem.
- An *idiom* is a low-level language-specific abstraction that guides some aspects of both design and implementation.

Among several existing formats for describing patterns [13–16,19], we choose the simple format described by Wellhausen and Fiesser [15], which presents the following structural elements 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, identifying the often contradictory 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 liabilities of using the pattern.

All of the above elements except Consequences are also prescribed by the MANDATORY ELEMENTS PRESENT pattern from Meszaros and Doble's Pattern Language for Pattern Writing [16]. In its OPTIONAL ELEMENTS WHEN HELPFUL pattern, the Consequences element is called the Resulting Context. Following their READABLE REFERENCES TO PATTERNS pattern, we show the pattern names using small capitals in this paper.

Although a fully specified software pattern should be published in the order given above, the elaboration of the pattern's elements usually does not proceed in that order. Instead, it spirals through the elements and may require multiple iterations over a period of time. In this paper, we adopt the Wellhausen and Fiesser *writer's path* [15] to guide us in writing the pattern because it is a simple methodology that enables us to explore the problem domain systematically and refine the description of the pattern incrementally. We enhance its steps by using other established methods such as Scope, Commonality, and Variability (SCV) analysis [20] and the pattern-writing patterns from Meszaros and Doble's [16] and Harrison's [21,22] pattern languages. We carefully record our steps to help others use this methodology to write patterns for other problems.

1. Explore the new pattern's rationale and scope.
   We consider questions such as: Why should we write a new pattern? What is included in and excluded from its scope? What concrete examples do we have that we can examine? We then state a crisp definition for the scope.
2. Examine existing solutions.
   We consider the answers to the questions from the previous step and discuss the solutions with others. We seek to determine what is common across all the solutions and what is variable among the solutions (i.e., holds for only some of the solutions).We briefly summarize the general solution, focusing on its essence. We collect a list of possible names for the pattern. We also list any clever ideas identified in the solutions for later consideration, even if they are not essential to the solution.

3. Describe the problem that leads to the solution.
   We strive to state this description in one sentence. We must be careful to separate the problem from its solution and make sure that the solution actually solves the problem.
4. Consider the consequences of the solution, both its benefits and its liabilities.
   We consider any "clever ideas" identified in Step 2. These may help us identify the consequences of applying the pattern. To identify the benefits, we consider the desirable outcomes that result from applying the pattern. (That is, we consider the difference in the result when the pattern is applied versus when the pattern is not applied.) To identify liabilities, we consider the complications that result from applying the pattern and what the possible undesirable outcomes are.
5. Identify the forces that make the problem difficult to solve.
   The forces usually conflict with one another, pushing in contradictory directions. We consider what differentiates the chosen solution from other possible solutions to the problem to help identify the different forces at work. We give each force a meaningful name.
6. Match each force with the corresponding consequences.
   A force makes the problem difficult to solve. How the solution resolves this difficulty leads to the corresponding consequences. Each force must be resolved and may have both benefits and liabilities. Each consequence must be matched by a force. The matching of forces and consequences helps guide us from the problem to the solution.
7. Elaborate the context in which the problem exists.
   We carefully consider all the assumptions made by the problem and its solution. The problem might not even exist outside of 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 elements of the pattern.
   We use the Context to describe the background and assumptions. We focus on devising a short, crisp Problem description. We put what makes the Problem difficult in the Forces and ensure the Solution solves the Problem and balances the Forces. We link the Forces with the Consequences.
10. Put the pattern elements in the standard order.
    We restate the Solution and Consequences appropriately to match the other elements, writing the pattern so that it flows smoothly from Context to Consequences.
11. Evolve the pattern based on feedback and experience.
    When writing the pattern, we seek feedback from experts in the technical area and in pattern writing. After a period of time, we reexamine and rewrite the pattern description. We continue to evolve the pattern as we gain deeper experience with its use. Patience is necessary because it takes time to ensure that the pattern description is accurate.

3. Exploring Rationale and Scope

In Step 1 on the writer's path, we explore the rationale and scope of the new pattern. Given the dynamic .NET GUI [6] and VR [5] case studies described in Section 1, there appears to exist "a recurring solution to a problem" that can potentially "be reused" by others. As suggested by Meszaros and Doble's PATTERN pattern [16], we seek to document "the solution using the pattern form". We begin by asking: What is the scope (i.e., the context) of the new pattern?

3.1. Implicit Invocation Architectural Pattern

For the research reported in this paper, we find that the IMPLICIT INVOCATION (II) architectural pattern [17,23–25] is useful to help us define the scope of the DCRC design pattern. Using typical software architecture terminology [18,26,27], Shaw describes the *system model* as a graph with software components at the nodes and connectors along the edges [24]. The *components* are high-level computational and data storage entities and the *connectors* are the interactions among the components. Furthermore, there is a *control structure* that governs how the system executes.

Figure 1 depicts the IMPLICIT INVOCATION architectural pattern. According to Shaw [24], an IMPLICIT INVOCATION system consists of a "loosely coupled collection" of "independent reactive processes" (i.e., "modules" [17]). 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" and asynchronous, so that the individual components are unaware of the recipients of their signals.

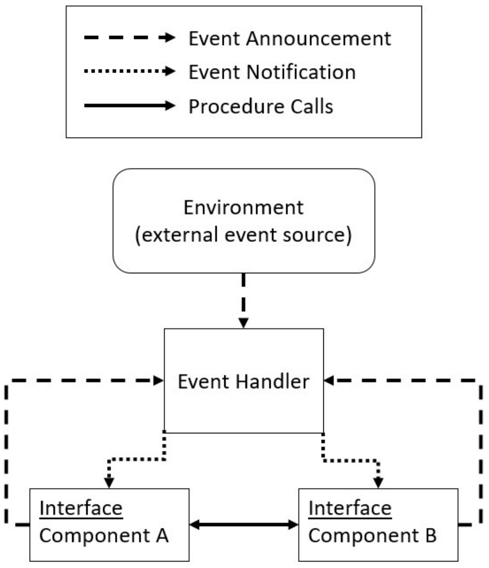

Figure 1. Implicit Invocation architectural pattern.

Implementing 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 signaled [24]. When a component registers interest in an event, it associates a procedure with that event. To notify the component that the event has been signaled, the event handler implicitly invokes the associated procedure [17]. We assume that the event handler is nondeterministic but fair. That is, once an event is signaled 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 [17,24,28,29]. Among the advantages is 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 classic OBSERVER [14] and PUBLISHER-SUBSCRIBER [13,28] design patterns. In this paper, we use the general term IMPLICIT INVOCATION, which seems to describe the overall concept and operation of the event-driven programming mechanisms in Marum's case studies and many other user interface platforms.

3.2. Identifying the Context

In both the .NET GUI [6,8] and VR [5] case studies, the built-in event handling systems follow the implicit invocation architectural pattern as described above. In both, we also observe transitional turbulence as described in Section 1. Both case studies also layer the solution to the transitional turbulence problem on top of the built-in event handling systems. Thus, to define the Context for our new pattern, we focus on the following characteristic:

(C1) The application is constructed according to the IMPLICIT INVOCATION architectural pattern, assuming nondeterministic but fair handling of events.

As we continue to write the pattern, we identify other assumptions about the Context in which the pattern is relevant. In writing the new pattern, we also constrain it in the following ways:

- As suggested by the CLEAR TARGET AUDIENCE [16] and CONSISTENT-“WHO” [22] 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 user interface platform in the general description.
- As suggested by the TERMINOLOGY TAILORED TO AUDIENCE and UNDERSTOOD NOTATIONS patterns [16], 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 [22], we seek to identify any “weasel words”—words that “imply meaning but have no real substance” or 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”. For example, we use a word such as “system” with care because it might have many different meanings in the discussion.

4. Examining Existing Solutions

In Step 2 on the pattern writer’s path, we examine existing solutions. Our primary objective in writing a new pattern is to unify the solutions that emerged from two related case studies: the dynamic .NET GUI [6,8] and VR [5] families of applications.

There is, of course, a wealth of other research on reactive programming languages and systems [3,7,9,10,30–35] that we could profitably examine. However, in this paper, we focus our attention on solutions that follow the IMPLICIT INVOCATION architectural pattern and work by augmenting the normal event handling mechanisms of the user interface technologies on which they are built. The solutions in both Marum et al. case studies satisfy these criteria. The new design pattern seeks to document how a developer should analyze an existing application, develop appropriate new software mechanisms to reduce transitional turbulence, and incorporate the mechanisms into a modified application.

As illustrated in Appendix A, both case studies develop a reactive programming approach that encodes the complex relationships between the components of a specialized IMPLICIT INVOCATION system in a dependency graph and then uses the graph to rearrange the updates of the components in any order consistent with the dependency constraints. As the case studies demonstrate experimentally [5,6,8], this approach enables faster updates and more accurate visualizations, potentially providing users with a more satisfying experience. Harrison’s “WHAT”-SOLUTIONS pattern [22] suggests writing the core idea of a solution in a one- or two-sentence summary. We thus state the Solution element’s summary as follows:

> 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.

This summary will form a prominent part of the full description of the new pattern’s Solution element.

As a result of our analysis, we identify at least three possible names for our pattern: Reactive Dependency Graph, Transitional Turbulence Reduction, and Dynamically Coalescing Reactive Chains. We choose among these names in Section 10.

5. 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 [22] 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 [22] 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.

The core issue addressed by the example applications in Section 4 is reducing transitional turbulence. Transitional turbulence can result in an external presentation that does not accurately represent the expected behavior of the system. This leads to the following statement of the Problem element for the new pattern:

> 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 the observers' expectations by increasing the accuracy of the external presentation.

Consider how the problem can be specifically observed in the case studies. In the example .NET GUI applications, 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—as Harrison's BIG PICTURE pattern [21] suggests it should. The Problem specifies *what* must be performed. The Solution proposes *how* that can be accomplished. The Context describes the environment in which the Problem and its Solution exist.

6. Considering the Consequences

In Step 4 on the pattern writer's path, we consider the consequences of applying the Solution to the Problem, both its benefits and liabilities. To identify these Consequences, we reexamine aspects of the example applications in Section 4.

6.1. Benefits

To identify the benefits, we consider what the desirable outcomes are from applying the pattern to the Problem to construct a Solution.

The example applications analyze the structure of the user interface and optimize its event processing by combining the state changes associated with sequences of related events into larger units. In doing so, they seek to mitigate the effects of transitional turbulence. Therefore, we state the first 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.

In the example applications, the structure of the user interface may change at run time. That is, components and events can be added, deleted, or modified. The applications dynamically adapt to these changes. They seek to preserve the benefits of the event processing optimizations that mitigate the effects of transitional turbulence. We state the second benefit:

- A solution dynamically adapts to changes in an application's component architecture at run time.

The example applications augment the standard (.NET or Unity3D) event processing system, but do not replace it. They use libraries, the C# reflection facilities, and other lightweight programming techniques to optimize event processing. In other situations, a solution might need to use other mechanisms, such as preprocessing tools. We state the third benefit:

- An application can be readily adapted to use the mechanisms that implement the solution.

6.2. Liabilities

To identify the liabilities, we consider the complications resulting from applying the pattern and the possible undesirable results.

In the example applications, the structure of the user interface can change at run time. These changes in the user interface's underlying structure may, in themselves, degrade the event processing performance, and thus increase latency and decrease accuracy. In addition, these changes may degrade the effectiveness of optimizations that are based on the user interface's structure. The application may need to undertake a costly reanalysis of that structure to incorporate different optimizations—as the example applications do. A solution should minimize the cost of adapting to structural changes at run time. We state the first liability:

- Changes to an application's component architecture at run time can increase latency and decrease accuracy.

Any Solution that reduces transitional turbulence likely requires that the structure of the user interface be analyzed and modified before its normal operation begins. This can be a costly operation, particularly if performed at run time—as the example applications do. A solution should minimize this startup overhead. We state the second liability:

- Implementing a solution often causes additional processing overhead at startup and shutdown of the application.

Similarly, any Solution that reduces transitional turbulence likely adds overhead to the normal processing of events. This overhead may be especially significant when the solution must adapt to changes in the user interface's structure. The example applications introduce this kind of overhead because they use dependency graphs to optimize the event processing and must rebuild the graph when the structure of the user interface changes. A solution should minimize this operational overhead. We state the third liability:

- Implementing a solution often causes additional run-time processing overhead, especially when the component architecture changes.

In addition, any Solution that reduces transitional turbulence likely makes the programs more complex and, hence, more costly to design, implement, test, and maintain. The added software mechanisms should be kept lightweight. The example applications use special libraries to handle most of the additional processing needed; the libraries work on top of the standard (.NET or Unity3D) event handler.

To enable the dependency graph to be built, the application developer must adapt some objects in the user interface to allow the library to manipulate them. The mechanisms are relatively lightweight but do make the application's code more complicated. We state the fourth liability:

- An application must be adapted to use the mechanisms that implement the solution. Modifying the application often complicates its design, implementation, testing, or use.

7. Identifying the Forces

In Step 5 on the pattern writer's path, we identify the forces. The Forces are the often contradictory aspects of the stated problem and its context that make it difficult to select and devise a solution [16]. Following the suggestion of the VISIBLE FORCES pattern [16], we assign each force in the new pattern a meaningful name and display the set of forces as a list. Following the suggestion of the FORCES HINT AT SOLUTION pattern [22], we order the Forces in the list from Problem-oriented issues toward Solution-oriented issues. To identify the Forces, we reexamine aspects of the example applications from Section 4.

Decreasing transitional turbulence is the primary motivation for attempting to solve the Problem. This gives rise to the first force we identify.

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

We adopt the same criteria as Marum et al. [5,6,8] to quantify transitional turbulence: latency (perhaps measured in update cycles) and error (i.e., inaccuracy) counts. In an implicit invocation architecture such as the applications we examine in Section 4, decreasing transitional turbulence probably requires a solution that optimizes event processing.

The structures of the example applications' user interfaces can change at run time. We want to handle this situation in any Solution to the Problem. This is the second force.

> *Run-time Reconfiguration:* We want to adapt to changes in an application's component architecture at run time.

Dynamically changing the structure may complicate any solution that optimizes the event processing based on the user interface's structure. For example, if the solution builds and uses a dependency graph of the controls in a .NET GUI, then changes in the the GUI's structure invalidates the graph. This requires that the dependency graph be updated whenever the structure changes, which likely makes the code more complex and degrades performance.

In the example applications (from Section 4), any Solution to the Problem likely requires that the user interface be analyzed and modified before its normal operation begins. Both steps probably require that new software mechanisms (i.e., code) be developed and executed. Additionally, the modified user interface probably has more complex code and a longer execution time. We want to avoid significantly increasing execution time. This gives rise to the third force.

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

Consider the example .NET GUI applications. The analysis may construct a dependency graph of the GUI's controls, and the modification may augment the GUI to use the dependency graph in optimizing the event processing.

- If the analysis and modification can be performed statically, then they can be conducted in a preprocessing phase and will thus have a limited impact on the startup and shutdown of the GUI's execution.
- If the analysis and modification must be performed dynamically, then they must be conducted at run time and can thus have a more significant impact on the startup and shutdown of the GUI's execution. We want these costs to be small.

Because of the requirement to support dynamic changes to the GUI (as discussed above), the example applications do the analysis and modification completely at run time. Some of the initial analysis and modification could have been performed in a preprocessing step, but that would require the mechanisms to be implemented in two completely different ways.

As noted above, the modified user interface has more overhead and more complex code for event processing. The costs of supporting Run-time Reconfiguration also adds processing overhead and code complexity. However, we want to keep the execution costs of event processing small. This is the fourth force.

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

All the mechanisms introduced in the discussion of the other forces above increase the complexity of the programs. This increases the cost of designing, implementing, testing, and maintaining the application. We want to keep this cost small. This is the fifth force.

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

We want any modifications of the programs to be simple and supported by libraries and/or tools. We also want the modifications to the event processing to work on top of the standard event processing mechanisms. The example applications designed and implemented libraries to handle most of the additional processing needed; the libraries work on top of the standard event handlers.

Each force is potentially in conflict with other forces, as shown in Figure 2. A solution must balance these forces.

- The *Transitional Turbulence Reduction* force is in conflict with all the other forces. Seeking to reduce transitional turbulence tends to increase the costs due to the other forces. Seeking to keep the costs due to the other forces low tends to make it difficult to reduce transitional turbulence.
- The *Code Cluttering* force is in conflict with all the other forces. They represent factors that can make the design and implementation of the code more complex. If no code can be added (i.e., the program is kept uncluttered), then the other forces cannot be satisfied.
- The *Run-Time Reconfiguration* force is in conflict with *Operational Overhead Creep*. Dynamically adapting to changes in the application's component architecture increases the operational overhead cost. If no increase in overhead cost is allowed, then the *Run-time Reconfiguration* force likely cannot be satisfied.

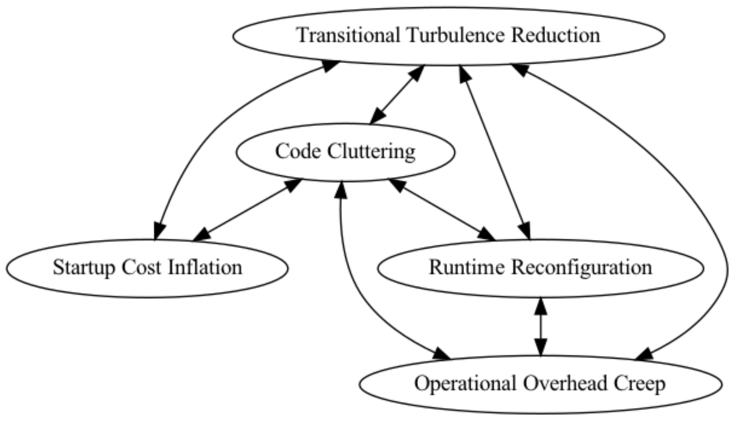

Figure 2. Conflicts among the pattern's forces.

8. Matching Forces with Consequences

In Step 6 on the pattern writer's path, we match each force with the corresponding consequences. Figure 3 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. The matching of forces and consequences helps guide us from the problem to the solution.

- Each force must be resolved; thus the force must be matched with at least one consequence.
- Each consequence must be matched with exactly one force. That is, it must be the unique result of the resolution of some force.

- A force may match both a benefit and a liability. A solution must seek to realize the benefit without incurring the liability. In Figure 3 note that the *Code Cluttering* and *Run-time Reconfiguration* forces each match with both a benefit and a liability.

If a force cannot be matched to a consequence, then it is likely part of the context of the problem that is not resolved by the solution.

Matching Forces with Consequences

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).

Run-time Reconfiguration: We want to adapt to changes in an application's component architecture at run time.

- Benefit: A solution dynamically adapts to changes in an application's component architecture at run time.
- Liability: Changes to an application's component architecture at run time can increase latency and decrease accuracy.

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 run-time processing overhead, especially when the component architecture 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 that implement 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 3. Matching the Forces with the Consequences.

9. Elaborating the Context

In Step 7 on the pattern writer's path, we elaborate 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" [15]. The context "imposes constraints on the solution" [16].

Section 3.2 states the basic Context as follows:

(C1) The application is constructed according to the IMPLICIT INVOCATION architectural pattern, assuming nondeterministic but fair handling of events.

Beyond that, we examine what additional assumptions the example applications from Section 4 make about contexts in which they execute.

In the previous discussion of the *Run-time Reconfiguration* force and the corresponding benefit, we assume that the following characteristic holds for the component architectures. We add this to the Context:

(C2) The application's component architecture may change at run time. The application organizes the components into a hierarchical structure. This structure may change dynamically at run time as a result of external stimuli or the actions of components.

In the previous discussion of transitional turbulence, we assumed that the display system operates independently from the application, but accesses the application's data structures before rendering a representation to the display screen. This situation means that periods of transitional turbulence can exist. Thus, we add the following characteristics to the Context:

(C3) The application presents some aspects of its state that can be observed periodically from outside the system. The timing of this presentation is not under the control of the application.

(C4) Because of the asynchronous nature of the application's operation, the externally observable presentation may exhibit periods of transitional turbulence.

The example applications assume that the components encapsulate their states behind interfaces and restrict all accesses to the states to functions defined in the interface [36–38]. A lack of encapsulation would make it more difficult to determine the relationships among the controls. Hence, we add the following to the Context:

(C5) 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 example applications build dependency graphs that record relationships among the controls. They do this dynamically at run time, so we assume that the implementation environment or the application itself allows a program to extract metadata about the components and their interfaces. Thus, we refine the Context to require that some kind of reflection capability be available:

(C6) 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 run time and extract metadata (such as names, types, and the type signatures of the procedures in component interfaces).

10. 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 [16] 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" [16]. The name should be memorable and suitable for adding 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 Pattern Name

DYNAMICALLY COALESCING REACTIVE CHAINS

because it seems to best meet these criteria. It is a noun phrase that metaphorically evokes how the solution achieves transitional turbulence reduction by coalescing a chain (sequence) of events into a single large-grained event at run time. For convenience, we sometimes use the acronym DCRC.

11. 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 independent of specific programming language and user interface technologies and do not want to overwhelm the reader with arcane details of particular implementations and implementation technologies.

11.1. Solution-Writing Guidelines

Several of Harrison's pattern-writing patterns [21,22] give us guidance on how to refine the Solution:

- As discussed in Section 5, the BIG PICTURE pattern [21] 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 [21] suggests that the Solution should solve the "whole" Problem "but not more".
- The CONVINCING SOLUTION pattern [21] suggests that pattern writers seek to make the Solution "compelling". Often, this means making it "narrower and deeper".
- As discussed in Section 4, the "WHAT"-SOLUTIONS pattern [22] 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 [22] 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 [22] 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 [21] challenging. The latter required us to tweak the statement of the Context to include subtle assumptions the Solution makes about the environment.

How do the forces hint at the solution? The primary purpose of a solution is to realize the benefit of the *Transitional Turbulence Reduction* force. It does so by using a dependency graph to reorder the updates of the components. To realize the benefit and avoid the liability of the *Code Cluttering* force, the solution works by layering lightweight software mechanisms on top of (i.e., by augmenting) the application's normal event processing system. To avoid the liability of the *Startup Cost Inflation* force, the solution must be able to build the dependency graph efficiently. To avoid the liability of the *Operational Overhead Creep* force, the solution must be able to reorder the events according to their dependencies and process them efficiently. To realize the benefit and avoid the liability of the *Runtime Reconfiguration* force, the solution must be able to detect a change in the component architecture and rebuild the dependency graph efficiently.

11.2. Solution: Summary (from Section 4)

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.

11.3. Solution: Definitions

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

- If the execution of some component `X` of an application can directly affect a subsequent execution of some other component `Y` in any way, then `Y` *depends on* `X`. For example, `X` might trigger an event for which `Y` listens; change the value of some attribute of its state that `Y` accesses; directly call one of `Y`'s mutator procedures; or create, delete, or modify `Y`.

- A *dependency graph* is a directed acyclic graph formed by placing the components at the nodes and adding a directed edge from some component `Y` to some component `X` only if `Y` depends on `X`.

Figure 4 shows a dependency graph for an application with ten components named with the upper case letters `A` through `J` and directed edges from every component to every other component on which it is directly dependent. The shaded area of the figure includes the six other components that are directly or indirectly dependent on component `A`. Any change in the state of component `A` may require changes in all other components in the shaded area. If each edge is implemented as an event, then six independent events must be processed to propagate the changes to all dependent components. This is the *update cycle*. The event handling system processes these events in a nondeterministic order, interleaved with any other pending events.

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.

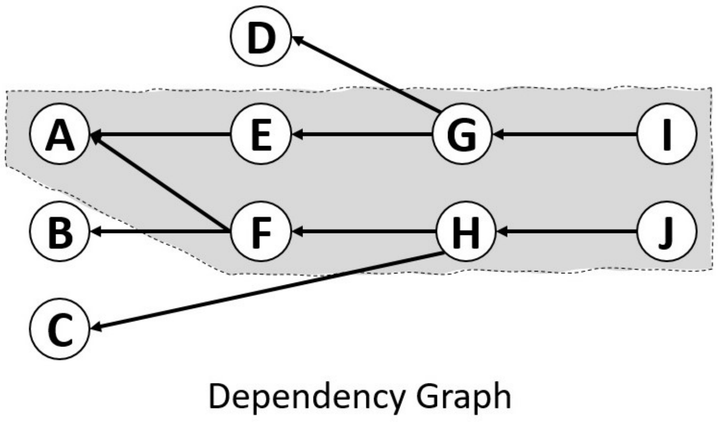

Figure 4. Dependency graph for an application with ten components, illustrating an update cycle.

11.4. Solution: Augmenting the Application

To apply the DCRC pattern to an application that satisfies the Context, we can augment the application with appropriate *software mechanisms*. For example, Figure 5 illustrates how a solution can augment an application's event handling to coalesce dependent events into larger-grained events without modifying the underlying event-handling mechanisms. Beginning with the application's II architecture (shown in panel 1), a solution first determines the dependency relationships between the components (panel 2 and also Figure 4) and then builds the corresponding dependency graph (panel 3). Then it can use the dependency graph to rearrange the component updates in any order that satisfies the dependency constraints (panel 4). In particular, the solution seeks to optimize the processing of an update cycle by performing all the updates in the cycle (the shaded area in Figure 4) directly as part of the processing of the first event.

The software 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 to the existing application. The "software mechanisms" needed and the meaning of "lightweight" depend on the application's specific implementation technologies and performance requirements.

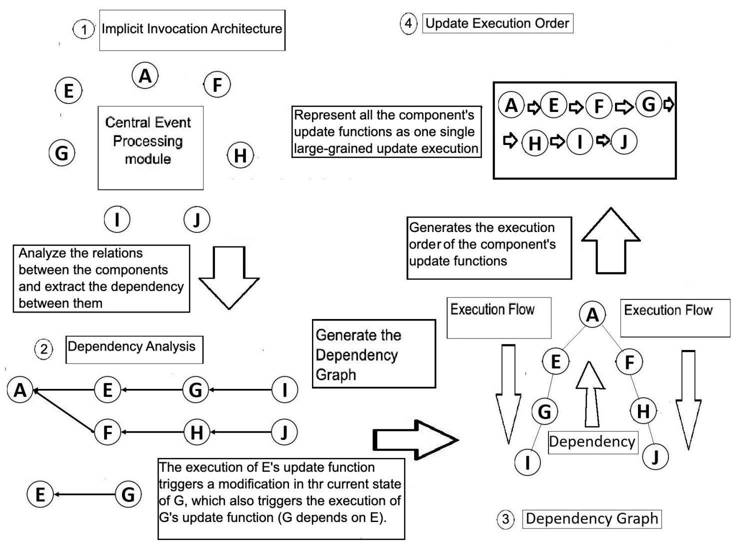
1 Implicit Invocation Architecture
Central Event Processing module
Analyze the relations between the components and extract the dependency between them
2 Dependency Analysis
The execution of E's update function triggers a modification in thr current state of G, which also triggers the execution of G's update function (G depends on E).
Generate the Dependency Graph
Execution Flow
Execution Flow
Dependency
3 Dependency Graph
Generates the execution order of the component's update functions
Represent all the component's update functions as one single large-grained update execution
4 Update Execution Order

Figure 5. Three-step process to coalesce dependent events by augmenting the event processing.

For applications that satisfy the Context, a developer can augment the application's event-handling mechanisms to solve the Problem. In general, to construct a Solution, the developer needs to design, implement, and install three primary software mechanisms: one to build the dependency graph at startup of the application, one to rebuild the dependency graph when needed during the application's execution, and one to coalesce all the component updates in an update cycle into a single event. The construction of these software mechanisms involves an *augmentation workflow* with three phases:

1. *Augmentation analysis*, which requires analyzing the application to identify how to add the necessary mechanisms;
2. *Augmentation development*, which requires developing (i.e., designing and implementing) the mechanisms;
3. *Augmentation incorporation*, which requires incorporating the mechanisms into the operation of the application.

Figure 6 summarizes the augmentation workflow, restating the tasks as questions to answer.

Augmentation Workflow

1. Augmentation Analysis

(AA1) How to iterate through components?
(AA2) How to extract dependency relationships between components?
(AA3) How to select which components to include in dependency graph?

2. Augmentation Development

(AD1) How to design and implement mechanism to differentiate between components included/excluded from dependency graph?
(AD2) How to design and implement mechanisms to detect whether component architecture or dependencies changed?
(AD3) How to design and implement mechanisms to construct/reconstruct dependency graph?

3. Augmentation Incorporation

(AN1) How to augment application to construct dependency graph at startup?
(AN2) How to augment application to use dependency graph to coalesce processing of event chains?
(AN3) How to augment application to update dependency graph when component architecture changes?

Figure 6. Summary questions for the augmentation workflow.

11.4.1. Solution: Augmentation Analysis

In the augmentation analysis phase, the solution developer must perform the three tasks AA1, AA2, and AA3 to analyze the original application and define the requirements for the new software mechanisms.

(AA1) Examine the hierarchical structure to identify how a program can iterate through the components (i.e., accessing each component exactly once).

(AA2) 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 between the components at run time.

Task AA2 may involve the use of the existing features of the components or the reflection capabilities of the implementation language. If sufficient capabilities do not exist, we can design lightweight modifications that implement sufficient application-specific capabilities.

(AA3) Examine the components and events to determine which of the relationships between the components 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, in task AA3, 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 an arbitrary delay.

11.4.2. Solution: Augmentation Development

In the augmentation development phase, the solution developer must perform the three tasks AD1, AD2, and AD3 to design and implement the new software mechanisms according to the requirements specified in the augmentation analysis phase.

(AD1) Design and implement a lightweight run-time mechanism that enables the program to differentiate between the components that are to be included in the dependency graph and those that are not.

Task AD1 involves 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 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 that is called by the dependency graph-building procedure. It must return a boolean value `true` if its argument should be inserted into the dependency graph and otherwise return `false`.

(AD2) Design and implement a lightweight run-time 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, the task AD2 assumes that a change to the hierarchical structure holding the components likely means a change to the component architecture.

(AD3) Design and implement lightweight mechanisms to construct the dependency graph initially and to reconstruct it when needed.

To build a dependency graph in task AD3, the program can traverse the hierarchical structure (e.g., perform 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.

11.4.3. Solution: Augmentation Incorporation

In the augmentation incorporation phase, the solution developer must perform the three tasks AN1, AN2, and AN3 to incorporate the new software mechanisms into the original application. This phase builds on the results of the augmentation development phase. Figure 7 shows how the augmented application can incorporate the three primary software mechanisms into a typical object-oriented GUI application at run time.

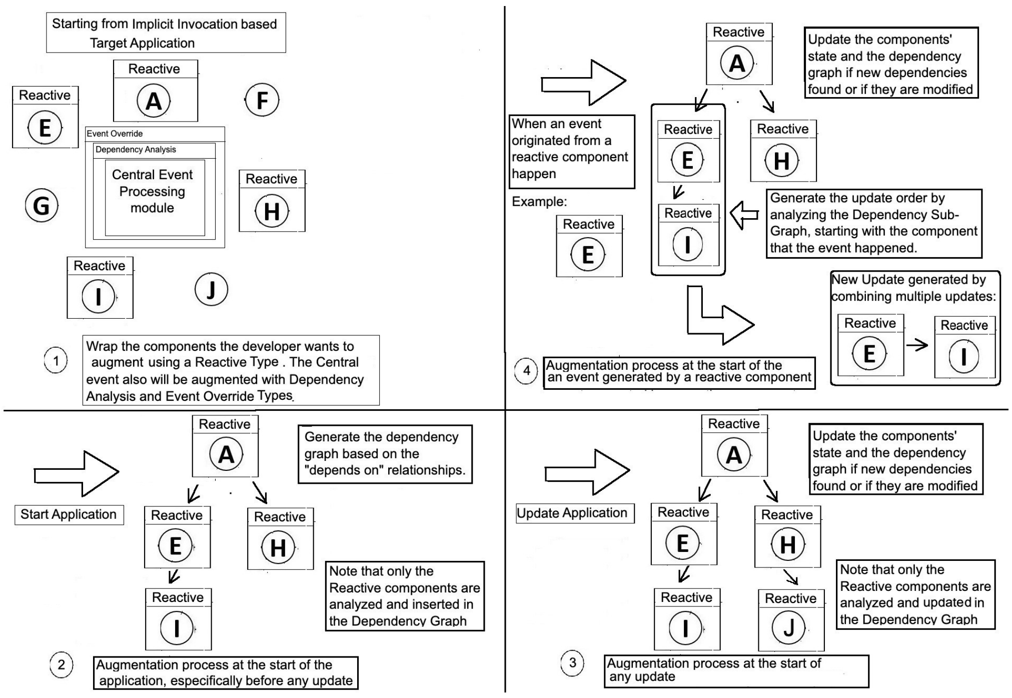
Starting from Implicit Invocation based Target Application
Reactive A
Reactive E
F
Event Override
Dependency Analysis
Central Event Processing module
G
Reactive H
Reactive I
J
1 Wrap the components the developer wants to augment using a Reactive Type . The Central event also will be augmented with Dependency Analysis and Event Override Types.
Reactive A
Update the components' state and the dependency graph if new dependencies found or if they are modified
When an event originated from a reactive component happen
Example:
Reactive E
Reactive E
Reactive H
Reactive I
Generate the update order by analyzing the Dependency Sub-Graph, starting with the component that the event happened.
New Update generated by combining multiple updates:
Reactive E
Reactive I
4 Augmentation process at the start of the an event generated by a reactive component
Start Application
Reactive A
Generate the dependency graph based on the "depends on" relationships.
Reactive E
Reactive H
Reactive I
Note that only the Reactive components are analyzed and inserted in the Dependency Graph
2 Augmentation process at the start of the application, especifically before any update
Update Application
Reactive A
Update the components' state and the dependency graph if new dependencies found or if they are modified
Reactive E
Reactive H
Reactive I
Reactive J
Note that only the Reactive components are analyzed and updated in the Dependency Graph
3 Augmentation process at the start of any update

Figure 7. Three-step process to incorporate the augmentation into an existing application.

(AN1) The application must construct the dependency graph at or before startup.

As shown in panel 1 of Figure 7, the augmentation process begins with an IMPLICIT INVOCATION application modified with the basic mechanisms developed in augmentation development task AD1. As shown in panel 2, it then uses the mechanisms developed in task AD3 to build the initial dependency graph at startup.

(AN2) 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 signaled by the listening components. This continues 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 meaning of "intercept" depends on the specific application's implementation technologies.

As shown in panel 4 of Figure 7, the augmented system combines the updates of all the components in the update cycle into a sequence of direct procedure calls. This coalesces the processing of a whole chain of events into a single event.

(AN3) After processing 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 the dependency graph must be updated appropriately to reflect the new component architecture.

As shown in panel 3 of Figure 7, at the beginning of any update, the augmentation process uses the mechanisms developed in augmentation development task AD2 to determine whether the dependency graph needs to be rebuilt. If so, it uses the mechanisms developed in task AD3 to rebuild the graph.

11.4.4. Solution: 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).

Run-Time Reconfiguration

Frequently during normal operation of the application, the augmented application checks if 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 to determine dependency relationships and build 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. Therefore, the new software mechanisms must be carefully designed, implemented, and documented 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 that implements the algorithms to build/reconstruct the dependency graph and uses it to coalesce chains into "large-grained" events.

12. Putting the Elements in Standard Order

In Step 10 on the pattern writer's path, we put the pattern elements in the standard order as defined in Section 2. We seek to organize the DCRC pattern according to the SINGLE-PASS READABLE [16], SKIPPABLE SECTIONS [16], and FINDABLE SECTIONS [16] patterns. 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 also seek to indicate the pattern's elements clearly and enable readers to skip past elements or detailed descriptions when they are trying to understand and use the pattern.

As shown in Appendix B, the full pattern description consists of the Pattern Name from Section 10, the refined Context from Section 9, the Forces from Section 7, the refined Solution from Section 11, and the Consequences from Section 6 restructured to show the mapping from the Forces in Figure 3. For a presentation of the pattern separate from this paper, it might be helpful to provide additional elements such as those suggested by the GLOSSARY [16] and OPTIONAL ELEMENTS WHEN HELPFUL [16] patterns.

13. Applying the Pattern

In Sections 3–12, we elaborated the DCRC software pattern by following the steps on the writer's path, applying relevant pattern-writing patterns along the way. In this section, we first demonstrate the technical feasibility and efficacy [4,39] of the DCRC pattern by applying it to an arbitrary GUI application developed using C# and the built-in .NET GUI framework.

A software pattern corresponds to a design science technological rule [40] of the form:

To solve an instance of the Problem in the Context apply the Solution.

Thus, to apply the DCRC pattern, we must show that the beginning application satisfies its Context and Problem, then we modify the application as described by its Solution.

13.1. Satisfying the Context and Problem

Section 9 presented the DCRC pattern's Context, which captures six characteristics that must be satisfied by the problem's environment. We list those below and identify the features of a C#/.NET GUI application that satisfy each characteristic.

(C1) The application is constructed according to the IMPLICIT INVOCATION architectural pattern, assuming nondeterministic but fair handling of events.

(C2) The application's component architecture may change at run time. The application organizes the components into a hierarchical structure. This structure may change dynamically at run time as a result of external stimuli or the actions of components.

The structure and operation of the built-in event handling system of the C#/.NET GUI follows the IMPLICIT INVOCATION architectural pattern. The GUI's "components" are its controls, each of which is represented by an object. The GUI arranges the objects representing the controls into a hierarchical data structure internally, for example, the Document Object Model (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. Thus, a C#/.NET GUI application satisfies the characteristics C1 and C2.

(C3) The application presents some aspects of its state that can be observed periodically from outside the system. The timing of this presentation is not under the control of the application.

(C4) Because of the asynchronous nature of the application's operation, the externally observable presentation may exhibit periods of transitional turbulence.

A C#/.NET GUI application consists of controls that are executed asynchronously and communicate through the event handling mechanism. Due to 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 C#/.NET GUI application can exhibit transitional turbulence and therefore satisfies characteristics C3 and C4.

(C5) 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).

In a C#/.NET GUI application, each control 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 on 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" with a well-defined interface [36,37,41]. Thus a C#/.NET GUI application satisfies characteristic C5.

(C6) 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 run time and extract metadata (such as names, types, and the type signatures of the procedures in component interfaces).

The primary programming language of the .NET framework is the object-oriented language C#. The language's extensive reflection facilities enable a program to examine its objects at run time 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). Thus a C#/.NET GUI satisfies characteristic C6.

Therefore, a .NET GUI application satisfies the Context of the DCRC software pattern. Now let us consider the pattern's Problem element, which states:

> 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 noted above, a C#/.NET GUI application can exhibit transitional turbulence. 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. Therefore, the DCRC pattern addresses a problem that is relevant for C#/.NET GUI applications.

13.2. Constructing a Solution

Given an arbitrary GUI application that uses the built-in C#/.NET GUI framework and satisfies the DCRC pattern's Context and Problem elements, we now show how to modify the application to achieve a Solution. Section 11 states the basic requirement in the pattern's Solution summary as follows:

> 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.

For a C#/.NET GUI application, we say that a control `B` "depends on" a control `A` if the execution of `A` can directly affect a subsequent execution of `B` in any way. We form the "dependency graph" by placing the controls at the nodes and adding a directed edge from one node to another if the corresponding components have a depends-on relationship.

The pattern's Solution calls for us to augment the C#/.NET GUI with "lightweight software mechanisms" to reduce transitional turbulence without extensive modification of the built-in event handling mechanism. The required mechanisms may include C# interfaces, classes, methods, functions, data structures, or combinations thereof. The pattern's Solution element describes the construction of these software mechanisms using an augmentation workflow consisting of three phases:

1. augmentation analysis, which requires analyzing the application to identify how to add the necessary mechanisms
2. augmentation development, which requires developing (i.e., designing and implementing) the mechanisms
3. augmentation incorporation, which requires incorporating the mechanisms into the operation of the application

To show that we can construct the desired software mechanisms, we answer the questions given in Figure 6.

13.2.1. Augmentation Analysis

In the augmentation analysis phase, we must perform the three analysis tasks AA1, AA2, and AA3 given in Section 11.4.

(AA1) How can we enable a C#/.NET GUI application to iterate through its components?

A GUI application provides a hierarchical collection of its controls. For a Web application, this collection holds the Document Object Model (DOM). For a desktop application, the `Designer` class holds a collection of controls as objects of class `Control` or one of its subclasses. We can augment the application to iterate through this collection and examine each of its controls, as task AA1 requires.

(AA2) How can we enable a C#/.NET GUI application to extract the dependency relationships between its components?

C#'s `Type` class enables a program to examine any of its 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. We can augment the application to examine its control objects to determine the dependency relationships among them, as task AA2 requires.

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 the control `A`.

(AA3) How can we enable a C#/.NET GUI application to select which components to include in and exclude from its dependency graph?

We likely should exclude any control from the event processing optimization that we cannot modify, such as a control that is part of of the .NET system or a third-party library. We should also exclude any control that requires excessive execution time. We should include all other controls, which we call the "reactive" controls. These controls and their interrelationships must be included in the dependency graph.

To allow us to designate a control as reactive, we define a C# `interface` named `iReactive` that declares a special event handling method `reactiveUpdate()`. We require that any class that implements `iReactive` defines an appropriate method body for `reactiveUpdate()`. If we need to mark an existing control as reactive, we can "wrap" the object with an instance of a class that implements `iReactive`. When the augmented application builds the dependency graph, we can include all controls that implement `iReactive` and exclude all those that do not.

13.2.2. Augmentation Development

In the augmentation development phase, we must perform the three design and implementation tasks AD1, AD2, and AD3 given in Section 11.4. In this phase, we use the results of the augmentation analysis above to design and implement the mechanisms. We then incorporate the resulting mechanisms into the GUI application as described in Section 13.2.3.

(AD1) How can we design and implement a mechanism for a C#/.NET GUI application to differentiate between the components to be included in and excluded from the dependency graph?

We require that all reactive control objects implement the `iReactive` interface as described in Section 13.2.1 above. We then develop a mechanism (e.g., a function) that uses the C# reflection facilities to determine whether or not an object implements the `iReactive` interface. (If an object does, it must be included in the dependency graph; if it does not, it must be excluded from the dependency graph.)

(AD2) How can we design and implement a mechanism to detect whether a C#/.NET GUI application's component architecture or the dependencies among the components have changed?

To determine whether the GUI has changed, we develop a mechanism to store a snapshot of the GUI's structure at the beginning of an update cycle (as defined in Section 13.2.3 below). The snapshot consists of the dependency graph, where each node has a reference to its associated control object. At the end of the update cycle, the mechanism must check whether the GUI structure has changed since the beginning of the cycle. In particular, it must detect GUI changes that modify existing dependencies or add new ones.

To determine whether there are changes in the dependencies, the mechanism must examine each reactive control. 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 if 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 mechanism examines the control's attributes and methods using C#'s reflection facilities. 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.

(AD3) How can we design and implement the mechanism to construct the dependency graph initially and to reconstruct it when needed?

As discussed in Section 13.2.1 above, a .NET GUI consists of a hierarchical collection of control objects. A C# program can iterate through this collection and examine each control using C#'s reflection facilities. If some control object `C` implements the `iReactive` interface (meaning it should be included in the dependency graph), then the mechanism inserts a new node for `C` into the dependency graph. For every other control that depends on `C`, the mechanism inserts a directed edge from that node to `C`'s node.

The algorithms are essentially the same for the initial construction of the dependency graph and for the graph's reconstruction because of a change in the GUI's structure. The reconstruction is different in that it only needs to iterate through the controls referenced by the previous dependency graph.

13.2.3. Augmentation Incorporation

In the augmentation incorporation phase, we must perform the three application augmentation tasks AN1, AN2, and AN3 given in Section 11.4. In this phase, we take the software mechanisms developed in Section 13.2.2 above and incorporate them into the GUI application.

(AN1) How can we augment the C#/.NET GUI application to construct the dependency graph at or before startup?

A C# GUI is an instance of the class `Form` or one of its subclasses. This class defines an event handler method `form_start()` that executes during the `Form`'s instantiation—after it instantiates all its controls but before it renders the form to the display. We modify this method to incorporate the construction of the dependency graph into the event handling system.

To designate a `Form` as reactive, we define a C# `interface` named `iUpdatable` that declares a `form_start()` method. We require that any class implementing `iUpdatable` defines an appropriate method body for `form_start()`. Using the mechanisms developed in Section 13.2.2 above, this method must examine the GUI and construct the initial dependency graph as an object in the `Form` subclass.

(AN2) How can we augment the C#/.NET GUI application to use the dependency graph to coalesce the processing of chains of events into a "large-grained event"?

As discussed in Section 13.2.1, we define an interface `iReactive` that declares a special event handler method `reactiveUpdate()`. Any `Control` subclass that implements `iReactive` must define `reactiveUpdate()` to have appropriate behavior (which may include the behavior of 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 instead of the control's standard event handling code.

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 handler method `reactiveUpdate()`. Based on the constraints in the dependency graph, this method constructs a sequence of updates of the dependent controls. Then it explicitly invokes the `reactiveUpdate()` methods of each control in the sequence. 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 is executed as one "large-grained" event in the augmented GUI application.

(AN3) How can we augment the C#/.NET GUI application to update the dependency graph when the component architecture changes during operation?

We must modify the application so that, at the end of the update cycle, it checks whether the GUI structure has changed since the beginning of the cycle (as described in Section 11.4.2 above). If the GUI has changed, we must rebuild the dependency graph (using mechanisms from Section 11.4.2). We add this check to the augmented event handler `reactiveUpdate` (as described above). It must do this immediately after inferring the execution order from the dependency graph and calling the `reactiveUpdate()` methods of the dependent controls.

13.3. Evaluating the Pattern

Section 13.1 argued that an arbitrary GUI application developed using C# and its built-in .NET GUI framework satisfies the DCRC pattern's Context and Problem elements. Section 13.2 then demonstrated how to modify that application to achieve a Solution. Furthermore, the original Marum et al. .NET GUI case study [6,8] discussed in Section 1 implemented and tested several such applications. Thus, the DCRC pattern is technically feasible.

To investigate the efficacy of the pattern's Solution in reducing transitional turbulence, we can examine the results of the Marum et al. .NET GUI case study [8]. The case study developed a library that embodied the solution approach. The library carries out the dependency graph construction at startup, the graph's reconstruction when needed, and the coalescing of events at run time.

Using the library, the case study developed three different application scenarios (i.e., three different, relatively complex self-completing forms) for the two different user interface platforms (i.e., desktop and Web) and conducted a set of experiments comparing the unmodified .NET GUI applications against the augmented .NET GUI applications. The experiments investigated how transitional turbulence and performance were affected. That is, they evaluated how well the augmented applications balanced the *Transitional Turbulence Reduction* force against the conflicting *Startup Cost Inflation*, *Run-time Reconfiguration*, and *Operational Overhead Creep* forces.

The experiments measured the startup costs for each application [8]. The average startup costs for the augmented .NET GUI applications was 2.6 times the average startup costs of the corresponding unmodified applications (i.e., 55 ms versus 21 ms). Therefore, as expected, there was *Startup Cost Inflation* that had to be mitigated by overall improvements in performance and transitional turbulence reduction.

The experiments also measured the overall execution time for each application, which included times for start-up, operational overhead, run-time reconfiguration, and component execution [8]. The augmented .NET GUI applications executed in about half of the total time required by the corresponding unmodified .NET GUI applications. Thus, for these experiments, the *Startup Cost Inflation*, *Run-time Reconfiguration*, and *Operational Overhead Creep* forces were appropriately balanced.

The experiments characterize transitional turbulence by determining the average number of errors per cycle and the number of cycles required for transitional turbulence to subside [6,8]. One of the application scenarios shows a decrease in the average errors per cycle by 80% from the corresponding unmodified .NET GUI application to the augmented .NET GUI application (i.e., from five errors to one). The other two application scenarios showed only one error per cycle for both the unmodified or augmented .NET GUI applications. The three application scenarios showed that it took an average of 75% fewer cycles for the transitional turbulence to subside in the augmented .NET GUI applications than it did in the unmodified.NET GUI applications (i.e., one cycle instead of four). Therefore, these experiments show that when transitional turbulence exists in an unmodified application, the augmented application exhibits a reduction in transitional turbulence, which shows that the three performance-related forces are in balance. The augmented applications decreased both the transitional turbulence and the overall execution time.

The experiments did not measure to what extent the design and implementation of the augmented applications became more complex [6,8]. However, by expressing the solution as a separate library and a set of wrapper classes for the C#/.NET GUI components, the augmented applications seem to have kept the added complexity small compared to the significant improvements in transitional turbulence and overall performance.

These experiments indicate that the Solution provided by the DCRC pattern is efficacious in a variety of circumstances. That is, it can reduce transitional turbulence in situations where it exists. In a separate case study, Marum et al. [5] demonstrated the feasibility and efficacy of the solution approach for a variety of VR applications using Unity3D. Of course, more experiments should be conducted on the C#/.NET GUI and other applications of the pattern to explore the efficacy of the pattern more fully.

This research extracted the DCRC pattern from two specific Marum et al. case studies and elaborated it systematically using the writer's path. We sought to capture all the assumptions the case studies made about the user interfaces and event-handling systems in the Context element. As a result, we expect the pattern to be applicable to any application that satisfies the Context.

In this section, we have argued that the DCRC pattern is applicable to an arbitrary C#/.NET GUI application. We expect that it is also applicable in other situations (e.g., other applications, languages, and user interface technologies) that satisfy the context. Additional experimentation and evaluation will be needed to determine whether that is indeed the case. If not, it may be necessary to evolve the pattern's context and other elements to handle the additional situations.

14. Discussion

In this section, we reflect on this pattern-writing research, examine related work, and suggest future research on the DCRC pattern and writer's path methodology.

14.1. Evolving the Pattern

The final step on the pattern writer's path is to evolve the pattern based on feedback and experience. The patterns community [42] often uses a process called *shepherding* to assist pattern writers [21,22]. This is a "process in which a pattern author receives feedback from another, experienced pattern author" [15]. 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 [21] pattern 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) [42].

We incorporated a feedback mechanism into our pattern writing process. The DCRC pattern is being written by a diverse four-person team. Two members of the team have expertise in software architecture and two in the application areas and technologies underlying the two case studies. The pattern authors include one member from each group, and the pattern reviewers include the other member from each group. Three members of the team were involved in the initial case studies [5,6] and one was unfamiliar with the case studies before joining the team. The team seeks to further revise the DCRC pattern as it gains more experience using the pattern and writing other patterns.

In the future, we plan to continue to evaluate the generality of the DCRC pattern by conducting new case studies or replicating previous case studies using different programming languages and user interface technologies. For example, we plan to replicate the .NET GUI case study using Java and JavaFX [43,44] and the VR case study using C++ and the CryEngine game engine [45].

In the DCRC pattern description, we seek to specify a design pattern with a relatively broad context. It would have been easier for us to specify idioms for the narrower contexts of C#/.NET and Unity3D by drawing on our understanding of the work in the Marum et al. case studies [5,6,8]. As work on the pattern continues in the future, it may be useful to specialize the general pattern to specific technologies, which may enable the definition of related idioms that are much simpler and more straightforward to apply than the current description of the DCRC pattern.

Software patterns are, in some sense, always works in progress that can incorporate "deeper experience gained when applying patterns in new and interesting ways" [46]. 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.

A better understanding of the DCRC pattern may enable the definition of a network of more specific patterns that can be woven into a pattern language [46–48]. For example, the relatively complex Solution element suggests that refactoring into several fine-grained patterns would be helpful. In addition, studying common variations of the event-driven, IMPLICIT INVOCATION architecture and implementation platforms can potentially lead to a family of related patterns.

14.2. Reflecting on the Writer's Path Methodology

As our primary methodology for writing patterns, we adopted the writer's path from Wellhausen and Fiesser's tutorial [15], integrating other established methods [16,20–22,46] where helpful. The writer's path is promoted as accessible to novice pattern writers. However, we could not find detailed examples of its use, so we chose to record our steps along the path systematically for the possible benefit of other pattern writers. This record also gives us the opportunity to identify challenges and issues for possible future research related to the methodology.

- As we began to write the DCRC pattern, we observed that the Context, Problem, and Solution elements were entangled with each other and with the incidental details of the implementation technologies, the nature of the application domains, the specific program implementations, and the history of their development. In future research, we suggest that steps 1–3 on the writer's path be refined further to help writers articulate clear, precise Context, Problem, and Solutions descriptions at an appropriate level of abstraction.
- To match the Forces with the Consequences, we found it necessary to refactor both the Forces and the Consequences to ensure that the issues were covered in compatible ways. In future research, we suggest that writer's path steps 4–6 be enhanced to help pattern writers identify the Forces and Consequences and state them compatibly.
- As stated in Section 11, the DCRC pattern's Solution element is complex and, thus, difficult to understand. In future research, we suggest that the writer's path be refined further to guide pattern writers in extracting essential information from existing solutions, narrowing the scope, and simplifying the pattern description.
- As we were rewriting the Solution (in writer's path Step 9), we found it necessary to revise the Context to capture several subtle assumptions made by the full Solution description (e.g., support for reflection). In future research, we suggest that the earlier steps of the writer's path be enhanced to help pattern writers identify the Solution's assumptions about the application's environment.
- The writer's path methodology does not currently address how to collect feedback from users and incorporate changes into the pattern, except by repeating the relevant steps. In future research, we suggest extending the methodology to guide pattern writers during this maintenance phase of the pattern life cycle, in particular, on when and how to evolve a pattern into a pattern language [46–48].

Although some researchers have begun efforts to better ground pattern writing in the scientific method [49,50], our purpose in this paper is pragmatic. We use the pattern-writing process to help us systematically deconstruct a set of related applications to reveal their hidden common structure, separating the essential features of the solution from the incidental features of the implementations. For the cases we studied, the writer's path enabled us to capture this structure and draft an appropriate software pattern. In the future, it may be useful to revise the writer's path methodology to incorporate the insights of Riehle et al. [50], Iba [19], design science methodologists [4,40], and others.

14.3. Leveraging Related Research

There are many different variations of the IMPLICIT INVOCATION software pattern. These include the simple OBSERVER [14], advanced OBSERVER [14], revisited OBSERVER [51], EVENT NOTIFICATION [52], and PROPAGATOR [53] design patterns. Mijač et al. [29] evaluates these patterns extensively and finds "a great similarity between considered design patterns, especially in their overall idea and intent", but identifies "features that should be considered when dealing with complex propagation scenarios." In subsequent work, Mijač et al. [4] proposes the REACTOR design pattern to include these improved features. Mijač et al. [35] then incorporates these ideas into an application framework named RE-FRAME, which "provides built-in abstractions, mechanisms and tools for handling reactive dependencies" in the C#/.NET context.

Both the REACTOR and the DCRC design patterns place the dependency graph in a central role, but the contexts of the two patterns differ. The DCRC pattern seeks to augment an existing user interface implementation (such as one that uses the built-in C#/.NET GUI) by adding software mechanisms that build the dependency graph and use it to reduce transitional turbulence. Future enhancements to the DCRC can take advantage of insights from the REACTOR pattern. Similarly, if restricted to specific languages and user interface technologies, the ideas of the DCRC pattern can potentially form the basis of an application framework, as REFRAME does with the REACTOR pattern.

The Marum et al. case studies [5,6] included comparisons of (what we now call) the DCRC solution approach with similar applications developed using the reactive programming packages Sodium [9], Rx.NET [10], and UniRx [12]. In the future, these and other reactive programming approaches [7] should be examined more closely to determine what new ideas can be incorporated into a future revision of the DCRC pattern. Of interest are approaches such as FrTime [3], functional reactive programming [32], FlapJax [33], Elm [30], Distributed REScala [31], and DOM-based functional reactive programming [34].

Our primary focus in this paper has been on defining a pragmatic design pattern that is useful to both practitioners and researchers. To date, we have not paid attention to formulating a formal specification or model. However, the ongoing work to evolve the pattern's Solution could benefit from a better formal understanding of the IMPLICIT INVOCATION architectural pattern and its variants. These have been the focus of formal methods research using a variety of different formalisms, including Z notation [23], process algebra and trace semantics [54], temporal logic [55], model checking [56,57], aspect-oriented programming concepts [58], pattern contracts [59,60], category theory [61], and colored Petri nets [62].

15. Conclusions

Two recent studies addressed the problem of reducing transitional turbulence in applications developed in C# on .NET. The first investigated this problem in desktop and Web GUI applications [6,8] and the second in virtual and augmented reality applications using the Unity3D game engine [5]. The studies used similar solution approaches, but both were somewhat embedded in the details of their applications and implementation platforms.

To answer question RQ1 posed in Section 1, we examined these two families of applications, extracted the common aspects of their problem definitions and solution approaches, and codified this problem-solution pair as a new software design pattern named DYNAMICALLY COALESCING REACTIVE CHAINS (DCRC). We developed the pattern incrementally in Sections 3–12 and then demonstrated its technical feasibility and efficacy in Section 13. In Section 14, we discuss related work and how the DCRC pattern might evolve in the future. This pattern enables the problem-solving approach to be reused in a range of related applications and implementation technologies. This work lays a foundation for further research on transitional turbulence and related software architecture issues.

To answer question RQ2 posed in Section 1, we adopted the writer's path methodology from Wellhausen and Fiesser's tutorial [15] to write new software patterns in a step-by-step manner. We outlined the writer's path in Section 2 and then, in Sections 3–12, followed the path systematically to write the DCRC pattern, carefully recording our reasoning at each step. In Section 14, we discuss related work and possible future enhancements to the writer's path methodology. There are many published patterns, but few well-documented examples of how those patterns were written. The writer's path methodology and detailed example in this paper can assist future pattern writers in navigating through the complications and subtleties of the pattern-writing process. By examining the use of the methodology in this example, we also identified ways in which the methodology itself can be improved.

Writing software patterns is a pragmatic art that has been practiced successfully for more than three decades. It is not possible to capture all the useful processes for pattern writing in one simple software engineering methodology. Even if that were possible, it probably would not match the thinking styles of all software engineers. However, carefully worked examples with thoughtful reflection on the thinking processes involved can be quite useful to others who seek to write or update patterns. That was a motivation for this paper's attention to detail.

Author Contributions: Conceptualization (seminal VR case study) J.P.O.M. and J.A.J.; conceptualization (subsequent GUI case study) J.P.O.M. and H.C.C.; conceptualization (DCRC design pattern) J.P.O.M., H.C.C. and Y.L.; software J.P.O.M.; methodology (case studies) J.P.O.M., J.A.J. and H.C.C.; methodology (enhanced writer's path) H.C.C., Y.L. and J.P.O.M.; writing (dissertation chapter) J.P.O.M.; writing (journal paper draft) H.C.C.; writing (revising and editing) all authors; and supervision H.C.C. and J.A.J. All authors have read and agreed to the published version of the manuscript.

Funding: The first author's (J.P.O.M.) work was supported by CAPES, Coordination for Enhancement of Academic Level Individuals—Brazil and by the University of Mississippi Department of Computer and Information Science.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Acknowledgments: A preliminary version of this work appeared as Chapter 5 of the first author's dissertation [63], which was supervised by the second and third authors. The authors are grateful for constructive feedback from reviewers who have evaluated previous drafts of this paper. We especially thank Cesar Rego of the University of Mississippi for his extensive comments.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

AR	Augmented Reality
DCRC	Dynamically Coalescing Reactive Chains design pattern
GUI	Graphical User Interface
II	Implicit Invocation architectural pattern
MVC	Model-View-Controller design pattern
VR	Virtual Reality

Appendix A. Examining Two Existing Solutions

In Step 2 on the pattern writer's path, we examine existing solutions. Our primary objective in writing a new pattern is to unify the solutions that emerged from two related case studies:

- Dynamic Web and desktop graphical user interface (GUI) applications implemented with C# on the .NET platform [6,8]
- Dynamic virtual reality (VR) and augmented reality (AR) applications implemented in the Unity3D game engine using C# [5].

Appendix A.1. Dynamic GUI Application

In the first case study, Marum et al. [6,8] explored the issue of transitional turbulence occurring in Web and desktop GUI applications 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 performed 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 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, controls within a C# desktop GUI are organized by a separate class named `Designer`; this class abstracts the user interface's visual representation and contains a hierarchical set of controls. The display system uses these data structures when it periodically renders the GUI on the screen.

This is where transitional turbulence can arise. 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's expectations of the user interface's behavior. To combat this problem, this case study developed a reactive programming [7] approach that analyzes the complex relationships among the GUI controls, encodes these dependencies into a dependency graph, and then uses the graph to rearrange the updates of the controls 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 unmodified system into a single large-grained event that updates the states of many controls at once.

Due to the nature of the display system, the approach cannot totally eliminate the transitional turbulence that can cause inaccurate or misleading displays. However, coalescing multiple events into large-grained events does potentially decrease the number of inaccuracies displayed for the rendered state of the system by simplifying the isolated updates of each individual component of the system and agglutinating them into larger execution flows consisting of several components linked into a chain. So, even though the approach makes the code more complex, it flattens the multiple execution flows into a single flow while maintaining the overall performance of the system in terms of starting and update processing times.

To evaluate the approach, this case study developed a prototype library and used it to perform several experiments [6,8]. 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). The experiments performed an automated test a large number of times on each form and measured the startup time, the total time, and the total number of inaccuracies (i.e., errors) when compared against the predicted visual and overall state of the system after the chain of events occurred. Marum et al. [6,8] compared the approach with other approaches that used the .NET GUI library. Figure A1 shows, in general, how the case study's approach constructs the dependency graph for this GUI application and modifies the GUI's event handling mechanism accordingly.

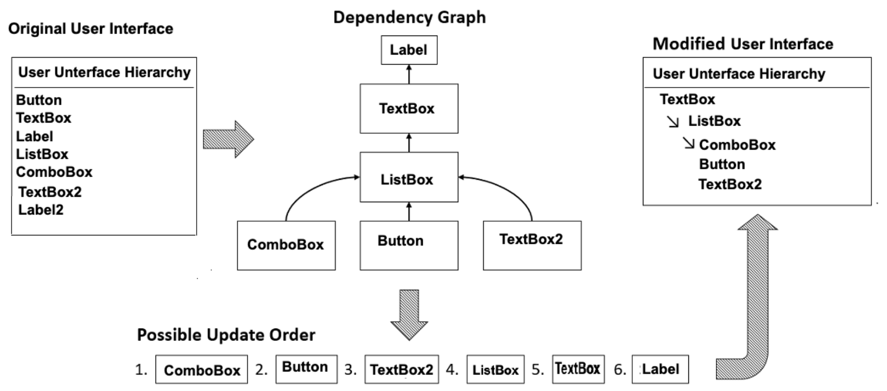

Figure A1. Constructing a dependency graph for the dynamic GUI and determining a new update order.

The experiments indicate that, on average, the approach requires less total time and exhibits fewer visual inaccuracies, at the cost of a modest increase in startup time compared to the three alternatives. Each application developed with the prototype library required approximately 2.6 times 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 about half of the time the corresponding unmodified .NET application required. Furthermore, it exhibited significantly fewer visual inaccuracies for complex self-completion forms than the corresponding unmodified .NET application exhibited. Based on the results of their experiments, they concluded that their approach improves performance and results in a more accurate behavior in many situations.

The Marum et al. solution approach seems to work well for the kind of problems envisioned and the technologies used in the first case study [6,8]. Given how the approach works, it seems reasonable that it will work for other applications requiring similar solutions. In the next subsection, we examine a related case study that uses somewhat different technologies, seeking a more precise understanding of the general solution and the problem it solves.

Appendix A.2. Dynamic VR Applications

In the second case study, Marum et al. [5] extended and systematized the research from a preliminary study [64] conducted a few months earlier. This preliminary work also motivated the case study that we examined in the previous subsection.

The case study sought to eliminate the instability corresponding to the transitional turbulence that occurs in virtual reality (VR) and augmented reality (AR) applications implemented with C# in the popular Unity3D game engine [11]. 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 each other, 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 case study, 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. The 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. Marum et al. [5] 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 occur 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 in the dynamic GUI case study, this case study developed a reactive programming 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. As in the previous case study, because of the nature of the display system, the approach cannot totally eliminate the transitional turbulence that can cause inaccurate or misleading displays, but coalescing multiple events into large-grained events does potentially decrease the number of inaccuracies and increase the system's performance.

To evaluate the approach, this case study developed a prototype library and used it to perform several experiments [5]. The experiments involved a three-way comparison among Unity3D applications using their approach, the built-in Unity3D event system, and UniRx, the Reactive Extensions library for the Unity3D platform [12]. Figure A2 shows, in general, how the case study constructs a dependency graph for this VR application and modifies the game scene accordingly.

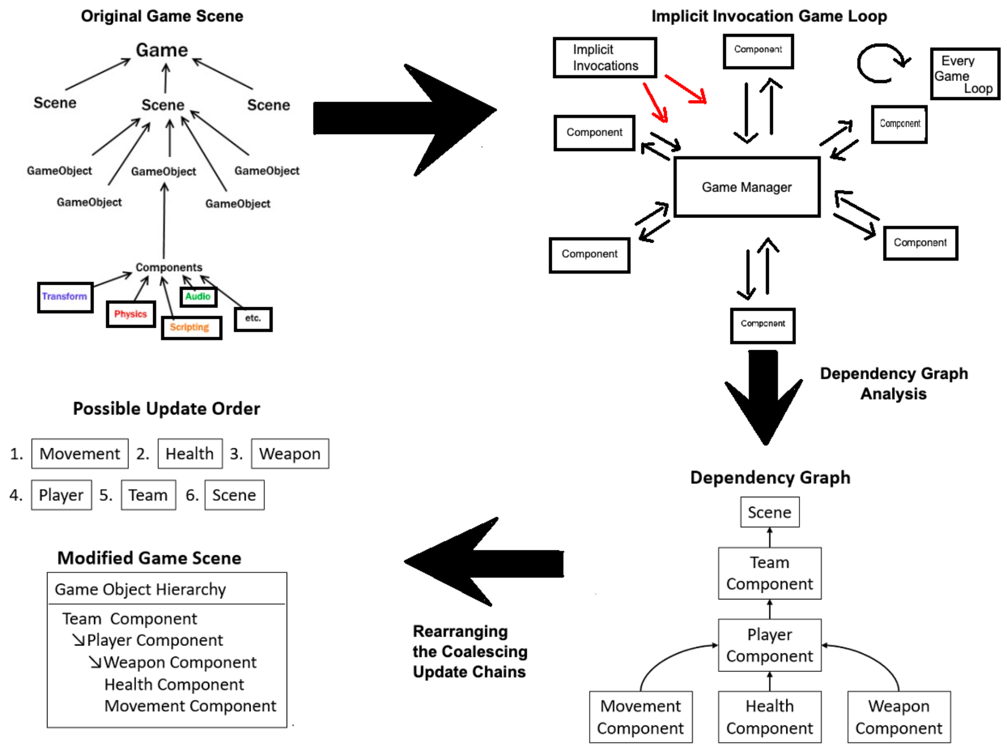

Figure A2. Three-step augmentation of a dynamic VR application using the design pattern's model.

The experiments 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. The test scenario 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, the experiments include (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, the experiment 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 the number of miscalculations that occurred in a test that failed. The experiments indicate that, on average, the approach exhibited a shorter latency and fewer errors, at the cost of a modest increase in the startup time compared to the other two alternatives. Marum et al. [5] concluded that the approach improves performance and results in more accurate behavior.

Thus, the Marum et al. solution approach also seems to work well for the types of problem envisioned and the technologies used in the second case study [5]. In Sections 4–15, Appendixs A.1 and A.2, the task is to identify the commonalities of the two specific solutions and codify a general, technology-independent approach as a new design pattern.

Appendix B. Final Design Pattern

Appendix B.1. Pattern Name

DYNAMICALLY COALESCING REACTIVE CHAINS

Appendix B.2. Context

(C1) The application is constructed according to the IMPLICIT INVOCATION architectural pattern, assuming nondeterministic but fair handling of events.

(C2) The application's component architecture may change at run time. The application organizes the components into a hierarchical structure. This structure may change dynamically at run time as a result of external stimuli or the actions of components.

(C3) The application presents some aspects of its state that can be observed periodically from outside the system. The timing of this presentation is not under the control of the application.

(C4) Because of the asynchronous nature of the application's operation, the externally observable presentation may exhibit periods of transitional turbulence.

(C5) 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).

(C6) 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 run time and extract metadata (such as names, types, and the type signatures of the procedures in component interfaces).

Appendix B.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 the observers' expectations by increasing the accuracy of the external presentation.

Appendix B.4. Forces

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

Run-time Reconfiguration: We want to adapt to changes in an application's component architecture at run time.

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, or use.

Appendix B.5. Solution

Appendix B.5.1. Solution: 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.

Appendix B.5.2. Solution: Definitions

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

- If the execution of some component `X` of an application can directly affect a subsequent execution of some other component `Y` in any way, then `Y` *depends on* `X`. For example, `X` might trigger an event for which `Y` listens; change the value of some attribute of its state that `Y` accesses; directly call one of `Y`'s mutator procedures; or create, delete, or modify `Y`.
- A *dependency graph* is a directed acyclic graph formed by placing the components at the nodes and adding a directed edge from some component `Y` to some component `X` only if `Y` depends on `X`.

Figure 4 shows a dependency graph for an application with ten components named with the upper case letters `A` through `J` and directed edges from every component to every other component on which it is directly dependent. The shaded area of the figure includes the six other components that are directly or indirectly dependent on component `A`. Any change in the state of component `A` may require changes in all other components in the shaded area. If each edge is implemented as an event, then six independent events must be processed to propagate the changes to all dependent components. This is the *update cycle*. The event handling system processes these events in a nondeterministic order, interleaved with any other pending events.

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.

Appendix B.5.3. Solution: Augmenting the Application

To apply the DCRC pattern to an application that satisfies the Context, we can augment the application with appropriate *software mechanisms*. For example, Figure 5 illustrates how a solution can augment an application's event handling to coalesce dependent events into larger-grained events without modifying the underlying event-handling mechanisms. Beginning with the application's II architecture (shown in panel 1), a solution first determines the dependency relationships between the components (panel 2 and also Figure 4) and then builds the corresponding dependency graph (panel 3). Then it can use the dependency graph to rearrange the component updates in any order that satisfies the dependency constraints (panel 4). In particular, the solution seeks to optimize the processing of an update cycle by performing all the updates in the cycle (the shaded area in Figure 4) directly as part of the processing of the first event.

These software 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 to the existing application. The "software mechanisms" needed and the meaning of "lightweight" depend on the application's specific implementation technologies and performance requirements.

For applications that satisfy the Context, a developer can augment the application's event-handling mechanisms to solve the Problem. In general, to construct a Solution, the developer needs to design, implement, and deploy three primary software mechanisms: one to build the dependency graph at startup of the application, one to rebuild the dependency graph when needed during the application's execution, and one to coalesce all the component updates in an update cycle into a single event. The construction of these software mechanisms involves an *augmentation workflow* with three phases:

1. *Augmentation analysis*, which requires *analyzing* the application to identify how to add the necessary mechanisms;
2. *Augmentation development*, which requires *developing* (i.e., designing and implementing) the mechanisms;

3. *Augmentation incorporation*, which requires *incorporating* the mechanisms into the operation of the application.

Figure 6 summarizes the augmentation workflow, restating the tasks as questions to answer.

Solution: Augmentation Analysis

In the augmentation analysis phase, the solution developer must perform the three tasks AA1, AA2, and AA3 to analyze the original application and define the requirements for the new software mechanisms.

(AA1) Examine the hierarchical structure to identify how a program can iterate through the components (i.e., accessing each component exactly once).

(AA2) 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 between the components at run time.

Task AA2 may involve the use of the existing features of the components or the reflection capabilities of the implementation language. If sufficient capabilities do not exist, we can design lightweight modifications that implement sufficient application-specific capabilities.

(AA3) Examine the components and events to determine which of the relationships between the components 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, in task AA3 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 an arbitrary delay.

Solution: Augmentation Development

In the augmentation development phase, the solution developer must perform the three tasks AD1, AD2, and AD3 to design and implement the new software mechanisms according to the requirements specified in the augmentation analysis phase.

(AD1) Design and implement a lightweight run-time mechanism that enables the program to differentiate between the components that are to be included in the dependency graph and those that are not.

Task AD1 involves 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 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 that is called by the dependency graph-building procedure. It must return a boolean value `true` if its argument should be inserted into the dependency graph and otherwise return `false`.

(AD2) Design and implement a lightweight run-time 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, task AD2 assumes that a change to the hierarchical structure holding the components likely means a change to the component architecture.

(AD3) Design and implement lightweight mechanisms to construct the dependency graph initially and to reconstruct it when needed.

To build a dependency graph in task AD3, 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.

Solution: Augmentation Incorporation

In the augmentation incorporation phase, the solution developer must perform the three tasks AN1, AN2, and AN3 to incorporate the new software mechanisms into the original application. This phase builds on the results of the augmentation development phase. Figure 7 shows how the augmented application can incorporate the three primary software mechanisms into a typical object-oriented GUI application at run time.

(AN1) The application must construct the dependency graph at or before startup.

As shown in panel 1 of Figure 7, the augmentation process begins with an IMPLICIT INVOCATION application modified with the basic mechanisms developed in augmentation development task AD1. As shown in panel 2, it then uses the mechanisms developed in task AD3 to build the initial dependency graph at startup.

(AN2) 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. This continues 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 meaning of "intercept" depends on the specific application's implementation technologies.

As shown in panel 4 of Figure 7, the augmented system combines the updates of all the components in the update cycle into a sequence of direct procedure calls. This coalesces the processing of a whole chain of events into a single event.

(AN3) After processing 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 the dependency graph must be updated appropriately to reflect the new component architecture.

As shown in panel 3 of Figure 7, at the beginning of any update, the augmentation process uses the mechanisms developed in augmentation development task AD2 to determine whether the dependency graph needs to be rebuilt. If so, it uses the mechanisms developed in task AD3 to rebuild the graph.

Appendix B.5.4. Solution: 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).

Run-Time Reconfiguration

Frequently during normal operation of the application, the augmented application checks if 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 to determine dependency relationships and build 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. Therefore, the new software mechanisms must be carefully designed, implemented, and documented 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 that implements the algorithms to build/reconstruct the dependency graph and uses it to coalesce chains into "large-grained" events.

Appendix B.6. Consequences

Appendix B.6.1. 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).
- *Run-time Configuration:* A solution dynamically adapts to changes in an application's component architecture at run time.
- *Code Cluttering:* An application can be readily adapted to use the mechanisms that implement the solution.

Appendix B.6.2. Liabilities

- *Run-time Reconfiguration:* Changes to an application's component architecture at run time can increase latency and decrease accuracy.
- *Startup Cost Inflation:* Implementing a solution often causes additional processing overhead at startup and shutdown of the application.
- *Operational Overhead Creep:* Implementing a solution often causes additional run-time processing overhead, especially when the component architecture changes.
- *Code Cluttering:* An application must be adapted to use the mechanisms that implement the solution. Modifying the application often complicates its design, implementation, testing, or use.

References

1. Chandy, M.K.; Misra, J. *Parallel Program Design: A Foundation*; Addison Wesley: Boston, MA, USA, 1988.
2. Lorenz, E.N. Deterministic Nonperiodic Flow. *J. Atmos. Sci.* **1963**, *20*, 130–141. [CrossRef]
3. Cooper, G.H.; Krishnamurthi, S. Embedding Dynamic Dataflow in a Call-by-Value Language. In Proceedings of the Programming Languages and Systems, 15th European Symposium on Programming, Vienna, Austria, 27–28 March 2006; pp. 294–308.
4. Mijač, M.; García-Cabot, A.; Strahonja, V. Reactor Design Pattern. *TEM J. Technol. Educ. Inform.* **2021**, *10*, 18–30. [CrossRef]
5. Marum, J.P.O.; Jones, J.A.; Cunningham, H.C. Dependency Graph-based Reactivity for Virtual Environments. In Proceedings of the IEEE VR 2020 Workshop on Software Engineering and Architectures for Interactive Systems (SEARIS), Atlanta, GA, USA, 22–26 March 2020; pp. 246–253.
6. Marum, J.P.O.; Cunningham, H.C.; Jones, J.A. Unified Library for Dependency Graph Reactivity on Web and Desktop User Interfaces. In Proceedings of the ACM Southeast Conference, ACMSE 2020, Tampa, FL, USA, 2–4 April 2020; pp. 26–33.
7. Bainomugisha, E.; Carreton, A.L.; van Cutsem, T.; Mostinckx, S.; de Meuter, W. A Survey on Reactive Programming. *ACM Comput. Surv.* **2013**, *45*, 1–34. [CrossRef]
8. Marum, J.P.O.; Cunningham, H.C.; Jones, J.A. *Unified Library for Dependency Graph Reactivity on Web and Desktop User Interfaces: ADDENDUM*; Technical Report; University of Mississippi, Department of Computer and Information Science: Oxford, MS, USA, 2020. Available online: https://john.cs.olemiss.edu/~hcc/papers/Addendum_ACMSE_2020.pdf (accessed on 16 January 2024).
9. Blackheath, S.; Jones, A. *Functional Reactive Programming*; Manning: Shelter Island, NY, USA, 2016.
10. ReactiveX Project. ReactiveX: An API for Asynchronous Programming with Observable Streams. 2023. Available online: http://reactivex.io (accessed on 16 January 2024).
11. Unity Technologies. *Unity User Manual 2020.3*; Unity Technologies: San Francisko, CA, USA, 2023. Available online: https://docs.unity3d.com/Manual (accessed on 16 January 2024).
12. Kawai, Y. UniRx: Reactive Extensions for Unity3D. GitHub. 2024. Available online: https://github.com/neuecc/UniRx (accessed on 16 January 2024).
13. Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*; Wiley: Chichester, UK, 1996; Volume 1.
14. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison Wesley: Boston, MA, USA, 1995.
15. Wellhausen, T.; Fiesser, A. 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, Irsee, Germany, 11–15 July 2011; pp. 1–9.
16. Meszaros, G.; Doble, J. A Pattern Language for Pattern Writing. In *Pattern Languages of Program Design 3*; Addison Wesley: Boston, MA, USA, 1998; pp. 529–574.
17. Garlan, D.; Shaw, M. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering*; Ambriola, V., Tortora, G., Eds.; World Scientific: Singapore, 1993; pp. 1–39.
18. Qian, K.; Fu, X.; Tao, L.; Xu, C.W.; Diaz-Herrera, J.L. *Software Architecture and Design Illuminated*; Jones & Bartlett Learning: Burlington, MA, USA, 2010.
19. Iba, T. How to Write Patterns: A Practical Guide for Creating a Pattern Language on Human Actions. 2021 Pattern Languages of Programs Conference, PloPourri, a Methodological, Philosophical, and Educational Study on Pattern Languages. 2022. Available online: https://hillside.net/plop/2021/plopourri/PLoP21_PLOPOURRI_Iba_Methodology4.pdf (accessed on 16 January 2024).
20. Coplien, J.; Hoffman, D.; Weiss, D. Commonality and Variability in Software Engineering. *IEEE Softw.* **1998**, *15*, 37–45. [CrossRef]
21. Harrison, N.B. The Language of Shepherding: A Pattern Language for Shepherds and Sheep. In *Pattern Languages of Program Design*; Harrison, N., Foote, B., Rohnert, H., Eds.; Addison Wesley: Boston, MA, USA, 1999; Volume 4, pp. 507–530.
22. Harrison, N.B. Advanced Pattern Writing: Patterns for Experienced Writers. In *Pattern Languages of Program Design*; Manolescu, D.A., Voelter, M., Noble, J., Eds.; Chapter 16; Addison Wesley: Boston, MA, USA, 2006; Volume 5, pp. 433–451.
23. Garlan, D.; Notkin, D. Formalizing Design Spaces: Implicit Invocation Mechanisms. In Proceedings of the VDM'91, Formal Software Development Methods: Proceedings of the International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, 21–25 October 1991; pp. 31–44.

24. Shaw, M. Some Patterns for Software Architectures. In *Pattern Languages of Program Design*; Vlissides, J., Coplien, J.O., Kerth, N.L., Eds.; Addison Wesley: Boston, MA, USA, 1996; Volume 2, pp. 255–269.
25. Qian, K.; Fu, X.; Tao, L.; Xu, C.; Diaz-Herrera, J.L. Implicit Asynchronous Communication Software Architecture. In *Software Architecture and Design Illuminated*; Chapter 8; Jones & Bartlett Learning: Burlington, MA, USA, 2010; pp. 177–198.
26. Shaw, M.; DeLine, R.; Klein, D.V.; Ross, T.L.; Young, D.M.; Zelesnik, G. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. Softw. Eng.* **1995**, *21*, 314–335. [CrossRef]
27. Garlan, D. Software Architecture. In *Wiley Encyclopedia of Computer and Science Engineering*; Wah, B.W., Ed.; Wiley Online Library: Hoboken, NJ, USA, 2007.
28. Eugster, P.T.; Felber, P.A.; Guerraoui, R.; Kermarrec, A. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.* **2003**, *35*, 114–131. [CrossRef]
29. Mijač, M.; Kermek, D.; Zlatko, S. Complex Propagation of Events: Design Patterns Comparison. In *Information Systems Development: Transforming Organisations and Society (ISD2014 Proceedings)*; Strahonja, V., Vrček, N., Plantak Vukovac, D., Barry, C., Lang, M., Linger, H., Schneider, C., Eds.; University of Zagreb, Faculty of Organization and Informatics: Varaždin, Croatia, 2014; pp. 306–316.
30. Czaplicki, E.; Chong, S. Asynchronous Functional Reactive Programming for GUIs. In Proceedings of the 34th SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, 16–19 June 2013; pp. 411–422.
31. Drechsler, J.; Salvaneschid, G.; Mogk, R.; Mezini, M. 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 2014, Portland, OR, USA, 20–24 October 2014; pp. 361–376.
32. Elliott, C.M. Push-Pull Functional Reactive Programming. In Proceedings of the 2nd SIGPLAN Symposium on Haskell, Haskell ’09, Edinburgh, UK, 3 September 2009; pp. 25–36.
33. Meyerovich, L.A.; Guha, A.; Baskin, J.; Cooper, G.H.; Greenberg, M.; Bromfield, A.; Krishnamurthi, S. Flapjax: A Programming Language for Ajax Applications. In Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2009, Orlando, FL, USA, 25–29 October 2009; pp. 1–20.
34. Reynders, B.; Devriese, D.; Piessens, F. Experience Report: Functional Reactive Programming and the DOM. In Proceedings of the Companion to the First International Conference on the Art, Science and Engineering of Programming, Programming ’17, Brussels, Belgium, 3–6 April 2017; pp. 23:1–23:6.
35. Mijač, M.; Garcia-Cabot, A.; Strahonja, V. REFRAME—A Software Framework for Managing Reactive Dependencies in Object-oriented Applications. *SoftwareX* **2023**, *24*, 101571. [CrossRef]
36. Parnas, D.L. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* **1972**, *15*, 1053–1058. [CrossRef]
37. Britton, K.H.; Parker, R.A.; Parnas, D.L. A Procedure for Designing Abstract Interfaces for Device Interface Modules. In Proceedings of the 5th International Conference on Software Engineering, San Diego, CA, USA, 9–12 March 1981; pp. 195–204.
38. Parnas, D.L. The Secret History of Information Hiding. In *Software Pioneers: Contributions to Software Engineering*; Broy, M., Denert, E., Eds.; Springer: Berlin/Heidelberg, Germany, 2002; pp. 398–409.
39. Mijač, M. Evaluation of Design Science Instantiation Artifacts in Software Engineering Research. In Proceedings of the Central European Conference on Information and Intelligent Systems, Varaždin, Croatia, 2–4 October 2019; pp. 313–321.
40. Engström, E.; Storey, M.; Runeson, P.; Höst, M.; Baldassarre, M.T. How Software Engineering Research Aligns with Design Science: A Review. *Empir. Softw. Eng.* **2020**, *25*, 2630–2660. [CrossRef]
41. Cunningham, H.C.; Zhang, C.; Liu, Y. Keeping Secrets within a Family: Rediscovering Parnas. In Proceedings of the International Conference on Software Engineering Research and Practice (SERP), Las Vegas, NV, USA, 21–24 June 2004; pp. 712–718.
42. The Hillside Group. A Group Dedicated to Design Patterns. 2023. Available online: https://hillside.net (accessed on 16 January 2024).
43. Chin, S.; Vos, J.; Weaver, J. *The Definitive Guide to Modern Java Clients with JavaFX*; Apress: New York, NY, USA, 2019.
44. OpenJFX Project. JavaFX. Gluon. 2024. Available online: https://openjfx.io (accessed on 16 January 2024).
45. Cleary, A.; Vandenbergh, L.; Peterson, J. Reactive Game Engine Programming for STEM Outreach. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education, SIGCSE ’15, Kansas City, MO, USA, 4–7 March 2015; pp. 628–632.
46. Buschmann, F.; Henney, K.; Schmidt, D.C. *Pattern-Oriented Software Architecture, On Patterns and Pattern Languages*; Wiley: Chichester, UK, 2007; Volume 5.
47. Iba, T.; Isaku, T. A Pattern Language for Creating Pattern Languages: 364 Patterns for Pattern Mining, Writing, and Symbolizing. In Proceedings of the 23rd Conference on Pattern Languages of Programs, PLoP ’16, Monticello, IL, USA, 23 October 2016; pp. 1–63.
48. Iba, T.; Kanai, T. Systematization of Patterns for Weaving a Pattern Language as a Whole. 2021 Pattern Languages of Programs Conference, PloPourri, a Methodological, Philosophical, and Educational Study on Pattern Languages. 2022. Available online: https://hillside.net/plop/2021/plopourri/PLoP21_PLOPOURRI_Iba_Methodology2.pdf (accessed on 16 January 2024).
49. Kohls, C.; Panke, S. Is That True...? Thoughts on the Epistemology of Patterns. In Proceedings of the 16th Conference on Pattern Languages of Programs, PLoP ’09, Chicago, IL, USA, 28–30 August 2009; pp. 1–14.
50. Riehle, D.; Harutyunyan, N.; Barcomb, A. Pattern Discovery and Validation Using Scientific Research Methods. In *Transactions on Pattern Languages of Programming V (TPLoP)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2021; pp. 1–25. Available online: https://arxiv.org/abs/2107.06065 (accessed on 16 January 2024).

51. Eales, A. The Observer Pattern Revisited. In *Educating, Innovating & Transforming: Educators in IT: Concise Paper*; The Pennsylvania State University: State College, PA, USA, 2005.
52. Riehle, D. The Event Notification Pattern—Integrating Implicit Invocation with Object-Orientation. *Theory Pract. Object Syst.* **1996**, *2*, 43–52. [CrossRef]
53. Feiler, P.H.; Tichy, W.F. Propagator: A Family of Patterns. In Proceedings of the TOOLS USA 97. International Conference on Technology of Object Oriented Systems and Languages, IEEE, Santa Barbara, CA, USA, 1 August 1997; pp. 355–366.
54. Garlan, D.; Jha, S.; Notkin, D.; Dingel, J. Reasoning about Implicit Invocation. In Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lake Buena, SIGSOFT '98/FSE-6, Vista, FL, USA, 3–5 November 1998; pp. 209–221.
55. Mikkonen, T. Formalizing Design Patterns. In Proceedings of the 20th International Conference on Software Engineering, Kyoto, Japan, 19–25 April 1998; pp. 115–124. [CrossRef]
56. Garlan, D.; Khersonsky, S. Model Checking Implicit-Invocation Systems. In Proceedings of the Tenth International Workshop on Software Specification and Design, IWSSD, San Diego, CA, USA, 5–7 November 2000; pp. 23–30.
57. Garlan, D.; Khersonsky, S.; Kim, J.S. Model Checking Publish-Subscribe Systems. In Proceedings of the International SPIN Workshop on Model Checking of Software, Portland, OR, USA, 9–10 May 2003; pp. 166–180.
58. Xu, J.; Rajan, H.; Sullivan, K. Aspect Reasoning by Reduction to Implicit Invocation. In Proceedings of the Foundations of Aspect-Oriented Languages Workshop (FOAL), Lancaster, UK, 23 March 2004; pp. 31–36.
59. Soundarajan, N.; Hallstrom, J.O. Responsibilities and Rewards: Specifying Design Patterns. In Proceedings of the 26th International Conference on Software Engineering, IEEE, Edinburgh, UK, 28 May 2004; pp. 666–675.
60. Soundarajan, N.; Hallstrom, J.O.; Shu, G.; Delibas, A. Patterns: From System Design to Software Testing. *Innov. Syst. Softw. Eng.* **2008**, *4*, 71–85. [CrossRef]
61. Fiadeiro, J.L.; Lopes, A. An Algebraic Semantics of Event-based Architectures. *Math. Struct. Comput. Sci.* **2007**, *17*, 1029–1073. [CrossRef]
62. Valero, V.; Macia, H.; Díaz, G.; Cambronero, M.E. Colored Petri Net Modeling of the Publish/Subscribe Paradigm in the Context of Web Services Resources. In Proceedings of the Formal Methods for Industrial Critical Systems: Proceedings of the 20th International Workshop, FMICS 2015, Oslo, Norway, 22–23 June 2015; pp. 81–95.
63. Marum, J.P.O. Dependency-based Reactive Change Propagation Design Pattern Applied to Environments with High Unpredictability. Ph.D. Thesis, University of Mississippi, Department of Computer and Information Science, University, MS, USA, 2021. Available online: https://egrove.olemiss.edu/etd/2122 (accessed on 16 January 2024).
64. Marum, J.P.O.; Jones, J.A.; Cunningham, H.C. Towards a Reactive Game Engine. In Proceedings of the 50th IEEE SouthEastCon, Huntsville, AL, USA, 11–14 April 2019; pp. 1–8.