# Concurrency and Distribution in Reactive Programming

**vom Fachbereich Informatik der Technischen Universität Darmstadt**

zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)

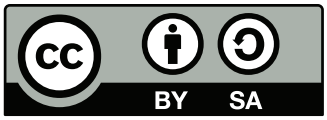
**Dissertation von Joscha Drechsler**

Erstgutachterin: Prof. Dr.-Ing. Mira Mezini
Zweitgutachter: Prof. Dr. Wolfgang De Meuter

Darmstadt 2019

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Drechsler, Joscha: Concurrency and Distribution in Reactive Programming
Darmstadt, Technische Universität Darmstadt,
Jahr der Veröffentlichung der Dissertation auf TUprints: 2019
URN: urn:nbn:de:tuda-tuprints-52284
Tag der mündlichen Prüfung: 11.02.2019



"Whoever said 'the definition of insanity is doing the same thing over and over again, but expecting different results' wasn't de-bugging concurrency issues."

User Arpeggi42 on reddit.com.

# Abstract

Distributed Reactive Programming is a paradigm for implementing distributed interactive applications modularly and declaratively. Applications are defined as dynamic distributed dataflow graphs of reactive computations that depend upon each other, similar to formulae in spreadsheets. A runtime library propagates input changes through the dataflow graph, recomputing the results of affected dependent computations while adapting the dataflow graph topology to changing dependency relations on the fly. Reactive Programming has been shown to improve code quality, program comprehension and maintainability over modular interactive application designs based on callbacks. Part of what makes Reactive Programming easy to use is its synchronous change propagation semantics: Changes are propagated such that no computation can ever observe an only partially updated state of the dataflow graph, i.e., each input change together with all dependent recomputations appears to be instantaneous.

Traditionally, in local single-threaded applications, synchronous semantics are achieved through glitch freedom consistency: a recomputation may be executed only after all its dependencies have been recomputed. In distributed applications though, this established approach breaks in two ways. First, glitch freedom implies synchronous semantics only if change propagations execute in isolation. Distributed applications are inherently exposed to concurrency in that multiple threads may propagate different changes concurrently. Ensuring isolation between concurrent change propagations requires the integration of additional algorithms for concurrency control. Second, applications' dataflow graphs are spread across multiple hosts. Therefore, distributed reactive programming requires algorithms for both glitch freedom and concurrency control that are decentralized, i.e., function without access to shared memory. A comprehensive survey of related prior works shows, that none have managed to solve this issue so far.

This dissertation introduces FrameSweep, the first algorithm to provide synchronous propagation semantics for concurrent change propagation over dynamic distributed dataflow graphs. FrameSweep provides isolation and glitch freedom through a combination of fine-grained concurrency control algorithms for linearizability and serializability from databases with several aspects of change propagation algorithms from Reactive Programming and Automatic Incrementalization. The correctness of FrameSweep is formally proven through the application of multiversion concurrency control theory. FrameSweep decentralizes all its algorithmic components, and can therefore be integrated into any reactive programming library in a syntactically transparent manner: Applications can continue to use the same syntax for Reactive Programming as before, without requiring any adaptations of their code. FrameSweep therefore provides the exact same semantics as traditional local single-threaded Reactive Programming through the exact same syntax. As a result, FrameSweep proves by example that interactive applications can reap all benefits of Reactive Programming even if they are concurrent or distributed.

A comprehensive empirical evaluation measures through benchmarks, how FrameSweep's performance and scalability are affected by a multitude of factors, e.g., thread contention, dataflow topology, dataflow topology changes, or distribution topology. It shows that FrameSweep's performance compares favorably to alternative scheduling approaches with weaker guarantees in local applications. An existing Reactive Programming application is migrated to FrameSweep to empirically verify the claims of semantic and syntactic transparency.

# Zusammenfassung

Verteilte Reaktive Programmierung ist eine Programmiertechnik zur modularen und zeitgleich deklarativen Implementierung verteilter interaktiver Anwendungen. Diese werden als ein über mehrere Rechner verteilter dynamischer Datenfluss-Graph aus voneinander abhängenden Berechnungen definiert, ähnlich zu Formeln in Tabellenkalkulation. Eine Laufzeitbibliothek propagiert eingehende Änderungen durch diesen Datenfluss-Graphen, indem sie die Ergebnisse betroffener Berechnungen neu berechnet und zugleich die Topologie des Datenfluss-Graphen an möglicherweise geänderte Abhängigkeitsbeziehungen anpasst. Nutzerstudien haben ergeben, dass Reaktive Programmierung Code-Qualität sowie Verständlichkeit und Wartbarkeit von Programmen verbessert im Vergleich zu modularen Designs interaktiver Anwendungen basierend auf dem Beobachter-Entwurfsmuster. Ein Teil dieser Verbesserungen basiert auf synchroner Semantik für die Propagation von Änderungen: keine Neuberechnung können einen nur teilweise aktualisierten Zustand des Datenfluss-Graphen beobachten. Dies führt dazu, dass sämtliche Neuberechnungen infolge einer eingehenden Änderung augenscheinlich instantan stattfinden.

Klassischerweise wurde synchrone Semantik in lokalen und nicht nebenläufigen Anwendungen durch das Sicherstellen der Konsistenzeigenschaft “Glitch-Freiheit” erreicht: jede Neuberechnung darf nur ausgeführt werden, nachdem alle nötigen Neuberechnungen ihrer Abhängigkeiten zuvor abgeschlossen wurden. In verteilten Anwendungen sind existierende Ansätze aufgrund zweierlei Gründe jedoch nicht mehr anwendbar. Zum einen impliziert Glitch-Freiheit synchrone Semantik nur solange Propagationen von Änderungen isoliert voneinander ausgeführt werden. Verteilte Anwendungen sind unweigerlich nebenläufig, was bedeutet, dass mehrere Änderungen zeitgleich durch denselben Datenfluss-Graphen propagiert werden. Um mehrere nebenläufig Änderungspropagationen voneinander zu isolieren, müssen daher Algorithmen zur Einschränkung der Nebenläufigkeit in die Laufzeitbibliothek integriert werden. Zum anderen ist der Datenfluss-Graph in verteilten Anwendungen über mehrere Rechner verteilt, welche nur über Netzwerkverbindungen miteinander kommunizieren können. Es können daher, sowohl zum Einhalten von Glitch-Freiheit als auch zur Einschränken von Nebenläufigkeit, ausschließlich Algorithmen eingesetzt werden, die dezentralisiert sind, was bedeutet dass sie ohne Zugriff auf gemeinsamen Arbeitsspeicher funktionieren müssen. Eine umfangreiche Studie themenverwandter früherer Arbeiten zeigt, dass bisher keine Lösung für diese Problemstellung existiert.

Die vorliegende Dissertation präsentiert FrameSweep, den ersten Algorithmus der synchrone Semantik für nebenläufige Propagation von Änderungen auf dynamischen und verteilten Datenfluss-Graphen sicherstellt. FrameSweep isoliert nebenläufige Propagationen und stellt ihre Glitch-Freiheit sicher, indem es mehrere Algorithmen aus dem Feld der Datenbanken für atomar erscheinende Transaktionen mit Aspekten von Algorithmen zur Änderungspropagation aus den Gebieten der Reaktiven Programmierung und der Automatischen Inkrementalisierung kombiniert. Die Korrektheit von FrameSweep wird durch Anwendung von etablierter Theorie zur Nebenläufigkeitskontrolle durch Multiversionen formal bewiesen. FrameSweep dezentralisiert all seine algorithmischen Bestandteile und kann dadurch syntaktisch transparent in jegliche Laufzeitbibliotheken für Reaktive Programmierung integriert werden, was bedeutet, dass existierende Programme FrameSweep ohne Anpassungen ihres Quelltexts verwenden können. FrameSweep implementiert somit auf Basis identischer Syntax identische Semantik wie klassische, lokale, nicht-nebenläufige Reaktive Programmierung. Dadurch ist FrameSweep ein beispielhafter Beweis, dass interaktive Anwendungen alle Vorteile Reaktiver Programmierung genießen können, selbst wenn sie nebenläufige oder verteilte Systeme sind.

Eine umfangreiche empirische Evaluation misst durch Benchmarks, wie sich die Performanz und Skalierbarkeit von FrameSweep verhalten unter diversen Faktoren wie mehr oder weniger häufige Zugriffskonflikte zwischen nebenläufigen Prozessen, Topologie des Datenfluss-Graphen, Änderungen

dieser Topologie, oder Topologie der Verteilung des Datenfluss-Graphen über verschiedene nur durch Netzwerk verbundene Rechner. Die Evaluation zeigt, dass Vergleiche der Performanz lokaler Anwendungen gegen globalen wechselseitigen Ausschluss und transaktionalen Speicher vorteilhaft zugunsten von FrameSweep ausfallen. Durch Migration einer bestehenden, mittels Reaktiver Programmierung implementierten Anwendung zu FrameSweep werden darüber hinaus die Behauptungen zu unveränderter Syntax und Semantik empirisch verifiziert.

# Contents

# List of Figures

# List of Tables



# List of Listings

# 1 Introduction

Smartphones and other mobile computing devices have become ubiquitous, as has mobile internet. In this environment of always-connected always-available user devices, *distributed interactive applications* have become a wide-spread and important class of software applications.

*Interactive* applications are applications that execute continuously, updating their internal state and executing reactions in response to external input stimuli, referred to as *activations* [Halbwachs and Mandel, 2006]. Activations can take many forms, e.g., user interface inputs, elapsing system time, or new sensor readings. It is desirable to implement interactive applications modularly, e.g., have a generic UI button implementation that is separate from the behavior that any specific instance will execute upon clicks. In traditional programming, modules can be reused in the form of custom user code calling generic library functions. In interactive applications though, modularity requires inversion of control, where this relation is reversed. Activations originate within generic library functions, e.g., a UI button, which must then pass control flow to custom user code. Implementing this inversion of control requires the use of callbacks, with the Observer design pattern [Gamma et al., 1995] being the most popular blueprint.

Direct use of callbacks has significant disadvantages on applications' code quality. It requires a lot of boilerplate code that defines, implements and maintains callback interfaces and registries. More critically though, it obfuscates applications' control flow. Multiple callbacks may execute following a single activation, and the order in which they execute is often incidental, particularly if multiple callbacks are chained together. "Callback hell" [Meyerovich et al., 2009] has been coined as a label for interactive applications that have grown into a jungle of interconnected callbacks so complex that their reactive control flow becomes essentially untraceable and developers can only interpret it as non-deterministic. Such non-deterministic control flow makes it very difficult to prevent inconsistencies while updating applications' internal state, as one can never be certain, which parts have already been updated and which have not.

*Distributed* applications are applications that execute across a federation of multiple networked computers, i.e., across multiple processors without shared memory. Callbacks readily transfer to the distributed setting, e.g., through synchronous remote procedure calls or asynchronous message passing to invoke remote callbacks on other hosts. Therefore, established practices for interactive applications can also be used to implement distributed interactive applications. But, this amplifies the problem of apparent non-deterministic control flow twofold. In distributed interactive applications, activations may occur on any host at any time, which can result in multiple callbacks executing concurrently across the entire application. These executions may overlap and interact with each other on mutually affected state, with no easy way to consistently resolve race conditions due to the lack of shared memory. Distributed interactive applications therefore suffer from more sources of non-determinism in callback execution order than single-threaded applications. Moreover, beyond the vagueness of which callback will execute next, distribution also makes it unclear on which host the next callbacks will execute. In addition to more sources, the non-determinism of control flow between callbacks in distributed interactive applications is therefore also yet harder to trace and track than in single-host applications.

Reactive Programming (RP) [Bainomugisha et al., 2013] is a paradigm for implementing interactive applications that solves the problems of callbacks. RP is declarative in that application developers specify applications' state as data flows of transformations, combinations, and derivations of values, similar to formulae in spreadsheets. A runtime library tracks the data flow dependencies between different values and automates recomputing and executing reactions to their changes. This way, RP absolves the need for boilerplate code that callbacks usually require. Further, the runtime library implements *synchronous*

semantics for the automatic change propagation. The process of recomputing all affected values and executing reactions in following an activation is called an *instant* and executed such that it appears to be instantaneous. All recomputations of data flow derivations and all executions of reactions are ordered such that all other state that they observe has always already been completely updated beforehand. RP thus provides deterministic and easy to understand semantics, which protects applications from descending into callback hell [Salvaneschi et al., 2014a].

The declarativeness of RP transfers to distributed applications as easily as callbacks. Application developers can simply specify distributed data flows through transformations, combinations, and derivations over local state and remote state that is defined on other hosts. The runtime library then automatically executes corresponding reactions and recomputations across the network. The concept of RP's synchronous propagation semantics also transfers easily. Equally deterministic and intuitive behavior is achieved if instants simply appear to be instantaneous across the entire distributed application. All recomputations of data flow derivations and all executions of reactions, *regardless on which host*, are ordered such that all other state that they observe, *regardless from which host*, has always already been completely updated beforehand. From application developers' perspective, distributed RP with synchronous propagation semantics would therefore be as beneficial for distributed interactive applications as non-distributed RP is for non-distributed interactive applications, or even more beneficial because the problem it solves is amplified.

## 1.1 Problem Statement

While several prior works have argued the case of distributed RP [Lombide Carreton et al., 2010, Salvaneschi et al., 2013, Reynders et al., 2014, Drechsler et al., 2014, Proença and Baquero, 2017, Mogk et al., 2018, Weisenburger et al., 2018], no feasible implementations with distributed synchronous propagation semantics exist yet. This is because synchronous propagation semantics transfer to distributed applications easily only in concept, but not in implementation. In fact, the opposite is true: providing synchronous propagation semantics for distributed RP is a difficult challenge. RP systems, regardless of distribution, must order recomputations and reactions within each instant to achieve synchronous propagation semantics. In distributed RP systems, this is more difficult though, because the algorithms that compute this order must be decentralized, i.e., function without access to shared memory. Further, distributed systems are naturally multi-processor systems, and thus subject to concurrency in the form of multi-threading, which breaks the established practices of achieving synchronous propagation semantics.

Traditionally, RP systems addressed local-only single-threaded applications and advocated *glitch freedom* [Cooper and Krishnamurthi, 2006] as their central semantic guarantee: Any data flow derivation or reaction may be recomputed only such that it observes all accessed values with either all or none of the changes of each instant completed. In other words, no recomputation or reaction must ever witness effects of an incomplete change propagation, where an instant already has updated some of the accessed values but will still update others. Glitch freedom is a prohibitive property and thus worthless on its own: a system that never recomputes any values is trivially glitch-free, but obviously useless. Glitch freedom implies synchronous propagation semantics for local-only single-threaded applications though, if it is combined with *liveness* [Margara and Salvaneschi, 2014]: If any value changes, the results of every data flow derivation that used this value must be recomputed before the instant ends. In other words, instants must not complete before having executed all recomputations and reactions that are affected (including transitively) by the underlying activation. All RP systems implement liveness, but it is only rarely discussed, simply because it is such a fundamental requirement for reactive applications that one naturally assumes it. All local-only single-threaded RP systems execute recomputations and reactions following some topological order of their data flow dependencies, which is sufficient to achieve both glitch freedom and liveness, and thus synchronous propagation overall.

Some works have transferred the combination of glitch freedom and liveness into distributed applications [Proença and Baquero, 2017], but do not provide synchronous semantics. This is because

multi-threading breaks the implication between glitch freedom, liveness, and synchronous propagation semantics. The changes of any number of concurrent instants from multiple activations can be incorporated into a value all at once, in a single glitch-free recomputation. This violates neither liveness nor glitch freedom, but does not provide synchronous semantics because the instants observably overlap in time and thus are clearly not instantaneous. In addition to liveness and glitch freedom, synchronous semantics require *isolation* [Drechsler et al., 2018] between concurrent instants: For any two concurrent instants, all changes and observations made by one instant must proceed all those by the other, or vice versa. RP systems addressing local-only single-threaded applications can ignore isolation only because it is free if there are no concurrent instants. To achieve synchronous semantics in multi-threaded RP though, isolation must be carefully and deliberately ensured through additional algorithms for *concurrency control*. In addition to ordering recomputations and reactions *within each* instant, concurrency control must order them and prevent race conditions *between multiple* concurrent instants. Moreover, it must do so consistently across all hosts in the entire distributed application, meaning that any additional algorithms for concurrency control must also be decentralized.

Concurrency control for isolation of RP instants is difficult to solve even before considering decentralization, because instants' control flow cannot be predicted precisely. The order of recomputations within every instant follows the data flow of the user-defined derivations, which provides a rough blueprint for their control flow. But the extent to which each instant affects an application's state depends dynamically on the results of these recomputations. A new sensor reading may or may not newly cross some threshold that varies depending on other inputs, meaning even identical activations can result in very differently unfolding instants. Further, any conditional instructions within derivations can dynamically change the data flow topology of applications while they are executing, by aggregating different parts of their application's state than during previous recomputations. Particularly in distributed applications, support for such dynamic changes to the data flow topology is crucial to accommodate network topology changes, e.g., crashing or disconnecting peers.

Concurrency control in the form of coarse-grained mutual exclusion requires very pessimistic assumptions to accommodate unpredictable control flow, and would therefore result in very extensive mutual exclusion. A global lock is the most extreme example, excluding all but a single instant from executing in any location at all times. Extensive exclusion implies significant overhead in terms of sequential communications between concurrent instants. E.g., the global lock must await a completion notification from one instant before sending a start notification to the next. This is particularly costly in distributed applications because network communication is orders of magnitude slower than local processing. Further, extensive exclusion hinders scalability, which is also particularly important in distributed applications because their data flow topologies can grow much larger than local-only applications. Coarse-grained mutual exclusion is therefore not a feasible approach.

Concurrency control through fine-grained mutual exclusion on the other hand – addressing many small-scale race conditions and ordering ambiguities individually – becomes highly susceptible to deadlocks under unpredictable control flow. In transaction systems, e.g., from databases, it has become common practice to address this problem by detecting deadlocks, and breaking them by aborting some of the involved transactions, discarding or rolling back their changes, and restarting them [Bernstein et al., 1986]. For RP though, since instants may execute reactions containing side-effects which cannot be undone, applying such an approach to RP would turn concurrency control into a leaky abstraction, where side-effects leave lasting and observable changes that would not be possible under single-threaded execution. A scheduling algorithm is needed, which executes a sophisticated orchestration of all instants' individual operations such that deadlocks never occur in the first place, so that aborts are not needed. In essence, concurrency control should appear *semantically transparent*, in that user applications cannot observe any difference between executing in a single-threaded versus a multi-threaded environment.

For generic imperative applications with unpredictable user-influenced control flow, this problem does not have a general solution. Finding a solution that is specific to RP, but still general for all RP applications, is therefore a challenging task, even before considering that the solving algorithm also needs to be

decentralized. Prior work has been able to solve this problem only for applications with static data flow topologies [Czaplicki and Chong, 2013], which is much easier due to significantly less unpredictability in instants' control flow. Even with dynamic data flow topologies though, part of instants' control flow is still governed by the propagation algorithm, which reifies some knowledge about the applications' data flow dependencies and follows certain restrictions to ensure glitch freedom. Thus, RP applications pose an environment that offers both unique challenges and unique opportunities for integrating concurrency control.

This dissertation will show that semantically transparent fine-grained concurrency control, general for arbitrary RP applications even with dynamic dataflow topology changes, is possible and can be fully decentralized. As a consequence, synchronous propagation semantics are feasible to achieve for distributed RP applications. Moreover, the integration of this concurrency control into an RP library is also *syntactically transparent*, in that the library remains usable purely through its established syntax, without requiring any specific adaptations. This means that concurrent and distributed interactive applications can benefit from the same improvements in code quality that RP has been shown to provide for local and single-threaded applications. Overall, this implies that distributed RP not only appears appealing conceptually, but is actually feasible and can be used to simplify the implementation of distributed interactive applications.

## 1.2 Contributions

In detail, this dissertation makes the following contributions.

- **An analysis of how concurrency and distribution affect the established model of RP.** The analysis breaks down RP starting from the API that applications interact with, and analyzes the problems that concurrency and distribution introduce. It further breaks down synchronous propagation semantics into its elementary building blocks and re-assembles them into a correctness definition for synchronous propagation semantics suitable for both concurrent and distributed environments.

- **An extensive survey of related work.** The survey covers frameworks based on or inspired by reactive programming, reactive distributed systems, synchronous programming languages, and event-based programming. It provides an intuition of what application scenarios each framework is designed for and how they differ from each other. It categorizes the frameworks based on the features they support (dynamic topologies, concurrency, distribution) and focusses particularly on the semantic guarantees that they provide and the algorithms they use to implement those.

- **FrameSweep, an all-in-one decentralized scheduling algorithm for synchronous propagation semantics in distributed RP.** While the survey shows that several systems and languages with synchronous semantics exist, FrameSweep is – to the best of our knowledge – the first algorithm to provide these semantics with support for concurrent instants in face of dynamic data flow topology changes, and also the first to do so that is suitable for distributed applications. FrameSweep treats each instant similar to a database transaction, except that any executed operation is immediately final; there is no explicit commit at the end and roll-backs do not exist. FrameSweep implements fine-grained isolation for these instant transactions in the form of *serializability*: Instants execute highly concurrent, but all reads and writes, including dynamic topology changes, are scheduled such that they interact with each other equivalently to a *serial* schedule, where instants execute always one after the other, single-threaded.

  FrameSweep integrates and decentralizes a multitude of algorithms to achieve all of its goals. FrameSweep provides serializability without deadlocks by integrating the established concurrency control algorithms *conservative two-phase locking* and *multiversion concurrency control* through *serialization graph testing*, in combination with a custom technique by the name of *retrofitting*. That this combination works without deadlocks or roll-backs is enabled by multiple properties that are

specific to RP, such as a having a reified data flow topology available at run-time and the control flow of instant transactions adhering to its topological order. FrameSweep implements glitch freedom and liveness by topologically ordering recomputations within each isolated instant, thereby achieving synchronous propagation semantics overall. It minimizes the cost of computing said topological order by computing it through a mark-and-sweep algorithm that reuses the infrastructure for isolation of each instant as the "mark". Mark-and-sweep propagation is straight-forward to decentralize by using the template of diffusing computations. The management of multiversions and locks associated with all individual data flow derivations is naturally decentralized. Only decentralization of serialization graph testing, which connects the other concurrency algorithms across the data flow topology, is challenging. FrameSweep solves this by further integrating partially dynamic graph algorithms for reachability in directed and undirected graphs.

- **A formal proof that a pseudocode implementation of FrameSweep fulfils the given correctness definition.** The proof shows that the serialization graph testing of FrameSweep accurately represents the multiversion serialization graph as defined by established multiversion theory over all executed operations. This shows that the premise of the multiversion theorem holds at all times, which thereby implies that all possible executions that FrameSweep can produce are serializable.
- **An implementation of FrameSweep in the REScala ecosystem [Salvaneschi et al., 2014b], and a detailed performance evaluation thereof.** The evaluation uses several microbenchmarks to measure the effects of different data flow topologies, thread contention, and network topologies on different aspects of the performance and multi-threading scalability of FrameSweep. It also evaluates the requirements and effects of integrating FrameSweep into an existing, previously single-threaded RP application, verifying both its semantic and syntactic transparency. FrameSweep therefore proves by example, that the benefits of RP can be applied to distributed interactive applications without incurring prohibitive performance overhead. The source code of FrameSweep and all benchmarks is included in the REScala code base, publicly available at https://github.com/guidosalva/REScala.

## 1.3 Structure

This dissertation is structured into the following chapters:

- Chapter 2 introduces the concepts and features of RP by example from an application developer's perspective.
- Chapter 3 analyzes the interactions of RP with concurrency and distribution. It breaks down the semantic properties of synchronous propagation, providing a formal foundation and a definition of correctness suitable for concurrent and distributed RP.
- Chapter 4 provides an extensive survey that categorizes related work by the features and consistency guarantees they provide as well as how they provide them, and explains why each falls short of providing synchronous propagation for distributed RP.
- Chapter 5 presents the FrameSweep scheduling algorithm in a distribution-agnostic pseudocode implementation and demonstrates it correct function on a running example.
- Chapter 6 formally proves correctness of the pseudocode implementation.
- Chapter 7 explains parallelization optimizations, changes and further engineering concerns necessary for a decentralized implementation of FrameSweep.
- Chapter 8 evaluates the performance characteristics of FrameSweep through empirical benchmarks.

- Chapter 9 summarizes left-open challenges and future work, and concludes.

## 1.4 List of Publications

In the context of this dissertation, the following research papers have been published:

- Salvaneschi, G., Drechsler, J., and Mezini, M. (2013). Towards distributed reactive programming. In De Nicola, R. and Julien, C., editors, *Coordination Models and Languages*, pages 226–235, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Drechsler, J., Salvaneschi, G., Mogk, R., and Mezini, M. (2014). Distributed REScala: An update algorithm for distributed reactive programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 361–376, New York, NY, USA. ACM.
- Drechsler, J., and Salvaneschi, G. (2014). *Optimizing Distributed REScala*, presented at the Workshop on Reactive and Event-Based Languages and Systems 2014, REBLS '14.
- Mogk, R., and Drechsler, J. (2017). Reactive Programming with REScala. In *Companion to the First International Conference on the Art, Science and Engineering of Programming*, Programming '17, pages 7:1–7:2, New York, NY, USA. ACM.
- Drechsler, J., Mogk, R., Salvaneschi, G., and Mezini, M. (2018). Thread-safe reactive programming. *Proc. ACM Program. Lang.*, 2(OOPSLA).
- Drechsler, J., and Mezini, M. (2018). *Distributing Thread-Safety for Reactive Programming*, presented at the Workshop on Reactive and Event-Based Languages and Systems 2018, REBLS '18.

# 2 Reactive Programming in REScala

The outline of this chapter is as follows. Section 2.1 introduces single-threaded, local-only RP from the developer's perspective by implementing an example application in REScala. The design is inspired by the dining philosophers [Hoare, 1985], with philosophers sharing forks around a table. For flexibility in later examples and benchmarks, the table's `SIZE` (number of philosophers and forks) is parametric. Section 2.2 then dives into details about the execution properties that RP provides for such programs, and how it provides them. Finally, Section 2.3 provides a comprehensive account of all the elementary variables and operations on these variables that RP is built from, which will serve as the foundation for building the FrameSweep integrated scheduler throughout the remainder of this dissertation.

## 2.1 The Application Developer's Perspective

This dissertation discusses RP with two kinds of *reactives*: Events and Signals. This section introduces both, as well as conversions between them and the imperative interactions they support.

### 2.1.1 Signals

Philosophers are modelled as `Vars` (input Signals) of type `Phil`, i.e., with the two possible values `Thinking` or `Eating`, initially `Thinking`.

```
1 cases Phil = { Thinking, Eating }
2 val phils = for (j <- 1 until SIZE) yield
3   Var[Phil](Thinking)
```

**Listing 2.1:** Philosopher Input Vars.

A REScala `Var` is a kind of mutable variable. Like ordinary variables, the value of a `Var` can be imperatively read (through `now`, e.g., `phils(1).now` reads the current value of the philosopher with index 1) and imperatively overwritten (through `set(...)`, e.g., `phils(1).set(Eating)` changes the value to `Eating`). Unlike ordinary variables though, changes of `Vars` are automatically propagated to self-updating *derived* Signals that use the `Vars` in their definition. In the following, forks are also modelled as derived Signals of type `Fork`.

```
4 cases Fork = { Free, Taken(by: Int) }
5 val forks = for (idx <- 0 until SIZE) yield Signal[Fork] {
6   val nIdx = (idx + 1) % SIZE
7   (phils(idx).depend, phils(nIdx).depend) match {
8     case (Thinking, Thinking) => Free
9     case (Eating, Thinking) => Taken(idx)
10    case (Thinking, Eating) => Taken(nIdx)
11    case (Eating, Eating) =>
12      throw new AssertionError()
13 } }
```

**Listing 2.2:** Derived Fork Signals.

The `Signal` keyword instantiates a derived Signal given a user-defined computation, akin to a spreadsheet formula. Like Vars, derived Signals' current values can be imperatively read (e.g., `forks(0).now`),

but unlike Vars, not set. Instead, the defining computation can *depend on* (the `depend` keyword[1]) values of multiple other reactives, its *dependencies*. The derived Signal's value is then updated automatically through *reevaluation*, i.e., re-execution of the its defining user computation, whenever the value of any dependencies changed. Concretely, in our example, each fork's value depends on the values of the philosopher with the same index and the next circular index (Line 7). If both are `Thinking` (Line 8), the fork is `Free`. If one philosopher is `Eating` (Lines 9 and 10), the fork is `Taken(idx)`, with the `Eating` philosopher's index. Otherwise (Line 11), the fork raises an error (can't be used by two philosophers simultaneously). Upon, e.g., `phils(1).set(Eating)`, both `forks(0)` and `forks(1)` depend on `phils(1)` and will be reevaluated. The re-execution of their user computations will now match the second, respectively third case (Lines 9 and 10), and thus both will change their value to `Taken(1)`.

### 2.1.2 Dynamic Dependencies

In the previous example of derived `forks` Signals, each fork has static dependencies on the same two philosophers at all times. In general though, derived reactives may have *dynamic* dependencies.

```
14 cases Sight = { Ready, Done, Blocked(by: Int) }
15 val sights = for (idx <- 0 until SIZE) yield Signal {
16   val prevIdx = (idx - 1 + SIZE) % SIZE
17   forks(prevIdx).depend match {
18     case Free =>
19     forks(idx).depend match {
20       case Taken(neighbor) => Blocked(neighbor)
21       case Free => Ready
22     }
23     case Taken(‘idx‘) =>
24       assert(forks(idx).depend == Taken(idx))
25       Done
26     case Taken(neighbor) => Blocked(neighbor)
27 } }
```

**Listing 2.3:** Derived Sight Signals with Dynamic Dependencies.

`Sight` (line 14) models philosophers' possible perceptions of their forks. According `sights` Signals are instantiated in line 15. Each `sights(i)` first depends on the philosopher's left fork (Line 17) to distinguish three cases:

- Left fork is `Free` (Line 18). Then, `sight` also depends on the right fork (Line 19). If the latter is `Taken` (Line 20), `sight` is `Blocked` with neighbor's index; otherwise (Line 21), `sight` is `Ready`.
- Left fork is `Taken` by the philosopher himself (Lines 23 to 25). Then `sight` is `Done` (he himself is eating). In this case, `sight` also depends on the right fork (Line 24) to assert that it is also taken by the philosopher himself.
- Left fork is `Taken` by a neighbor (Line 26). Then `sight` shows `Blocked` by that neighbor and does *not* depend on the right fork.

Since each `sight` accesses its right fork in the first and second, but not in the last case, this dependency is dynamic. We say, a dependency is *discovered*, when `depend` is called on a reactive during a reevaluation, but was not during the previous reevaluation. Conversely, we say a dependency is *dropped* when `depend` on a reactive was not called during a reevaluation, but was during the previous reevaluation. If a reactive changes, reevaluations are triggered only for those reactives that have dependencies established on that reactive at that point in time. For each `sight`, the dependency on its right fork is discovered (respectively

[1] In the REScala API, `r.depend` is abbreviated as `r()`; we use the explicit `.depend` notation here for naming consistency.

dropped) when, a reevaluation switches into (respectively away from) the computation's last case, in comparison to the previous reevaluation. If the dependency on its right fork last was dropped, changes of the right fork's value will not reevaluate `sight` until it discovers the dependency again during another reevaluation (necessarily triggered by a change of its left fork as the only other remaining dependency).

Dynamic dependencies enable important features. One example is the creation and removal of new Signals at run-time, which must be newly (dis-)connected with their dependencies. Another example are *higher-order* Signals, i.e., Signals whose value contains other (*inner*) Signals. For instance, consider a live score card of a game. The score card is modeled by a Signal whose value is the current list of `Player` objects, sorted by the players' current score. Each `Player` object references its current `score` modeled by a numeric Signal. The score card list Signal is therefore a higher-order Signal, with all players' scores as inner Signals. When new players join or leave the game, the player list Signal is changed, which causes dynamic dependency changes in that it adds (or removes) the additional players' score `Signals` to the sorting computation's dependencies.

### 2.1.3 Events and Conversions

Events are the temporal complement to Signals. While Signals model continuously existing values, such as the current mouse cursor coordinates, Events model instantaneous occurrences of values, such as the coordinate increments whenever the mouse cursor moved. While no changes are being propagated, Events therefore have no value, and thus – different from Signals – cannot be read imperatively. Only if an Event emits a value during the propagation of some changes, this value can be read through `depend` calls only by reevaluations of other reactives triggered within the same changes' propagation. Analogously to input Vars and derived Signals, there are input Events (not shown), which can be fired imperatively through `evt.fire(value)`, and derived Events, which depend on other reactives and may emit values when reevaluated. Events and Signals can also be converted to and derived from each other, as shown below.

```
28 val sightChngs: Seq[Event[Sight]] =
29   for (i <- 0 until SIZE) yield
30     sights(i).changed
```

**Listing 2.4:** Changed Events: Signal/Event Conversion.

The `.changed` derivation converts a Signal (`sights(i)`) into an Event that emits each new value of the Signal.

```
31 val successes: Seq[Event[Sight]] =
32   for (i <- 0 until SIZE) yield
33     sightChngs(i).filter(_ == Done)
```

**Listing 2.5:** Event Filtering.

The `.filter` derivation forwards emitted values only if they match a given predicate. Here, `successes` Events fire whenever the respective `sightChngs` emit `Done`.

```
34 val counts: Seq[Signal[Int]] =
35   for (i <- 0 until SIZE) yield
36     successes(i).fold(0) { (acc, _) => acc + 1 }
```

**Listing 2.6:** Fold Signals: Event/Signal Conversion.

Finally, `.fold` derivation converts the `successes` Events back into Signals. Event to Signal conversion is more complex than the reverse direction, because the resulting Signals always have a value, while the original Events do not. Folding Signals therefore are not pure transformations of their dependencies'

values that can be recomputed at any time, but their values are unique autonomous state. For this reason, `.fold` takes two parameters: the Signal's initial autonomous state value, and an accumulator function. Whenever the underlying Event emits, the accumulator function is invoked to accumulate the emitted value into the Signal's own previous state and return the newly updated state. Folding Signals therefore accumulate Event values with semantics similar to folding over infinite lists. In above code, each folding `counts` Signals is initialized with value `0`, which is incremented whenever the respective `successes` Event fires, i.e., each counts the philosopher's successful `Eating` iterations.[2]

### 2.1.4 Imperative Reactions

So far we have seen that the imperative environment can fire (`evt.fire(value)` or set (`var.set(value)`) input reactives to update all of the application's derived reactive state. Further, the imperative environment can "pull" updated values from all Signals through imperative reads (`s.now`). Naturally, applications want to execute effects as a consequence of reactive updates, which is possible but very cumbersome to implement with only these two interactions, and in particular doesn't work with Events. As a third interaction between imperative and reactive abstractions, the computations defining derived Events and Signals may therefore contain *side-effects* to "push" changes back into the imperative environment.

```
37 val totalCount = successCounts.reduce(_ + _)

38 val output = totalCount.map{v => println(... v ...)}
```

**Listing 2.7:** Side-Effects in User Computations.

To introduce a side-effect reaction in our philosophers example, we first `reduce` (a method of Scala's Seq collection, unrelated to REScala) the collection of all philosophers' `successCounts` by adding them up one by one. This yields the `totalCount` Signal (Line 37), whose value is the overall sum of all philosophers' successful `Eating` iterations. We then `map` the `totalCount` Signal using the `println` function, i.e., console output. Normally, `s.map(f)` is used to create a new derived Signal of all values of `s` transformed via the function `f` (`s.map(f)` is a shorthand notation `Signal{f(s.depend)}`). Passing `println` for `f` doesn't match this intention, since `println` is a `void` function that always only returns the `Unit` value, meaning the resulting `output` Signal is unusable because its value can never change. But, `output` is still reevaluated whenever `totalCount` changed, and while all its reevaluations never result in a change, `println` still executes the side-effect of printing its parameter (the latest value `v` of `totalCount`) to console.

Segregating side-effects into separate "terminal" reactives, i.e., that are not used in other user computations, makes application's designs more accessible.[3] This segregation is voluntary though, instead of a hard requirement. In particular for debugging purposes it is often desirable to insert `println` statements, break points or other side-effects into any derived reactive's defining computation. Due to its synchronous propagation semantics, which we discuss next, RP supports user computations that arbitrarily mix side-effects and pure value transformations for all derived reactives.

## 2.2 RP Behind the Scenes

This section discusses the aspects of RP bring life to applications implemented in the programming style described in the previous section. We introduce the dependency graph as a tool for visualizing and implementing change propagation, and discuss its life cycle: The bird's eye view of synchronous instants

[2] REScala's API readily offers a predefined `successes(i).count` derivation for this exact behavior; we use `fold` manually here to visualize otherwise hidden aspects that are relevant for scheduling later.

[3] REScala even offers a dedicated `observe` transformation, which has the same semantics as `map` transformation, but makes the reactive's "side-effect reaction only" intention clearly visible.

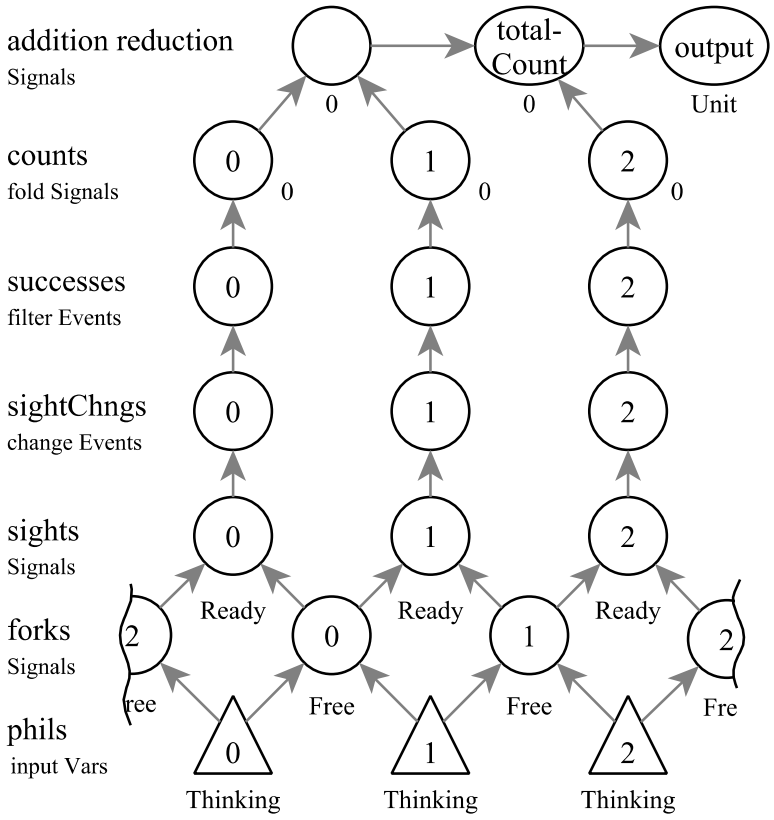



**Figure 2.1:** Initial Philosophers Dependency Graph

as a whole, and the semantic properties for the execution of individual operations within instants to achieve this view.

### 2.2.1 Dependency Graph

The runtime representation of a RP program is its directed acyclic *dependency graph DG*, built while the execution of a reactive program unrolls. Figure 2.1 shows the initial state of the dependency graph that is created when executing the philosophers' code presented so far with `SIZE := 3`. Nodes represent reactives, with input reactives visualized as triangles and derived reactives as circles. The labels on each node correspond to the respective index, i.e., `phils(0), phils(1), phils(2)` for the triangular input Vars.

The dependency graph does not distinguish between Events and Signals, since both fit into the same common interfaces for input and derived reactives: Input reactives can have new values applied imperatively, derived reactives can reevaluate after any dependencies changed, and both these actions result in the reactive either changing (Events emit a value or Signals change their value) or not changing (Events do not emit a value, Signals' values remain unchanged). Still we show Signals' current values printed next to their respective nodes, as a visual reference for which values the user application sees when accessing reactives. E.g., in this initial state, the Signals `phils` are `Thinking`, `forks` are `Free`, `sights` are `Ready`, `counts` are 0, and `output` has the Unit value, whereas all Events have no value.

Edges represent the dependency relations between reactives, pointing in the direction of the reactive data and control flow. I.e., an edge $\texttt{r} \underset{DG}{\longrightarrow} \texttt{d}$ corresponds to the user-defined computation of the derived reactive `d` depending on the value of reactive `r`. The *DG* topology always corresponds to the *current* state of dependency relations, meaning if dependencies are discovered (dropped), corresponding edges are added (removed). The *DG* topology therefore changes with dynamic dependency changes. In Figure 2.1,

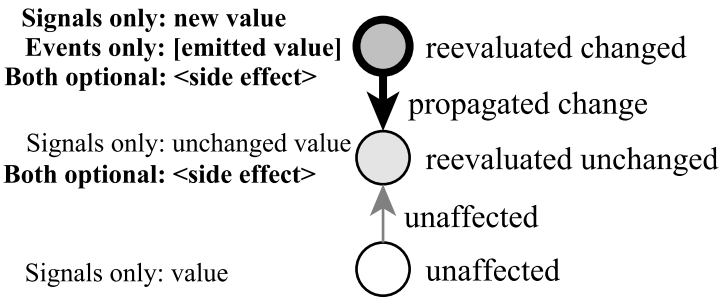



**Figure 2.2:** Change Propagation Visualization Legend

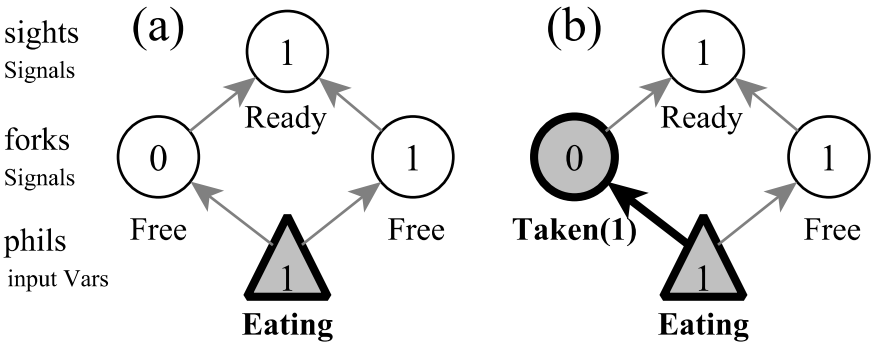



**Figure 2.3:** Elementary Change Propagation

all edges between any `forks` and `sights` with the same index are dynamic and will be repeatedly removed/added throughout the application's lifetime, all others edges are static.

### 2.2.2 Synchronous Change Propagation

When a RP application starts, it first instantiates all its reactives. All Events have no value, and all Signals are created with their value up-to-date in terms of their defining computation over all dependencies' up-to-date values. The application is said to be *quiescent* (no changes are being propagated), and the *DG* in a *consistent resting state*.

```
1 val idx = 1
2 while(System.in.available == 0) {

3   if (sights(idx).now == Ready)
4     phils(idx).set(Eating)

5   if (sights(idx).now == Done)
6     phils(idx).set(Thinking)
7 }
```

**Listing 2.8:** Driving Thread.

Once some inputs change from having new values applied imperatively (`set(...)` and `fire(...)` calls), the consistent resting state is broken in that some derived reactives' values become outdated. E.g., consider an imperative thread that executes the loop in Listing 2.8, which continuously tries to switch `phils(1)` between `Thinking` and `Eating`. Figure 2.2 shows how we use shading and bolding to visualize change propagation on the dependency graph. Figure 2.3 (a) shows a section of the dependency graph in Figure 2.1 after `phils(1).set(Eating)` has changed `phils(1)` to `Eating`. The respective *DG* node is shaded gray and has a bold outline, with the changed value in bold font.

In response to an input change, the RP system reevaluates each affected derived reactive, in order to get its value up-to-date with respect to the changed input's new value. Each reevaluation re-executes the derived reactive's defining computation, which may or may

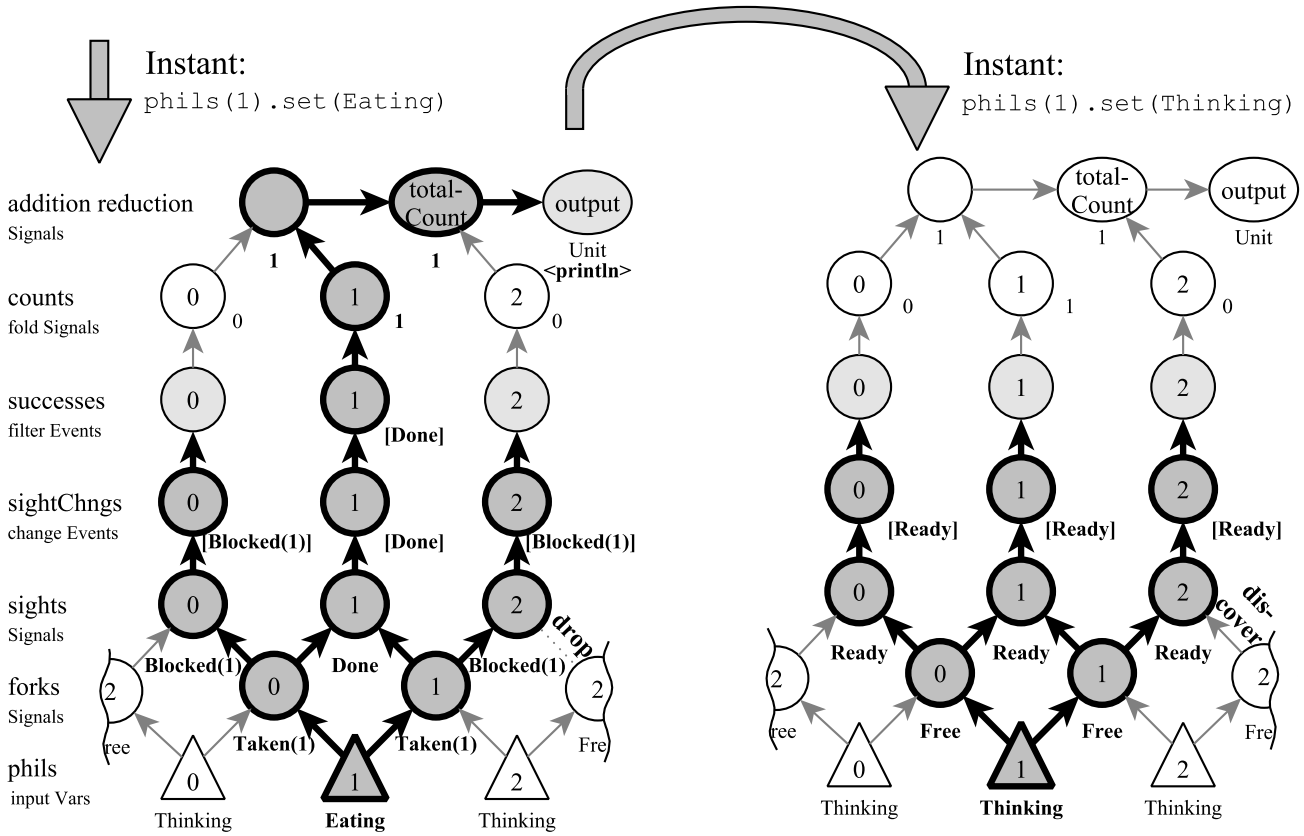



**Figure 2.4:** Two Synchronous Philosopher Instants

not change both its dependencies and its value. E.g., in Figure 2.3 (b), the input change of `phils(1)` has propagated to `forks(0)`, visualized by the bold dependency edge, but not to `forks(1)` yet. `forks(0)` in response reevaluated (shaded gray) and changed its value to `Taken(1)` (bold outline and bold value), while its dependencies remained unchanged. If a reevaluated reactive's value changed, this causes further derived reactives to become outdated and thereby recursively prompts further reevaluations. This means, reactives can be transitively affected by the same input change via multiple paths in the *DG*, and thus may have multiple causes and possible timings for reevaluation.

To provide deterministic and easy-to-use semantics, RP implements *synchronous* change propagation. The process of applying an imperative change to an input reactive together with executing all reevaluations that this change triggers, including transitive ones, is called an *instant* and appears to instantaneous. From the outside perspective of the imperative thread admitting the input change, this means that any `.set(...)` or `.fire(...)` call returns only after all triggered reevaluations have been completed. More importantly though, from the inside perspective of a user computation that executes because the reactive that it defines is reevaluated during an instant, any other reactives whose values it reads either already have been or will not be reevaluated during the instant.

For Events, synchronous instants introduces *simultaneity* in that all Events that fire during an instant fire simultaneously. Application developers thus need not worry about event timings when implementing a reactive that depends on multiple events: There will only be a single reevaluation, during which it will see for all Events it depends on, if and which value each fires during the instant. This also gives rise to deterministic for, e.g., a "xor" Event combinator, where a derived Event fires only if one of two but not both dependency Events fire within the instant of a single imperative input change.

For Signals, an equivalent simultaneity is introduced. Depending on multiple Signals means, that during a reactives' single reevaluation in an instant, all dependency Signals' values are already completely updated. Application developers thus also need not worry about potentially observing some mix

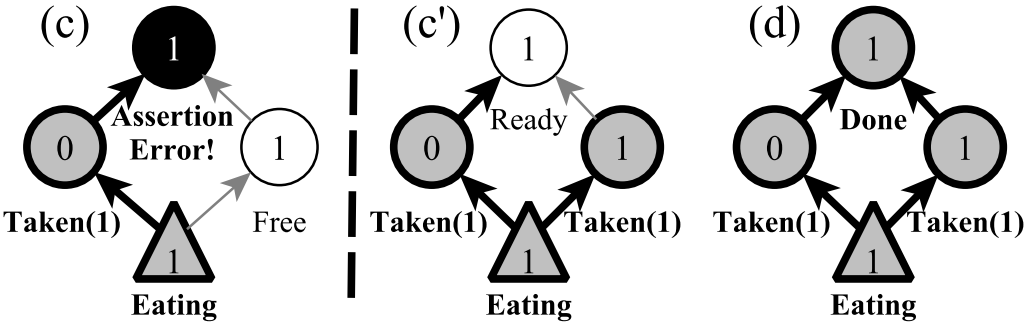



**Figure 2.5:** Glitched vs. Glitch-Free Reevaluation Order

of partially outdated values. All accessed Signals simply have always been updated already. Repeated reevaluations of Signals (without folding) thus behave identically as if the respective Signal is newly instantiated with the rest of the application in a different consistent resting state.

Lastly, synchronous instants provide straight-forward semantics also for Event/Signal conversions (e.g., `changed` or `fold`). A Signal derived from an Event or vice versa is reevaluated in the same instant. Converting a Signal to an Event and back to a Signal yields the same simultaneity as the original Signal. Events and Signals simply follow the same notion of simultaneity. This also provides well-defined semantics for derived reactives that combine Event *and* Signal dependencies.

In summary, synchronous propagation semantics mean that the lifecycle of RP applications can be thought of as a series of instants, each taking the application from one quiescent consistent resting state to the next, with updated values and updated *DG* topology. Figure 2.4 shows the results of two successive instants, setting `phils(1)` to Eating and back to Thinking, based on the application's initial state from Figure 2.1. Note in particular the dependency edge between `forks(2)` and `sights(2)` being first dropped and then discovered again.

### 2.2.3 Liveness and Glitch Freedom

The most fundamental property for any reactive system is that once no more input changes are admitted, i.e., if the application becomes quiescent, then the *DG* must eventually reach the next consistent resting state. This property is called *liveness*: if any reactive's value changed, all other reactives that depend on it must be reevaluated before the application becomes quiescent. Achieving synchronous execution from the outside, i.e., from the perspective of the imperative thread or other instants, is equivalent to liveness in single-threaded RP. The propagation algorithm can only execute reevaluations while it has control, i.e., during a `set` or `fire` call. To ensure liveness, it must therefore complete all reevaluations before returning control to the imperative thread. The imperative thread then can only execute `now` reads during quiescent consistent resting states, and cannot observe incomplete change propagations. Further, only a single instant can execute at any point in time, meaning also no instant can observe incomplete change propagation of another.

Achieving synchronous execution from the inside, i.e., from the perspective of user computations during reevaluations, is done through *glitch freedom* consistency, i.e., the absence of *glitches*. A glitch is defined as a user computation observing partially updated state of an active change propagation, i.e., a user computation that reads the value of two or more dependencies, one of which has already been changed by an instant, and another that has not but still will be changed later by the same instant. Figure 2.5 (c) visualizes an exemplary glitch as the next progression step from Figure 2.3 (b). The change of `forks(0)` has propagated to and triggered a reevaluation of `sights(1)`. The user computation of `sights(1)` then jumps into the case of `forks(0)` being `Taken(1)` by the philosopher himself (Line 23 in Section 2.1) and executes the associated assertion (Line 24). This assertion is based on the invariant

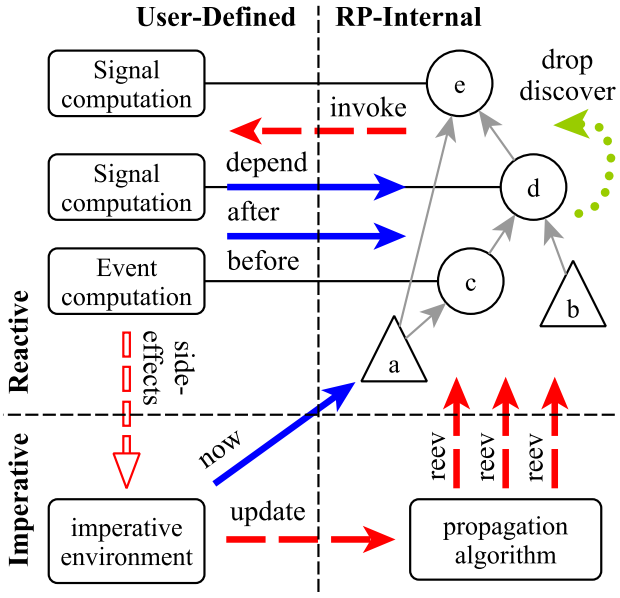



**Figure 2.6:** Operations between RP Components

that all philosophers always hold either both or neither of their two forks. In all possible consistent resting states of the application (except those where two neighboring philosophers eat at the same time and cause the fork to throw an error, which is not the case here), this assertion holds. During this reevaluation here though, the assertion fails, because the change propagation is incomplete with `forks(1)` still having the outdated value `Free`.

Glitch freedom by itself is a filtering problem: at every potential trigger for a reevaluation, execute a reevaluation only if no glitch will occur, i.e., only if all dependencies' current values are part of a single consistent resting state. With liveness though, a potential reevaluation that is not executed because it would not be glitch-free must not be discarded, but instead becomes pending to be executed at a later point in time. Glitch freedom with liveness can thus be interpreted as an ordering problem instead of a filtering problem: given a set of pending reevaluations, compute an order such that after all preceding reevaluations were executed, the next one in the order is guaranteed to be glitch-free. Single-threaded RP systems have traditionally chosen this latter approach. They can always execute only one reevaluation at a time, even if multiple ones are pending, and therefore must maintain a queue of the remaining pending reevaluations. Glitch freedom is then achieved by keeping this queue sorted to produce a glitch-free reevaluation order.

Any topological order of those reactives in the *DG* that are reachable by a changed input is sufficient to guarantee both glitch freedom and liveness in single-threaded RP. The glitch in Figure 2.5 (c) occurred because all visualized reactives are affected by the change of `phils(1)` and the exemplary reevaluation order is inconsistent with the *DG* topology: `sights(1)` depends on both `forks(0)` and `forks(1)`, meaning any topological order must order `sights(1)` after both forks, but it was reevaluated before `forks(1)`, and thus the glitch occurred. If – instead – the topological order is respected, by first reevaluating `forks(1)` (Figure 2.5 (c')) and only then `sights(1)` (Figure 2.5 (d)), no glitch occurs. Lastly, note that propagation algorithms must compute reevaluation orders ad-hoc also at run-time. Since dynamic dependency changes modify the *DG* topology as part of regular change propagation, any reevaluation orders pre-computed at compile-time might become incorrect.

| Available to / Called by | API method | Description |
|---|---|---|
| Imperative Environment | `s.now` | Retrieve the current value of Signal `r`. |
| | `i.set(v)`, `i.fire(v)` | Shorthand for `update(i -> v)` |
| | `update(`$i_1$ `->` $v_1$`,` $i_2$ `->` $v_2$`,` $i_3$ `->` $v_3$`, ...)` | Synchronously emit/change all values $v_j$ from input Event/Signal $i_j$ and subsequently reevaluate all dependent (cf. `r.depend`) Events and Signals. |
| Signal/Event Computations | `s.before` | Read the value of Signal `s` from before the current update changed it. |
| | `r.after` | Read the glitch-free value of Event/Signal `r`, i.e., from after the current update changed `r`, or no value/the value from before if `r` is unaffected. |
| | `r.depend` | Depend on the value of Event/Signal `r`, i.e., read glitch-free value (cf. `r.after`) and register to be recomputed upon the next emission/change of `r`. |
| Propagation Algorithm | `r.reevaluate` | Update value of `r` by re-executing its user computation, return all derived reactives currently registered on `r` for reevaluation together with whether or not `r` changed. |
| Derived Reactives `d` | $comp_d$`.invoke` | Execute the user-defined computation of `d` to, if Event, possibly return a value to be emitted by `d`, or if Signal, return a new (possibly changed) value for `d` |
| | `r.discover(d)` | Register `d` to be reevaluated upon subsequent changes of the value of `r`. |
| | `r.drop(d)` | Unregister `d` to no longer be reevaluated upon subsequent changes of the value of `r`. |

**Table 2.1:** User-Facing (Top Half) and Internal (Bottom Half) Operations of Reactive Programming

## 2.3 Anatomy of RP Applications

To systematically study the effects of concurrency and distribution on RP, this section presents in detail the anatomy of a RP system – all components and all operations through which they interact to implement reactive change propagation. The anatomy of a RP system is depicted in Figure 2.6, with all components represented by boxes. The "imperative environment" in the lower-left corner represents user-defined threads and regular variables of the host programming language (e.g., the loop in Listing 2.8). The "event/signal computations" in the top-left corner represent the user-defined computations of derived reactives (e.g., all code from Section 2.1). The *DG* of all corresponding reactives and their dependency relations is depicted in the top-right corner. Finally, the propagation algorithm is shown in the lower-right corner.

The arrows in Figure 2.6 visualize all operations through which the components interact. These operations are the formal foundation on which concurrency scheduling will later be built. We therefore show how all RP features discussed so far map to these operations and explain each of them in detail, including which data they access and how this data is stored in the *DG* nodes. For later reference, Table 2.1 provides a brief summary of each operation. We distinguish three categories of operations, which we visualize through different kinds of arrows:

- reading the values of reactives (solid blue arrows),
- propagating changes (dashed red arrows), and
- performing dynamic dependency changes (dotted green arrows).

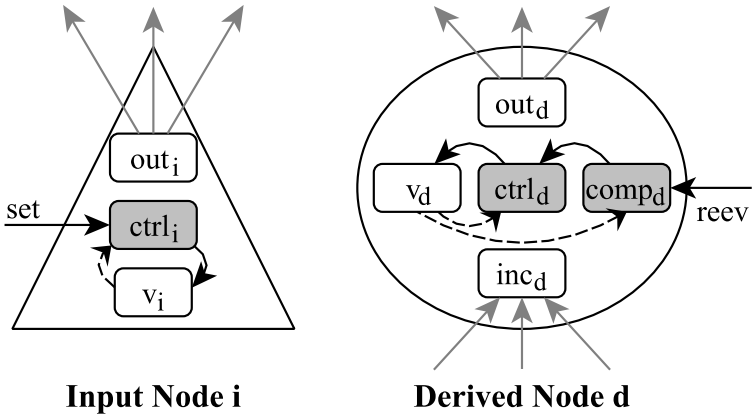



**Figure 2.7:** Composition of Reactives as *DG* Nodes

### 2.3.1 Operations for Reading Reactives' Values

Operations for Reading Reactives' Values are `now`, `after`, `before`, and `depend`, visualized as solid blue arrows in Figure 2.6. The simplest one is `s.now` (`s` is a Signal), which reads and returns the current value of `s`. When a derived reactive `d` reevaluates, its user computation, $comp_d$, may also read the value of other reactives `r`, which thereby become its parameters. Reading parameters happens through `r.before` or `r.after`, instead of `r.now`, because `r.now` is ambiguous in this context. If `r` is also affected by the change propagation that reevaluates `d` and no (transitive) dependency between `d` and `r` exists, `r` may be reevaluated before or after `d`, i.e., `r.now` may return the value of `r` before or after it is reevaluated. In most cases, $comp_d$ reads the glitch-free value of its parameters through `r.after` – these calls may possibly suspend until `r` is glitch-free.

The simplest use of `before` is in a folding Signal `f`: To compute the updated value of `f`, $comp_f$ must first read the old value of `f` through `f.before`. Folding Signals thus implement self-loops on individual *DG* nodes. Calling `before` on other Signals further down the *DG* topology generalizes this pattern to "backwards edges" that close cycles across multiple reactives (cf. `last` keyword in [Sawada and Watanabe, 2016]). But, there are also use cases for `before` not related to *DG* cycles. E.g., a `submit` Event in a chat GUI reevaluates the `text` input field to remove the sent text, but simultaneously needs to emit a `message` Event to the server with the value of `text` from *before* it was cleared. On Events `e`, only the `e.after` operation is available for reading their value. This is because Events only have an emitted value after they were reevaluated, but not before (`before`) and not outside of change propagation (`now`).

Most parameter reads inside any $comp_d$ are `r.depend`. E.g., all reads in all $comp_d$ in Section 2.1 are `r.depend`, except for the folding `counts(i).before` self-call. This includes all internally executed reads by any `changed`, `filter`, `map`, etc. derivations. The `r.depend` operation behaves identical to `r.after`, but additionally registers `d` for reevaluation upon changes of `r`. The choice between `r.depend` and `r.after` thus grants all $comp_d$ fine-grained control over which `d` are dependencies or just parameters, i.e., which values' changes do or do not trigger the next reevaluation of `r`.

### 2.3.2 Operations for Change Propagation

Operations for implementing change propagation through the *DG* are `update`, `reevaluate`, and `invoke`, visualized as solid red arrows in Figure 2.6. Change propagation is always initiated by an imperative `update(`$i_1$ `->` $v_1$`,` $i_2$ `->` $v_2$`, ...)` call. Calls `var.set(v)` or `evt.fire(v)` are shorthand for `update(var -> v)` or `update(evt -> v)`. `update(...)` calls do not directly update the values of the given input reactives, but are dispatched through the propagation algorithm. The latter translates them

```
1  procedure DerivedReactive.reevaluate(reactive, instant):
2    let (userValue, inc) = shepherdExecution(instant){
3      # shepherding the reactive's reevaluation means:
4      # a) ensure that any before/after/depend call by the user computation
5      #    is executed with the correct instant context parameter,
6      # b) record the set of reactives on which the user computation called
7      #    depend, and return it together with the user computation result.
8      invoke comp_reactive
9    }
10   execute updateDeps(reactive, instant, inc)
11   dispatch control(reactive, instant, userValue)

12 procedure Event.control(event, instant, userValue):
13   # userValue is optional None or Some(value).
14   execute reevOut(event, instant, userValue)

15 procedure Signal.control(signal, instant, userValue):
16   if(userValue == execute before(signal, instant)):
17     execute reevOut(signal, instant, None)
18   else:
19     execute reevOut(signal, instant, Some(userValue))

20 procedure DerivedReactive.updateDeps(reevaluating, instant, inc):
21   foreach dep ∈ (inc \ inc_reactive):
22     execute discover(dep, instant, reactive)
23   foreach dep ∈ (inc_reactive \ inc):
24     execute drop(dep, instant, reactive)
25   update inc_reactive := inc
```

**Listing 2.9:** Infrastructure for Reevaluations.

into a series of `reevaluate` calls on derived reactives to propagate the input changes glitch-free through the entire application. The execution of each `reevaluate` involves the respective *DG* nodes' values and variables, which are depicted as grey boxes (values) and white boxes (variables) in Figure 2.7.

The propagation algorithm treats all reactives through a common interface. Any reactive can be reevaluated, and responds by reporting as having either changed or not changed, and returning the set of derived reactives that that currently depend on it, i.e., the set of reactives that potentially reevaluate next due to this change or no-change. The only minor difference is that input reactives' reevaluations are given a value imperatively, while derived reactives' reevaluations compute this value themselves. For concurrency scheduling, though, a thorough understanding also of how Events and Signals process values behind this common interface is necessary, too. Listing 2.9 shows pseudocode for both the common and the specific aspects of reactives' reevaluations.

Input reactives `i` (depicted on the left of Figure 2.7) consist of variables $v_{\texttt{i}}$ and $out_{\texttt{i}}$, and *control code* $ctrl_{\texttt{i}}$. $v_{\texttt{i}}$ holds the emitted/stored value of `i` and $out_{\texttt{i}}$ its *outgoing dependencies set*, i.e., the set of derived reactives currently registered on `i` for reevaluations. Initially, the propagation algorithm reevaluates each input reactive `i` in `update(..., i -> v, ...)`, by passing the corresponding `v` to $ctrl_{\texttt{i}}$. For Events, $ctrl_{\texttt{i}}$ (Line 12) stores every given `v` as the emitted value $v_{\texttt{i}}$ and reports `i` as changed. For Signals, $ctrl_{\texttt{i}}$ (Line 12) stores `v` as $v_{\texttt{i}}$ only if it is different from the previous $v_{\texttt{i}}$ (`before` self-call in Line 16, visualized in Figure 2.7 by the thin dashed arrow from $v_{\texttt{i}}$ to $ctrl_{\texttt{i}}$). Together with the whether or not `i` changed, $ctrl_{\texttt{i}}$ returns $out_{\texttt{i}}$ to the propagation algorithm. If `i` changed, the propagation algorithm must eventually reevaluate each reactive in $out_{\texttt{i}}$, once it can guarantee glitch-freedom. If `i` did not change, reevaluations of reactives in $out_{\texttt{i}}$ triggered by other changes no longer need to wait for `i`.

Derived reactives `d` (depicted on the right of Figure 2.7) consist of the same building blocks as input reactives, plus the user computation $comp_{\texttt{d}}$ and variable $inc_{\texttt{d}}$ that stores the *incoming dependencies set* – the set of reactives on which `d` is currently registered for reevaluation.[4] Derived reactives and Signals use the same control codes $ctrl_{\texttt{d}}$ as input reactives. However, the value `v` passed to $ctrl_{\texttt{d}}$ is not taken from the `update(...)` call's parameters, but computed by `reevaluate` as the return value of (re-)executing $comp_{\texttt{d}}$ (Line 1). While $comp_{\texttt{d}}$ executes, the set of reactives `r` on which it calls `r.depend` is recorded (called `inc` throughout Listing 2.9). After $comp_{\texttt{d}}$ returns, but before the returned `v` is passed to $ctrl_{\texttt{d}}$, `d` updates its dependency edges in the *DG* to match `inc` (Line 10). The edges to add, respectively remove, are computed as the difference between $inc_{\texttt{d}}$ and the newly recorded set of `r.depend` reactives. If $comp_{\texttt{d}}$ executed `r.depend`, i.e., $\texttt{r} \in \texttt{inc}$, but $\texttt{r} \notin inc_{\texttt{d}}$ (Line 21), `d` executes `r.discover(d)`. If $\texttt{r} \in inc_{\texttt{d}}$, i.e., $\texttt{r} \notin \texttt{inc}$, but $comp_{\texttt{d}}$ did not execute `r.depend` (Line 23), `d` executes `r.drop(d)`. Afterwards, `inc` is written as the new $inc_{\texttt{d}}$ (Line 25). Only then, the execution of `reevaluate` is finally completed by passing the `v` returned by $comp_{\texttt{d}}$ to $ctrl_{\texttt{d}}$ (Line 11).

### 2.3.3 Operations for Dynamic Dependency Edge Changes

Operations for dynamic changes of *DG* edges are `r.discover(d)` and `r.drop(d)`, visualized as dotted green arrows in Figure 2.6. They are executed by one reactive on another reactive inside the *DG*, and are thus visualized as a loop between *DG* nodes. The operation `r.discover(d)` updates $out_{\texttt{r}} := out_{\texttt{r}} \cup \{\ \texttt{d}\ \}$, which discovers edge $\texttt{r} \underset{DG}{\longrightarrow} \texttt{d}$, thereby registering `d` for reevaluation upon changes of $v_{\texttt{r}}$. The operation `r.drop(d)` vice versa updates $out_{\texttt{r}} := out_{\texttt{r}} \setminus \{\ \texttt{d}\ \}$, which drops edge $\texttt{r} \underset{DG}{\longrightarrow} \texttt{d}$, thereby un-registering `d` to no longer reevaluate upon changes of $v_{\texttt{r}}$.

[4] The sets $inc_{\texttt{d}}$ and $out_{\texttt{r}}$ mirror each other in that every edge $\texttt{r} \underset{DG}{\longrightarrow} \texttt{d}$ corresponds to $\texttt{d} \in out_{\texttt{r}}$ and $\texttt{r} \in inc_{\texttt{d}}$.

# 3 Introducing Concurrency and Distribution

Section 3.1 discusses two ways in which RP interacts with concurrency: concurrent execution of multiple independent reevaluations within single instants, and concurrent execution of multiple instants. Both dimensions are orthogonal: both have different challenges and solutions, and RP systems can support neither, either individually, or both of them together. Section 3.2 discusses, how lack of shared memory affects RP. In particular, distributed applications require support for both of dimensions of concurrency, albeit for entirely different reasons. Therefore, having first a detailed understanding and proper handling of concurrency is indispensable for providing distributed RP.

## 3.1 Implications of Concurrency

Concurrent execution of multiple independent reevaluations within single instants refers to making each instant internally a multi-threaded process. If two reactives must reevaluate within the same instant and do not transitively depend on each other in either direction, they can reevaluate in either order, or even at the same time, without violating glitch freedom. In the context of using a topological reevaluation order to achieve synchronous propagation, the opportunities for parallelizing independent reevaluations appear in the form of ambiguities in the topological order of the *DG*. For instance, changing one philosopher's input Var to `Eating` requires two `forks` to reevaluate next. All `forks` are independent of each other's values. Single-threaded topological orders can order either fork first, i.e., either fork can reevaluate first without causing a glitch. In multi-processor systems, reevaluating both `forks` at the same time becomes a third option, that also does not affect cause a glitch, i.e., retains synchronous propagation semantics.

Concurrent reevaluations per instant is a *voluntary* feature in the sense that this kind of concurrency does not appear by itself. Each `update(...)` call comes from and executes in a single imperative thread from the user application initially. For independent reevaluations to execute in parallel, the propagation algorithm must intentionally be implemented to off-load pending reevaluations into a worker thread pool for multi-threaded processing. The RP anatomy (Figure 2.6) is not affected by this change – the only difference is that some of the multiple *reev* calls depicted therein now execute in parallel on different threads, instead of all sequentially on the same thread.

Parallelization of independent reevaluations is also comparatively easy to implement correctly. The only race conditions it introduces are multiple reactives that complete reevaluations at the same time. These may cause a single derived reactive to be assessed for whether or not it is ready for a glitch free reevaluation by multiple worker threads concurrently. To retain correct semantics, only one of these threads must actually execute a reevaluation if the reactive is found to be ready. Since reevaluations are distributed across arbitrary worker threads already though, it does not matter which thread this will be. These race conditions therefore have commutative solutions, and thus are easily solvable ad-hoc with localized case-by-case synchronization.

Concurrent instants on the other hand pose a much more complex problem. First, concurrent instants are not a voluntary feature that RP libraries can choose to implement to some degree that is still easy to handle. Instead, full support for concurrent instants is mandatory for any RP library to be used in a multi-threaded application, because such an environment inevitably imposes this concurrency onto the library. Figure 3.1 depicts the RP anatomy as in Figure 2.6, but in a multi-threaded instead of single-threaded environment. In general, when a RP system is embedded into a multi-threaded environment, an arbitrary number of imperative threads can concurrently read Signals (`now`) and admit input changes

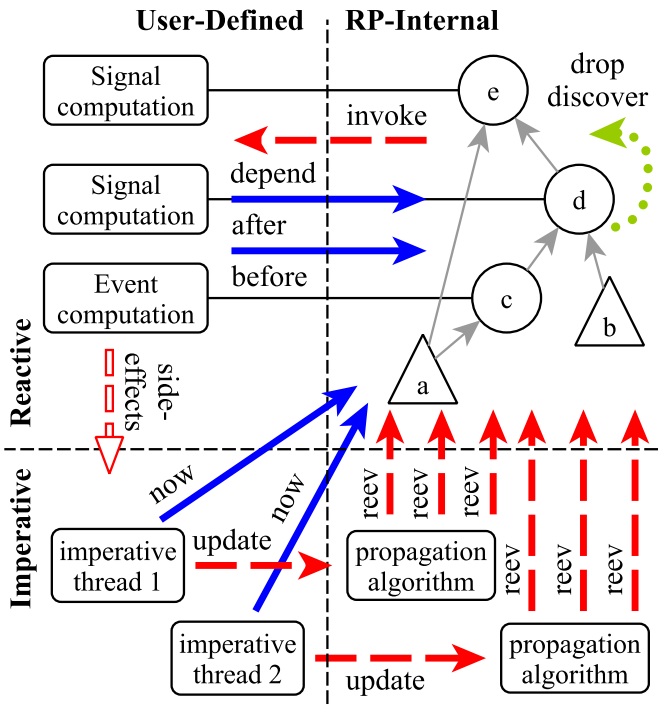



**Figure 3.1:** Anatomy of Multi-Threaded RP

(`update(...)`). Concurrent input changes result in concurrently executing instants, which execute concurrent reevaluations (`reev`), which in turn concurrently update and read (`before`, `after`, `depend`) values and dependency edges (`drop`, `discover`), that *can* depend on each other. Concurrent instants therefore result in race conditions that involve any of the operations introduced in the previous chapter, and that generally do not have commutative solutions. Moreover, yet more complex issues arise when multiple concurrent instants have more than one reactive in common, which they will reevaluate. If they reach and reevaluate one of these in a different order than the other, it can become impossible to execute glitch-free reevaluations for yet other reactives later down the *DG* topology.

To provide a systematic analysis of these race conditions, Section 3.1.1 introduces concurrent instants to the philosophers application from Section 2.1. It will exemplify complex race conditions and ordering ambiguities between operations of concurrent instants, and derive increasingly complex restrictions on their interleaving that are necessary to achieve thread-safety under concurrent instants. Next, Section 3.1.2 will introduce terminology from concurrency and database communities for formally describing, discussing and evaluating such thread-safety. Finally, Section 3.1.3 presents a formal correctness definition for synchronous propagation semantics of RP in a multi-threaded environment, i.e., with concurrent instants.

### 3.1.1 Race Conditions between RP Operations

To introduce concurrent instants into the philosophers running example, consider one instance of the imperative driver thread from Listing 2.8 to execute per philosopher on a given table. We first discuss race conditions between concurrent operations on the same reactive (i.e., same *DG* node). Consider two threads, each executing an `Eating` instant on a philosopher, from the applications initial state (i.e., all `counts` are 0). Assume they race such that each increases the value of one of the two dependencies of the `totalCount` Signal to 1, and then both simultaneously reevaluate `totalCount`. Both reevaluations read both updated inputs, and thus both compute the latest value for `totalCount` as 2. Both simultaneously read the previous value of `totalCount` as 0, and thus both update it to 2 and both report `totalCount`

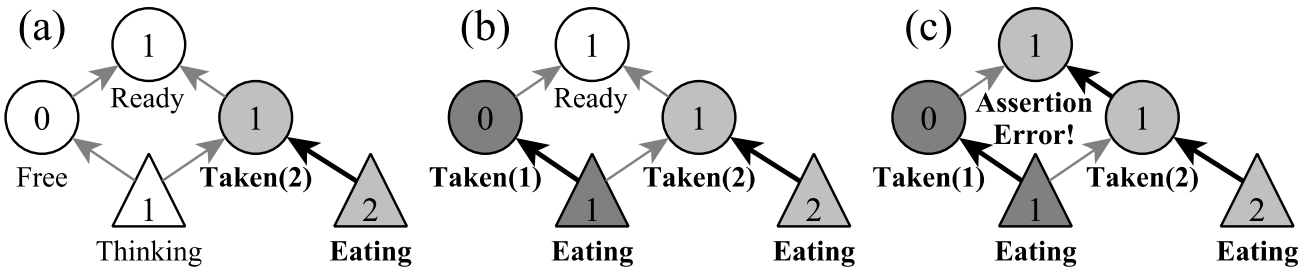



**Figure 3.2:** A Glitch from Concurrent Propagations Interacting

to have changed. In response, both reevaluate `output`, and thus the application outputs 2 as the new total `Eating` count twice – a result that would not be possible under synchronous execution. This very scenario can play out this way even if both instants follow a topological reevaluation order in an attempt to achieve glitch freedom and thereby also synchronous propagation semantics. It shows that the established single-threaded approach for synchronous propagation semantics is insufficient under multi-threading.

The cause of the exemplified problem is a race condition between both instants concurrently executing a compare-then-update on $v_{\texttt{totalCount}}$. To prevent this, such compare-then-update operations on nodes' values must execute mutually exclusively. Note that, e.g., folding Signals generalize this problem, in that they read the node's own value (`before` self-call) already at the very beginning of its reevaluation. Thus, reevaluations as a whole must execute mutually exclusively on each node. With mutually exclusive reevaluations, the second instant's reevaluation of `totalCount` would always read the previous value as 2, thus determine the node as unchanged, and thus the new value 2 would correctly be printed only once instead of twice.

Mutually exclusive reevaluations would only resolve half of the problem at hand, though. With synchronous execution, the application would always first output a new `totalCount` of 1, and not jump straight to 2. But, in the concurrent example this intermediate step is skipped, because the first instant's reevaluation already encompassed both instants' changes instead of just its own. Some may consider this issue harmless in this particular instance, but taking into account its more general implications shows that it is not. If `totalCount` were an Event instead of a Signal, this same issue would result in the Event spuriously skipping some values, which is not tolerable. Moreover, in more complex situations even with only Signals, other instants' changes may be observed only partially, which may cause glitches even if all instants by themselves follow a glitch-free reevaluation order. Figure 3.2 demonstrates this, showing mutually exclusive reevaluations per node with glitch-free reevaluation orders are insufficient to achieve even only glitch freedom, let alone synchronous execution.

The graph section shown in Figure 3.2 is the same as in Figure 2.5, but with `phils(2)` included additionally. In (a), thread $T_2$ executes `phils(2).set(Eating)` and reevaluates `forks(1)` to `Taken(2)` (nodes shaded light grey). In (b), thread $T_1$ reads `sights(1)` as `Ready`, changes `phils(1)` to `Eating`, and reevaluates `forks(0)` to `Taken(1)` (nodes shaded dark grey). $T_1$ has not yet reevaluated `forks(1)`, hence `sights(1)` is not ready for a glitch-free reevaluation. In (c), $T_2$ propagates its change of `forks(1)` to `sights(1)`. From the local perspective of $T_2$, both predecessors of `sights(1)` are now glitch-free: `forks(0)` is not affected by the propagation of $T_2$, and the reevaluation of `forks(1)` is already completed. Thus, unaware of the (partial) changes by $T_1$, $T_2$ deems `sights(1)` ready for a glitch-free reevaluation. But this is incorrect, because the changes by $T_1$ are incomplete, and the reevaluation produces the same glitch we demonstrated in Figure 2.5 (`sights(1)` fails its assertion).

Given that instants individually executing reevaluation in a glitch-free order is insufficient to achieve glitch freedom overall, one might attempt a more comprehensive approach that ensures glitch freedom as a global property shared across concurrent instants. E.g., vector clocks can be used, based on one

logical clock per input reactive, to defer reevaluations of derived reactives until all their dependencies' latest values are associated with the same timestamp for all sources that they (transitively) depend on [Proença and Baquero, 2017]. Doing so prevents the glitched reevaluation shown in Figure 3.2 (c), because `forks(0)` and `forks(1)` disagree on the timestamp of the `phils(1)` input Var. Instead, `sights(1)` can reevaluate only after all changes of both $T_1$ and $T_2$ to both `forks(0)` and `forks(1)` have been completed. Such an algorithm ensures glitch freedom across concurrent instants, but this is still insufficient to ensure synchronous propagation semantics: Every executed reevaluation is glitch-free, but fewer reevaluations are executed. Namely, in the example, `sights(1)` is affected by two instants, but only reevaluated a single time for both of their changes. As a more drastic example, in very "busy" applications, some reactives may never observe a glitch-free snapshot of all their dependencies, because new instants keep introducing new timestamp discrepancies before the previous instants progressed enough to resolve theirs. Reactives may therefore skip any number of reevaluations, i.e., suspend updating for any length of time. This shows that glitch freedom in a multi-threaded environment no longer implies synchronous propagation semantics, but yields only a form of eventual consistency where reactives are only guaranteed to become updated once the application becomes quiescent.

The reason why glitch freedom implies synchronous propagation semantics in single-threaded applications is, that outside synchronous appearance of instants comes for free in single-threaded applications, and glitch freedom only completes the picture by filling in inside synchronous appearance. In multi-threaded environments, this implication breaks down because outside synchronous appearance is lost, meaning implementing glitch freedom as a shared property of concurrent instants is a lost cause when aiming for synchronous propagation semantics. Instead, traditional approaches for glitch freedom can be reused, but only after first ensuring outside synchronous appearance by isolating concurrent instants against each other, i.e., implementing some form of mutual exclusion between reevaluations across multiple reactives. Unlike for reevaluations on individual reactives though, ad-hoc mutual exclusion across multiple reactives is infeasible, because it can result in deadlocks. Figure 3.3 demonstrates this on a table of only (for simplicity) two philosophers and forks. The associated threads $T_0$ and $T_1$ simultaneously `set(Eating)` (nodes are shaded light grey for $T_0$ and dark grey for $T_1$), and propagate this to their respective left fork. Each fork should reevaluate, but each already has one of its dependencies reevaluated by one instant, but the other by the other instant. Thus it is already too late to establish any kind of mutual exclusion – the two instants are deadlocked.

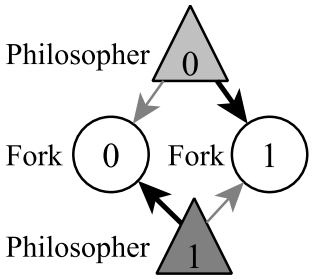



**Figure 3.3:** A Deadlock

It is common practice in, e.g., databases, to work around deadlocks by aborting one of the involved parties and undo its changes, which then allows some other parties to proceed, and restart the aborted execution from the beginning. Due to side-effects in reactives' user computations though, which we cannot undo, this workaround is also is inapplicable here. Thus the only remaining alternative is to establish necessary synchronization between reevaluations of different reactives by different threads pre-emptively, before any are executed, to ensure that deadlocks do not occur in the first place.

### 3.1.2 Established Thread-Safety Properties

Desiring processes of multiple individual operations to appear as atomic, i.e., as if all their operations occurred at once at a single point in time, is a common occurrence in concurrent systems. As a result, several established properties and tools exist, to formally define and verify the correctness of implementations that provide such atomicity. The most intuitive of these properties is *linearizability* [Herlihy and Wing, 1990]. Linearizability means that operations take effect atomically at some *linearization point* during their execution. Linearizability is a good fit for describing outside synchronous appearance from

the perspective of imperative threads: Every `n.now` and `update(...)` call should appear to execute atomically at some linearization point between its call and its return, across the entire application.

But, linearizability addresses only individual operations on singular data structures. It is therefore not applicable to discuss synchronous appearance of instants that include the execution of multiple user computations on different reactives. Linearizability matches when considering the entire *DG* of an RP application forms one big data structure. With user computations bound to individual reactives though, users unavoidably are aware that each node in the *DG* is a separate data structure on its own, and that each `update(...)` call involves the execution of mutliple $comp_{\mathrm{d}}$, `s.before`, `n.after` and `n.depend`, all on different reactives. Instead, atomic appearance must be achieved by ensuring that any concurrent threads can only see the state from before or after the entire execution of an instant, but never any intermediate, only partially updated state.

*Transaction processing* [Bernstein et al., 1986] provides such a model. Any number of reads and writes, which should appear as one atomic unit of work, are grouped into one transaction. Applied in the RP setting, every `s.now` call thus is a transaction containing a single read, and every `update(...)` call is a transaction that contains all the reads and writes of all reevaluations executed by the corresponding instant. Transaction processing provides a well-established formal tool set to describe and verify the correctness of transaction systems. During execution, a transaction system produces a *history* – a sequence of all operations of every transaction in the order of their execution. Different levels of correctness of transaction systems are defined in terms of permutations of these histories that a system can produce.

*Serial* histories are those where all transactions execute without interleaving of operations. Single-threaded systems naturally correspond to serial histories, so serial histories are considered the baseline for correctness. This matches the intention of concurrent RP, where single-threaded applications inherently provide outside synchronous appearance of instants. Serial histories very restrictive though, since they do not allow any parallelism. To ensure a serial history for RP applications, instants would have to execute under mutual exclusion, one after the other.

A less restrictive property is built on the notion of *view-equivalence*. View-equivalence is defined over *reads-from relations* of the form "transaction `A` reads the value of variable `x` that was written by transaction `B`". Two histories from the same set of transactions are called view-equivalent, iff they are permutations of each other, but still have identical reads-from relations. View-equivalence implies that all reads of all transactions return the same value, which implies that transactions cannot distinguish between view-equivalent histories. Therefore, any histories that are view-equivalent to serial histories are equally acceptable as correct. Such histories are called *serializable* histories.

Yet, serializability alone is not enough. A `s.now` transaction $t_{now}$ could read a value of Signal `s` that was written by $t_{past}$ half an hour earlier, and was already overwritten shortly after by another $T_{past'}$. This would conform with serializability, as $t_{now}$ can formally be ordered between the $t_{past}$ and $t_{past'}$. Yet, this would confuse users, as one would naturally expect such a read to return a current, up-to-date value of `s`. The foundation for a correctness definition for RP is thus the combination of serializability *and* linearizability, known as *strict serializability*: Transactions must execute their operations in a serializable fashion, and at the same time must have a linearization point within the actual real-time window of their execution.

### 3.1.3 Thread-Safety Correctness

The definition of thread-safety for RP is built the following way: Strict serializability ensures outside synchronous appearance of instants through the combination of linearizability from the perspective of imperative threads, and serializability from the perspective of user computations executing inside of instants. Within this outside synchronous propagation semantics, inside synchronous appearance for user computations within each instant can be achieved as in single-threaded contexts, i.e., through glitch freedom and liveness. In particular, this allows reusing existing propagation algorithms, e.g., ordering reevaluations topologically is again sufficient to achieve inside synchronous appearance. To

further increase flexibility without loss of correctness guarantees, we make the following adaptations to strict serializability, based on various properties of RP.

1. Schedulers usually consider transactions of reads and writes on individual variables. The discussion in Section 2.3 revealed that all variables ever accessed by the RP system are stored inside DG nodes. Moreover, all operations presented there (`now`, `reev`, `before`, `after`, `depend`, `drop` and `discover`) affect variables of single nodes only, and hence are easily executed linearizable per node, e.g., through simple mutual exclusion using Java's `synchronized` scope on the node instance. Thus, we design schedulers in terms of these more complex operations. This allows us to address scheduling solutions at a higher level of abstraction.

2. We require strict serializability only for user-visible operations, i.e., those whose arrows in Figure 2.6 have one end on the left-hand side. Other operations (e.g., `drop` or `discover`) may be executed arbitrarily, as long as they add up such that all user-visible operations are executed correctly. This gives schedulers more freedom to act in order to provide their guarantees, without changing users' experience.

3. While a given propagation algorithm may be able to produce only a subset of all correct reevaluation orders, other reevaluation orders still remain correct. We thus accept any glitch-free order of reevaluations as correct, even if an underlying propagation algorithm is unable to produce this order in a single-threaded setting. This makes it possible to devise and verify the correctness of scheduling algorithms independently of concrete propagation algorithms. Further, it also gives more freedom to schedulers by allowing them to disregard and overrule propagation algorithms' reevaluation orders, as long as they still preserve glitch freedom.

4. Lastly, recall that transactions of RP instants may contain side-effects, and thus cannot be aborted to resolve deadlocks. Because such aborts have become a common, widely accepted, and often even expected practice in transaction systems (e.g., databases, software transactional memory), we consider it necessary to explicitly state *abort-free* as an additional criterion for correctness.

Summarizing, we therefore define *correctness for the execution of concurrent RP instants* as abort-free strict serializability for user-visible operations (`now`, `update`, `reev`, `after`, `before`, `depend`), with reevaluations of every instant being executed in any glitch-free order.

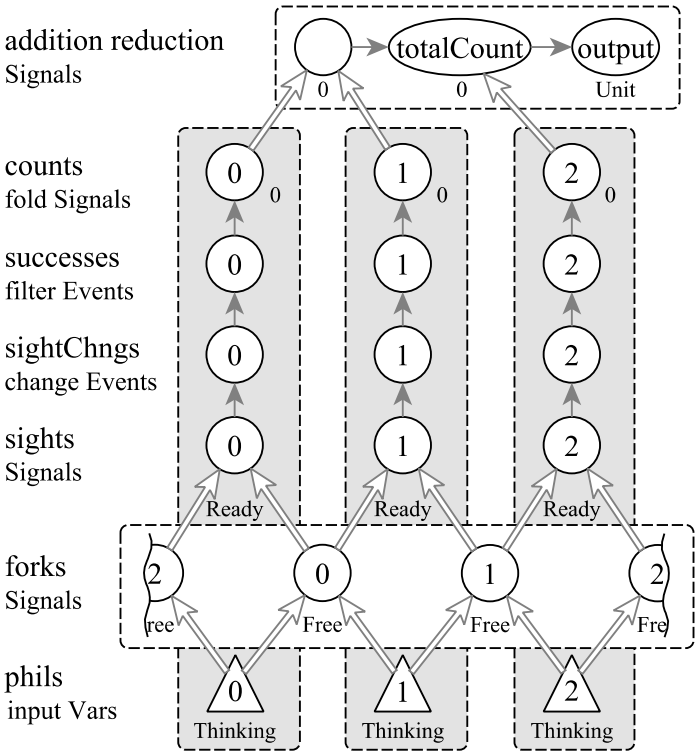



**Figure 3.4:** Distributed Reactive Philosophers

## 3.2 Implications of Distribution

Several reactive systems use distribution as a mostly hands-off feature to provide users with increased throughput, e.g., to support big data processing. They offload parts of their computations into computing clusters, and are thereby able to utilize enormous amounts of processing power. For distributed RP though, we approach distribution not as a feature for scaling performance, but as a necessity instead. Programs and data originate in many different places, and in order for computations to occur, both the program and all the data it needs must come together in the same physical location. Application developers may have reactives on one host, but require their values for computations on different hosts. As with concurrent instants, this means distributed RP cannot choose to support only a limited amount of use cases that are easy to handle, but must fully support all its features across arbitrarily distributed *DG* topologies.

This is not to say that RP systems cannot exploit the idea of off-loading computations into computing clusters to increase their throughput. On the contrary, automated tools for off-loading subsections of the *DG* to computing clusters can be implemented on top of the support for arbitrarily distributed applications. Such tools would have to limit the RP model or statically ensure that only user computations that do not contain side effects are off-loaded, as off-loading side effects would likely break them. Based on this, they would have to identify suitable cuts through the dependency graph such that they off-load computations as optimal for performance improvements as possible. This same technique could also be used to provide automated redundancy for increased fault-tolerance, where one subsection of the *DG* is cloned to multiple remote hosts so that a working instance remains alive and available even if some of these hosts fail or disconnect. But all this is beyond the scope of this dissertation and will be left to potential future work and not discussed further.

Figure 3.4 shows the overlaid *DG* and network topology of a distributed implementation of the philosophers, with separate hosts visualized by boxes with a dashed outline. Each philosopher lives on its own host (grey dashed boxes), containing its "personal" reactives: Its `phils` input var, `sights` Signal,

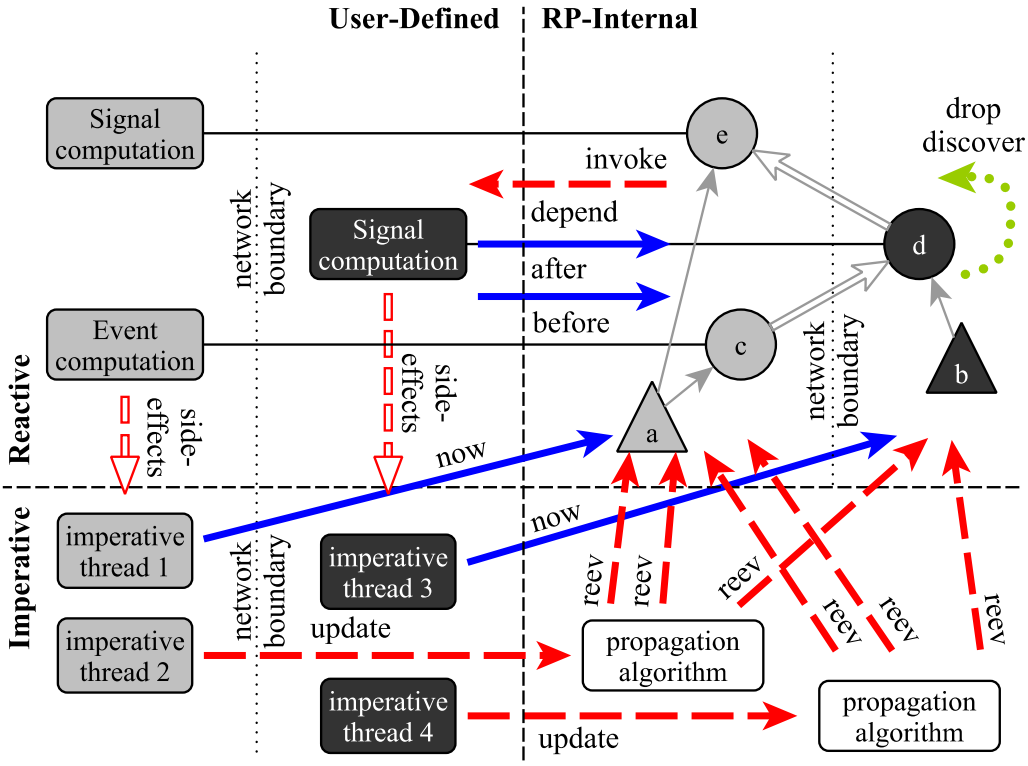



**Figure 3.5:** Anatomy of Distributed RP

`sightChngs` and `successes` Event, and `count` folding Signal. Two additional dedicated network hosts (white dashed boxes) are responsible for all forks that the philosophers share amongst each other, and for keeping track of the sum of all philosophers' counts. Dependency edges that cross network boundaries are emphasized as hollow arrows.

Figure 3.5 shows the effects of distribution on the anatomy of RP applications. Every reactive along with its user-defined computation is placed on some host in the network, visualized through their nodes/boxes being colored in different shades of grey for different hosts. Imperative threads are also placed on some host, equally visualized through shading. Instants on the other hand, despite originating from specific hosts' imperative threads, are not associated with any specific hosts, but transcend network boundaries. They may affect reactives on any host, and their semantic guarantees are upheld across the entire distributed *DG* as a whole.

### 3.2.1 Concurrency in Distributed RP

Support for both dimensions of concurrency – concurrent reevaluations of independent reactives within each single instant and concurrent instants – is essential for distributed RP. To exemplify the need for parallel reevaluations of independent reactives, consider a `phils(1).set(Eating)` instant in the distributed philosophers example. This instant affects `sights(0)`, `sights(1)` and `sights(2)`. All `sights` are independent of each other (no data dependencies between them, topologically unordered) and each is placed on a different host. If the RP system does not support parallel reevaluations of independent reactives, the reevaluations of these `sights` would have to be coordinated in some way. Locally, this would result only in suboptimal use of available processors (there is at least one processor available for each of the `sights`). In this distributed topology though, due to the lack of shared memory between the philosophers' hosts, such coordination would require some form of sequential remote communication where no actual data dependencies exist in the *DG* topology. Network communication is orders of mag-

nitude slower than local-only computations, meaning such unnecessary remote communication would add an immense overhead cost. To avoid overhead cost for synchronizing between reactives that do not depend on each other, full support for concurrent reevaluations of independent reactives within each instant is thus crucial.

To exemplify the need for concurrent instants, observe that the distributed philosophers in Figure 3.4 have each philosopher's input Var placed on the philosopher's own host. Naturally, each host has its own imperative environment. In particular, each philosopher's driving thread from Listing 2.8 executes in this environment on the philosophers' host. This shows that not only do concurrent instants occur in distributed applications, but they may originate arbitrarily from threads on all hosts. While not exemplified by the philosophers example, it is also still possible for each host to be multi-threaded in itself. Concurrent instants therefore interact everywhere throughout distributed RP applications, both within individual hosts and across multiple ones. Distributed RP therefore also requires a correctness condition for concurrent instants.

### 3.2.2 Correctness in Distributed RP

Recall the correctness guarantees to achieve synchronous execution in multi-threaded RP systems are liveness, glitch freedom and isolation in the form of abort-free strict serializability. Each of these guarantees is straight-forward to transfer into a global property across the entire distributed application.

- Liveness: Whenever a reactive changes, all other reactives that depend on it – including those on other hosts that depend on it remotely – must be reevaluated before the instant completes. In the distributed philosophers, the `phils(1).set(Eating)` instant should only complete after all the same nodes as in the local version have reevaluated, regardless of them being distributed across multiple hosts.
- Glitch Freedom: A user computation must never read the value of two or more dependencies, where one has already been changed by an instant but another will still be changed later by the same instant – regardless of which reactive is placed on which host, and which host the instant originated from. As an example, the demonstrated glitch from Figure 2.5 can be constructed identically across the same four reactives in the distributed philosophers application from Figure 3.4, where these reactives are connected only by network edges.
- Strict Serializability: All reads and writes of instants must be orchestrated such that they are equivalent to those instants having executed serially, one after the other, and this order must correspond to the real time of their executions – and must be consistent across all hosts in the network.

With these three guarantees together in a distributed RP application, distributed synchronous propagation semantics are achieved: Each instant appears to be instantaneous *across the entire distributed application*.

The algorithms that provide these consistency guarantees *are* affected by the distributed environment though, in that the *DG* of distributed RP applications becomes a distributed data structure. The *DG* is the foundation for computing liveness and glitch freedom, meaning the propagation algorithms must be suitable to operate on a distributed *DG*. Strict Serializability on the other hand requires a consistent order for instants across all hosts. Since instants can originate from any hosts in distributed RP applications, this order also creates distributed relations between elements from different hosts. The concurrency control algorithms for strict serializability must therefore also be able to operate on distributed data.

All of these algorithms, regardless of whether they are used to provide change propagation semantics or concurrency control, should be decentralized as much as possible, i.e., make as many decisions as possible locally, without having to communicate with other hosts. The previous section already presented concurrent reevaluations of independent reactives within each instant as one example to motivate this. A second possibly even more significant motivation is ease of use. In order for the code quality and

program comprehension benefits of non-distributed reactive programming to apply equally in distributed applications, distributed RP should ideally require no additional code or configuration in order to work correctly. In particular, this means that applications should not have to specify some form master or slave hosts or any kind of hierarchy. Decentralized algorithms, where all hosts are equal and can synchronize to form any required relations on-the-fly, are essential to facilitate this.

Concluding, the challenges of adding distribution on top of multi-threaded RP can be summarized as follows. Unlike multi-threading, distribution does not break any assumptions or guarantees on which synchronous propagation semantics for instants is built. All properties of RP's synchronous execution apply the same way they do in local-only applications. Instead, distribution puts restrictions on how the algorithms that ensure these properties can be implemented. The data structures, which the semantic guarantees of RP are computed from, become distributed, but still must be assessed wholly by the propagation and concurrency control algorithms. The algorithms for all of these guarantees therefore require implementations that are decentralized as much as possible.

# 4 A Survey of Reactive Frameworks

This chapter provides a comprehensive survey of prior works related to synchronous propagation semantics, reactive programming or just event-based programming in general, as well as automatic incrementalization. It classifies prior works by a multitude of different aspects: the programming style that they facilitate, what programming abstractions they provide, their relation to concurrency and distribution, the consistencies they provide for their change propagation, and the algorithms they use to provide these consistencies.

Table 4.1 lists all the prior works considered in the survey. It summarizes, which of the following features they do (filled dots) or do not (empty dots) support, and which consistency guarantees they provide, both for their local propagation within each individual host and – if applicable – for their distributed propagation over the network.

- **Dynamic Graphs:** This column shows if changes to the dependency graph topology at run-time are supported, or if applications' topology is fixed at compile-time.
- **Signals, Events:** These columns show whether or not Signals and/or Events, or which other programming abstractions are supported (local propagation), and can be shared remotely (distributed propagation).
- **Concurrent Reevaluations per Instant, Concurrent Instants:** For framework that provide synchronous propagation semantics, these columns show whether or not the framework supports concurrent reevaluations inside each synchronous instant, and parallel execution of multiple concurrent instants. Frameworks that do not provide synchronous propagation semantics to not have a concept of instants that separates concurrency into these two separate dimensions. For those, a single entry fills both columns, indicating whether or not or which kind of concurrency the frameworks support for their propagation of changes in general.
- **Glitch freedom (or Use of Topology):** This column shows whether or not each framework provides glitch freedom or derives other properties from the dependency graph topology.
- **Isolation (or Concurrency Consistency)**: This columns shows whether or not each framework provides isolation for concurrent instants, or if it provides some other form of consistency in regards to concurrent change propagation, unrelated to synchronous propagation semantics.

## 4.1 Preface: Selective Topological Sorting

As discussed in Section 2, a sufficient condition to achieve glitch freedom is executing reevaluations in a topological order of the *DG*. This survey will show many variants of propagation algorithms that implement topological sorting in different ways. We introduce the notion of *minimalism*, to be able to classify them in more detail.

The change propagation of each individual instant usually affects only a subset of all reactives in an application. Figure 4.1 exemplifies the extent of changes on the philosophers application from Section 2 for an instant that updates a philosopher from `Eating` back to `Thinking` (cf. Figure 2.3): Black reactives are reevaluated and change (the philosopher's input Var, both his `forks` Signals, his and his direct neighbors' `sights` Signals and `sightChngs` Events). Dark gray reactives are reevaluated but do not change (`filter` Events), meaning the change propagation does not progress further. Minimalism classifies the

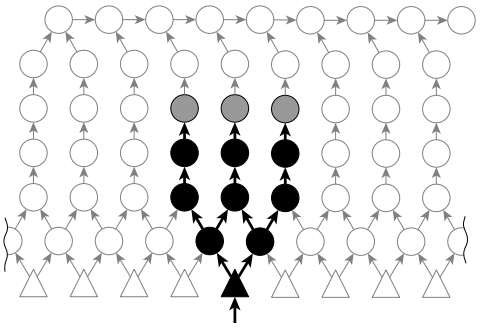


**Figure 4.1:** Depth-Minimal Traversal

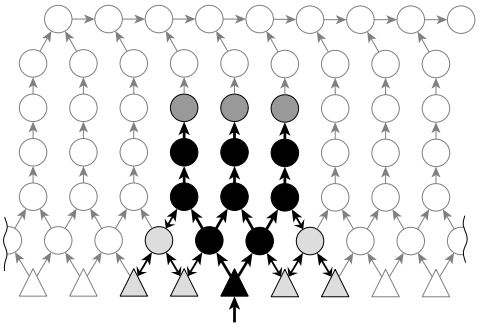


**Figure 4.2:** Reverse-Minimal Traversal

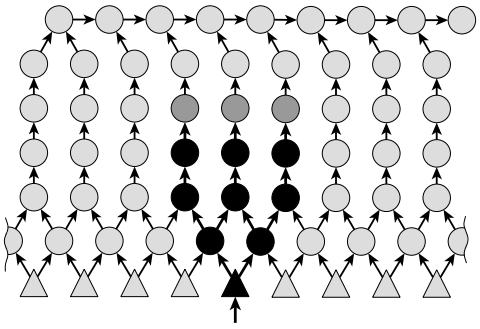


**Figure 4.3:** Non-Minimal Traversal

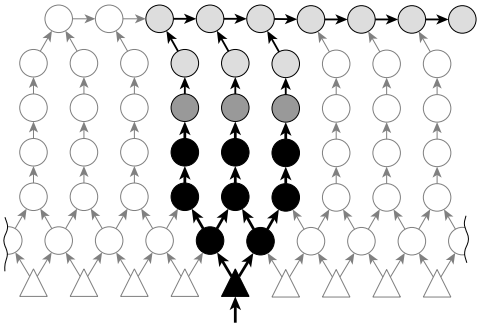


**Figure 4.4:** Breadth-Minimal Traversal

capability of propagation algorithms to compute a topological order of only the affected reactives of each instant, while traversing as few other reactives as possible. Especially in distributed applications, minimalism is highly desirable, since limiting the scope of change propagation to as few hosts as possible will avoid significant amounts of slow remote communication. We call propagation algorithms that are able to propagate a change glitch-free while traversing only those reactives that are actually reevaluated *depth-minimal*, because they do not traverse deeper into the *DG* than actual changes.

Traditionally topological sorting [Kahn, 1962] of a directed acyclic graph can be achieved by repeatedly removing a node with no predecessors from the graph until no nodes remain. This implementation of topological sorting executes in linear time in terms of graph size, which is optimal since the resulting order includes every node. Using this algorithm for change propagation though would imply that each instant, regardless of the extent of changes, would traverse every reactive in the entire *DG*. Figure 4.3 visualizes the same instant as Figure 4.1, but additionally highlights all unaffected reactives with a light gray background, because the algorithm would traverse them unnecessarily. We call such algorithms as *non-minimal*.

Several propagation algorithms achieve a middle ground between non-minimal and depth-minimal execution: they traverse only over the subset of all reactives that are reached by a forward or backwards graph traversal from affected reactives.

Figure 4.2 visualizes the extent of unnecessarily traversed reactives for propagation algorithms that utilize backwards graph traversals to determine glitch freedom for any affected reactive. We call such algorithms *reverse-minimal*. Similar to depth-minimal algorithms, they do not traverse the full depth of the *DG* topology. They will however always traverse an unnecessarily broad subset of reactives, since the direction in which they fan is the opposite of change propagation.

Figure 4.4 visualizes the extent of unnecessarily traversed reactives for propagation algorithms that utilize forwards graph traversals. They always traverse the full depth of the *DG* topology, but they limit their breadth as much as possible. Therefore, we call such *breadth-minimal*. Unlike reverse-minimal algorithms breadth-minimal algorithms can be optimal: For instants, during which every reevaluated reactive changes, breadth-minimal and depth-minimal traversals are identical. Breadth-minimal change propagation traverses unnecessary reactives only after some affected reactives did not change after reevaluation.

## 4.2 Synchronous Programming Languages.

For historic reasons, we discuss synchronous programming languages first. In general, these languages allow developers to define applications as a composition of many individual concurrent components, and then compose these components in a way that yields deterministic execution semantics for the overall program. As such, these programs often do not actually execute with any kind of concurrency at run-time, but only manage the concurrency of applications' components at compile-time.

We first discuss **ReactiveC** [Boussinot, 1991], which serves as a kind of foundation for the other languages in this group. ReactiveC extends the C programming language, introducing additional keywords that allow regular imperative programs to use aspects of synchronous execution semantics. Programs consist of a set of parallel processes that use regular variables and progress synchronously in instants. ReactiveC introduces the `stop` keyword, which allows processes to halt until the next instant. It introduces `loop` to repeat a process in every instant, and extends this to conditional `repeat`, `every` and `watching` statements, which repeat processes a fixed number of instants, only at instants where a certain condition is true, or at each instant until an abort condition is true for the first time. In combination, these keywords allow concurrent processes to synchronize their processing steps in a well-defined manner.

### 4.2.1 Esterel: Imperative Synchronous Programming

**Esterel** [Berry and Gonthier, 1992] is the earliest of the synchronous programming languages. It uses a similar syntax as ReactiveC, but additionally introduces *signals* – not to be confused with RP Signals – and

| RP(-related) Frameworks and their Features | | | | Dynamic Graphs | Properties of Local Propagation | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Signals | Events | Propagation Algorithm | Concurrent Reevaluations per Instant | Concurrent Instants | Glitch Freedom (or Use of Topology) | Isolation (or Concurrency Consistency) |
| Synchronous Propagation Semantics | Reactive Programming | Single-Core | **FrTime** | ● | ● | ● | Quadratic Topsort | ○ | ○ | ● | n/a |
| | | | **REScala** | ● | ● | ● | Height-based PQ | ○ | ○ | ● | n/a |
| | | | **Scala.React** | ● | ● | ● | Height-based PQ | ○ | ○ | ● | n/a |
| | | | **FlapJax** | ● | ● | ● | Height-based PQ | ○ | ○ | ● | n/a |
| | | | **Bacon.js** | ● | ● | ● | DFS + Reverse DFS | ○ | ○ | ● | n/a |
| | | | **emfrp** | ○ | ● | ○ | List Compilation | ○ | ○ | ● | n/a |
| | | Multi-Core | **Elm** | ○ | ● | ○ | Sweep + Queues | ● | ● | ● | ● |
| | | | **MV-RP** | ● | ● | ● | Txn CC + abstracted | ○/● **) | ● | ● | ● |
| | | Dist., but Serial | **SID-UP** | ● | ● | ● | Source ID Sets | ● | ○ | ● | n/a |
| | | Distributed | **DREAM** | ○ | ● | ○ | Vector Clocks + Locks | ● | ● | ○/…/● | ○/…/● |
| | | | **FrameSweep** (this work) | ● | ● | ● | Txn CC Mark & Sweep | ● | ● | ● | ● |
| | Synchronous Languages | Single-Core | **SLf, SLi (Self-Adjusting Computation)** | ● | ● | ○ | Timestamp PQ | ○ | ○ | ● | n/a |
| | | Multi-Core, Autom. Increment. | **PAL (Parallel Self-Adjusting Computation)** | ● | ● | ○ | Mark & Sweep | ● | ○ | ● | n/a |
| | | | **Incremental Concurrent Revision Types** | ● | ● | ○ | Mark & Sweep | ● | ○ | ● | n/a |
| | | | **iThreads** | ● | ● | ○ | Mark & Sweep | ● | ○ | ● | n/a |
| | | Multi-Core, Synchronous Languages | **ReactiveC** | n/a | variables | | n/a | ● | ○ | n/a | n/a |
| | | | **Esterel** | ○ | hybrid | | FSM Compilation | ● | ○ | ● | n/a |
| | | | **SL** | ○ | hybrid | | FSM Compilation | ● | ○ | ● | n/a |
| | | | **Céu** | ○ | hybrid | | FSM Compilation | ● | ○ | ● | n/a |
| | | | **SIGNAL** | ○ | hybrid | | FSM Compilation | ● | ○ | ● | n/a |
| | | | **Lustre** | ○ | hybrid | | FSM Compilation | ● | ○ | ● | n/a |
| | | Distributed, but Serial | **CoReA** | ○ | hybrid | | FSM Compilation | ● | ○ | ● | n/a |
| | | | **Distributing Reactive Systems** | ○ | hybrid | | FSM Compilation | ● | ○ | ● | n/a |
| | | | **ULM** | ○ | hybrid | | FSM Compilation | ● | ○ | ● | n/a |
| No or Local-Only Synchronous Propagation Semantics | Reactive Progr. | Single-Core | **Cycle.js** | ● | ○ | ● | [RxJS] | ○ | | ○ | n/a |
| | | | **Knockout.js** | ● | ● | ○ | n/a | ○ | | ○ | n/a |
| | | | **Reactor.js** | ● | ● | ○ | n/a | ○ | | Obs. Only | n/a |
| | | Multi-Core | **Scala.Rx** | ● | ● | ○ | Height-based PQ | ● | | Obs. Only | Single Nodes |
| | | | **Containers, Aggregates, Mutators, Isolates** | ● | ● | ● | n/a | ● | | ○ | Single Nodes |
| | Event-Based Programming | | **Observers/Callbacks** | man. | ○ | ●*) | n/a | ● | | ○ | n/a |
| | | | **Reactive Extensions** | man. | ○ | ●*) | n/a | ● | | ○ | Single Nodes |
| | | | **Complex Event Processing (abstractly)** | ○ | ○ | ● | n/a | ● | | Query Optimization | Single Nodes |
| | | Distributed, Only Events, for Responsiveness | **Publish-Subscribe (abstractly)** | man. | ○ | ● | n/a | ● | | ○ | ○, FIFO, Causal, Serial |
| | | | **Actors, Microservices** | man. | abstracted | | | | | | |
| | | | **Stream Processing (Spark, Flink, etc.)** | ○ | ○ | ● | n/a | Data Parallelism | | Query Optimization | n/a |
| | Au. Inc. | | **Incoop** | ○ | ● | ○ | Mark & Sweep | Data Parallelism | | ○ | n/a |
| | | | **Self-Adjusting MapReduce** | ○ | ● | ○ | Mark & Sweep | Data Parallelism | | ○ | n/a |
| | | Only Signals, for Eventual | **Reactive Caching for Composed Services** | ○ | ● | ○ | Mark & Sweep | ● | | ○ | n/a |
| | Reactive Programming | | **AmbientTalk/R** | ● | ● | ○ | Height-based PQ | ○ | ○ | ● | n/a |
| | | | **Direst** | ○ | ● | ○ | Height-based PQ | ○ | ○ | ● | n/a |
| | | | **REScala CRDTs** | ● | ● | ● | Height-based PQ | ○ | ○ | ● | n/a |
| | | | **Quarp** | ● | ● | ○ | Source Vector Clocks | ● | | ● | ○ |
| | | Both, for Usability | **Multi-Tier FRP** | ● | ● | ● | [Bacon.js] | ○ | ○ | ● | n/a |
| | | | **ScalaLoci** | ● | ● | ● | Height-based PQ | ○ | ○ | ● | n/a |

*) "Events" is used synonymously with method calls or (a)synchronous messages here.

**) Concurrent reevaluations per propagation supported, but disabled for performance optimization.

**Table 4.1:** Reactive and Related Frameworks and the Features They Support

| Frameworks (repeated) | Properties of Distributed Propagation | | | | | | |
|---|---|---|---|---|---|---|---|
| | Signals | Events | Propagation Algorithm | Concurrent Reevaluations per Instant | Concur-rent Instants | Glitch Free-dom (or Use of Topology) | Isolation (or Concurrency Consistency) |
| **FrTime** | | | | | | | |
| **REScala** | | | | | | | |
| **Scala.React** | | | | | | | |
| **FlapJax** | | | | | | | |
| **Bacon.js** | | | | | | | |
| **emfrp** | | | | | | | |
| **Elm** | | | | | | | |
| **MV-RP** | | | | | | | |
| **SID-UP** | ● | ● | Source ID Sets | ● | ○ | ● | n/a |
| **DREAM** | ● | ○ | Vector Clocks | ● | ● | ○/…/● | ○/…/● |
| **FrameSweep** (this work) | ● | ● | Mark & Sweep | ● | ● | ● | ● |
| **SLf, SLi (Self-Adjusting Computation)** | | | | | | | |
| **PAL (Parallel Self-Adjusting Computation)** | | | | | | | |
| **Incremental Concurrent Revision Types** | | | | | | | |
| **iThreads** | | | | | | | |
| **ReactiveC** | | | | | | | |
| **Esterel** | | | | | | | |
| **SL** | | | | | | | |
| **Céu** | | | | | | | |
| **SIGNAL** | | | | | | | |
| **Lustre** | | | | | | | |
| **CoReA** | hybrid | | Single Tick Delay | ● | ○ | ● | n/a |
| **Distributing Reactive Systems** | hybrid | | Resynchronization Messages | ● | ○ | ● | n/a |
| **ULM** | Process Migration | | | n/a | ● | ○ | Single Hosts |
| **Cycle.js** | | | | | | | |
| **Knockout.js** | | | | | | | |
| **Reactor.js** | | | | | | | |
| **Scala.Rx** | | | | | | | |
| **Containers, Aggregates, Mutators, Isolates** | | | | | | | |
| **Observers/Callbacks** | | | | | | | |
| **Reactive Extensions** | | | | | | | |
| **Complex Event Processing (abstractly)** | | | | | | | |
| **Publish-Subscribe (abstractly)** | ○ | ● | n/a | ● | ○ | | ○, FIFO, Causal, Serial |
| **Actors, Microservices** | ○ | ●*) | n/a | ● | Load Balancing | | Single Hosts + FIFO |
| **Stream Processing (Spark, Flink, etc.)** | ○ | ● | n/a | Data Parallelism | Load Balancing | | n/a |
| **Incoop** | ● | ○ | Mark & Sweep | Data Parallelism | Load Balancing | | n/a |
| **Self-Adjusting MapReduce** | ● | ○ | Mark & Sweep | Data Parallelism | Load Balancing | | n/a |
| **Reactive Caching for Composed Services** | ● | ○ | Mark & Sweep | ● | ○ | | n/a |
| **AmbientTalk/R** | Peers | ○ | n/a | ● | ○ | | Single Hosts |
| **Direst** | CRDT | ○ | n/a | ● | ○ | | Single Hosts |
| **REScala CRDTs** | CRDT | ○ | n/a | ● | ○ | | Single Hosts |
| **Quarp** | ● | ○ | Source Vector Clocks | ● | ● | | ○ |
| **Multi-Tier FRP** | ● | ● | n/a | ● | ○ | | Single Hosts + FIFO |
| **ScalaLoci** | ● | ● | n/a | ● | ○ | | Single Hosts + FIFO |

*) "Events" is used synonymously with method calls or (a)synchronous messages here.
CRDT: Conflict-Free Replicated Data Type

**Table 4.2:** Reactive and Related Frameworks' Distributed Properties (Continuation of Table 4.1)

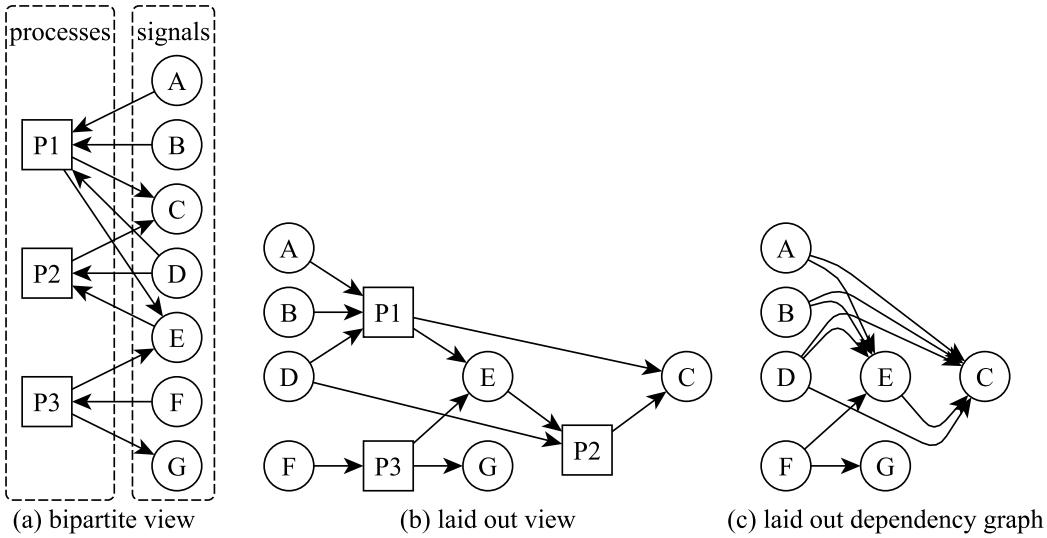



**Figure 4.5:** Deriving Dependency Graphs from Esterel Programs.

bases synchronous execution around these signals. A signal is, in essence, a hybrid of an RP input Signal (Var) and an RP input Event. Like a Var, an Esterel signal holds a value at all times, which processes can imperatively reassign in each instant. Reassigning the value makes the signal "present" for this instant, i.e., it emits like an RP input Event. Processes can also set the signal present without changing its value, similar to RP input Events firing the same value repeatedly in successive instants. Esterel connects its synchronous semantics to its signals twofold. Conditional statements, similar to those in ReactiveC above, are all based on signals being present or not present, instead of arbitrary boolean expressions. Further, signals are exclusively either present or not present and have only a single value associated throughout each entire instant. To maintain this appearance towards programs' processes, Esterel must control the execution of processes such that signals are always written before being read within each instant.

Esterel and the other synchronous programming languages pre-date the term "glitch freedom". Yet, this is precisely what they implement within their synchronous semantics, although particularly in Esterel applications, this is difficult to see. Viewed abstractly, Esterel programs are a set of processes reading and writing on a set of signals with no recognizable structure. Nevertheless, Esterel's description of semantics itself introduces a method of deriving a dependency graph from Esterel programs, namely by introducing an edge between any two signals where one is read before the other is written within one instant of any single process. Figure 4.5 visualizes this transformation. It shows a straight-forwards bipartite graph visualization of an Esterel Program (a), with reads as edges from signals (circles) to processes (squares) and writes in the reverse direction. By giving this graph a different layout, the program's directed acyclic graph structure becomes apparent (b). The mandated edges for the signal dependency graph transformation then are a subset of the transitive edges of each pair of incoming and outgoing edges of every process (c).

Esterel's execution semantics describe a topological sorting implementation, with minimalism being irrelevant since every process executes in every instant, even if its only operation is to immediately pause again until the next instant. At the beginning of every instant, all signals are considered invalid. Processes can preliminarily make signals present and write new values at any time. Once all processes can no longer write a given signal, which is inferred by lexical analysis of the concurrent expressions that still remain to be evaluated, the signal is marked as finalized, with its presence depending on its preliminary state. Processes can evaluate any statements that read the value or verify the presence of a signal only after that signal has been marked as finalized. This way, signals in the above described dependency graph are finalized in topological order, i.e., glitch-free. The actual implementation of

Esterel's semantics is not applicable to distributed RP though: In order to maximize efficiency, Esterel uses a compiler that optimizes and reduces programs into finite state machines that execute on top of ReactiveC.

### 4.2.2 Other Synchronous Imperative Languages

The **SL Synchronous Language** [Boussinot and de Simone, 1996] is closely related to Esterel, following the same model and differing only in the way it handles reactions to absent signals. In Esterel, programs are not compositional, because they may produce causality cycles. These may cause programs to be incorrect, e.g., `present S else emit S`, where if signal `S` is not present, it should be made present in the same instant, which in turn means the statement should not have executed. They may also cause instants to have more than one possible correct outcome, i.e., an instant's outcome may have some signal present or not present, both options being correct outcomes, but the program may behave differently. The SL Synchronous Language allows the absence of any signals to be determined only at the very end of an instant, and according reactions execute only during the next instant. This reduces the language expressiveness, but in turn makes it impossible to define ambiguous or incorrect programs.

**Céu** [Sant'Anna et al., 2013] is also closely related to Esterel, but is much younger. It differs mainly in two aspects. First, it provides even more determinism, in that it ensures a consistent order even between concurrent processes that do not have data dependencies with each other. Second, whereas Esterel executes one instant after the next, with system inputs polled in every instant, Céu instead hibernates applications between instants. More similar to RP, the next instant in Céu is started only when any of the systems' inputs actually changed. This makes Céu suitable for energy-constrained computing devices, such as mobile sensors, since hardware components can be turned off during hibernation phases, thereby reducing power consumption to minimally necessary heights.

**CoReA** [Boniol and Adelantado, 1993] attempts to bring the synchronous programming of Esterel to distributed systems. All hosts' instants are based on the same global clock. It only implements "weak synchronous" semantics though: Every host in the distributed application runs is a "strong synchronous" system, equal to, e.g., regular local Esterel applications. Communication between hosts takes place only between clock ticks though, meaning one host can react to signals from processes on other hosts only in the next instant. This is intended to find a semantics that is both concurrent and deterministic, as a middle ground between the multi-threaded imperative programming style and the other "strong synchronous" languages in this group: The former is concurrent but non-deterministic, while the latter is considered deterministic but not concurrent, due to an interpretation of "strong synchronous" semantics as implying that programs *must* be sequentialized into single-threaded applications. This contrasts the interpretation taken by this dissertation for FrameSweep, which is that there is well-defined space for concurrent executions within ("strong") synchronous semantics.

**"Distributing Reactive Systems"** [Caspi et al., 1994] also addresses distributing synchronous Esterel programs. It provides algorithms for distributing values and process steps based on assigning signals to certain network locations, and then inserting dummy communications into the distributed components to ensure synchronous semantics. It provides two different options for synchronous semantics, which it also calls "strong" and "weak". They relate to either allowing or not allowing different hosts in the application to diverge in terms of their current instant. "Strong", as above, equals the semantics of local applications, meaning every instant occurs simultaneously on all hosts. This work therefore does implement distributed synchronous semantics, but still without concurrent instants, i.e., still not scalable in the way that this dissertation intends to achieve. "Weak" on the other hand is defined differently from CoReA. In this work, "weak" synchronous semantics means that instants across hosts are allowed to diverge in time, once they do not await further signals from each other, but only so far that they still overlap, such that an event producer cannot run away from a slower consumer. For every instant, there must be a moment where either all hosts or every connected pair of hosts (both variants are possible) is processing that instant at the same time. This weak synchronous semantics does not change the internal

synchronicity of a given program, but weakens only its outside appearance, in that an external observer may see one host process a second instant before another host processing the first. As such, this weak synchronous semantics is the first appearance of concurrent instants as considered in this dissertation, albeit still in a very narrowly restricted form.

Lastly, there is **ULM** [Boudol, 2004], which also intends to bring the SL synchronous language to distributed applications. ULM does not implement any form of distributed synchronization though, but instead extends processes to agents that can migrate between hosts. Each host individually is a synchronous system, and an agent that migrates to a given host executes its process synchronously with all other processes on that host. In order to synchronize with processes on a different host though, the agent has to migrate to that other host. The process can never synchronize with processes on two hosts at the same time. As such, ULM also does not bring anything new to the table in regards to implementing distributed synchronous semantics.

### 4.2.3 Synchronous Data Flow Languages

Differently from the imperative synchronous style pioneered by Esterel, there are two prominent data flow synchronous languages, **SIGNAL** [Gautier et al., 1987] and **Lustre** [Caspi et al., 1987]. SIGNAL and Lustre do not have imperative processes. Instead, programs are described as a set of equations (computations) that each define a variable, like derived Signals of RP. Differently from RP though, which variables are recomputed at which points in time is not based on their changes, but on *clocks*, similar to clocked processors in hardware. In SIGNAL, each input variable comes with its own associated clock, whereas Lustre initially has only a basic clock that simply ticks every instant. In both languages though, custom clocks can be defined directly in the programs in the form of boolean variables that tick in every instant where they are present and true. In order to combine any two variables with any operator, both variables must either have the same clock, or must be converted to the same clock, and the operator will then recompute the variable's value whenever that clock ticks. Converting variables between clocks can be done, e.g., by reusing variables' latest values, or re-sampling them at slower clock rates.

SIGNAL and Lustre both provide calculi to compile programs into finite state machines, equally to the imperative synchronous languages discussed above. The dependency graph of SIGNAL and Lustre programs is derived from the dependency relations between variables, in particular including boolean variables that are used as the clocks of other variables, and used to order variable recomputations in each transition, achieving glitch freedom. Differently from the imperative synchronous languages, minimalism does play a role in SIGNAL and Lustre since variables are only recomputed in instants where their associated clocks tick. This minimalism is integrated into the program at compile-time by using the boolean expressions of clock variables to determine, which recomputations must be executed at which transitions of the resulting finite state machine. Since all this is still solved by statically at compile-time, this approach is also not suitable for in the presence of dynamic dependencies.

### 4.2.4 Summary

Unsurprisingly, all synchronous programming languages share the concept of synchronous instants. They are all closely related, using nearly identical algorithmic approaches to provide synchronous semantics, namely compiling their programs into sequentialized finite state machines. This approach is not suitable to implement distributed RP with fine-grained concurrency control, as envisioned in this dissertation.

More recently, the limitation of processing power of single CPUs has begun to impact synchronous programming languages as well. As a result, research has started towards getting synchronous programs to execute in parallel [Yip et al., 2016]. Therein, synchronous programs are no longer sequentialized, but concurrent execution of independent computations within each instant is introduced. This matches what is considered concurrent reevaluations within instants in terms of this dissertation. Due to their

focus on upholding the real-time predictability and constraints however, the implementation of this is based on real-time scheduling, using computed worst-case execution times of each task, rather than algorithmic ordering, and thus this is not transferable to RP. Moreover, they retain the restriction that only a single instant executes globally at a time, and thus support only one of the two dimensions of concurrency (for good reason, as supporting concurrent instants would make bounded execution time much more difficult). As such, these recent advancements not applicable to distributed RP either.

## 4.3 Reactive Programming

The term "Reactive" in relation to programming is overloaded with several partially overlapping meanings. This dissertation derives the term "Reactive Programming" from Functional Reactive Programming. This section addresses prior works from this area of research. A second meaning, which has recently gained a lot of popularity, is the notion of "Reactive Distributed Systems", spearheaded by the Reactive Manifesto [Bonér et al., 2014]. Therein, "reactive" describes the quality of distributed systems to be resilient to failures in that they remaining responsive and continue to function even if some parts of the application fail. The Reactive Manifesto advocates event-based programming as the fundamental design abstraction, on which all distributed systems should be built to ensure that they are "reactive". Therefore, prior works in the category of reactive distributed systems are not addressed here, but as part of Section 4.4 instead.

Works on Functional Reactive Programming can be separated into two fundamental categories: discrete or continuous. In discrete RP frameworks – which is the model followed in this dissertation – the values of Events and Signals are recomputed in instants as discrete logical steps. Continuous Functional Reactive Programming [Elliott and Hudak, 1997] instead defines Signals and Events as mathematical continuous functions time to values. Syntactically, both categories are similar, in that applications define Events and Signals declaratively through formulae or functions, and the control flow between them is added by the runtime library or framework. The semantics behind discrete vs. continuous Functional Reactive Programming are very different though, and have no relation with synchronous propagation semantics. As such, works on continuous Functional Reactive Programming are not included in this survey.

### 4.3.1 Local, Single-Threaded Reactive Programming

Discrete Functional Reactive Programming has been pioneered by **FrTime** [Cooper and Krishnamurthi, 2006]. FrTime embeds RP into Scheme as its host language, using a model very similar to the one described in this dissertation. It provides Events and Signals as first-class entities, with support for dynamic changes to the dependency graph topology for higher-order reactives. It does not address any form of concurrency, and thus achieves synchronous propagation semantics through glitch freedom by executing reevaluations in a topological order. It computes this topological order by labelling all reactives with their height. Input reactives as height zero, and derived reactives as one height above the maximum height of all their dependencies. After a reactive changed, all outgoing dependencies are added into a priority queue, and change propagation is implemented by reevaluating all reactives in this queue, lowest height first. In essence, heights allow to conclude the absence of transitive reachability, but without exhaustively traversing the dependency graph. If a first reactive has a smaller height than a second, then there cannot be path in the *DG* from the second reactive to the first, hence the first can be reevaluated without risk of future reevaluations of the second causing a glitch. As a result, height-based priority queue change propagation is depth-minimal, i.e., optimal. Unfortunately, heights significantly over-approximate transitive dependencies, which makes this algorithm unsuitable for use in distributed applications. If any two reactives have different heights, the propagation algorithm must assume that a path in the *DG* exists from one to the other, and therefore cannot reevaluate them concurrently even if

they do not actually depend on each other. In distributed applications, this unavoidably implies a large amount of remote synchronization between reactives where no data dependencies actually exist, which causes a prohibitive performance overhead.

**REScala** [Salvaneschi et al., 2014b] and **Scala.React** [Maier and Odersky, 2012] embed RP into Scala. They also implement a model of RP very similar to this dissertation, and also use the height-based priority queue. Scala.React acknowledges multi-threading in that it binds applications' *DG* to a scheduler. This scheduler executes all instants' change propagation on a specific thread (e.g., the UI thread for UI applications), i.e., it deliberately prevents concurrency. Scala.React additionally implements Reactors and incremental list reactives inside the RP model of Signals and Events. Reactors are processes similar to those from imperative synchronous programming languages. Reactors' execution semantics are closest to processes in Céu, in that they do not execute in every instant, but only when they are actually affected by changes. Differently though, Reactors depend on and change regular RP Events and Signals, rather than using the event/signal hybrid abstractions from imperative synchronous programming languages. Lastly, an extension to Scala.React describes, how incremental collection reactives can be integrated into an RP system with Events and Signals [Maier and Odersky, 2013]. Incremental collection reactives [Maier and Odersky, 2013] are in essence Signals, whose values are collections of elements, and that propagate incremental changes between each other, rather than propagating an entirely new collection instance for every change. While it should be feasible to integrate these features into the model of distributed RP of this dissertation without having to significantly changes, they do not yield insights towards concurrent or distributed change propagation.

With many interactive applications being browser-based distributed web applications, a plethora of frameworks inspired by RP has been built in JavaScript. These come in varying shapes and sizes, with many implementing only a subset of the features of RP presented in this dissertation. Generally though, they support the implementation of only the user interface in the single-threaded browser environment, and not the server. Thus, despite being designed for use in distributed applications, they are not implementations of distributed RP, and do not address any form of concurrency. Since there are far too many such frameworks to discuss all off them, we only address several examples.

**FlapJax** [Meyerovich et al., 2009] and **Bacon.js** [`https://baconjs.github.io`] are examples that implement similar model including all the features discussed in this dissertation. Bacon.js on the other hand uses a two-phased approach. First, a depth-minimal forwards depth-first graph traversal either directly reevaluates reactives that do not request glitch freedom, or places those that do in a queue. Second, for every reactive in this queue, a backwards depth-first traversal is executed. When the traversal has finished traversing all dependencies of a reactive and returns, this reactive is ready for a glitch-free reevaluation. If it is in the queue of pending reevaluations, it is reevaluated, and if this results in it changing, another forwards traversal is executed before the backwards traversal continues. This forwards traversal also places affected reactives that require glitch freedom in the queue though, so all reevaluations that are susceptible to glitches are still executed only when the backwards over all their dependencies has completed. Thus this algorithm achieves topological reevaluation order, and therefore glitch freedom and synchronous propagation semantics, and is reverse-minimal overall. Due to the global queue of pending glitch-susceptible reevaluations this algorithm is unsuitable for use in distributed RP. Even if there was a way to implement it without this queue though, it would likely complicate concurrency control due to introducing the additional requirement for safely executing backwards graph traversals, on top of inherent requirement for safely executing forwards graph traversal for change propagation.

**Cycle.js** [`https://cycle.js.org`], **Knockout.js** [`https://knockoutjs.com`] and **Reactor.js** [`https://github.com/fynyky/reactor.js`] are examples of JavaScript RP libraries that provide only a subset of the RP features discussed in this dissertation. Cycle.js supports only Events, not Signals, by re-using RxJS, the JavaScript implementation of Reactive Extensions (discussed in Section 4.4). Knockout.js and Reactor.js conversely support only Signals, but not Events. All three do not use a propagation algorithm for glitch freedom, and thus do not provide synchronous propagation semantics. Reactor.js still attempts

to support the appearance of synchronous propagation though, by providing a kind of observer-only glitch freedom. Symmetric to the model of this thesis including dedicated input reactives that do not have dependencies, Reactor.js supports observers as dedicated output reactives that no other reactives can depend upon. It expects side-effects to be part of user computations of observers only, while user computations of regular derived reactives are expected to be functionally pure. Instants are processed in two phases. First, changes are propagated through derived reactives in an arbitrary order. Affected derived reactives are reevaluated with glitches, while affected observers are added to a queue instead. After this first phase, i.e., once all changes have been propagated and all temporarily glitched values have been overwritten by correct ones, the second phase reevaluates all queued observers. Due to not having outgoing dependencies, observers cannot depend on each other, i.e., they are all independent of each other and therefore can be reevaluated in any order without causing glitches. Thus, even though overall change propagation is not glitch-free, glitches cannot interact with side-effects, and therefore cannot cause any harm. Restricting side-effects to dedicated observer reactives only does not align with the model used in this dissertation though, where side-effects are allowed in all user computations, and therefore is not useful towards the envisioned goal.

Lastly, **emfrp** [Sawada and Watanabe, 2016] is a standalone RP language designed for use with resource-constrained embedded devices. It supports only Signals, not Events, and does provide glitch freedom and thus synchronous propagation semantics. To minimize its computational requirements, it does not use a propagation algorithm. Instead, it is restricted to static dependency graphs only, and provides synchronous propagation semantics by computing a glitch-free reevaluation order for every input change statically at compile-time. This means, it is algorithmically similar to the FSM compilation of synchronous programming languages, and does not provide any new insights towards distributed RP.

### 4.3.2 Reactive Programming Libraries with Concurrency

**Scala.rx** [https://github.com/lihaoyi/scala.rx] is another embedding of RP in Scala. It supports dynamic *DG* topologies of only Signals, not Events. Scala.rx uses the same height-based priority queue change-propagation as REScala, Scala.React and FlapJax described above. On top of that though, it integrates support for both concurrent reevaluations per instant and concurrent instants. For concurrent reevaluations per instant, Scala.rx simply reevaluates all reactives of the current lowest height in the priority queue in parallel, instead of only a single one. Unfortunately, since heights still over-approximate transitive dependency relations, this support for concurrent reevaluations per instant is far from optimal, and does not improve on the requirement for over-extensive remote synchronization between independent reactives. Without concurrent instants, this introduction of concurrent reevaluations per instant remains glitch-free, and the algorithm therefore does still provide synchronous propagation semantics. For concurrent instants though, Scala.rx implements only minimal concurrency control that does not preserve synchronous propagation semantics. Namely, instead of ensuring isolation for entire instants, Scala.rx ensures only mutual exclusion for reevaluations on single reactives. Concurrent instants can reevaluate different reactives in different orders though, and as such, glitch freedom, synchronous propagation semantics, and the concept of instants is lost as soon as applications admit input changes concurrent. To compensate for concurrency glitches, Scala.rx implements the same concept of observer-only glitch freedom as Reactor.js, but – even for observers – this restores only glitch freedom, but not isolation and therefore also not synchronous propagation semantics. In summary, the propagation algorithm of Scala.rx is equally infeasible to use for distributed RP as the single-threaded priority queue, and it provides no new insights towards supporting concurrency.

**Containers, Aggregates, Mutators, Isolates** [Prokopec et al., 2014] is also another embedding of RP in Scala. It supports dynamic *DG* topologies of Events, Signals and incremental collection reactives similar to Scala.React. It does not use a propagation algorithm though, and therefore does not provide glitch-freedom and synchronous propagation semantics. It does support parallel propagation of multiple concurrent changes, and provides two features to support this concurrency. First, it allows reactives to

be bound to specific threads, so that all their reevaluations are executed on that specific thread. Second, it ensures mutual exclusion for reevaluations of each single reactive, but does not implement a more sophisticated form of consistency in relation to concurrency.

**Elm** [Czaplicki and Chong, 2013] is another Reactive Programming framework for JavaScript that implements only a subset of RP features. Namely, Elm supports only Signals, not Events, and only static dependency graphs, i.e., no dynamic changes of dependency relations. Despite JavaScript being a traditionally single-threaded environment and Elm therefore not actually utilizing any multi-threading, Elm has a propagation algorithm that provides synchronous propagation semantics while fully supporting fine-grained parallel execution of concurrent instants and concurrent reevaluations within each instant. Elm's propagation algorithm uses asynchronous message passing to implement traditional topological sorting, and adds message queues on all dependency edges. After a reactive changed, it queues its new value in the message queues of all its outgoing dependency edges. Any reactive that does not change instead queues a "no change" message into each queue. This particularly includes input reactives, meaning if a change is admitted to some input reactive, all other input reactives must be iterated to emit "no change" messages. Every derived reactive can execute a glitch-free reevaluation once it has a message in the queue of every incoming dependency edge. To execute this reevaluation, it dequeues the first message from each incoming dependency edge message queue. Newly received values are used to update the parameters with which the reactive executes its user computation, while it retains the previous value of a parameter if it receives a "no change" messages. If all queued messages are "no change" messages, "no change" message is forwarded to all outgoing dependencies immediately, without executing a reevaluation – we call this process *no-change propagation*.

Due to several aspects, Elm's change propagation algorithm is not suitable for distributed RP, but it still yields several novel insights. First, the algorithm is non-minimal, meaning it would require a lot of unnecessary slow remote communication. More crucially though, the necessity to starting no-change propagation from all inputs for every instant would require some form of central coordinator that knows all input reactives of the entire distributed application. Due to this central coordinator, a decentralized implementation impossible. Nevertheless, the actual propagation of changes after this initialization supports concurrent reevaluations per instant in a fully decentralized fashion, sending messages exclusively along dependency edges in the *DG* topology with each node capable of determining glitch freedom of its reevaluations autonomously. Further, the concept of no-change propagation shows, how such decentralized message-based propagation can accurately reevaluate only those reactives that are actually affected by changes, even if its graph traversal is not depth-minimal. Finally, its use of message queues on *DG* edges demonstrates that having reactives' values from different points of time accessible in different locations of the *DG* makes it possible to simultaneously support fine-grained parallelism between concurrent instants and provide synchronous propagation semantics. Unfortunately though, retaining these values in queues bound to *DG* edges is fundamentally incompatible with dynamically discovering such edges, because it is not possible to reconstruct correct contents of the associated queue when it is created.

**MV-RP** [Drechsler et al., 2018] is one of the stepping stones developed in the context of this dissertation towards distributed RP. MV-RP is a scheduler add-on that creates a thread-safe concurrent RP implementation when combined with any propagation algorithm for glitch freedom. The scheduler is built from a combination of concurrency control algorithms from databases that cooperate with the propagation algorithm. It treats each execution of an instant by that propagation algorithm as a transaction, and allows multiple concurrent ones to execute with fine-grained parallelism, while maintaining isolation in the form of abort-free strict serializability, and thus retaining synchronous propagation semantics. Most crucially, and responsible for its name, one of MV-RP's concurrency control algorithms is *multiversion* concurrency control. Multiversion concurrency control retains the old values of variables when new values are written, which bears notable similarity to how Elm supports concurrent instants through having reactives' values buffered for different lengths of time in different message queues. Differently from Elm though, MV-RP stores these multiple values not on *DG* edges, but on the reactives

where they originate from, which is what enables it to support dynamic dependency edge changes. A second difference to Elm is, that MV-RP is breadth-minimal instead of non-minimal, but in turn requires two *DG* traversals instead of just one. Since the implementation of FrameSweep includes all the same algorithms and data structures of MV-RP, we do not go into further detail here on how MV-RP works. These details are instead explained as part of the description of FrameSweep in Chapter 5.

### 4.3.3 Signal-only Distributed Reactive Programming for Eventual Consistency

This section surveys a significant number of frameworks under the label of Distributed Reactive Programming that build on the insight that Signals without synchronous propagation semantics are a very natural fit to express eventually consistent values. In their model, a Signal simply represents the most recent known state or result value of some computation, and may update at arbitrary times for any reason. This way, they become naturally tolerant towards network delays and partial failures. On the downside though, the lack of synchronous semantics means that the changes of these Signals become non-deterministic. Therefore, the model of these frameworks is fundamentally unable to support Events, because non-deterministically firing Events are not a useful programming abstraction. The earlier-discussed Scala.rx also fits into this category, with the minor variation that it uses eventually consistency for Signals to tolerate race conditions from concurrency, rather than network delays and partial failures from distribution. As a result, equal to Scala.rx, none of the frameworks presented here yield any new insights for the implementation of distributed RP with synchronous propagation semantics.

**AmbientTalk/R** [Lombide Carreton et al., 2010] provides distributed RP for loosely-coupled peer-to-peer scenarios. Locally on each host, it uses height-based priority queue change propagation under global mutual exclusion, i.e., provides synchronous propagation semantics with no support for concurrency. For distribution, local Signals can be assigned to interfaces that can be published for discovery by remote peers. Peers can query their environment for specific interfaces, which returns a Signal of a collection of all instances of this interface published by peers in the proximity. These collections of connected peers' publications form the basis for all distributed reactivity. If a peer changes the value of a published signal, this value changes in all connected peers' collection. If peers lose or gain connection, their published values are added or removed from the collection. The distributed *DG* topology is thus entirely transient, meaning AmbientTalk/R does support defining fixed dependency relations across a specific network topology. Correspondingly, it does not address any form of consistency for distributed change propagation.

**Direst** [Myter et al., 2016] and the **REScala's CRDT Signals** [Mogk et al., 2018] use eventually consistent Signals for state replication in distributed applications. Locally, equal to AmbientTalk/R, both use height-based priority queue propagation under global mutual exclusion to support synchronous propagation semantics with no concurrency for Signals of arbitrary values. They differ only slightly, with REScala additionally supporting Events, and Direst not supporting dynamic dependency changes. Distribution on the other hand can be built, for both, only through multiple hosts collaboratively sharing replicated Signals of certain conflict-free replicated data type (CRDT) values. CRDTs are types whose values can be updated incrementally, and all possible updates are commutative with each other, e.g., an integer counter with only increment and decrement operations, but no setter. Due to this commutativity, these replicated Signals retain their eventual consistency regardless in which order and with which delays changes are replicated between hosts. Neither Direst nor REScala's CRDTs implement any consistency guarantees for their distributed propagation beyond this eventual consistency.

**Quarp** [Proença and Baquero, 2017] follows the an approach more that is closer to the model used in this dissertation, with applications defined as a distributed dependency graph of remotely shared Signals of arbitrary values. Quarp's propagation algorithm ensures glitch freedom and is fully decentralized. If other algorithms – such as a decentralized implementation of MV-RP – ensure isolation, Quarp would be suitable to add glitch freedom and thus achieve decentralized synchronous propagation semantics, Quarp is thus highly relevant for distributed RP as envisioned by this dissertation. Without such other

algorithms, Quarp provides glitch freedom under fully concurrent distributed change propagations, but does not ensure isolation, and therefore does not provide synchronous propagation semantics. Similar to the propagation algorithm of Elm, QUARP's change propagation is message-based and utilizes no-change propagation. Differently though, its propagation is breadth-minimal, but in turn it adds a vector timestamp to every message. These vector timestamps are based on logical clocks on all input reactives that are individually incremented with every admitted change. The vector timestamps added to the outgoing messages of a reactive map every transitively reachable input reactive to its latest known timestamp. Each reactive can autonomously compute whether or not a glitch-free reevaluation is possible by comparing the timestamp of each input reactive across all its dependencies. If two dependencies' values map the same input reactive to two different timestamps, the reevaluation would combine values associated with two different points in time, and therefore might produce a glitch. A glitch-free reevaluation is guaranteed only if, for each input reactive, all dependencies that do associate a timestamp associate the same timestamp with that input reactive.

### 4.3.4 Other Distributed RP Frameworks

**Multitier Functional Reactive Programming** [Reynders et al., 2014] and **ScalaLoci** [Weisenburger et al., 2018] are Scala-based languages for the implementation of distributed systems using a combination of RP and multitier programming, i.e., describing all components of a distributed system as a single compilation unit with the compiler later separating them. Multitier FRP is focussed purely on distributed RP, and targets specifically client-server web applications. ScalaLoci on the other hand supports distributed RP as one of several features for implementing distributed applications, and targets arbitrary distributed application scenarios (client-server, peer-to-peer, and more). Both support remote sharing of both Events and Signals of arbitrary types, and dynamic changes to the dependency graph topology. On each individual host, both use height-based priority queue change propagation under mutual exclusion and thus provide synchronous propagation semantics with no concurrency. Because both focus on language design improvements from the synergy of combining multitier and reactive programming though, neither implements sophisticated distributed propagation algorithms. They emulate distributed propagation by triggering separate, unconnected instants on the remote hosts when remote dependencies change, ensuring only FIFO order preservation for remote changes across each individual network link. Both leave the challenge for providing consistent distributed change propagation semantics to future work.

**SID-UP** [Drechsler et al., 2014] is another stepping stone on the path of this dissertation towards distributed RP. It supports Signals, Events, dynamic dependency changes and concurrent reevaluations per instant, but not concurrent instants. Equal to Quarp, SID-UP ensures glitch freedom and is fully decentralized, and if other algorithms – such as a decentralized implementation of MV-RP – ensure isolation, SID-UP would be suitable to add glitch freedom and thus achieve decentralized synchronous propagation semantics. SID-UP is therefore also highly relevant for distributed RP as envisioned by this dissertation. Differently from Quarp though, SID-UP is incompatible with concurrent instants, and thus not usable without such other algorithms. SID-UP uses an approach similar to Quarp's vector timestamps, but with slightly less complex data structures. Due to not supporting concurrent instants, only two different vector timestamps could exist throughout the entire application at any point in time: those that corresponds to before and after the currently executing instant. SID-UP thus achieves equivalent behavior to Quarp's vector timestamps by storing on each reactive only the set of transitively reachable input reactives – to determine if a given instant affects the reactive – and whether or not the reactive was already reevalutated by the current instant. SID-UP therefore does not associate timestamps with each transitively reachable input reactive, and does not require logical clocks on input reactives. In summary, SID-UP uses slightly less complex data structures than Quarp, but in turn can not provide glitch freedom for concurrent instants. Assuming that some external environment provides isolation though –

such as MV-RP – the latter becomes irrelevant though, meaning SID-UP may be superior to Quarp for implementing distributed glitch freedom in the context of synchronous propagation semantics.

**DREAM** [Margara and Salvaneschi, 2018] is an RP implementation designed to compare the cost of different consistency levels for distributed change propagation. DREAM supports all aspects of concurrency during propagation, but only over static distributed *DG* topologies of only Signals, i.e., it supports neither Events nor dynamic dependency changes. DREAM allows applications to chose – on top of liveness – one of the following consistency levels, which it implements through different combinations of vector clocks and locks:

- FIFO: If one reactive changes twice in succession, any depending reactive is reevaluated for these changes in the same order. This is equivalent to the consistency provided by Multitier FRP and ScalaLoci for their distributed propagation.

- Causal: If a change of a first reactive propagates and causes a second reactive to change, any third reactive that depends on both of them is reevaluated for the first reactive's change before it is reevaluated for the second reactive's change.

- Single Source Glitch Freedom: This consistency is defined ambiguously. Its name and motivating example suggest that it provides glitch freedom for applications where updates originate from a single input reactive only, but does not provide glitch freedom if concurrent updates from different sources interact. This is equivalent to the consistency provided by Scala.rx, which was discussed earlier. The formal definition of single source glitch freedom and its implementation however suggest that it provides glitch freedom even under concurrent instants, but without isolation. This is equivalent to the consistency provided by Quarp, which was discussed earlier. DREAM assumes that Signals always change after reevaluations, therefore its implementation of Single Source Glitch Freedom corresponds to a breadth-minimal *DG* traversal without no-change propagation.

- Complete Glitch Freedom: Glitch-freedom and isolation, i.e., synchronous propagation semantics, but only for reevaluations executed during instants. Imperative reads by outside processes may still observe partial effects of instants (i.e., glitches). In other words, Complete Glitch Freedom ensures abort-free serializability for transactional instants, but not strict, i.e., without linearizability for imperative reads. Similar to MV-RP, which was discussed earlier, this consistency is implemented by preceding the breadth-minimal glitch-free change propagation traversal of each instant by a breadth-minimal lock acquisition traversal. The way in which DREAM acquires the correct set of locks for each instant relies on the static nature of the dependency graph though. Moreover, DREAM relies on a centralized lock manager to avoid deadlocks between instants. As such, this implementation does is unsuitable for distributed RP as envisioned by this dissertation.

- Atomic: Glitch freedom and isolation, i.e., synchronous propagation semantics, including transactional execution of imperative reads by outside processes. This corresponds to the consistency envisioned by this dissertation for distributed RP (abort-free strict serializability) and implemented by MV-RP for local multi-threaded RP. The implementation of Atomic consistency is even more similar to MV-RP than Complete Glitch Freedom. On top of Complete Glitch Freedom, it also acquires locks for imperative reads in a similar fashion. But, this does not change that the implementation of Complete Glitch Freedom is unsuitable for this dissertations goal.

Lastly, **FrameSweep** – the contribution of this dissertation – belongs into this category. FrameSweep supports all of the features discussed by this dissertation: dynamic distributed *DG* topologies of both Signals and Events, glitch-free concurrent reevaluations within each instant and fine-grained parallel execution of multiple concurrent instants in isolation, thus overall still achieving synchronous propagation semantics. The algorithmic details of how FrameSweep implements its consistency guarantees are described in Chapter 5, and how it adapts to distribution in Chapter 7.

## 4.4 Event-Based Programming

The most basic programming paradigm that can be considered event-based is the **Observer** pattern [Gamma et al., 1995], or **callback**-based application designs in general When chaining together multiple observers, a *DG* topology forms. Since each instance of the observer pattern is often an individual implementations with custom interfaces, such topologies are not accessible in a uniform way. As a result, control flow of Observer-based application designs emerges incidentally, and there are no overarching control facilities. Observer subscriptions are dynamic, but not automatically inferred from their computations. Observers subscriptions are established or cancelled only through explicit (un-)subscription instructions. Graph traversals are depth-minimal, but provide no consistency guarantees. Applications can choose to fire several observer notifications concurrently, but no consistency is provided unless manual synchronization is implemented within some individual observers.

**Reactive Extensions** [Meijer, 2010] are sometimes referred to as "observers on steroids". They provide *Observables* as a uniform model for finite or infinite event streams, with convenient syntax to specify new Observables as derivations and combinations of others. Ready-to-use implementations exist in many different languages, most popularly Rx.NET [Liberty and Betts, 2011], RxJava, and RxJS (JavaScript). Reactive Extensions do determine most required dependency subscriptions automatically, but establishing or cancelling must still be done through explicit (un-)subscription instructions. Change propagation in Reactive Extensions is again depth-minimal, but does not provide any consistency guarantees. Reactive Extensions do address multi-threading, not in a way that provides overarching consistency guarantees such as eventual consistency or isolation. They only ensure that incoming events on each individual Observable are processed under mutual exclusion. Further, they allow individual Observables to be bound to a specific thread, e.g., to ensure that the processing of incoming events of an Observable that modifies an applications' user interface can be bound to always execute on the UI thread.

**Complex Event Processing** [Wu et al., 2006, Cugola and Margara, 2012] is a model for processing event streams almost as if they were database tables. New event streams can be defined as relational algebra queries over other event streams. Complex Event Processing implementations do not address consistency guarantees for change propagation, but may analyze the topology of the operator graph of queries in order to rearrange operators and optimize query execution. They may support concurrent propagation of changes, but also do not provide any overarching consistency guarantees, ensuring only that individual queries' data structures (e.g., join indices) do not get corrupted by race conditions.

### 4.4.1 Event-based Distributed Systems

Event-based programming has been advocated, most prominently by the Reactive Manifesto [Bonér et al., 2014], as the fundamental paradigm on which distributed systems should be built. This is because, similar to eventually consistent Signals in purely Signal-based distributed RP frameworks, event streams in event-based programming almost naturally integrate handling for network delays or failures. Pull-based access to remote state may stall in the presence of network delays or failures, thereby making the application unresponsive. With event-based programming on the other hand, there is no pull-based access to remote state, only push-based event propagation. In case of delays or failures, some event streams will not emit new values for a while, but this does not impact the responsiveness of the remaining application. This section discusses frameworks in this category.

**Publish-Subscribe Systems** [Eugster et al., 2003] enjoy widespread popularity. They consist of a central broker, which allows subscribers to subscribe to specific topics or content patterns of events, and publishers to submit events. The broker ensures that each published event is delivered to all interested subscribers, thereby decoupling event consumers and produces from another. Any publishers or subscribers can connect or disconnect freely without affecting any others, meaning the event propagation topology is dynamic. Publish-Subscribe systems are naturally subject to multiple events being published concurrently, and do provide various forms of consistency guarantees to address this. These include FIFO

or Causal or Serial delivery order for events, but since the event propagation topology does not form an acyclic dependency graph, glitch freedom is not applicable.

**Actors** [Agha, 1986] and **microservice** architectures in general are essentially distributed and concurrent object-oriented programming models. Each object (actor or microservice) has its own thread and communicates with others through asynchronous message passing. They are only distant relatives of event-based programming though, because any object can arbitrarily send a message to any other object. There is no concept of subscriptions to form a dependency graph, and therefore there they cannot implement any topology-based consistency guarantees such as glitch freedom. Each object has a queue of received messages though, and its thread processes incoming messages one after the other, meaning there is only unsophisticated consistency in that messages are processed under mutual exclusion per individual object. The main benefit of actor and microservice architectures lies in their scalability and fault tolerance models. Each actors or microservices is automatically restarted if it crashes or stalls, and it can have multiple instances distributed throughout arbitrarily large computing clusters to implement immensely scalable applications. Some systems even support persistence for individual actors' and microservices' internal state, which is then automatically restored if they are restarted.

**Stream processing** [Abadi et al., 2005], with recent frameworks such as **Spark** [Han and Zhang, 2015] and **Flink** [Carbone et al., 2015], provides big data processing through distributing event processing operator graphs into computing clusters. Implementations mix to varying degrees features similar to those Reactive Extensions and features from Complex Event Processing. Input data is then streamed through this distributed operator graph, exploiting data parallelism for concurrency where possible. Equal to CEP, stream processing frameworks do not provide any consistency guarantees for change propagation bayed on the operator graph topology, but may analyze the topology in order to rearrange operators and optimize execution. They are also not concerned with complex consistency guarantees in relation to concurrency either, since concurrency from data parallelism is commutative.

## 4.5 Automatic Incrementalization

Automatic incrementalization, also called self-adjusting computations, is an area of research that addresses the challenge of automatically deriving incremental implementations of existing batch algorithms. All works in this area of research take as input a traditional batch algorithm, i.e., a non-reactive program that execute once on a single set of input data to produce a single set of output data. Given an initial set of input data, they execute the algorithm normally while recording a trace of thunks of its execution and memoizing all thunks' results. Given a second set of input data similar to the first, they then re-execute the algorithm to compute the updated result, but minimize the number of re-executed thunks by reusing memoized results of thunks whose input values are unchanged.

While most challenges faced by automatic incrementalization are unrelated to Reactive Programming, the two topics are still closely related. In essence, automatic incrementalization transform traditional batch algorithms into reactive applications, which differ from the model considered in this dissertation only in that their state is persisted and the application terminated after an instant completes, and rebooted at the start of the next instant. Traditional batch algorithms are usually iterative, using some form of control flow loops, and are therefore difficult to implement in Reactive Programming, because they are not easy to model as acyclic dependency graphs. By recording an execution trace of such iterative algorithms though, automatic incrementalization essentially unrolls such control flow loops into an acyclic dependency graph, whose nodes are interdependent thunks with memoized results, i.e., similar to RP Signals. Reformulating incremental re-runs not as "which thunks' results can be re-used because of unchanged inputs", but as "which thunks' results must be recomputed because of changed inputs", the very close relation to RP change propagation becomes obvious.

For automatically incrementalized algorithms, glitch freedom and support for dynamic *DG* topology changes are indispensable. Glitch Freedom is indispensable because the original batch algorithms were designed, implemented and tested as regular algorithms executed with well-defined control flow. None

of their thunks can be expected to tolerate any inconsistencies in their inputs from memoized results being recomputed in an order that would not be possible under regular non-incremental execution. Therefore, thunks must be recomputed in a glitch-free order. Support for dynamic *DG* topology changes is indispensable due to the *DG* topology potentially corresponding to unrolled control flow loops whose iteration count may depend on values from the input data. If changed input data results in an increased or decreased number of iterations that a given loop will perform, this decreases or increases the depth of the loop's unrolled dependency graph. While executing an incremental update, different sections of the recorded *DG* may therefore grow and/or shrink.

The original dissertation on self-adjusting computations [Acar, 2005] incrementalizes batch algorithms defined in the functional language **SLf** and the imperative language **SLi**. It implements glitch freedom for its incremental change propagation by recording a logical timestamp at which each thunk originally executed. Because these languages are single-threaded, these timestamps yield a total order over all thunks, and executing recomputations in order of the timestamps therefore achieves glitch freedom. This change propagation is depth-minimal, but not applicable for distributed RP with concurrency.

Several subsequent works address automatic incrementalization of algorithms written in computing models that include some form of parallelism. **PAL** [Hammer et al., 2007] addresses functional programs with parallel `let` expressions. Another work other addresses fork-join-parallel programs with **concurrent revision types** [Burckhardt et al., 2011] for merging the changes to shared variables on joins. **iThreads** [Bhatotia et al., 2015] is a transparent drop-in replacement for the POSIX threading library pthreads, and automatically incrementalizes multi-threaded shared memory programs. All of these use the same mark-sweep change propagation algorithm: Starting from all changed inputs, a first breadth-minimal traversal marks all transitively affected thunks as dirty. A second breadth-minimal traversal then removes all dirty marks by recomputing thunks that have no remaining dirty predecessors. No-change propagation is used where thunks produce unchanged results after recomputations. Any thunks that executed in parallel in the initial execution of the batch algorithm are also re-executed in parallel during incremental change propagation. Mark-sweep change propagation therefore provides glitch freedom and supports concurrent reevaluations per instant in two breadth-minimal graph traversals per instant.

Lastly, some works address automatic incrementalization for certain distributed programs. **Incoop** [Bhatotia et al., 2011] and **Self-Adjusting MapReduce** [Acar and Chen, 2013] incrementalize map-reduce [Dean and Ghemawat, 2008] algorithms. Incoop incrementalizes algorithms automatically by splitting the input data of map and reduce tasks into groups and memoizing each groups' results. Self-Adjusting MapReduce on the other hand requires the map and reduce tasks to be implemented as deliberate self-adjusting computations, and this way even incrementalizes their internal executions. Both of these works also use mark-sweep change propagation. **Reactive Caching for Composed Services** [Burckhardt and Coppieters, 2018] defines a microservice architecture model with automatic incremental change propagation. Each microservice request is a subscription to the request's result, which can change and will result in subscribed microservices recomputing their own results, similar to change propagation over RP Signals. Differently from other works on self-adjusting computations, this work provides only eventual consistency instead of glitch freedom, to integrate fault tolerance into its programming model with similar reasoning as the purely Signal-based distributed RP frameworks discussed earlier.

# 5 FrameSweep: A Scheduler for Thread-Safe Reactive Programming

This chapter introduces the FrameSweep integrated scheduling algorithm, which provides glitch freedom, liveness and isolation in the form of abort-free strict serializability in a single unified implementation by controlling the execution of all RP operations introduced in Section 2, specifically Figure 2.6. Section 5.1 presents, how the architectural integration of FrameSweep into the RP runtime of REScala is derived from, relates to, and improves over which prior works from Chapter 4. Section 5.2 gives an overview of the algorithmic building blocks of FrameSweep, and how the remaining sections of this chapter is structured. The remaining sections present the detailed workings of each building block on a pseudocode implementation of FrameSweep. The pseudocode implementation is simplified in that it does it ignores practical requirements for decentralization, but easier to understand and prove correct. This chapter argues the correctness of this implementation mostly informally though, supported by a continuous running example of several concurrent instants executing on the reactive philosophers application from Chapter 2. Chapter 6 will then provide a thorough formal proof. Finally, Chapter 7 will address the practical concerns for decentralizing the implementation, by replacing several aspects with alternative implementations that are more sophisticated, but semantically equivalent.

## 5.1 RP Architecture with an Integrated Scheduler

The survey in Chapter 4 showed, that Elm [Czaplicki and Chong, 2013] and MV-RP [Drechsler et al., 2018] provide the only algorithms that support synchronous change propagation for concurrent instants executing with fine-grained parallelism. It further showed that Elm does this in a way that is fundamentally incompatible with dynamic dependency graphs and unsuitable for decentralization. Therefore, FrameSweep's design is an evolution of MV-RP, which we summarize briefly. Figure 5.1 shows the architectural blueprint for RP systems of MV-RP, including all components and operations from Figure 2.6. MV-RP adds a concurrency scheduler component into the RP architecture that provides isolation (abort-free strict serializability) to any propagation algorithm that ensures for glitch-freedom and liveness through topological change propagation. The scheduler controls the execution of all interactions between user-defined code (both imperative and reactive), the propagation algorithm, and the dependency graph that stores all reactives' state. In terms of the execution profile of instants, MV-RP adds one breadth-minimal traversal of the dependency graph before the concrete propagation algorithm's change propagation, in order to plan out a safe execution schedule for the instants' reevaluations. In essence, MV-RP therefore allows RP systems to "buy" support for concurrent instants at the cost of executing one additional breadth-minimal graph traversal before each instant.

To be usable in distributed applications, FrameSweep must be decentralized. The concurrency scheduling of MV-RP relies on global data structures, and therefore additional effort is necessary to find a decentralized implementation, which will be addressed as part of Chapter 7. Decentralized propagation algorithms to integrate with MV-RP's concurrency scheduler do exist in prior work though. Assessing these shows that they are all breadth-minimal, and that providing glitch freedom through breadth-minimal traversals generally requires a twofold cost:

- **Mark-sweep propagation from automatic incrementalization** pays the cost of two graph traversals: It first executes a non-topological breadth-minimal graph traversal to mark all reachable thunks as dirty, and second executes another breadth-minimal traversal that clears the dirty marks in topological order.

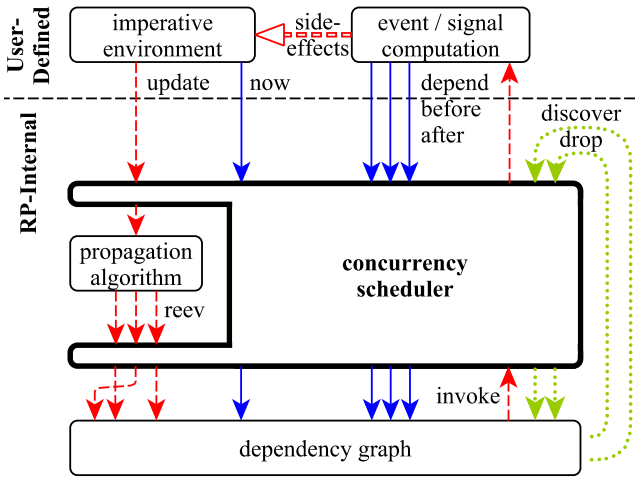



**Figure 5.1:** MV-RP Architecture Blueprint

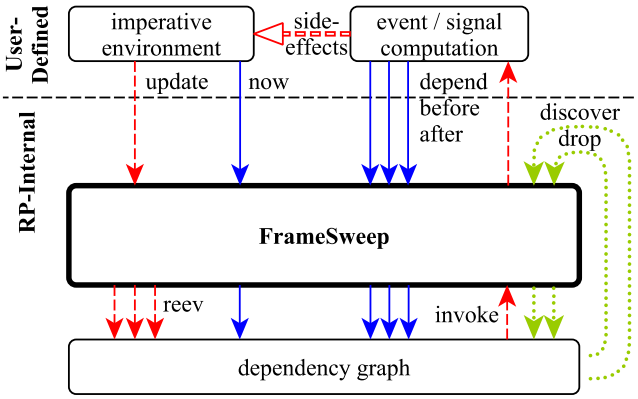



**Figure 5.2:** FrameSweep Integrated Architecture

- **Propagation algorithms from Distributed RP** pay the cost of one breadth-minimal graph traversal plus the cost of tracking, on each reactive, the set or vector clock of all transitively reachable input reactives.

This is corroborated by the following two examples that both are unsuitable for decentralization because they provide weaker properties for lower costs, or better properties for higher costs:

- **Traditional topological sorting** pays the cost of a single graph traversal only, but in turn is non-minimal, which is weaker than breadth-minimal, and unsuitable for distribution because each instant must be started through a central entity that is aware of all input reactives in the entire graph.
- **Height-based priority queue change propagation** pays the cost of one graph traversal plus the cost of tracking, on each reactive, the length of the longest path from any transitively reachable input reactive. While heights are cheaper to compute and store than sets or vector clocks of transitively reachable inputs, they actually cause a higher cost for instants, because their over-approximation of transitive reachability between reactives prevents a lot of parallelization unnecessarily. The communication required to prevent this parallelization is simultaneously what makes this algorithm unsuitable for distribution. In return though, its graph traversal is depth-minimal, which is stronger than breadth-minimal.

Given this observation, a straight-forward integration of any of these algorithms with MV-RP would imply a threefold cost:

- **MV-RP + Pure topological sorting:** Two graph traversals plus the loss of breadth-minimalism.
- **MV-RP + Mark-Sweep propagation from automatic incrementalization:** Three graph traversals.
- **MV-RP + Propagation algorithms from Distributed RP:** Two graph traversals plus maintaining the set of transitively reachable input reactives per reactive.

FrameSweep improves over this expectation though. In the original design of MV-RP (Figure 5.1), the responsibility of scheduling reevaluations is split between propagation algorithm and concurrency scheduler, with both implementing different aspects of it. FrameSweep removes the separation between a propagation algorithm and concurrency scheduler though, and instead uses an integrated design as visualized in Figure 5.2 with a single scheduler for both concurrency and glitch freedom (and liveness). This

allows FrameSweep to exploit a synergy in that it can avoid redundancies between separately maintained infrastructure for glitch freedom and infrastructure for isolation. As a result, FrameSweep achieves a breadth-minimal implementation that ensures glitch freedom, liveness and isolation in the form of abort-free strict serializability at only a twofold cost. Specifically, FrameSweep executes each instant through only two breadth-minimal graph traversals, without any additionally maintained topology information per reactive. Both the concurrency planning of MV-RP and the dirty marks of a mark phase are established by the first graph traversal, and the second traversal then uses this infrastructure to execute reevaluations in an order that is both serializable and glitch-free.

## 5.2 Main Building Blocks

Overall, FrameSweep combines intertwines the following building blocks from prior works into a single algorithm:

- The mark-sweep change propagation algorithm from automatic incrementalization for glitch-freedom and liveness.
- The counter-based optimizations for the sweep phase developed in the context of the SID-UP distributed change propagation algorithm [Drechsler and Salvaneschi, 2014].
- The algorithm components of the MV-RP concurrency scheduler for abort-free strict serializability: several established algorithms for concurrency control from databases and one custom concurrency control technique.

The following list discusses the origins of each building block, gives a brief overview of its functionality, and explains, which aspects of FrameSweep's consistency guarantees it is responsible for. In this list, the responsibility of glitch freedom is split into the aspects of marking and sweeping. The responsibility of isolation, i.e., consistently ordering non-commutative interactions between of concurrent instants, is split into four different aspects, depending on what operations interact on which variables: Interactions of reads with reads on any variable are irrelevant because they are commutative. Write-write conflicts, i.e., ordering writes against other writes of concurrent instants, are addressed as one aspect together for all variables. Read-write conflicts, i.e., ordering reads against writes of concurrent instants, are addressed as three aspects, one for each variable: user value, incoming dependencies set or outgoing dependencies set.

- **C2PL/Mark.** *Conservative two-phase locking* (C2PL) [Bernstein et al., 1986] is a pessimistic lock-based concurrency control technique from databases. C2PL provides abort-free strict serializability for transactions, by acquiring locks for all variables they will access before starting their execution. To do so, the set of variables of each transaction must be known before starting their execution. This is inconvenient to specify manually, because it clutters the syntax for executing transactions, and is not feasible to determine automatically, if transactions are expressed in a language too complex for a sound static analysis. In the context of RP though, it can be automated – such that it does not affect the syntax of executing instants – for all reevaluations of every instant: Given the input reactives that an instant is about to change, a traversal of the *DG* starting from there (i.e., breadth-minimal) will reach all reactives that the instant may potentially reevaluate. FrameSweep exploits this traversal for a two-fold purpose.

  First, the traversal implements C2PL by locking every reached reactive in advance of the instant later potentially executing a reevaluation (dashed red arrows in Figure 2.6/5.1/5.2). This allows ordering ambiguities between reevaluations of concurrent instants to be sorted ahead of their execution. As the proofs in Section 6 will show, all write-write conflicts on all variables, as well as all read-write conflicts on incoming dependencies sets occur only between reevaluations of the same

reactive. C2PL therefore pre-solves all these conflicts, such that none will occur during the instants' following change propagation, and thereby providing abort-free strict serializable execution for reevaluations. Read-write conflicts on user values and outgoing dependencies sets may occur between reevaluations of different reactives though, and are therefore not managed by C2PL, but addressed below.

Second, the traversal locks precisely those reactives that a mark-sweep propagation algorithm would mark as dirty during its mark phase. By using acquired locks to simultaneously serve as dirty markers for the respective instant's glitch freedom scheduling, the C2PL traversal thus also implements the mark phase at no extra cost.

- **Sweep.** FrameSweep achieves liveness and glitch freedom by implementing the propagation phase of instants as only the topological sweep phase (again breadth-minimal), on top of the dirty-marking locks from C2PL. This is how FrameSweep provides synchronous propagation semantics (liveness, glitch-freedom and isolation) at the cost of only two breadth-minimal graph traversals, without maintaining heights or source sets or source vector clocks on all reactives. To implement the sweep traversal more efficiently and make its visualization more intuitive, FrameSweep employs the counter-based optimizations developed in the context of the SID-UP propagation algorithm for distributed RP: Rather than repeatedly determining each reactives' readiness for serializable and glitch-free reevaluation by counting their number of locked/dirty and changed dependencies, they track these numbers incrementally in the form of stored `pending` and `changes` counters for each instant. After the reevaluation of a reactive completes, all successor reactives' `pending` counters are decremented and `changes` counters incremented if the reevaluation changed the reactive. Once a reactive's `pending` counter reaches zero, it is ready for a glitch-free reevaluation, because it has no further predecessor that are still marked dirty (locked). If, at that point, the `changes` counter is also zero, its reevaluation is skipped as unchanged because none of its dependencies values changed, which in turn decrements all successors reactives' `pending` counters, but does not increment their `changes` counters. This implements no-change propagation, to ensure that change propagation behaves as if it was depth-minimal, even though the sweep traversal is only breadth-minimal.

- **MVCC.** Unlike reevaluations, read operations (`before`, `after`, `depend` and `now`, solid blue arrows in Figure 2.6/5.1/5.2) by user computations and imperative threads may target reactives arbitrarily, i.e., cannot be predicted ahead of time. Thus C2PL is inapplicable to prevent read-write conflicts on reactives' user values. Instead, FrameSweep employs *multiversion concurrency control* (MVCC) [Bernstein et al., 1986], also from databases. MVCC maintains a backlog of past values (called *versions*) that were written to every reactive's user value under the control of C2PL. Because read operations are idempotent, MVCC can safely execute them "in the past" by returning old versions from the backlog, if necessary to ensure a serializable return value. MVCC thus consistently resolves read-write conflicts on reactives' user values on-the-fly, and thereby provides abort-free serializable execution – and with some additional care also strict – for read operations. Since writes are not idempotent, they cannot be executed in the past, meaning the combination of C2PL for reevaluations and MVCC for reads is indispensable for FrameSweep to remain abort-free.

- **Retrofitting.** Read-write conflicts on outgoing dependencies are caused by dynamic dependency edge changes (`drop` and `discover`, dotted green arrows in Figure 2.6/5.1/5.2). They cannot be prevented by C2PL, since `drop` and `discover` are executed on the same reactives where `depend` was executed, and are thus equally impossible to predict ahead of time. These operations can also not be executed "in the past" through MVCC, because they are write operations and therefore not idempotent. Instead, FrameSweep employs *retrofitting*, a custom technique that provides abort-free strict serializable execution of dynamic *DG* edge changes by enabling their execution "in the past" despite their writes affecting the system's state (i.e., not idempotent). Retrofitting

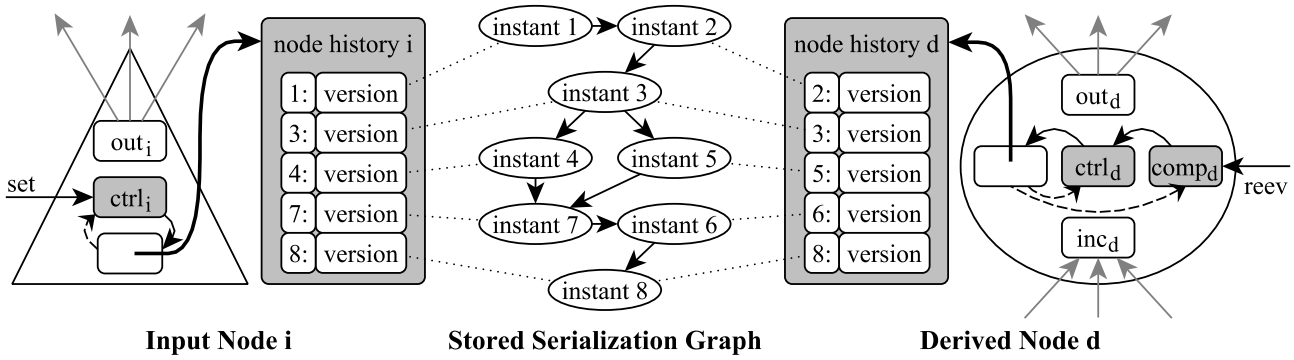



**Figure 5.3:** Node Composition with Node Histories Storing Node Versions

exploits a synergy of topological propagation for glitch freedom and liveness, and consistent ordering of reevaluations between concurrent instants for isolation, provided by the combination of C2PL/Mark, Sweep and MVCC outlined so far. This synergy allows retrofitting to safely rewrite the history of affected instants, un- or re-doing effects as necessary, to restore serializability after executing effectful writes to outgoing dependencies sets "in the past", while maintaining both glitch freedom and liveness. Retrofitting therefore forms a symbiotic relation with the other algorithms, in that it is applicable only because of the precise set of properties that they provide, and then serves to resolve read-write conflicts on reactives' outgoing dependencies sets which the other algorithms cannot handle, upholding this very set of properties.

- **Serialization Graph Testing.** Established database theory states, that if a transaction system handles different non-commutative interactions between concurrent transactions with different scheduling algorithms, serializability is achieved as long as all non-commutative conflicts are handled, and all algorithms apply the same *serialization order*, i.e., the logical order in which transactions appear to execute [Bernstein and Goodman, 1981]. Above outline already discussed, in which classes FrameSweep splits non-commutative conflicts, and which classes are handled by which algorithms: C2PL handles write-write conflicts on all variables and read-write conflicts on incoming dependencies sets, MVCC handles read-write conflicts on user values, and Retrofitting handles read-write conflicts on outgoing dependencies sets. To facilitate the common serialization order across these algorithms, FrameSweep uses *Serialization Graph Testing* [Bernstein et al., 1986], also from databases. Serialization graph testing tracks at run-time a model of the serialization order as formally defined over the transaction system's execution history, in form of the directed acyclic *stored serialization graph SSG* over all transaction as nodes, or instants in the case of FrameSweep. When a non-commutative conflict between operations of concurrent instants occurs, testing for which paths exist in the *SSG* between these instants – which is what gives the technique its name – tells in which order these operations should appear to execute for the systems' history to remain serializable. The expression $a \xrightarrow[SSG]{*} b$ denotes a path from instant a to instant b, which means that operations by a must appear to execute before those of b. When no such path exists in either direction, the order between a and b has not yet been determined, and must be chosen and recorded as a new edge $a \xrightarrow[SSG]{} b$ so that subsequent conflicts will be resolved the same way. The *SSG* therefore incrementally grows with new edges while FrameSweep executes instants' operations.

FrameSweep connects all of its algorithmic components through a single data structure, called *node history*. Each reactive has one node history, which replaces the user value variable in the reactives' composition, as visualized in Figure 5.3 (cf. Figure 2.7). Each node history stores one *node version* for each instant that affected the respective reactive through some operation. Node histories interact with

serialization graph testing in that their contained versions are sorted according to the order described in the *SSG*. Figure 5.3 visualizes an exemplary *SSG* of eight instants in the center. Dotted lines connect each instant to its corresponding node versions in the node histories of input node $i$ and derived node $d$. The *SSG* globally describes only a partial order. In Figure 5.3, e.g., instants 4 and 5 are unordered. But, the *SSG* describes a total order over all node versions within each reactive's node history. In Figure 5.3, e.g., for the node history (1, 3, 4, 7, 8) on $i$, the *SSG* order is total, with existing paths $1 \xrightarrow[SSG]{*} 3 \xrightarrow[SSG]{*} 4 \xrightarrow[SSG]{*} 7 \xrightarrow[SSG]{*} 8$.

The functionalities of every other algorithm component in FrameSweep are implemented as operations that modify or extract information from reactives' node histories. After Section 5.3 gives a precursory overview on how FrameSweep supports or restricts concurrent execution for all its operations, the remaining sections discuss these operations in detail: Section 5.4 presents, how C2PL/Mark models locking queues and marks reactives as dirty by creating *placeholder* node versions. Section 5.5 discusses, how the topological sweep propagation uses `pending` and `changes` counters stored in these placeholders to decide when pending reevaluations can be executed glitch-free and serializable, and how resulting changed values are stored in written versions that replace these placeholders. Section 5.6 explains how MVCC implements the different read operations' semantics by selecting, when and which value to return for each read operations based on the placeholders and written versions that are present within reactives' node histories. Finally, Section 5.7 shows how retrofitting rewrites applications' execution history by rearranging the contents affected reactives' node histories after non-serializable dynamic *DG* edge changes.

## 5.3 Concurrency of FrameSweep's Operations

By default, concurrent instants may execute all of FrameSweep's operations concurrently. To ensure that this does not result in race conditions that may break the integrity of FrameSweep's data structures (node histories and the *SSG*), this chapter uses the simple solution of placing operations under different scopes of mutual exclusion. For any given reactive, only a single operation that modifies its node history is allowed to execute at any given point of time. In the following pseudocode listings, these procedures are annotated with the keyword `locally-exclusive`. Some of these operations, for a part of their execution, must simultaneously interact with the *SSG*. Of all these operations executing concurrently on all reactives, only a single one is allowed to do so at any given point of time. These procedures are annotated with the keyword `globally-exclusive`. As Section 7.3 will discuss in more detail, this simpler approach has certain downsides, e.g., global locking is incompatible with decentralization. It will presents a more elaborate solution that requires less restrictive synchronization and thereby avoids these downsides.

FrameSweep further increases the amount of concurrency by parallelizing both its graph traversals by the use of a thread pool. Whenever a graph traversal recurses to the outgoing dependencies of a reactive, it does so by submitting one task for each connected outgoing dependency to the task pool. Applied to the propagation phase, this parallelization implements the second dimension of concurrency, concurrent execution reevaluations within individual instants. The pseudocode presented throughout the remainder of this chapter interacts with the thread pool through the `submit` keyword and `activeTasks` counters. A `submit <call>` statement is similar to a `execute <call>` statement, but adds the specified call as a new task to the thread pool's task queue for asynchronous execution, rather than executing it directly. As a result, the call escapes any mutual exclusion scopes that enclosed the `submit` statement, as well as the call hierarchy. Instants instead determine termination of their graph traversals by the help of the thread pool counting the number of submitted, but not yet completed tasks. Each instant has an `activeTasks` counter. In every `submit` statement, the specified task always has at least two parameters: first the targeted reactive, and second the instant with which this task is associated. When a task is submitted, the thread pool increments the `activeTasks` of the associated instant. When the task's execution later

completes, the thread pool afterwards decrements the associated instant's `activeTasks` counter again. This means, all tasks of a given instant's graph traversal have completed once the instant's `activeTasks` becomes zero again.

## 5.4 Framing Phase

FrameSweep transparently executes every `update(`$\texttt{i}_1$ `->` $\texttt{v}_1$`,` $\texttt{i}_2$ `->` $\texttt{v}_2$`, ...)` call as a transactional instant through `procedure update`, defined in Listing 5.1 Line 26ff., consisting of the framing phase, discussed here, followed by the propagation phase, whose various aspects are discussed throughout the following sections. The goal of the framing phase is to create a *placeholder* in the node history of every reactive that the instant may potentially reevaluate. These placeholders serve two purposes. First – for serializability – all placeholders on a given reactive represent a queue for the reactive's exclusive lock. Only the instant that has the first (in terms of instants' order in the *SSG*) placeholder on a given reactive is allowed to reevaluate that reactive, while all other placeholders must wait. When the first placeholder is eventually removed, this privilege immediately falls to the next placeholder, if further ones already exist. Second – for glitch freedom – an instant's placeholder in a node history represents that the corresponding reactive is currently marked dirty by that instant. Since a planned reevaluation means that the value of the reactive may change in the future, any reactives that depend on it might produce a glitch if they were reevaluated while the placeholder still exists.

The entry point for the framing phase is `procedure framingPhase`, defined in Line 30ff. It is implemented as a graph traversal over the *DG*, starting from all declared inputs (Line 32), e.g., `declaredInputs = { i1, i2 }` for an `update(`$\texttt{i}_1$ `->` $\texttt{v}_1$`,` $\texttt{i}_2$ `->` $\texttt{v}_2$`...)` call. The recursive task, which executes on every reached reactives to ensure that a placeholder exists, is `procedure Reactive.frame`, defined in in Line 38ff. First, it executes `procedure Reactive.ensureVersion` (Line 44ff), which encapsulates the synchronization of creating a new node version `reactive[instant]` – unless one already exists(Line 45) – with the necessary updates to the *SSG*.

The logic to newly create version `reactive[instant]` carries a certain complexity, but most of it becomes relevant only later, and the following simplification suffices for now. First, `ensureVersion` collects in set `P` (Line 46ff.) all concurrent instants `conc` that already have a node version `reactive[conc]` on the parameter `reactive`, but are not yet ordered in *SSG* against the parameter `instant` (set `F` remains empty since there no earlier phase than the framing phase exists). If any such `conc` instants exist (Line 54), new ordering relations must be added to the *SSG* to ensure that it still defines a total order over all node version on the parameter `reactive` after the a new node version for the parameter `instant` has been added. To do so, `ensureVersion` selects `pred` as the latest instant in terms of the *SSG* order from set `P` (Line 55). Next, it creates edge `pred` $\xrightarrow[SSG]{}$ `instant` (Line 56), which by transitivity with the previously existing *SSG* edges orders all `conc` instants as `conc` $\xrightarrow[SSG]{*}$ `instant`, reflecting that the parameter `instant` was the last to arrive at the reactive. Lastly, `ensureVersion` creates the new version `reactive[instant]` with initially both the `pending` and `changed` counter set to zero and an empty user value (Line 60).

Returning to `frame`, once `reactive[instant]` is guaranteed to exist, the framing traversal increments its `pending` counter (Line 40). The framing traversal must recurse to successor reactives only the first time that it reaches the parameter `reactive` and newly marked it dirty, but not if it reaches it again. This is ensured by recursing to successor reactives (Line 42f.) only if the increment changed the counter's value from 0 to 1 (Line 41). If the counter value changed in any other way, e.g., from 1 to 2, the traversal does not recurse to all successors again. The incremented `pending` counter is retained though, because it reflects that the parameter `reactive` has an additional dirty predecessor. This way, the framing traversal not only creates placeholders on all reachable reactives to queue for their lock and mark them as dirty, but it also directly initializes the `pending` counters for the following sweep phase to the correct number of dirty predecessors.

```
26 procedure update(inputChanges):
27   let instant = <new>
28   execute framingPhase(instant, inputChanges.keys)
29   execute propagationPhase(instant, inputChanges.keys, inputChanges)

30 procedure framingPhase(instant, declaredInputs)
31   # instant.phase == Framing
32   foreach (input ∈ declaredInputs):
33     submit frame(input, instant)
34   suspend until ( globally-exclusive {
35     instant.activeTasks == 0 ∧∀ pred ⟶_SSG instant: pred.phase ≥ Propagating
36   })
37   globally-exclusive { update instant.phase := Propagating }

38 locally-exclusive procedure Reactive.frame(reactive, instant):
39   execute ensureVersion(reactive, instant)
40   increment reactive[instant].pending += 1
41   if (reactive[instant].pending == 1):
42     foreach (derived ∈ reactive.outDeps):
43       submit frame(derived, instant)

44 globally-exclusive procedure Reactive.ensureVersion(reactive, instant):
45   if (∄ reactive[instant]):
46     let P := { conc | ∃ reactive[conc] ∧
47                       ∄ conc ⟶*_SSG instant ∧
48                       ∄ instant ⟶*_SSG conc ∧
49                       conc.phase >= instant.phase }
50     let F := { conc | ∃ reactive[conc] ∧
51                       ∄ conc ⟶*_SSG instant ∧
52                       ∄ instant ⟶*_SSG conc ∧
53                       conc.phase < instant.phase }
54     if (P ≠ ∅):
55       let pred := max_SSG(P)
56       add edge pred ⟶_SSG instant.
57     if (F ≠ ∅):
58       let succ := min_SSG(F)
59       add edge instant ⟶_SSG succ.
60     create reactive[instant] := (pending := 0, changed := 0, value := None)
```

**Listing 5.1:** Pseudocode implementations of framing phase routines.

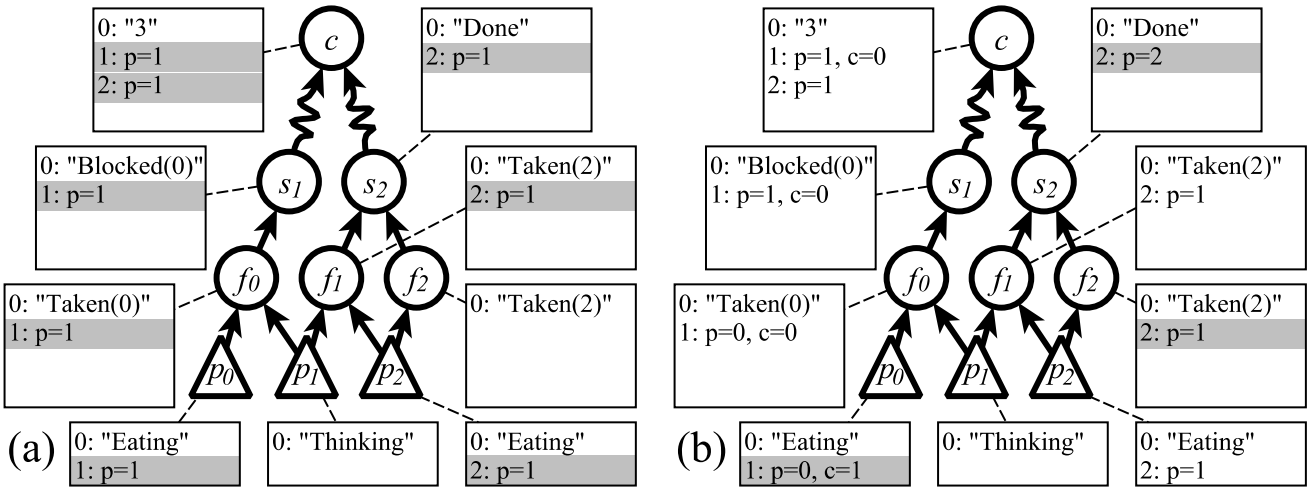



**Figure 5.4:** FrameSweep Example Trace: Framing Phase

Returning finally to `framingPhase`, any parameter `instant` must await two conditions to be fulfilled before switching to the propagation phase. First, the `activeTasks` counter must reach zero to indicate that all submitted tasks of the framing traversal have been completed. Additionally, all instants that were established as predecessor instants so far, i.e., all `pred` for which a path `pred` $\xrightarrow[SSG]{*}$ `instant` exists, must have transitioned to their own propagation phases. This second restriction establishes a global invariant in FrameSweep, that no instant must ever have a predecessor instant that is in an earlier phase, e.g., no propagating instant must have a predecessor that is still framing. For the transition from framing to propagating here, the rationale for this invariant is as follows. Consider a propagating instant executing a reevaluation for its first placeholder in a reactives' node history. If a predecessor instant existed that is still executing its framing traversal, this might still create an earlier placeholder on that same reactive. This means, that predecessor instant would have reevaluate the reactive before the propagating instant, but since the propagating instant already started its reevaluation, serializability is lost. Thus, to ensure that first placeholders are reliable and cannot no longer be challenged, propagating instants must never have predecessor instants that are still framing. Suspending instants after their framing traversal until after all predecessor instants have begun propagating establishes this invariant, and since new framing instants will always only order themselves after other instants through `ensureVersion` (explained above), it is subsequently maintained.

### Demonstration

Figure 5.4 (a) shows the first step of a running example that demonstrates FrameSweep's processing of concurrent instants on the reactive philosophers application introduced in Section 2. In the center, it visualizes the relevant section of the *DG* through bold nodes and edges in the middle of each step. The *DG* section consists of input Vars `phils(1)` through `phils(3)` ($p_1$, $p_2$, $p_3$), derived Signals `forks(1)` through `forks(3)` ($f_0$, $f_1$, $f_2$), `sights(1)` and `sights(2)` ($s_1$, $s_2$), and `totalCount` ($c$). Both $s_1$ and $s_1$ are shown connected to $c$ through squiggly edges, because these connections are not direct edges $s_1 \xrightarrow[DG]{} c$, but indirect paths $s_1 \xrightarrow[DG]{*} c$ over other reactives that are omitted. The node histories of all reactives are visualized as rectangles attached to the respective nodes via a dashed line. Each line of text in these rectangles visualizes one node version. The lines starting with label "0:" (white background) correspond to node versions that form the initial state for the running example: Both $p_0$ and $p_2$ are `Eating` and $p_1$ is `Thinking`, all derived reactives' values are up to date with respect to these inputs, and $c$ has counted up to a value of 3 so far; as a consequence of $s_1$ having changed to `Blocked(1)` during its latest reevaluation, edge $f_2 \xrightarrow[DG]{} s_1$ was dropped and currently does not exist.

Lines with a grey background correspond to node versions that have been changed or newly created in each step of the example. In Figure 5.4 (a), these are placeholders created by the framing phase of two concurrent instants: $t_1$ and $t_2$, which admitted `Thinking` to $p_0$ and $p_2$ respectively. The framing traversal of $t_1$ has already visited all reachable reactives: lines "1: p=1", which stand for "placeholder by $t_1$ with `pending` counter at value 1", exist in the node histories of $p_0$, $f_0$, $s_1$ and $c$. The `changes` counters of placeholders are not visualized on placeholders of framing instants, because they all are at value zero and not relevant yet. The framing traversal of $t_2$ is still in progress. It already created placeholders "2: p=1" on $p_2$, $f_1$, $s_2$ and $c$, but has not visited $f_2$ yet – the corresponding `frame` task is currently queued in the thread pool's task queue. On $c$, $t_2$ created its placeholder after $t_1$, and hence established the order $t_1 \xrightarrow[SSG]{} t_2$. The *SSG* is not visualized throughout the running example, but as a mnemonic, the example crafted such that instants order themselves matching their indices, i.e., $t_1 \xrightarrow[SSG]{*} t_2 \xrightarrow[SSG]{*} t_3$.

As $t_1$ did not establish any predecessor relations during its execution, it can transition to the propagation phase at this point. In Figure 5.4 (b), $t_1$ has performed this transition, visualized by all its placeholders now also showing the `changes` counters, e.g., "c=0". The framing traversal of $t_2$ has newly traversed over $f_2$, where it created a new placeholder, to $s_2$, where it incremented the `pending` counter to "p=2". This corresponds to $s_2$ now having two dirty predecessors ($f_1$ and $f_2$). Moreover, it means that the traversal did not recurse past $s_2$, because it already did so when it visited $s_2$ for the first time in Figure 5.4 (a). Thus, the framing traversal of $t_2$ is now also complete and $t_2$ is also allowed to begin its propagation phase, because while it does have $t_1$ as a predecessor, $t_1$ too is already propagating.

## 5.5 Propagation with Pipeline Parallelism

During the propagation phase, FrameSweep executes a second breadth-minimal topological sweep traversal of the $dg$ for each instant. The propagation traversal removes all placeholders that the instant created during the framing phase by executing the planned reevaluations in a glitch-free order. If any reactives' reevaluation yields a new user value, then the corresponding placeholder is replaced with a written versions that store this new value.

The entry point for the propagation phase of instants is `procedure propagationPhase`, defined in Listing 5.2 Line 61ff. The parameter `changedInputs` is the map specified by the imperative `update(`$\texttt{i}_1$ `-> `$\texttt{v}_1$`, ...)` call that started the instant, associating each changed input with its new value. Equal to the framing traversal, the propagation traversal starts from all `declaredInputs` (Line 63). The propagation traversal is noticeably more complex though, for several reasons. It needs to treat input reactives differently from derived reactives, because their values are assigned rather than recomputed. Further, it needs to handle many different cases, e.g., depending on what kind of reactives are reevaluated and whether or not reevaluations change reactives' values, and therefore consists of multiple mutually recursive functions.

### Reevaluating Input Reactives

For every input reactive, FrameSweep first executes `procedure awaitFirstPlaceholder`, defined in Line 75ff. This suspends execution until the placeholder of the parameter `instant` has become the first placeholder in the node history of the parameter `reactive`, i.e., until the parameter `instant` is allowed to reevaluate the `reactive`. Afterwards, FrameSweep updates the placeholder's counters from `pending = 1, changed = 0` to `pending = 0, changed = 1` (Line 65f.), to ready this placeholder for reevaluation. Then, FrameSweep passes the newly assigned value[1] to the input reactive's control code

[1] The propagation phase tolerates a discrepancy where `changedInputs` assigns new values to only a subset of the `declaredInputs` from the framing phase (Line 69f.). This discrepancy can originate from an API for extended transactions that contain more operations than only a single `update(`$\texttt{i}_1$ `->` $\texttt{v}_1$`, ..)` calls, which will be introduced later in Section 7.5. Supporting this discrepancy does not complicate the scheduler beyond this single-line else case though, and therefore there is no need for a deeper understanding of this use case yet. For input reactives that were included as

```
61 procedure propagationPhase(instant, declaredInputs, inputChanges):
62   # instant.phase == Propagating, inputChanges.keys ⊆ declaredInputs
63   foreach (input \in declaredInputs):
64     execute awaitFirstPlaceholder(changedInput, instant)
65     update input[instant].changes := 1
66     update input[instant].pending := 0
67     if (inputChanges.get(input) matches Some(newValue)):
68       submit control(changedInput, instant, newValue)
69     else:
70       submit reevOut(unchangedInput, instant, None)
71   suspend until ( globally-exclusive {
72     instant.activeTasks == 0 ∧∀ pred ⟶(SSG) instant: pred.phase ≥ Completed
73   } )
74   globally-exclusive { update instant.phase := Completed }

75 procedure Reactive.awaitFirstPlaceholder(reactive, instant):
76   suspend until (∀ reactive[pred]: pred ⟶*(SSG) instant ⇒
77                  reactive[pred].pending == 0 ∧ reactive[pred].changes == 0)

78 locally-exclusive procedure Reactive.reevOut(reactive, instant, maybeNewValue):
79   # maybeNewValue is optional, i.e., either None or Some(newValue)
80   if (maybeNewValue != None):
81     update reactive[instant].value := maybeNewValue
82     update reactive[instant].changes := 0
83     foreach derived ∈ reactive.outDeps:
84       submit notify(derived, instant, Changed)
85   else:
86     update reactive[instant].changes := 0
87     foreach derived ∈ reactive.outDeps:
88       submit notify(derived, instant, Unchanged)
89   let succs = { succ | ∃ reactive[succ] ∧
90                        (reactive[succ].pending > 0 ∨ reactive[succ].changes > 0) }
91   if (succs != ∅):
92     let immediateSucc = min_SSG(succs)
93     if (reactive[immediateSucc].pending == 0):
94       submit reevaluate(reactive, immediateSucc)

95 locally-exclusive procedure DerivedReactive.notify(reactive, instant, change):
96   # change is boolean Changed or Unchanged
97   execute ensureVersion(reactive, instant)
98   if (change == Changed):
99     increment reactive[instant].changes += 1
100  decrement reactive[instant].pending -= 1
101  if (reactive[instant].pending == 0):
102    if (∃ reactive[pred]: pred ⟶*(SSG) instant ∧
103        (reactive[pred].pending > 0 ∨ reactive[pred].changes > 0)):
104      if (reactive[instant].changes == 0):
105        foreach derived ∈ reactive.outDeps:
106          submit notify(derived, instant, Unchanged)
107    else if (reactive[instant].changes == 0):
108      execute reevOut(reactive, instant, None)
109    else:
110      submit reevaluate(reactive, instant)
```

**Listing 5.2:** Pseudocode implementations of propagation routines.

(Line 67). The control code (cf. Listing 2.9) will in turn execute `procedure reevOut`, defined in Listing 5.2 Line 78ff., with the `maybeNewValue` parameter either defined or not, depending on whether or not the newly assigned value actually constitutes a change for the specific type of input reactive (e.g., reassigning its current value to a Var is not a change).

#### Propagating Reevaluation Results

With `reevOut`, the propagation phase enters the mutual recursion that implement the propagation traversal over derived reactives. If the reevaluation result constitutes a change for the reactive (i.e., parameter `maybeNewValue` is defined, Line 80 ff.), `reevOut` replaces the placeholder by a written version, by zeroing the `changes` counter (`pending` is already zero) and storing the new user value instead. If the reevaluation result does *not* constitute a change for the reactive (parameter `maybeNewValue` is undefined, Line 85 ff.), the placeholder is instead replaced by an *idempotent* version, by only zeroing the `changes` counter (again, `pending` is already zeroed) without storing a user value. Idempotent version are called such because they have no further user-visible effect on other operations: they neither block other operations from executing, nor influence the value returned by any read operations. The only purpose of idempotent versions is to leave a record that an instant affected the reactive in question in some way and ensure that any instants that later execute further operations on the reactive will order themselves in the *SSG* against that instant.

Replacing a placeholder during `reevOut` has two further effects. First, the parameter `reactive` is no longer marked dirty for the parameter `instant`, but may now be marked changed instead if it was replaced by a written version. Consequently, the `pending` and `changes` counters on successor reactives' placeholders must be updated accordingly, which may make them ready for glitch-free reevaluations. This is done by submitting a `notify` task for every successor reactive – the inner workings will be discussed in the next paragraph – with the `change` parameter set as `Changed` or `Unchanged` respectively. Second, because `reevOut` removes the first placeholder, the parameter `instant` simultaneously "unlocks" the parameter `reactive`. This eager unlocking allows the next instant to immediately start a reevaluation of the reactive, assuming it is glitch-free. As a final step, `reevOut` therefore collects all `succ` instants that have further placeholders on the same reactive (Line 89f., a successor relation `instant` $\underset{SSG}{\longrightarrow}$ `succ` is implied because the placeholder of the parameter `instant` was the first). If any such `succ` instants exist (Line 91), `reevOut` selects the earliest one in terms of the *SSG* order (Line 92). If this earliest placeholder is ready for glitch-free reevaluation (Line 93, `pending == 0` meaning no more dirty dependencies), `reevOut` additionally submits this subsequent reevaluation by `succ` as a new task to the thread pool (Line 94). FrameSweep thus enables a very high degree of parallelism, called *pipeline parallelism* [Czaplicki and Chong, 2013], where concurrent instants can make progress simultaneously even in mutually affected regions of the *DG*.

#### Reevaluating Derived Reactives

The arrival of propagated changes on successor reactives is implemented by `procedure notify`, defined in Line 95ff. The first step of `notify` is to update the counters of the placeholder `reactive[instant]`. In the case that the `change` parameter is set as `Changed`, `notify` will increment the `changes` counter (Line 98f.) to reflect that the dependency where `reevOut` replaced a placeholder with written version is now marked changed for the parameter `instant`. Further, regardless of whether `Changed` or `Unchanged`, `notify` always decrements the `pending` counter (Line 100) to reflect that the dependency is no longer marked dirty for the parameter `instant`, regardless of which kind of node version the placeholder was replaced with. If `reactive[instant]` has become glitch-free from the `pending` counter update (Line 101, `pending == 0` meaning no remaining dirty dependencies for the parameter `instant`), further actions may be necessary. This depends on whether or not an earlier placeholders

`declaredInputs` in the framing phase, but then are not associated with a new value in `changedInputs`, FrameSweep simply skips the control code invocation and propagates a no-change by directly executing `reevOut` with the `maybeNewValue` parameter undefined.

exists on the parameter `reactive` (Line 102), and whether or not the version has become idempotent, i.e., if `changed == 0` (Line 104 and Line 107) in addition to `pending == 0` from above.

- If an earlier placeholder exists and `reactive[instant]` has become idempotent (Line 105f), `notify` submits further `notify` tasks with the `changed` parameter set to `Unchanged` for all successor reactives. This process implements no-change propagation of instants on reactives where they are pipelined behind earlier instants' placeholders, through recursion of just `notify`. Note that this implies that no-change propagation of a later instant can "overtake" the incomplete propagation of earlier instants, i.e., no-change propagation may not adhere to the correct serialization order. This does not violate FrameSweep's correctness though, because no-change propagation does not involve the execution of any user computations, and therefore is not user-visible.

- If no earlier placeholder exists and `reactive[instant]` has become idempotent (Line 108), `notify` executes an `reevOut` with no new user value. This will in turn equally submit an `Unchanged` `notify` task for all successor reactives. Additionally though, if another placeholder becomes the new first placeholder and is ready for glitch-free reevaluation, `reevOut` will spawn a subsequent reevaluation. This process implements no-change propagation of instants on reactives where they are not pipelined behind incomplete propagations of earlier instants, through mutual recursion of `notify` and `reevOut`.

- If no earlier placeholder exists and `reactive[instant]` has not become idempotent (Line 109f., meaning `changes > 0`), then a reevaluation for the parameter `instant` itself is submitted. This executes the user computation and passes the eventually returned result to the reactive's control code, using the unchanged implementation of Listing 2.9 introduced in Chapter 2. In particular, the implementation remains without any mutual exclusion scopes. While the placeholder locking queue ensures that only a single reevaluation executes at any time, this reevaluation does not prevent any other operations such as `frame`, `notify`, any read operations or dynamic dependency edge changes from executing concurrently on the same reactive. Only when the reactive's control code eventually passes its result to `reevOut`, this executes mutually exclusively against other scheduling operations again. Then, `reevOut` again submits an `Unchanged`, or in this case potentially also a `Changed` `notify` task for all successor reactives, alongside again potentially triggering a subsequent reevaluation. This process implements regular change propagation of instants through mutual recursion of `notify`, `reevaluate` with its sub-calls, and `reevOut`.

- In the remaining case, i.e., if `reactive[instant]` has not become idempotent with `changes > 0` but an earlier placeholder still exists (this corresponds to the missing `else`-case of the `if` condition in Line 104), nothing is done. This is because the corresponding reevaluation, while ready for execution in terms of glitch freedom, is *not* ready for execution in terms of serialization order with other concurrent instants' reevaluations. The reevaluation will instead be submitted as a subsequent reevaluation at a later point, when the instant owning the last preceding placeholder removed it through its own execution of `reevOut`. This process implements the case where the change propagation traversal temporarily becomes dormant on reactives instead of recursing further, to adhere to the correct serialization order of pipelined placeholders.

#### Demonstration

For demonstrating the propagation phase, the running example already included first step in Figure 5.4 (b), where the placeholder of $t_1$ on $p_0$ was updated from "p=1, c=0" to "p=0, c=1". In Figure 5.5 (c), $t_1$ has now written its new value to $p_0$, i.e., replaced this placeholder by a written version that stores `Thinking`. Note that the initial state's `Eating` remains accessible for MVCC. To propagate this change, the reevaluation submitted a `Changed` `notify` for $f_0$. This has executed, and thus incremented the `changes` and decremented the `pending` counter of the the placeholder of $t_1$ on $f_0$, i.e., also updated

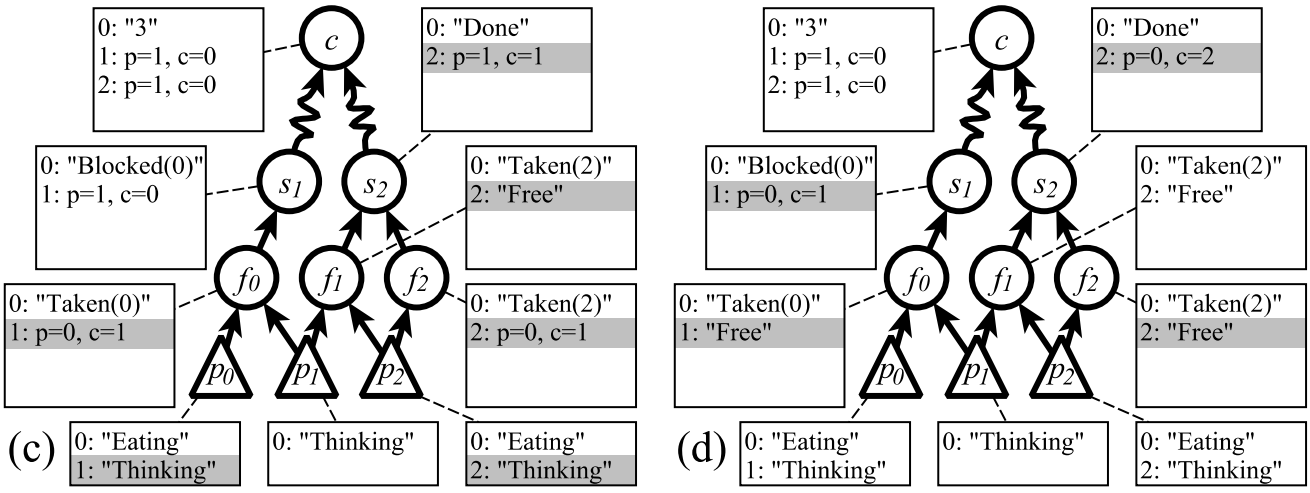



**Figure 5.5:** FrameSweep Example Trace: Propagation

this placeholder from "p=1, c=0" to "p=0, c=1"- With `pending` at zero, the corresponding reevaluation of $f_0$ by $t_1$ is now glitch-free and has been submitted. In Figure 5.5 (d), this reevaluation completed and changed $f_0$ to `Free`. Correspondingly, a subsequent `Changed` notification has been submitted for $s_1$, which there also updated the placeholder of $t_1$ from "p=1, c=0" to "p=0, c=1". This demonstrates how the propagation phase recursively progresses through the *DG*.

Stepping back again to Figure 5.5 (c), $t_2$ has already reevaluated $p_2$ to `Thinking` and $f_1$ to `Free`. This already demonstrates first effects of pipeline parallelism, in that $t_2$ is already making progress despite some of its later reactives still being locked for exclusive access by other instants, e.g., $c$ by $t_1$. The `Changed notify` submitted by the reevaluation of $f_1$ for $s_2$ has again incremented the `changes` and decremented the `pending` counter. Differently from before though, this updated the placeholder of $t_2$ on $s_2$ from "p=2, c=0" to "p=1, c=1", which demonstrates how FrameSweep ensures glitch freedom. Since $t_2$ will still reevaluate $f_2$ as a second dependency of $s_2$, $s_2$ is not yet ready for a glitch-free reevaluation yet. In the node histories, this is reflected by $f_2$ still having a placeholder for $t_2$, i.e., $f_2$ is still dirty, and accordingly `pending` counter of the placeholder of $t_2$ on $s_2$ has not reached zero yet. Stepping forward to Figure 5.5 (d), $t_2$ has completed its reevaluation of $f_2$, with the placeholder there also replaced by a written `Free`. The subsequent `Changed notify` again incremented the `changes` and decremented the `pending` counter, meaning the placeholder of $t_2$ on $s_2$ has now been updated to "p=0, c=2", corresponding to zero remaining dirty, but two changed predecessors. Thus $t_2$ is now ready to reevaluate $s_2$. Stepping forward further to Figure 5.6 (e), this reevaluation has completed, with the placeholder replaced by a written `Ready`.

## 5.6 Read Operations

This section presents, how FrameSweep implements the different semantics of the various read operations of RP (`depend`, `after`, `before` and `now`).

Listing 5.3 shows the pseudocode implementations for all read operations. During a reevaluation, `procedure Reactive.depend`, defined in Line 111ff., is executed when the user computation of the `reevaluating` reactive calls `reactive.depend`. It first registers the access to `reactive` with the execution shepherding of the ongoing reevaluation (Line 112), so that the according dependency edge `reactive` $\xrightarrow[DG]{}$ `reevaluating` will be either retained or created at the end of the reevaluation (cf. Section 2.3.1). Next, the execution checks whether edge `reactive` $\xrightarrow[DG]{}$ `reevaluating` already exists. If it exists, the read is dispatched through `procedure Event.readSynchronized` in Line 117 if `reactive` is

```
111 procedure Reactive.depend(reactive, instant, reevaluating):
112   add shepherding.accessedDependencies += reactive
113   if (reactive ∈ inc_reevaluating):
114     dispatch readSynchronized(reactive, instant)
115   else:
116     dispatch after(reactive, instant)

117 locally-exclusive procedure Event.readSynchronized(reactive, instant):
118   if (∃ reactive[instant] ∧ reactive[instant].value != None):
119     return reactive[instant].value
120   else:
121     return None

122 locally-exclusive procedure Signal.readSynchronized(reactive, instant):
123   if (∃ reactive[instant] ∧ reactive[instant].value != None):
124     return reactive[instant].value
125   else:
126     return beforeSynchronized(reactive, instant)

127 locally-exclusive procedure Signal.beforeSynchronized(reactive, instant):
128   let preds = { pred | ∃ reactive[pred] ∧
129                        pred -*->_SSG instant ∧
130                        reactive.value != None }
131   let pred = max_SSG(preds)
132   return reactive[pred].value

133 locally-exclusive procedure Reactive.after(reactive, instant):
134   execute ensureVersion(reactive, instant)
135   execute Reactive.awaitFirstPlaceholder(changedInput, instant)
136   suspend until (reactive[instant].pending == 0 ∧ reactive[instant].changes == 0)
137   dispatch readSynchronized(reactive, instant)

138 locally-exclusive procedure Signal.before(reactive, instant):
139   execute ensureVersion(reactive, instant)
140   execute Reactive.awaitFirstPlaceholder(changedInput, instant)
141   return beforeSynchronized(reactive, instant)

142 procedure Signal.now(signal):
143   let instant = <new>
144   execute framingPhase(instant, declaredInputs = ∅)
145   execute before(signal, instant)
146   execute propagationPhase(instant, declaredInputs = ∅, inputChanges = ∅)
```

**Listing 5.3:** Pseudocode implementations of MVCC reading routines.

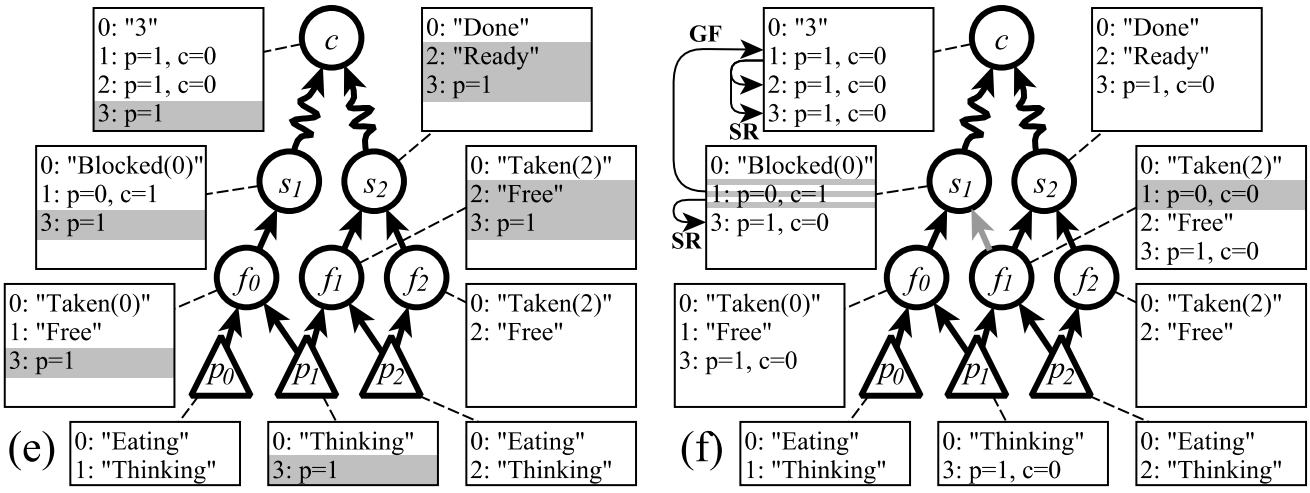



**Figure 5.6:** FrameSweep Example Trace: Read Operations

an Event, or `procedure Signal.readSynchronized` in Line 122 if `reactive` is a Signal, where the correct version to read from is determined. For both, if the reading `instant` has an own written version on the target `reactive`, the value of that version is returned (Line 118f. and Line 123f. respectively). Otherwise, the semantics differ for Events and Signals. For Events with no own written version, the correct result to return is always – regardless of any other versions in the node history – no value (Line 120). For Signals with no own written version on the other hand, the correct return value is chosen by `procedure Signal.beforeSynchronized`, defined in Line 127ff., as the latest (Line 131) of all written versions by earlier instants (Line 128). This ensures that the read returns a serializable value, even if it is executed after later instants have already written newer values to the `reactive`, by ignoring all such newer values.

### Reading Without Dependency Edges

If `depend` is executed, but edge $\texttt{reactive} \underset{DG}{\longrightarrow} \texttt{reevaluating}$ does not exist, the read is instead dispatched through `procedure Reactive.after`, defined in Line 133ff. This implements the semantics for `reactive.after` reads, i.e., the case where a read is executed deliberately without prompting a dependency edge. For reading without a corresponding dependency edge, the `reactive` to be read may not yet be in a state where the reading `instant` can read its value correctly, because the framing and propagation traversal ensure serializability and glitch freedom only along existing *DG* edges. Thus, FrameSweep executes the following steps to ensure that this state has been reached. First, `ensureVersion` is executed [2], but without updating `pending` or `changes` counters afterwards. This means, if an idempotent version exists or is newly created, it is not converted into a placeholder or written version, but left as idempotent.

[2] *Readers interested in understanding the intuition behind FrameSweep's procedures can safely ignore the following explanation. It does not contribute towards understanding, and only serves to clarify correctness concerns.* At this point, the complexity of `ensureVersion` for maintaining totality of the *SSG* order for the node history comes into full effect (cf. Listing 5.1 Line 44). Equally to the framing traversal, the newly inserted version is ordered as late as possible into the node history. But now, since the reading `instant` is already propagating, it must not order itself after other instants that are still in the framing phase, to maintain FrameSweep's invariant that instants must never have predecessor instants still in earlier phases. For this, all `conc` instants that are not yet ordered against the reading `instant` are split depending on whether they are still framing (set `F`), or already propagating (set `P`). The reading instant orders itself later than all `conc` instants in `P`, equal to the framing traversal, but earlier than those in `F` by doing the reverse: It selecting `succ` as the earliest instant from `F` and inserting edge $\texttt{instant} \underset{SSG}{\longrightarrow} \texttt{succ}$, thereby ordering all `conc` in `F` as $\texttt{instant} \underset{DG}{\longrightarrow} \texttt{conc}$ by transitivity. Throughout this process, all `conc` in `F` must be prevented from concurrently transitioning their phases, as this would invalidate the split between `F` and `P`. Otherwise, the reading `instant` might initially group a framing `conc` into `F`, but before it establishes the *SSG* edge, that `conc` instant could concurrently transition through its propagation phase all the way to completed. In that case, the *SSG* edge insertion would order the propagating `instant` earlier than the completed `conc`, despite being in an earlier phase still. The implementation presented here achieves this by executing all

This is because this version does not represent a planned or completed reevaluation, but only ensures that the reading `instant` is ordered in the *SSG* against all planned or completed reevaluations of the `reactive` by other instants. Once ordered, the execution suspends twice. First in Line 135 until the placeholders of all instants that are then ordered earlier have been removed, to ensure that the read does not execute too early in terms of serializability. Second in Line 136 until the version of the reading `instant` itself no longer is a placeholder, to ensure that the read is not executed too early in terms of glitch freedom. Afterwards, the `reactive` is guaranteed to be in a state where the reading `instant` can read its value correctly, and the read is dispatched back to `readSynchronized`.

#### Reading Past Values

The implementation of `s.before` reads, implemented by `procedure Signal.before` in Line 127ff, is similar to `s.after` in that it also deliberately does not prompt a dependency edge. It also executes `ensureVersion` in Line 139 to ensure that the reading `instant` is ordered in the *SSG* against all reevaluations by other instants. It also suspends in Line 140 until all placeholders by earlier instants have been removed, to ensure that the read is not executed too early in terms of serializability. It does not suspend for glitch freedom until the reading `instant` itself no longer has a placeholder though, because the semantics of `before` deliberately ignore values of the reading `instant` itself, and thus do not need to wait for glitch freedom. Instead, after only ensuring serializability, it dispatches to `beforeSynchronized` to read the latest value written by earlier instants.

#### Demonstration

For better illustration of the upcoming examples, Figure 5.6 (e) introduces a third instant to the running example. Initiated by `phils(1).set(Eating)`, $t_3$ has already executed its entire framing traversal, with placeholders created on $p_1$, $f_0$, $f_1$, $s_1$, $s_2$ and $c$. For the demonstration of read operations, consider then Figure 5.6 (f), where $t_1$ has just started its reevaluation of $s_1$, with the corresponding placeholder highlighted by a striped grey background. The first read executed by this reevaluation is `forks(0).depend` (cf. Listing 2.3 Line 17 with `prevIdx == 0` from `idx == 1` of $s_1$). Because the corresponding edge $f_0 \xrightarrow[DG]{} s_1$ exists, this read dispatches directly to `readSynchronized`. Since $t_1$ has an own written version on $f_1$, its value (`Free`) is returned. The same would be true if $f_0$ was an Event.

Based on that return value, the reevaluation next executes `forks(1).depend` (cf. Listing 2.3 Line 19 with `idx == 1` of $s_1$). Differently from $f_0$, there is no edge $f_1 \xrightarrow[DG]{} s_1$ (grey dependency arrow), and the read dispatches through `after`, which creates the idempotent version "1: p=0, c=0" on $f_1$. Since already no earlier placeholders exist anymore, all suspension conditions are met immediately and the version to read from is chosen immediately, again by dispatching to `readSynchronized`. Equal to $f_0$, there are also three versions on $f_1$, but they differ in that they are associated with instants $t_0$, $t_2$ and $t_3$ instead of $t_0$, $t_1$ and $t_3$, meaning $t_1$ does not have an own written version. If $f_1$ was an Event, the read would therefore return no value. Since $f_1$ is a Signal though, the read is forwarded to `beforeSynchronized`, which returns `Taken(2)` from the initial state as the latest written version ordered earlier than $t_1$.

#### Imperative Reads

To conclude the discussion of read operations, only `s.now` reads remain to be addressed. These are not executed within the context of `update(...)` calls, but are separate individual imperative interactions. Still, FrameSweep implements them as instants, through `procedure Reactive.now`, defined in Line 142ff.: An instant for `s.now` first executes the framing phase with an empty `declaredInputs` set in Line 144. This means, it does not create any placeholders, establishes no *SSG* ordering relations, and immediately transitions to `Propagating`. Then, the read is dispatched as `s.before`, ensuring that the instant is ordered against other instants' reevaluations and returning a serializable value from an earlier

phase transitions under the same global mutual exclusion as *SSG* accesses (Listing 5.1 Line 37 and Listing 5.2 Line 37ff.). A more sophisticated and alternative, which is compatible with decentralization, is presented in Section 7.3.2.

instant, since the `s.now` instant itself does not write any own values. Finally, the instant executes a propagation phase with an empty `declaredInputs` set and an empty `inputChanges` mapping in Line 146. This means, it does execute any change or no-change propagation, but only suspends until all preceding instants have completed before completing itself. This is necessary for linearizability, which the proofs in Chapter 6 will address in more detail.

## 5.7 Retrofitting

The latest state of the running example in Figure 5.6 (f) demonstrated the reevaluation of $s_1$ by $t_1$ executing a `depend` read of $f_1$. In addition to reading a user value from its target reactive, `depend` reads prompt corresponding *DG* edges to be created. In Figure 5.6 (f), the edge $f_1 \xrightarrow[DG]{} s_1$ is visualized in grey not only to show that the `depend` read executes while it does not exist, but because it must be created. To create it, $t_1$ must add $s_1$ to the outgoing dependencies set of $f_1$, i.e., execute a write operation on that variable on $f_1$. Both $t_2$ and $t_3$ are ordered later than $t_1$ in the *SSG* though, but already read the outgoing dependencies set of $f_1$ from the initial state during the graph traversals they have executed so far. If $t_1$ simply executed its edge insertion in this state, these already executed graph traversals by $t_2$ and $t_3$ would become incorrect, because they then executed on an outdated value of the outgoing dependencies set from $t_0$ and ignored the actual latest value that was just newly written by $t_1$.

### Reasoning

The key insight to reconciling these situations is, that while the execution of ordered-later instants are no longer correct, they are not arbitrarily incorrect either, but only "over-planned" or incomplete. In case of edge drops, later instants may have "over-planned" in that they created and updated too many placeholders. None of the corresponding reevaluations can have executed yet though, and therefore wrongly created and updated placeholders can still be removed or their updates reverted. In case of discoveries, inversely, later instants may be incomplete in that they have missed creating and/or updating some placeholders. No other operations, which would be affected by these placeholders, can have executed yet though, and therefore affected placeholders can still be added and updated belatedly. The reasoning for both cases is that the combination of glitch freedom and C2PL for reevaluations forms a sort of *protective shield* that prevents any potentially affected user-visible operations from executing. An instant can drop or discover a dependency edge `reactive` $\xrightarrow[DG]{}$ `reevaluating` only during an ongoing reevaluation of `reevaluating`. This means, the instant has a placeholder on `reevaluating` that it will not remove until after the dynamic edge change has been completed. In Figure 5.6 (f), this is the placeholder of $t_1$ on $s_1$, highlighted with a striped grey background.

From the framing phase for serializability and glitch freedom, every derived reactive `d` that is reachable from `reevaluating` (i.e., some path `reevaluating` $\xrightarrow[DG]{*}$ `d` exists) also has a placeholder by that instant. Due to glitch freedom, all reevaluations corresponding to these placeholders cannot start, because they must (transitively) wait for `reevaluating` to become glitch-free by completing the ongoing reevaluation. Figure 5.6 (f) visualizes with the arrow labelled "GF", how the placeholder by $t_1$ on $s_1$ implies through glitch freedom that the placeholder by $t_1$ exists on $c$ and cannot be removed while the reevaluation is still ongoing, and the same also applies for all omitted reactives reachable from $s_1$. Further, due to C2PL for serializability, on any such `d` and on `reevaluating` itself, no read operations or reevaluations by later instants can have executed yet (all these operations suspend until all earlier instants' placeholders have been removed). Figure 5.5 (f) visualizes with the arrows labelled "SR", how the placeholders of the "GF" arrow by $t_1$ precede and block all operations by the later instants $t_2$ and $t_3$. In summary, the protective shield formed by glitch freedom and C2PL for serializability – visualized in Figure 5.5 (f) by the combination of all "GF" and "SR" arrows – prevents all user-visible operations (reevaluations and reads) by later instants from executing. It therefore allows all placeholders of such later instants on

any `reevaluating` itself and on any transitively reachable derived reactive `d` to be added, removed or arbitrarily updated without causing any effects observable by user code.

#### Implementation

The entry point for dropping a dependency edge `reactive` $\xrightarrow[DG]{}$ `reevaluating` is `procedure Reactive.drop`, defined in Listing 5.4 Line 147ff. It first ensures, that the parameter `instant` has at least an idempotent version in the node history of the parameter `reactive` (Line 149, to guarantee that future transactions order themselves correctly against the edge-dropping write. Then it executes the edge write (Line 150) and starts retrofitting as explained below. The entry point for discovering a dependency edge `reactive` $\xrightarrow[DG]{}$ `reevaluating` is `procedure Reactive.discover`, defined in Listing 5.4 Line 153ff. Here, a version is guaranteed to exist already, because the discovery is executed in response to a `depend` read of the parameter `reactive` where edge `reactive` $\xrightarrow[DG]{}$ `reevaluating` did not exist yet. This means, the condition in Listing 5.3 Line 113 was not met, and the `depend` read was dispatched in Line 116 to `after`, where `ensureVersion` for the `instant` was executed in Line 134. The procedure therefore immediately executes the edge write (Line 156), before also starting retrofitting.

Note that the pseudocode here uses the keyword `subsecute` instead of `submit` or `execute`. This is because retrofitting as presented here uses a graph traversal that requires a semantics that is a mix of the two keywords. The retrofitting traversal visits each reactive under local exclusion, equal to the framing and propagation graph traversal. This means, recursing to successor reactives must escape the mutual exclusion scope within which the recursive call is made, equal to `submit`. On the other hand though, the retrofitting traversal must not execute asynchronously, but must complete within the execution of `drop` or `discover`, equal to `execute`. FrameSweep's actual implementation does not use a graph traversal for retrofitting (and therefore still truly is only a two-phase algorithm). But, the actual approach it uses – which is described in Section 7.2 – imposes significant increases in complexity for all other operations presented so far. Thus, in order to keep the explanation and proof of FrameSweep's correctness as accessible as possible, this section presents retrofitting as a third recursive graph traversal with this special recursion/termination semantics.

The first step of retrofitting is to compute, which instants – if any – had their already executed graph traversals invalidated by the edge write and therefore require fixing. FrameSweep's node histories are constructed such that the necessary corrections can be determined based on the contents of the node history of the parameter `reactive` whose outgoing edges were changed. Namely, retrofitting must be performed for every later instant `succ` (i.e., `instant` $\xrightarrow[SSG]{*}$ `succ`) that has a version `reactive[succ]` that is not idempotent. This applies equally for both drops and discoveries, they only differ in *how* they fix the affected instants' executions afterwards. The `procedure Reactive.computeRetrofit`, defined in Listing 5.4 Line 159ff., computes as `retrofitChange` the set of all such `succ` instants that have a written version `reactive[succ]`, and as `retrofitFrame` the set of all such `succ` instants that have a placeholder `reactive[succ]`. In Figure 5.6 (f), the idempotent version "p=0, c=0" of $t_1$ on $f_2$ (grey background) was created for discovering edge $f_2 \xrightarrow[SSG]{*} s_1$. It is followed by the written `Free` by $t_2$ and the placeholder "p=1, c=0" by $t_3$. Therefore, retrofitting is required for `retrofitChange = {` $t_2$ `}` and `retrofitFrame = {` $t_3$ `}`.

Performing the computed retrofitting is implemented in `procedure Reactive.retrofit`, defined in Listing 5.4 Line 167ff. It is used for retrofitting both drops and discoveries. For retrofitting after drops, the `delta` parameter is set to $-1$ to decrements affected placeholders' counters. For retrofitting after discoveries, the `delta` parameter is set to $+1$ to increment the counters instead.

- For every placeholder `reactive[succ]` that prompted retrofitting (Line 169f), `procedure Reactive.reframe` is executed, defined in Listing 5.4 Line 178ff. The first step of `reframe` on every reached `reactive` is to ensure that a version for the parameter `succ` instant exists (Line 180). Then, the counters of this version are updated.

```
147 locally-exclusive procedure Reactive.drop(reactive, instant, reevaluating):
148   # delete edge: reactive ⟶_DG reevaluating
149   execute ensureVersion(reactive, instant)
150   remove reactive.outDeps -= reevaluating
151   let (retrofitChange, retrofitFrame) = computeRetrofit(reactive, instant)
152   subsecute retrofit(reevaluating, retrofitChange, retrofitFrame, -1)

153 locally-exclusive procedure Reactive.discover(reactive, instant, reevaluating):
154   # create edge: reactive ⟶_DG reevaluating
155   # version reactive[instant] already exists from preceding readSynchronized(reactive, instant)
156   add reactive.outDeps += reevaluating
157   let (retrofitChange, retrofitFrame) = computeRetrofit(reactive, instant)
158   subsecute retrofit(reevaluating, retrofitChange, retrofitFrame, +1)

159 procedure Reactive.computeRetrofit(reactive, instant):
160   let retrofitChange := { succ | ∃ reactive[succ] ∧
161                                  ∃ instant ⟶_SSG succ ∧
162                                  (reactive[succ].value != None) }
163   let retrofitFrame  := { succ | ∃ reactive[succ] ∧
164                                  ∃ instant ⟶_SSG succ ∧
165                                  (reactive[succ].pending > 0 ∨ reactive[succ].changes > 0) }
166   return (retrofitChange, retrofitFrame)

167 locally-exclusive procedure Reactive.retrofit(reevaluating, retrofitChange, retrofitFrame, delta):
168   # delta is +1 or -1
169   foreach (succ ∈ retrofitFrame):
170     subsecute reframe(reevaluating, succ, delta)
171   foreach (succ ∈ retrofitChange):
172     execute ensureVersion(reevaluating, succ)
173     increment/decrement reevaluating[succ].changes += delta
174     if (reevaluating[succ].pending == 0 ∧ ((delta == +1 ∧ reevaluating[succ].changes == 1) ∨
175                                            (delta == -1 ∧ reevaluating[succ].changes == 0))):
176       foreach (derived ∈ reevaluating.outDeps):
177         subsecute reframe(derived, succ, delta)

178 locally-exclusive procedure Reactive.reframe(reactive, succ, delta):
179   # delta is +1 or -1
180   execute ensureVersion(reactive, succ)
181   increment/decrement reactive[succ].pending += delta
182   if (reactive[succ].changes == 0 ∧ ((delta == +1 ∧ reactive[succ].pending == 1) ∨
183                                      (delta == -1 ∧ reactive[succ].pending == 0))):
184     foreach (derived ∈ reactive.outDeps):
185       subsecute reframe(derived, succ, delta)
```

**Listing 5.4:** Pseudocode implementations of dynamic edge change retrofitting routines.

- For drops (`delta == -1`), `reframe` decrements the `pending` counter of `reactive[succ]` (Line 181), reflecting that the removal of the dependency edge disconnected a dirty dependency. If this turns `reactive[succ]` from a placeholder into an idempotent version (all counters zero, checked in Line 182f.), `reframe` recurses to all successor reactives. This process undoes the part of the framing traversal of `succ`, that `succ` executed superfluously because it observed the dependency edge that should have been dropped before its execution (in serializability time), but was only dropped now (in real time).
- For discoveries (`delta == +1`), these changes are reversed. The `pending` counter is incremented (Line 180), reflecting that the newly added edge connected an additional dirty dependency. If this turns `reactive[succ]` from an idempotent version into a placeholder (`changed` is zero while `pending` was updated from 0 to 1, checked in Line 182f.), `reframe` recurses to all successor reactives. This process belatedly executes the part of the framing traversal, that `succ` previously missed because it did not observe the dependency edge that should have been discovered before its execution (in serializability time), but was only discovered now (in real time).

- For every written `reactive[succ]` that prompted retrofitting (Line 171ff), retrofitting differs only in that it updates placeholders' `changed` instead of `pending` counters on the `reevaluating` reactive. This corresponds to the dropped or discovered incoming dependency edge having disconnected or connected an additional dependency that is changed because of the written version, instead of dirty because of the placeholder. The remaining recursive graph traversal of all transitively reached reactives is identical to above: If the update of `changes` turned `reevaluating[succ]` from a placeholder into an idempotent version after drop, or from an idempotent version into a placeholder after discovery (Line 174f), a `reframe` task is submitted for all successor reactives (Line 176), which will recursively update `pending` counters. In summary, retrofitting prompted by written `reactive[succ]` not only undoes the superfluously executed – or belatedly executes the missed – part of the framing traversal of the parameter `succ` instant, but also undoes the superfluously executed – or belatedly executes the missed – `Changed notify` task by `succ` over the added/removed `reactive` $\underset{DG}{\longrightarrow}$ `reevaluating` edge.

#### Demonstration

Figure 5.7 (g) shows the updated state of the running example after retrofitting has been executed for the newly discovered edge $f_1 \underset{DG}{\longrightarrow} s_1$. For $t_3$, the placeholder on $f_1$ prompted retrofitting to belatedly execute a missed part of its framing traversal. In Figure 5.6 (f), a placeholder by $t_3$ on $s_1$ already existed, meaning the original framing traversal already traversed over all successor reactives. In Figure 5.6 (g), the `reframe` traversal thus updated this placeholders' `pending` counter to "p=2", but then did not traverse to its successor reactives a second time. This counter value of 2 also shows, how retrofitting interacts with the mark-sweep change propagation for glitch freedom. The framing phase of $t_3$ originally set up the placeholder of $s_1$ in Figure 5.6 (f) with a value of 1, reflecting that the corresponding reevaluation must only wait for $f_0$ to become glitch-free. After retrofitting, in Figure 5.7 (g), the counter value of 2 correctly reflects that the reevaluation now must wait for two predecessors – $f_0$ and $f_1$ – to become glitch free, instead of just one.

The written `Free` by $t_2$ on $f_1$ prompted retrofitting to belatedly execute a missed part of the framing phase *and* change propagation of $t_2$. In Figure 5.6 (f), $t_2$ did not have a version on $s_1$, yet. In Figure 5.7 (g), retrofitting therefore created this version and retrofit the missed change propagation by updating its `changed` counter to "c=1", instead of its `pending` counter. Since this placeholder was newly created, recursive `reframe` tasks were submitted for all omitted reactives that depend on $s_1$. Some of these were already framed by $t_2$ via $s_2$ though, so this `reframe` traversal stopped somewhere on the path from $s_1$ to $c$, and the placeholder on $c$ remains unaffected. With retrofitting complete, the discovery of edge

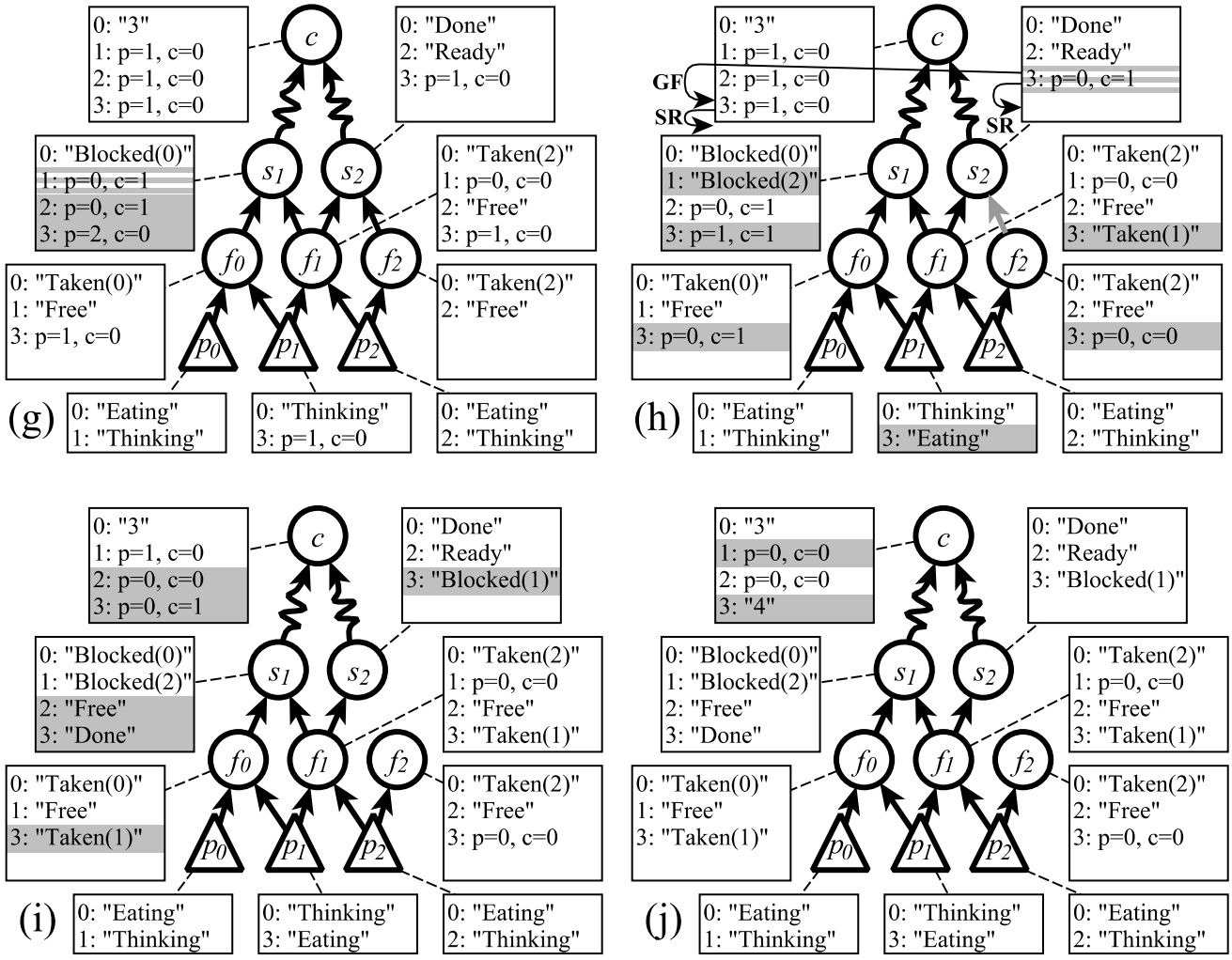



**Figure 5.7:** FrameSweep Example Trace: Retrofitting

$f_1 \xrightarrow[DG]{} s_1$ is also complete, and the still ongoing reevaluation of $s_1$ by $t_1$ can now continue and complete, too. Retrofitting has successfully corrected the applications' execution into the same state that could have been reached without retrofitting if $t_2$ and $t_3$ had executed – as the *SSG* order intends – only after $t_1$ discovered the edge.

Figure 5.7 (h) shows a further progressed state of the applications. The reevaluation of $s_1$ by $t_1$ has completed, which spawned the subsequent reevaluation for the placeholder of $t_2$ that was newly created during retrofitting. Further, $t_3$ has started its propagation phase, with reevaluations of $p_1$ and $f_1$ already completed. Subsequently, $t_3$ updated its previously retrofitted placeholder on $s_1$ to "p=1, c=1", reflecting that $f_1$ is now changed instead of dirty, but $f_0$ is still dirty. The reevaluation of $f_0$ by $t_3$ is ongoing, simultaneously with the reevaluation of $s_2$ also by $t_3$, demonstrating an example of concurrent reevaluations per instant. Given the new value of $f_1$, the reevaluation of $s_2$ (striped grey background) has computed `Blocked(1)` as its result without executing a `depend` read of $f_2$ (cf. Listing 2.3 Line 26). This means, the existing dependency edge $f_2 \xrightarrow[DG]{} s_2$ (grey bold edge) was not used, and therefore $t_3$ has created an idempotent version on $f_2$ in order to drop it. Since this idempotent version is not followed by any versions by later instants, the edge change does not prompt any retrofitting. The protective shield, which prevents any user-visible operations by successive transactions from executing on any reactives that retrofitting could affect, exists regardless, and is visualized again through "GF" and "SR" arrows.

In Figure 5.7 (i), the application has progressed further again. The reevaluation of $s_2$ by $t_3$ has completed without any retrofitting, and converted the placeholder into a written `Blocked(1)`. All instants – $t_1$, $t_2$, and $t_3$ – have reevaluated all reactives except for $c$. The final `notify` task of $t_1$ has not executed on $c$ yet, which therefore still has a placeholder with the `pending` counter at 1. The no-change propagation by $t_2$ (no-change because an update of $p_2$ to `Thinking` does not result in any changes past the `filter` Events) has overtaken the propagation by $t_1$, and replaced the placeholder of $t_2$ on $c$ with an idempotent version. The change propagation by $t_3$ (the update of $p_1$ to `Eating` *does* propagate changes all the way up to incrementing the overall `Eating` counter) has also overtaken $t_1$, and converted the placeholder of $t_1$ on $c$ into one that is ready for reevaluation. The latter reevaluation is not yet allowed to start though, because it is pipelined behind the earlier placeholder by $t_1$. Both $t_2$ and $t_3$ are dormant at this point, with no further queued or executing tasks, but may not complete yet because their predecessor $t_1$ is still propagating.

Finally, in Figure 5.7 (h), the no-change propagation by $t_1$ (again, an update of $p_0$ to `Thinking` does not result in any changes past the `filter` Events) has reached $c$ and skipped its reevaluation, turning the placeholder of $t_1$ into an idempotent version directly. At that point, $t_1$ can complete, and then also $t_2$. For $t_3$ though, $t_1$ submitted another reevaluation of $c$, demonstrating the reason for why $t_3$ was not allowed to transition to completed before the ordered-earlier $t_1$ did. That reevaluation has also already completed and recomputed the folding Signal, incrementing its counter value to 4. To conclude the introduction of FrameSweep, observe that the overall picture of Figure 5.7 (h) shows an end result that is indeed equivalent to a sequential execution following the serialization order that was decided on-the-fly: The exact same result would have been achieved, if – starting from the initial state $t_0$ – first $t_1$ executed all by itself, then $t_2$, and finally $t_3$. FrameSweep achieved this result without aborting any instant or otherwise reverting or rolling back any user-visible operation, and all reevaluations were executed without any glitches having occurred, despite non-commutative operations on reactives having executed in a different order than the serialization order on multiple occasions.

# 6 Proving Correctness

This section formally proves the correctness of FrameSweep's pseudocode implementation from Chapter 5 through the application of established multiversion theory [Bernstein et al., 1986]. Multiversion theory defines how to compute a `multiversion serialization graph` (*MVSG*) from a history of reads and writes executed by a transaction system. The definition of *MVSG* is accompanied by a proven theorem, stating that if the *MVSG* of a given history is acyclic, then this history is serializable. The proofs of FrameSweep's correctness cover all possible histories that FrameSweep can produce by induction, starting with an empty history as the induction base and covering as induction steps, case by case, the execution of every read and write operation from the pseudocode routines defined in Listing 2.9 and Listings 5.1 through 5.4. They show that the *SSG* is acyclic, and that for every possible history, the *SSG* tracked by FrameSweep at runtime is a correct representation of the formally defined *MVSG*. This implies that the corresponding *MVSG* is acyclic, and therefore all histories produced by FrameSweep are correct.

The proofs are structured as follows. Section 6.1 gives a summary of the relevant formal tools from established multiversion theory. Section 6.2 describes, how FrameSweep connects with these tools. Section 6.3 proofs several basic internal properties of FrameSweep. Section 6.4 proofs more complex and advanced properties, such as equivalences between FrameSweep's framing, propagation, and retrofitting graph traversals. Section 6.5 finally presents the induction proof that FrameSweep's pseudocode implementation is correct.

## 6.1 A Recap of Multiversion Theory

FrameSweep's proof is built relating the pseudocode implementation presented in this chapter to established multiversion theory from databases. The following definitions provide a minimal summary of the tools provided by multiversion theory.

**Definition 1** (Multiversion History)**.** *A multiversion history $H$ is the set of all reads and writes executed by a transaction system. When a transaction $t_i$ writes a variable $x$, this is denoted as $write_i(x^i) \in H$. When a transaction $t_j$ reads the value $x$ that was written by $t_i$, this is denoted $read_j(x^i) \in H$.*

Note that $read_j(x^i)$ a direct representation of a reads-from relation where $t_j$ read a value from $t_i$.

**Definition 2** (Version Order)**.** *A version order $\ll$ is a non-reflexive order that defines, for every variable $x$, a total order over all versions $write_i(x^i)$.*

**Definition 3** (Multiversion Serialization Graph)**.** *Given a multiversion history $H$, and a version order $\ll$, the multiversion serialization graph $MVSG(H, \ll) = (T, E)$ is a directed graph over all transactions $T = \{t_i\}$ and the following edges. For every $read_j(x^i) \in H$, with $i \neq j$ there is an edge $t_i \rightarrow t_j \in E$. Further, for every pair of $read_j(x^i) \in H$ and $write_k(x^k) \in H$, with $i$, $j$ and $k$ pairwise distinct, there is an edge:*

$$\left.\begin{array}{ll} t_j \rightarrow t_k, & \text{if } write_i(x^i) \ll write_k(x^k) \\ t_k \rightarrow t_i, & \text{if } write_k(x^k) \ll write_i(x^i) \end{array}\right\} \in E$$

In essence, $MVSG(H, \ll)$ is the collection of all deductions that can be drawn in relation to each reads-from relation, i.e., each $read_j(x^i)$, when asking "Assuming that $H$ was produced by a sequential application without multiversions, in which order must the contained transactions have executed to

produce $H$": Each $read_j(x^i)$ implies that $t_i$ was the last transaction to write $x$ before $t_j$ read $x$. This means first that $t_i$ executed before $t_j$, hence $t_i \rightarrow t_j \in E$, and second that every other $t_k$ that also wrote $x$, i.e., every $write_k(x^k)$, must have executed either before $t_i$, hence $t_k \rightarrow t_i$, or after $t_j$, hence $t_j \rightarrow t_k$. The version order $\ll$ is the mechanism by which this latter ambiguity is modelled.

**Theorem 1** (Multiversion Serializability)**.** *A multiversion history $H$ is serializable if a version order $\ll$ exists such that the multiversion serialization graph $MVSG(H, \ll)$ is acyclic.* □

The Multiversion Serializability theorem has been proven in literature [Bernstein et al., 1986]. In terms of above mental model, it essentially states that any history $H$ is correct (view-equivalent to a serial history without multiversions) if there is some combination of choices for ordering all writes before or after all reads-from relations such that the entire collection of deductions does not imply any temporal loops.

## 6.2 Connecting FrameSweep and Multiversion Theory

FrameSweep connects to multiversion theory in two ways: through the multiversion history it produces and through the version order it defines.

### 6.2.1 Multiversion Histories produced by FrameSweep

The operations that are relevant to the following proofs are all reads and writes on reactives' user values as well as all reads and writes on reactives' incoming and outgoing dependency sets. Reads and writes of reactives' user values are obviously user-visible operations, and thus relevant for FrameSweep's correctness. Reads and writes on reactives' incoming and outgoing dependency sets on the other hand are not user-visible operations and thus not directly relevant to FrameSweep's correctness. But, showing serializability also for them proves that FrameSweep provides its isolation in a way that retains instants' internal execution semantics from single-threaded execution. This in turn implies that each instant correctly executes mark-sweep propagation, which means that liveness and glitch freedom are ensured, which `are` relevant to FrameSweep's correctness.

For simplification, the proofs will use different models for operations on reactives' user values and incoming dependencies sets versus their outgoing dependencies sets. The user values and incoming dependencies set of a reactive $r$ are modelled as single variables $v_{\mathtt{r}}$ and $inc_{\mathtt{r}}$ with immutable values that are always read and written in their entirety. The outgoing dependencies set of a reactive $r$ on the other hand is modelled through a collection of one boolean variable $out_{\mathtt{r},\mathtt{d}}$ one for every derived reactive `d` in the entire application. If $out_{\mathtt{r},\mathtt{d}}$ is `false`, then $d$ is currently not in the set of outgoing dependencies of $r$; all $out_{\mathtt{r},\mathtt{d}}$ are initially `false` If $out_{\mathtt{r},\mathtt{d}}$ is `true`, then $d$ currently *is* in the set of the outgoing dependencies of $r$. Adding or removing $d$ to or from the outgoing dependencies of $r$ during discoveries or drops in FrameSweep's pseudocode means, performing a single write operation that updates $out_{\mathtt{r},\mathtt{d}}$ to `true` or `false` respectively. Reading the outgoing dependencies set of $r$ during graph traversals on the other hand means performing multiple read operations to collect all `d` with $out_{\mathtt{r},\mathtt{d}} == \mathtt{true}$, i.e., performing one read operation of $out_{\mathtt{r},\mathtt{d}}$ for every derived reactive `d` in the entire application. This model of outgoing dependencies sets is not feasible to actually implement in practice, but it is semantically equivalent in theory to any other representation of sets and therefore feasible to use in the proofs.

**Definition 4** (FrameSweep History)**.** *A multiversion history produced by FrameSweep consists of reads and writes over the following variables of every reactive `r`:*

- *user value $v_r$ per reactive $r$, with recorded operations $write_i(v_r^i)$ and $read_j(v_r^i)$.*
- *incoming dependencies set $inc_r$ per reactive $r$, with recorded operations $write_i(inc_r^i)$ and $read_j(inc_r^i)$.*

- *individual outgoing dependency connection flags* $out_{r,d}$ *per combination of every of reactive* $r$ *with every derived reactive* $d$*, with recorded operations* $write_i(out^i_{r,d})$ *and* $read_i(out^j_{r,d})$*.*

### 6.2.2 Stored Serialization Graph Version Order

FrameSweep defines a version order $\ll_{SSG}$ based on the *SSG*. All $write_i(x^i)$ of each variable $x$ are simply ordered according to how the associated instants $t_i$ are ordered in the *SSG*. The remainder of this chapter will use *MVSG* as shorthand notation for $MVSG(H, \ll_{SSG})$.

**Definition 5** (Version Order of FrameSweep)**.** *The version order of FrameSweep,* $\ll_{SSG}$*, orders* $write_i(x^i) \ll_{SSG} write_j(x^j)$ *iff* $t_i \xrightarrow[SSG]{*} t_j$ *exists.*

The definition of $\ll_{SSG}$ is susceptible to describing an ill-defined version order. If for any variable $x$, a pair of $write_i(x^i) \in H$ and $write_j(x^j) \in H$ exists such that both $t_i \xrightarrow[SSG]{*} t_j$ and $t_j \xrightarrow[SSG]{*} t_i$ exist, then $\ll_{SSG}$ describes an order that contains some reflexive relations. Alternatively, if for any such pair neither $t_i \xrightarrow[SSG]{*} t_j$ nor $t_j \xrightarrow[SSG]{*} t_i$ exists, then $\ll_{SSG}$ describes an incomplete order that is not total over all versions of $x$. The remainder of this section will prove that neither of these cases can occur.

**Lemma 1** (Representation of Writes)**.** *For every* $write_i(v^i_r) \in H$*,* $write_i(inc^i_r) \in H$ *and* $write_i(out^i_{r,d}) \in H$*,* `ensureVersion` *was executed to create a node version for* $t_i$ *in the node history of* $r$*, which represents this write.*

*Proof.* The only place where an instant $t_i$ can write $inc^i_r$ is Listing 2.9 Line 25 of `updateDeps`. This is called only in Line 10 of `reevaluate` for $t_i$, which in turn is called in two locations:

- in Listing 5.2 Line 109 of `notify` for $t_i$. In this case, `ensureVersion` is called for $t_i$ at the very beginning of `notify` in Line 97.
- in Line 94 of `reevOut` for a different instant $t_j$. In this case, `reevaluate` for $t_i$ is submitted only on the condition that a node version by $t_i$ exists, verified in Line 91. Since `ensureVersion` is the only place where node versions are created, it must have executed for $t_i$ on $r$ beforehand from somewhere.

An instant $t_i$ may write the user value $v_r$ in FrameSweep's pseudocode only in Listing 5.2 Line 81 of `reevOut`, which is in turn called in three locations:

- in Listing 5.2 Line 70, at the beginning of the propagation phase of $t_i$ on some input reactive that is part of the `declaredReactives` of $t_i$. Here, `ensureVersion` was previously executed in Listing 5.1 Line 39 of `frame`, which $t_i$ executed for all of its `declaredReactives` at the beginning of its framing phase in Line 32f.
- in Listing 2.9 Line 14 of `Event.control`, and Line 17 and Line 19 of `Signal.control`. In turn, there are two locations where `control` is called:
  - in Listing 5.2 Line 68, also at the beginning of the propagation phase of $t_i$ on some input reactive that is part of the `declaredReactives` of $t_i$. The same reasoning as above applies.
  - in Listing 2.9 Line 11 of `reevaluate` for $t_i$. The reasoning following `reevaluate` was already presented in the context of $inc^i_r$ above.
- in Listing 5.2 Line 108 of `notify` for $t_i$. This reasoning was also addressed above, in the first case of `reevaluate`.

An instant $t_i$ may write the outgoing dependency $out_{r,d}$ in FrameSweep's pseudocode in two locations.

- in Listing 5.4 Line 150 of `drop`. Here, `ensureVersion` is executed for $t_i$ in the preceding Line 149.
- in Listing 5.4 Line 150 of `discover`. Any execution of `discover` happens after a reevaluation by $t_i$ of a derived reactive $d$ executed a `depend` read on some reactive $r$ for which a corresponding edge $r \underset{DG}{\longrightarrow} d$ does not exist yet. This means, during the execution of `depend`, the condition in Listing 5.3 Line 113 was not met. As a result, the read was dispatched through Line 116 to `after`, wherein `ensureVersion` for $t_i$ was executed in Line 134.

□

**Lemma 2** (*SSG* is Acyclic)**.** *The SSG tracked by FrameSweep is acyclic at all times, and thus describes a non-reflexive partial order.*

*Proof.* We prove Lemma 2 by induction. The induction base is all applications' initial state, where no instants have executed any operations yet. In this case, the *SSG* does not contain any edges yet, and therefore is trivially acyclic. The induction steps are all insertions of new edges in FrameSweep's pseudocode procedures. FrameSweep's pseudocode procedures create new edges in the *SSG* in two locations, both inside of `ensureVersion` in Listing 5.1 Line 44ff.:

- In Line 56, edge `pred` $\underset{SSG}{\longrightarrow}$ `instant` is created. For this edge to close a cycle, path `instant` $\overset{*}{\underset{SSG}{\longrightarrow}}$ `pred` must already exist in the *SSG*. However, `pred` is selected in Line 55 from set `P` defined in Line 46ff., with was constructed to contain only instants for which path `instant` $\overset{*}{\underset{SSG}{\longrightarrow}}$ `pred` does *not* exist. Thus this edge insertion cannot create a cycle.
- In Line 59, edge `instant` $\underset{SSG}{\longrightarrow}$ `succ` is created. For this edge to close a cycle, path `succ` $\overset{*}{\underset{SSG}{\longrightarrow}}$ `instant` must already exist in the *SSG*. However, `succ` is selected in Line 58 from set `F` defined in Line 50ff., with was constructed to contain only instants for which path `succ` $\overset{*}{\underset{SSG}{\longrightarrow}}$ `instant` does *not* exist. Thus this edge insertion cannot create a cycle either.

□

**Lemma 3** (Total Order of *SSG*)**.** *For every node history, the SSG describes a total order over all contained node versions.*

*Proof.* All node versions are created through `ensureVersion`, defined in Listing 5.1 Line 44ff. Aside from creating a node version for $t_i$ on a reactive $r$, `ensureVersion` adds edges to the *SSG* such that $t_i$ is ordered against every other instant $t_j$ that also has a node version in the same node history. In Line 46ff., every instant `conc` that has a node version in the history of $r$, but is ordered neither `conc` $\overset{*}{\underset{SSG}{\longrightarrow}}$ `instant` nor `instant` $\overset{*}{\underset{SSG}{\longrightarrow}}$ `conc`, is placed either in set `P` or `F`. In Line 55, `pred` is selected as the latest instant of all those in `P`, i.e., for every `conc` $\in$ `P`, there is a path `conc` $\overset{*}{\underset{SSG}{\longrightarrow}}$ `pred`. By adding edge `pred` $\underset{SSG}{\longrightarrow}$ `instant`, all `conc` $\in$ `P` are ordered `conc` $\overset{*}{\underset{SSG}{\longrightarrow}}$ `instant` by transitivity. In Line 58, `succ` is selected as the earliest instant of all those in `F`, i.e., for every `conc` $\in$ `F`, there is a path `succ` $\overset{*}{\underset{SSG}{\longrightarrow}}$ `conc`. By adding edge `instant` $\underset{SSG}{\longrightarrow}$ `succ`, all `conc` $\in$ `F` are ordered `instant` $\overset{*}{\underset{SSG}{\longrightarrow}}$ `conc` by transitivity. Thus, afterwards, either `conc` $\overset{*}{\underset{SSG}{\longrightarrow}}$ `instant` or `instant` $\overset{*}{\underset{SSG}{\longrightarrow}}$ `conc` is established for all other instants `conc` that also have a node version in the node history of $r$. The order described by the *SSG* is therefore always total over all node versions in the node history of $r$. □

**Theorem 2** (The Version Order of FrameSweep is Well-Defined)**.** *The Version order of FrameSweep, $\ll_{SSG}$, is well-defined in that for any pair of $write_i(x^i) \in H$ and $write_j(x^j) \in H$, $i \neq j$, it is either $write_i(x^i) \ll_{SSG} write_j(x^j)$ nor $write_j(x^j) \ll_{SSG} write_i(x^i)$.*

*Proof.* Consider any pair of $write_i(v_r^i)$ and $write_j(v_r^j)$, $write_i(inc_r^i)$ and $write_j(inc_r^j)$, or $write_i(out_{r,d}^i)$ and $write_i(out_{r,d}^i)$, $i \neq j$. Lemma 1 implies, a corresponding pair of node versions for $t_i$ and $t_j$ exists in the node history of $r$. Lemma 2 together with Lemma 3 implies, that the *SSG* describes a total non-reflexive order over the node history of $r$, meaning either $t_i \xrightarrow[SSG]{*} t_j$ or $t_j \xrightarrow[SSG]{*} t_i$ exists. Applying Definition 5 for $\ll_{SSG}$, it follows that either $write_i(x^i) \ll_{SSG} write_j(x^j)$ or $write_j(x^j) \ll_{SSG} write_i(x^i)$ is the case. Thus, $\ll_{SSG}$ correctly defines a non-reflexive total order over all writes of each variable, meaning it is a well-defined version order overall. □

## 6.3 Basic Aspects Proofs

The lemmas in this section prove various comparatively simple aspects of FrameSweep, e.g., that reevaluations are executed only for first placeholders, or that reevaluations that are queued behind earlier placeholders are guaranteed to be submitted once these placeholders are cleared. The proof of FrameSweep's correctness relies on these lemmas, but they do not provide any insight into how FrameSweep ensures serializability.

**Lemma 4** (`reevaluate` Preconditions)**.** *An instant $t_i$ executes `reevaluate` for a reactive $r$ only if $t_i$ has the first placeholder on $r$, the placeholder's `pending` counter is zero, and placeholders' `changes` counter is non-zero.*

*Proof.* Consider all locations where `reevaluate` is executed.

- In Listing 5.2 Line 94 of `reevOut`. Here, `reevOut` is executed for some other instant $t_j$ with which the previous first placeholder was associated that was just removed. $t_i$ is selected in Line 89ff. as the earliest of all instants that have a following placeholder, i.e., its placeholder is the new first placeholder. The condition `pending == 0` is checked in the previous Line 93, while `changes > 0` can be shown by contradiction: If `changes == 0` were the case, then together with `pending == 0`, the version by $t_i$ would not have been a placeholder, and therefore $t_i$ could not have been part of the set from Line 89f.
- In Line 109. The absence of an earlier placeholder is given since this execution is part of the `else` case of the condition in Line 102f, which explicitly checks for the existence of an earlier placeholder. The condition `pending == 0` is explicitly checked in the preceding Line 101, while `changes > 0` is given since this is the `else` case to the condition of `changes == 0` in Line 107.

□

**Lemma 5** (`reevOut` Preconditions)**.** *An instant $t_i$ executes `reevOut` for a reactive $r$ only if $t_i$ has the first placeholder on $r$ and the placeholder's `pending` counter is zero.*

*Proof.* We consider all locations, where `reevOut` is executed.

- In Listing 2.9 Line 14, Line 17 and Line 19, `reevOut` is executed as as part of `control`. `control` in turn is executed in two places:
  - in Line 11 of `reevaluate`, for which both conditions were are precondition from Lemma 4.
  - On input reactives at the beginning of the propagation phase in Listing 5.2 Line 68. Here, the preceding Line 64 ensures that all preceding placeholders were removed first, and the preceding Line 66 explicitly sets `pending := 0`.
- In Listing 5.2 Line 70. Here too, the preceding Line 64 ensures that all preceding placeholders were removed first, and the preceding Line 66 explicitly sets `pending := 0`.

- In Line 108. The absence of an earlier placeholder is given since this execution is part of the `else` case of the condition in Line 102f, which explicitly checks for the existence of an earlier placeholder. The condition `pending == 0` is explicitly checked in the preceding Line 101.

□

**Lemma 6** (Subsequent Reevaluations)**.** *Reevaluations that are queued on preceding placeholders are guaranteed to be submitted when the last preceding placeholder is removed.*

*Proof.* An execution of `reevOut` always submits a subsequent reevaluation that is ready in terms of glitch freedom in Listing 5.2 Line 89ff. We show that every location where a placeholder may be removed will execute `reevOut` if such a subsequent reevaluation is queued. This then implies that no placeholder can become the first placeholder on a reactive without the queued reevaluation that it represents being submitted. A placeholders may be removed wherever `pending` or `changes` is decremented or updated to 0.

- Update of `pending` to 0 for each input reactive at the beginning of the propagation phase in Listing 5.2 Line 66. This update is accompanied by a second update of `changes` to 1, and thus never removes a placeholder.
- The update of `changes` to 0 in Line 82 and Line 86. These lines are themselves part of ongoing executions of `reevOut`.
- The decrement of `pending` in Line 100 of `notify`. Here, `reevOut` is executed in Line 108 precisely when the first placeholder is removed:
  - when `pending` has become 0, checked in Line 101
  - when no preceding placeholders exist, which is the else case in Line 107 of the condition in Line 102 that checks if a preceding placeholder *does* exist,
  - and when `changes` is also 0, checked in Line 107.
- The decrement of `changes` or `pending` during drop retrofitting in Listing 5.4 Line 173 and Line 163. Here, a preceding placeholder by the instant that is dropping the edge exists, which forms the base of the protective shield for retrofitting. Therefore, this case can only remove placeholders that are not the first on their reactive.

□

**Lemma 7** (Phase Ordering)**.** *Instants in the propagation phase cannot have any predecessor instants that are still in the framing phase. Instants that are completed cannot have any predecessors that are still in the propagation or framing phase. Expressed generically, for any $t_i \xrightarrow[SSG]{} t_j$, $t_i$ must always be in a greater or equal phase than $t_j$, for the order of phases Framing < propagating < Completed.*

*Proof.* By induction, with Lemma 7 as the induction hypothesis. As induction base, in their initial state of all applications, the *SSG* does not have any edges, and the hypothesis is trivially fulfilled because every instant has no predecessor instants. Induction steps are all parts of the code that change edges in the *SSG* or change instants' phase. There are four such steps:

- An instant $t_j$ can establish an instant $t_i$ as its predecessor, i.e., create edge $t_i \xrightarrow[SSG]{} t_j$ in Listing 5.1 Line 56. Here, $t_i$ is selected from set `P` that explicitly contains only instants whose phase is greater than or equal to that of $t_j$, due to the condition in Line 49.
- An instant $t_j$ can establish an instant $t_i$ as its successor, i.e., create edge $t_j \xrightarrow[SSG]{} t_i$ in Line 59. Here, $t_i$ is selected from set `F` that explicitly contains only instants whose phase is smaller than that of $t_j$, due to the condition in Line 49.

```
1  locally-exclusive procedure Reactive.reframe(reactive, succ, delta):
2    execute ensureVersion(reactive, succ)
3    increment/decrement reactive[succ].pending += delta -= 1
4    if (reactive[succ].changes == 0 ∧ ((delta == +1 ∧ reactive[succ].pending == 1) ∨
5                                       (delta == -1 ∧ reactive[succ].pending == 0))):
6      foreach (derived ∈ reactive.outDeps):
7        subsecute reframe(derived, succ, delta)

8  locally-exclusive procedure DerivedReactive.notify(reactive, instant, change):
9    execute ensureVersion(reactive, instant)
10   if (change == Changed):
11     increment reactive[instant].changes += 1
12   decrement reactive[instant].pending -= 1
13   if (reactive[instant].pending == 0):
14     if (∃ reactive[pred]: pred -*SSG-> instant ∧
15           (reactive[pred].pending > 0 ∨ reactive[pred].changes > 0)):
16       if (reactive[instant].changes == 0):
17         foreach derived ∈ reactive.outDeps:
18           submit notify(derived, instant, Unchanged)
19     else if (reactive[instant].changes == 0):
20       execute reevOut(reactive, instant, None)
21     else:
22       submit reevaluate(reactive, instant)
```

**Listing 6.1:** Equivalence of Drop Retrofitting to No-Change Propagation.

- An instant $t_j$ in the framing phase can transition to the propagation phase in Line 37. Here, the suspension in the preceding Line 34 ensures that all predecessor instants $t_i$, i.e., with $t_i \xrightarrow[SSG]{} t_j$, have transitioned to propagating, before $t_j$ does so itself. For all $t_k$ with $t_j \xrightarrow[SSG]{} t_k$, increasing the phase of $t_j$ when – by the induction hypothesis – it was already greater or equal than that of $t_k$ has no effect.
- An instant $t_j$ in the propagation phase can transition to completed in Line 74. Here, the suspension in the preceding Line 71 ensures that all predecessor instants $t_i$, i.e., with $t_i \xrightarrow[SSG]{} t_j$, have transitioned to completed, before $t_j$ does so itself. For all $t_k$ with $t_j \xrightarrow[SSG]{} t_k$, increasing the phase of $t_j$ when – by the induction hypothesis – it was already greater or equal than that of $t_k$ has no effect.

Seeing as Lemma 7 holds for the initial state of all applications, and does not get broken through all steps in FrameSweep's pseudocode that could affect it, it follows that the Lemma 7 is true at all times. □

## 6.4 Advanced Aspects Proofs

The lemmas in this section prove more complex aspects of FrameSweep, e.g., equivalences between retrofit, framing and propagation traversals, that placeholders exist precisely in the time frame between an instants' framing and propagation phase, or that framing and propagation phase affect all the same reactives – even in face of dynamic edge changes. They provide some insights into the foundations and effects on which FrameSweep's serializability – and especially retrofitting – is built.

**Lemma 8** (Drop Retrofitting Equivalence)**.** *The retrofitting traversal of* `reframe`*, if executed after a dynamic edge drop with* `delta == -1`*, is equivalent to a no-change propagation traversal.*

*Proof.* Listing 6.1 shows two copies of previous pseudocode procedures. They have been specialized and partially evaluated for certain parameters, with decidable conditions and resulting unreachable branches

```
1 locally-exclusive procedure Reactive.frame(reactive, instant):
2   execute ensureVersion(reactive, instant)
3   increment reactive[instant].pending += 1
4   if (reactive[instant].pending == 1):
5     foreach (derived ∈ reactive.outDeps):
6       submit frame(derived, instant)

7 locally-exclusive procedure Reactive.reframe(reactive, succ, delta):
8   execute ensureVersion(reactive, succ)
9   increment/decrement reactive[succ].pending += delta += 1
10  if (reactive[succ].changes == 0 ∧ ((delta == +1 ∧ reactive[succ].pending == 1) ∨
11                                     (delta == -1 ∧ reactive[succ].pending == 0)):
12    foreach (derived ∈ reactive.outDeps):
13      subsecute reframe(derived, succ, delta)
```

**Listing 6.2:** Equivalence of Discovery Retrofitting to Framing Phase.

crossed out in light gray. First, `reframe` (originally defined in Listing 5.4 Line 178ff.) is specialized for the fixed parameter `delta == -1`. Second, `notify` (originally defined in Listing 5.2 Line 95ff.) is specialized for fixed parameter `changed == Unchanged` and for the assumption, that the affected version is preceded by an earlier placeholder, corresponding to the placeholder on which the protective shield for retrofitting is founded. Except for the checks of `pending == 0` and `changes == 0` being verified in different orders, the remaining code of both procedures is identical. □

**Lemma 9** (Discover Retrofitting Equivalence)**.** *The retrofitting traversal of `reframe`, if executed after a dynamic edge discovery with `delta == +1`, is equivalent to a framing traversal.*

*Proof.* Listing 6.2 shows two copies of previous pseudocode procedures. First, `frame` (originally defined in Listing 5.1 Line 38) as originally defined. Second, `reframe` (originally defined in Listing 5.4 Line 178ff.), but specialized and partially evaluated for certain parameters, with decidable conditions and resulting unreachable branches crossed out in light gray. Specifically, `reframe` is specialized for the fixed parameter `delta == +1`, and for the assumption that – like `frame` – it executes for an instant in the framing phase. The latter implies, that the `changes` counters on all versions associated with the instant must still be at zero, because the `changes` counter can be incremented only two places, both of which are unreachable for framing transactions:

- In Listing 5.2 Line 99 of `notify`, which can only be executed during the propagation phase. Calls to `notify` are in Line 105f of `notify`, which is recursive, and in `reevout-recurse1` and `reevout-recurse2` of `reevOut`. Calls to `reevOut` can originate from Line 70 at the beginning of the propagation phase, from `notify-reevout` which is again recursive, and from `control` in Listing 2.9 Line 14, Line 17 or Line 19. Calls to `control` can originate from `reevaluate`, which is exclusively called recursively either in Listing 5.2 Line 94 or in Line 110 of `notify`, or in Line 68 again at the beginning of the propagation phase.
- In Listing 5.4 Line 173, but only on the condition that a written version was found for the instant to which retrofitting is applied. Changed versions are created exclusively in Listing 5.2 Line 81 of `reevOut`, which was shown in the previous point to be reachable only in the propagation phase.

The remaining code of both procedures is identical. □

**Lemma 10** (Placeholders Correspond to Traversal Discrepancies)**.** *An instant $t_i$ having a placeholder on a reactive $r$ is equivalent to a temporary discrepancy where the framing traversal of $t_i$ has recursed to the outgoing dependencies of $r$, but the propagation traversal has not yet.*

*Proof.* Every recursion to outgoing dependencies of the framing traversal and propagation traversal is accompanied by the creation – respectively removal – of a placeholder. Conversely, every creation or removal of a placeholder is accompanied by a recursion of the framing – respectively propagation – traversal. We address all four cases separately.

The framing traversal recurses to all outgoing dependencies of a reactive in the following locations.

- In Listing 5.1 Line 42f. of `frame`. This is done only on the condition that incrementing the `pending` counter in Line 40 changed its value from 0 to 1 (condition in Line 41. This implies that the respective version newly became a placeholder (`changes` counters are all 0 during the framing phase).
- In Listing 5.4 Line 176 and Line 184, if retrofitting is executed after a discovery, which following Lemma 9 is equivalent to a framing traversal. This is done on the condition that a placeholder was created, explicitly checked in Line 174f. and Line 182f. respectively, after incrementing the `changes` counter in Line 173 and the `pending` counter in Line 181 respectively.

The propagation traversal recurses to all outgoing dependencies of a reactive in the following locations.

- In Listing 5.2 Line 83f. or Line 87f. of `reevOut` Here, `changes := 0` is explicitly set by the respective preceding Line 82 or Line 86, while from Lemma 5, `pending == 0` is a precondition for `reevOut` to be executed.
- In Line 105f. of `notify`. Here, the preceding Line 104 and Line 101 ensure that the version has become idempotent (`pending == 0 ∧ changes == 0`) after the `pending` counter was decremented in Line 100.
- In Listing 5.4 Line 176 and Line 184, if retrofitting is executed after a drop, which following Lemma 8 is equivalent to a propagation traversal. This is done on the condition that a placeholder was removed, explicitly checked in Line 174f. and Line 182f. respectively, after decrementing the `changes` counter in Line 173 and the `pending` counter in Line 181 respectively.

A placeholder may be created wherever `pending` or `changes` is incremented or set to 1.

- The increment of the `pending` counter in Listing 5.1 Line 40 of `frame`. The condition in Line 40 explicitly checks, if this turned the version into a placeholder. If so, the framing traversal recurses to all outgoing dependencies in Line 42f.
- The update of `changes` to 1 in Listing 5.2 Line 65 at the beginning of the propagation phase. This is executed for input reactives in `declaredReactives`, replacing their `pending == 1` from the framing phase with `changes == 1`. Therefore, this does not create a placeholder, but just change the counters of a version that already is a placeholder.
- The increment of `changes` in Line 99 in `notify`. This is executed in response to a predecessor turning from dirty to changed, meaning it also replaces one `pending` count with one `changes` count. Therefore, this also does not create a placeholder, but just change the counters of a version that already is a placeholder.
- The increment of `changes` or `pending` during drop retrofitting in Listing 5.4 Line 173 and Line 163. Both are immediately followed by conditions that check if a placeholder was created in Line 174f. and Line 182f. respectively. If so, a discovery retrofit traversal is executed recursively for all outgoing dependencies, which following Lemma 9 is equivalent to a framing traversal.

A placeholders may be removed wherever `pending` or `changes` is decremented or updated to 0.

- Update of `pending` to 0 for each input reactive at the beginning of the propagation phase in Listing 5.2 Line 66. This update is accompanied by a second update of `changes` to 1, and thus never removes a placeholder.

- The update of `changes` to 0 in Line 82 and Line 86 in `reevOut`. These lines are immediately followed by the propagation phase recursing to all outgoing dependencies in Line 83 and Line 87 respectively.
- The decrement of `pending` in Line 100 of `notify`. If this removed a placeholder, this is detected by Line 101 for `pending == 0`, and either Line 104 or Line 107 for `changes == 0`. In the former case, the propagation phase recurses to all outgoing dependencies immediately in Line 105f. In the latter case, `reevOut` is executed in Line 108, which in turn recurses to all outgoing dependencies in either Line 83 or Line 87.
- The decrement of `changes` or `pending` during drop retrofitting in Listing 5.4 Line 173 and Line 163. Both are immediately followed by conditions that check if a placeholder was removed in Line 174f. and Line 182f. respectively. If so, a drop retrofit traversal is executed recursively for all outgoing dependencies, which following Lemma 8 is equivalent to a (no-change) propagation traversal.

□

**Lemma 11** (Framing Phase and Propagation Phase Match)**.** *The framing traversal and the propagation traversal of any instant $t_i$, after accounting for any applied retrofitting, visit every reactive equally often.*

*Proof.* Both traversals start from the same set of input reactives (`declaredReactives`). For the framing traversal, this is Listing 5.1 Line 32. For the propagation traversal, this is Listing 5.2 Line 63. Input reactives are unaffected by retrofitting.

Both traversals recurse to derived reactives through the same outgoing dependencies set variables on the same conditions on each reactive. The framing traversal increments the `pending` counter on every reached node (Listing 5.1 Line 40). On the condition that this increment changed the counter's value from 0 to 1 (Line 41), it recurses to all outgoing dependencies of the node in Line 42f. For the propagation traversal, there are multiple places where it recurses to successor reactives. In all these locations, both the `pending` update and the recursion condition are reversed. The propagation traversal decrements the `pending` counter on every reached node. On the condition that this decrement changed the counter's value from 1 to 0, it traverses to all outgoing dependencies of the node eventually. These locations are:

- On each initial input reactives, the `pending` counter is explicitly updated to 0 in Listing 5.2 Line 66. The condition that this decrement changed the counter's value from 1 to 0 is not checked, because it is always true: the framing phase visits each of these input reactive precisely once, so their `pending` counter is guaranteed to be 1 when the propagation phase starts, so this update is always equivalent to decrementing the counter from 1 to 0. As a consequence, unconditionally, either `control` is called in Line 68, which then calls `reevOut` in Listing 2.9 Line 14, Line 17 or Line 19, or `reevOut` is called directly in Listing 5.2 Line 70. Therein, the propagation traversal then recurses to all outgoing dependencies in either Line 83f. or Line 87f.
- On each reached derived reactive, `pending` is decremented in Line 100. The condition that this decrement changed the value from 1 to 0 is checked in Line 101. Afterwards, it recurses to all outgoing dependencies over four possible paths.
  - Directly in Line 105f.
  - Indirectly in Line 108 through `reevOut` as above.
  - Indirectly in Line 109 through `reevaluate`, which in Listing 2.9 Line 11 calls again `control` as above.
  - Delayed, where `reevaluate` is ready to execute in terms of glitch freedom but not yet in terms of serializability because a preceding placeholder still exists. The condition in Line 102f. ensures the existence of said placeholder. By Lemma 6, the `reevaluate` task is guaranteed

to be executed when all preceding placeholders were removed. The reasoning following the execution of `reevaluate` was given in the previous point.

Given that both traversals start on the same reactives, and recurse over the same outgoing dependencies set variables under equivalent conditions, both traversals traverse the same set of nodes, as long as the outgoing dependencies set variables do not change between two traversals. Given the possibility of dynamic dependency edge drops or discoveries though, this is not necessarily the case. A discrepancy between framing and propagation traversal of $t_i$ arises, if an edge $r \underset{DG}{\longrightarrow} d$ is dropped or discovered after the framing traversal of $t_i$ recursed to the outgoing dependencies of $d$, but before its propagation traversal did. Applying Lemma 10, this can be rephrased as a discrepancy between framing and propagation traversal of $t_i$ arises, if an edge $r \underset{DG}{\longrightarrow} d$ is dropped or discovered while $t_i$ has a placeholder on $d$. We distinguish three cases.

- $t_i$ itself drops edge $r \underset{DG}{\longrightarrow} d$. In this case, a discrepancy is impossible. Recall that $t_i$ must be actively reevaluating $r$ during its propagation phase to execute this drop. Because $r \underset{DG}{\longrightarrow} d$ still exists, glitch freedom implies that $t_i$ must have already completed its reevaluation of $r$. Thus $t_i$ can execute such a drop only after both its framing and propagation phase recursed to the outgoing dependencies of $r$.

- $t_i$ itself discovers edge $r \underset{DG}{\longrightarrow} d$. In this case, a discrepancy is avoided. Such a discovery is executed in response to the reevaluation of $d$ having executed a `depend` read of $r$ while $r \underset{DG}{\longrightarrow} d$ did not exist yet. The existence of this edge is verified during the execution of `depend` in Listing 5.3 Line 113. If it does not exist, the read was dispatched through Line 116 to `after`. Therein, the read's execution is suspended in Line 135f until all placeholders by $t_i$ and earlier transactions have been removed. Thus the execution of such a discovery by $t_i$ is actively delayed until both its framing and propagation phase recursed to the outgoing dependencies of $r$.

- A different transaction $t_j$ drops or discovers $r \underset{DG}{\longrightarrow} d$. In this case, both $t_i$ and $t_j$ must have a version on $r$, meaning by Lemma 3, it must be either $t_i \underset{SSG}{\longrightarrow} t_j$ or $t_j \underset{SSG}{\longrightarrow} t_i$.
  - For $t_j$ with $t_i \underset{SSG}{\longrightarrow} t_j$ executing a drop of $r \underset{DG}{\longrightarrow} d$, such a discrepancy is impossible. Because $r \underset{DG}{\longrightarrow} d$ existed beforehand, a placeholder by $t_i$ on $r$ would imply that $t_i$ also has a placeholder on $d$. This placeholder would precede that of $t_j$, thus $t_j$ could not be reevaluating $d$, and thus $t_j$ could not execute this drop. Thus, a drop of $r \underset{DG}{\longrightarrow} d$ by such a later $t_j$ cannot execute while $t_i$ has a placeholder on $r$.
  - For $t_j$ with $t_j \underset{SSG}{\longrightarrow} t_i$ executing a discovery of $r \underset{DG}{\longrightarrow} d$, such a discrepancy is impossible, too. As stated above, such a discovery is executed after read dispatched through `after` in Listing 5.3 Line 133ff., which suspended in Line 135f until all placeholders of $t_j$ and earlier transactions have been removed. Thus, a discovery of $r \underset{DG}{\longrightarrow} d$ by such a later $t_j$ can also not execute while $t_i$ still has a placeholder on $r$.
  - A $t_j$ with $t_i \underset{SSG}{\longrightarrow} t_J$, a discrepancy *is* possible. But, retrofitting by $t_j$ will detect this discrepancy in `computeRetrofity`, where it $t_i$ matches the criteria from which set `retrofitFrame` is built in Listing 5.4 Line 163. In case of a drop, retrofitting by $t_j$ will execute a `reframe` traversal to undo the previously executed part of the framing traversal of $t_i$. Lemma 8 has shown this to be equivalent to a (no-change) propagation traversal. This resolves the discrepancy that would otherwise remain when from the propagation traversal of $t_i$ later would not recurse from $r$ to $d$ due to the now dropped edge, while the original framing traversal did. In case of a discovery, retrofitting by $t_j$ will execute a `reframe` traversal to belatedly execute the missing part of the framing traversal of $t_i$. Lemma 9 has shown the equality of this traversal to the

regular framing traversal. This means, the propagation traversal of $t_i$ will not introduce a discrepancy, when it later naturally recurses from $r$ to $d$ over the newly discovered edge, while the original framing traversal did not.

□

## 6.5 Serializability Proofs

This section contains proofs related to serializability, the key aspect of FrameSweep's correctness.

**Lemma 12** (Prefix Placeholder Freedom is Final)**.** *Once a node version of a propagating instant has no preceding placeholders, it will never have any preceding placeholders again.*

*Proof.* Consider an instant $t_i$ in the propagation phase with a node version on some reactive $r$, and an earlier node versions on $r$, i.e., by some instant $t_j$ with $t_j \xrightarrow[SSG]{*} t_i$, are not placeholders. We show that any creations of placeholders can only occur for instants $t_k$ that are ordered later than $t_i$. A placeholder may be created wherever `pending` or `changes` is incremented or set to 1.

- The increment of the `pending` counter in Listing 5.1 Line 40 of `frame`. An instant $t_k$ executing this increment must be in the framing phase, and must already have some version on $r$. Since $t_i$ also has a version on $r$, $t_k$ must be ordered against $t_i$. With $t_k$ in the framing phase and $t_i$ in the propagation phase, Lemma 7 implies that $t_k$ must be ordered later than $t_i$.
- The update of `changes` to 1 in Listing 5.2 Line 65 at the beginning of the propagation phase. This is executed for input reactives in `declaredReactives`, replacing their `pending == 1` from the framing phase with `changes == 1`. Therefore, this does not create a placeholder, but just change the counters of a version that already is a placeholder.
- The increment of `changes` in Line 99 in `notify`. This is executed in response to a predecessor turning from dirty to changed, meaning it also replaces one `pending` count with one `changes` count. Therefore, this also does not create a placeholder, but just change the counters of a version that already is a placeholder.
- The increment of `changes` or `pending` during drop retrofitting in Listing 5.4 Line 173 and Line 163. Here, a preceding placeholder by the instant that is dropping the edge exists, which forms the base of the protective shield for retrofitting. Since the node version of $t_i$ does not have any preceding placeholders, this can only create placeholders for $t_k$ that are ordered later than $t_i$.

□

**Corollary 1** (Propagating First Placeholder Ages Monotonously)**.** *From Lemma 12, it follows that once a placeholder of a propagating instant $t_i$ has become the first placeholder on any reactive $r$, it will remain the first placeholder on $r$ until it is removed. Once it is removed, neither $t_i$ itself nor any earlier instant can have a placeholder again. Thus, the next propagating instant $t_j$ to have a first placeholder on $r$ must be ordered later than $t_i$, i.e., has $t_i \xrightarrow[SSG]{*} t_j$. Therefore, the first placeholder on each reactive $r$ belongs to propagating instants in monotonously increasing order of the SSG.*

**Corollary 2** (`reevOut` Execute in Order of *SSG*)**.** *From Lemma 5, any execution of `reevOut` by an instant $t_i$ on a reactive $r$ occurs only if $t_i$ has the first placeholder on $r$. From Lemma 12, the first placeholder on each reactive $r$ belongs to propagating instants in monotonously increasing order of the SSG. Hence, for any reactive $r$, the order in which instants execute `reevOut` corresponds to the order of these instants in the SSG.*

**Corollary 3** (Reevaluations Execute in Order of *SSG*)**.** *From Lemma 4, any execution of* `reevaluate` *by an instant $t_i$ on a reactive $r$ occurs only if $t_i$ has the first placeholder on $r$. From Lemma 12, the first placeholder on each reactive $r$ belongs to propagating instants in monotonously increasing order of the SSG. Hence, for any reactive $r$, the order in which instants execute reevaluations corresponds to the order of these instants in the SSG.*

**Lemma 13** (User Value Read Order)**.** *An instant $t_i$ executing any read operation of the user value $v_r$ of a reactive $r$ is ordered in SSG against all other placeholders and written versions on $r$, and execute the read only once no more placeholders by earlier instants can exist.*

*Proof.* An instant $t_i$ can read the user value $v_r$ in the following locations:

- In Listing 5.3 Line 119 of `Event.readSynchronized` and Line 124 of `Signal.readSynchronized`. These are in turn called in two locations:
  - In Line 113 of `depend`, under the condition that an edge $r \underset{DG}{\longrightarrow} d$ exists for some derived reactive $d$ that $t_i$ is currently reevaluating. Following Lemma 4, $t_i$ must have a placeholder on $d$ for this reevaluation. Any placeholder or written version on $r$ by any $t_j$ implies that the framing traversal and maybe also the propagation traversal of $t_j$ recursed to all outgoing dependencies of $r$, which includes $d$ because of the edge $r \underset{DG}{\longrightarrow} d$. Hence, $t_j$ must have create a placeholder on $d$, and may have replaced it with a written or idempotent version. In either case, both $t_j$ and $t_i$ have a version on $d$, and following Lemma 3 are thus ordered against each other. Further, again following Lemma 4, the placeholder by $t_i$ is the first placeholder on $d$, i.e., no placeholders by any earlier $t_j$ can exist. This implies that these $t_j$ cannot have a placeholder on $r$ either, because any such placeholder would imply that the propagation traversal of $t_j$ could not have recursed to $d$ yet, and thus $t_j$ would still have a placeholder on $d$ ordered earlier than that of $t_i$. Further, Lemma 12 applies on $d$, implying that no placeholders can be created by any earlier instant, which transfers to $r$ for the same reason.
  - In Line 137 of `after`. Here, the preceding Line 134 ensures that $t_i$ has a node version on $r$. By Lemma 3, $t_i$ is therefore ordered against all instants that also have a node version on $r$. Subsequently, Line 135 explicitly suspends the read's execution until all earlier instants' placeholders have been removed. At that point, Lemma 12 applies, ensuring that no new placeholders can be created by earlier instants.
- In Line 132 of `Signal.beforeSynchronized`. Here, the preceding Line 139 ensures that $t_i$ has a node version on $r$. By Lemma 3, $t_i$ is therefore ordered against all instants that also have a node version on $r$. Subsequently, Line 140 explicitly suspends the read's execution until all earlier instants' placeholders have been removed. At that point, Lemma 12 applies, ensuring that no new placeholders can be created by earlier instants.

□

**Lemma 14** (Properties of Outgoing Dependencies Reads)**.** *Any read of the outgoing dependencies sets of a reactive $r$ (a) is done in the context of some graph traversal of an instant $t_i$ recursing to all outgoing dependencies of $r$, (b) $t_i$ has a node version on $r$, and (c) occurs either before or while $t_i$ is in the propagation phase and has the first placeholder on $r$.*

*Proof.* The outgoing dependencies set is read in the following locations:

- In Listing 5.1 Line 42f. of `frame`. For (a), this occurs when the framing traversal of $t_i$ recurses to all outgoing dependencies of $r$. For (b), the preceding Line 39 ensures that $t_i$ has a node version on $r$. For (c), `frame` executes during in the framing phase, before $t_i$ even reaches its propagation phase.

- In Listing 5.2 Line 83f. or Line 87f. of `reevOut`. For (a), this occurs when the propagation traversal of $t_i$ recurses to all outgoing dependencies of $r$. Lemma 5 implies that $t_i$ must the first placeholder on $r$ to execute `reevOut`, which fulfils both (b) and (c).

- In Line 105f. of `notify`. For (a), this occurs when no-change propagation of $t_i$ recurses to all outgoing dependencies of $r$. For (b), the preceding Line 97 ensures that $t_i$ has a node version on $r$. For (c), the condition in the in preceding Line 102f explicitly checks, that a placeholder by an earlier instant still exists.

- In Listing 5.4 Line 176 and Line 184. For (a), this occurs when a retrofit traversal for $t_i$ recurses to all outgoing dependencies of $r$. For (b), the preceding Line 172 or Line 180 respectively ensures that a version exists. For (c), since retrofitting can affect only non-first placeholders (first placeholders are held by the instant whose dynamic dependency edge change necessitated retrofitting, forming the protective shield for its rewrites), this occurs before $t_i$ can have the first placeholder on $r$.

□

**Theorem 3** (*SSG* Correctly Represents *MVSG*)**.** *The SSG is a correct representation of the MVSG, in that for every edge* $t_i \xrightarrow[MVSG]{} t_j$ *there is a path* $t_i \xrightarrow[SSG]{*} t_j$.

*Proof.* Proof by induction, with Theorem 3 as the induction hypothesis. Recall from Definition 3 the construction rules for *MVSG* edges: Every $read_j(x^i)$ implies edge $t_i \xrightarrow[MVSG]{} t_j$, and every pair of $read_j(x^i)$ and $write_k(x^k)$ implies either $t_k \xrightarrow[MVSG]{} t_i$ if $x_k \ll_{SSG} x_i$, or $t_j \xrightarrow[MVSG]{} t_k$ if $x_i \ll_{SSG} x_k$ instead. As induction base, consider the initial state of any application, where no operations have been executed, i.e., $H = \emptyset$. In that case, none of these construction rules apply, meaning the *MVSG* contains no edges, so the hypothesis is trivially fulfilled. As induction steps, consider every read operation and every write operation on variables $v_\mathtt{r}$, $inc_\mathtt{r}$ and $out_{\mathtt{r},\mathtt{d}}$ from FrameSweep's pseudocode. Every execution of such an operation adds additional $write_k(x^k)$ and $read_j(x^i)$ elements to $H$, which creates additional cases for these construction rules to apply, meaning it adds new edges to the *MVSG*. It thus remains to prove that for every induction step, i.e., for every such read and write operation, that the *SSG* contains a path $t_i \xrightarrow[SSG]{*} t_j$ for every edge $t_i \xrightarrow[MVSG]{} t_j$ that is newly added to the *MVSG*.

First, consider all read operations. Every execution by an instant $t_j$ of a read operation on variable $x$ adds a $read_j(x^i)$ to $H$. From Definition 3, this adds the following edges to the *MVSG*:

- $t_i \xrightarrow[MVSG]{} t_j$ for the instant $t_i$ that wrote the value returned by the read. For each individual read operation, the proofs will show that the returned value is selected such that an path $t_i \xrightarrow[SSG]{*} t_j$ exists in the *SSG*.

- $t_k \xrightarrow[MVSG]{} t_i$ for every $write_k(x^k) \in H$ with $write_k(x^k) \ll write_i(x^i)$. Following Definition 5, $write_k(x^k) \ll write_i(x^i)$ is derived from $t_k \xrightarrow[SSG]{*} t_i$, which corresponds to the required edge. Thus, these *MVSG* edges are always covered by *SSG* paths, and need not be addressed individually for each read.

- $t_j \xrightarrow[MVSG]{} t_k$ for every $write_k(x^k) \in H$ with $write_i(x^i) \ll write_k(x^k)$. For each individual read operation, the proofs will show that either – in the case of $v_\mathtt{r}$ – a corresponding path $t_j \xrightarrow[SSG]{} t_k$ exists, or – in the case of $inc_\mathtt{r}$ and $out_{\mathtt{r},\mathtt{d}}$ – no such $write_k(x^k)$ can exist.

FrameSweep's pseudocode contains the following read operations:

- $read_j(inc_r^i)$ executed in Listing 2.9 Line 21 and Line 23 of `updateDeps`: Since `updateDeps` is executed only in Line 10 of `reevaluate`, $t_j$ is currently executing a reevaluation of $r$.
  - $t_i \xrightarrow[MVSG]{} t_j$ for the instant $t_i$ that wrote the value $inc_r^i$ returned by the read. Since FrameSweep stores only a single version of $inc_r$ on each $r$, $t_i$ is the last instant – in real time – that wrote $inc_r$. The only write of $inc_r$ is in Line 20 of `reevaluate`, so $t_i$ is the last instant – in real time – that reevaluated $r$ before $t_j$. Corollary 3 has shown that the real time order in which reevaluations execute matches the *SSG* order. Thus $t_i \xrightarrow[SSG]{*} t_j$ must exist, covering the *MVSG* edge.
  - $t_j \xrightarrow[MVSG]{} t_k$ for every $write_k(inc_r^k) \in H$ with $write_i(inc_r^i) \ll write_k(inc_r^k)$. Following Definition 5, $write_i(inc_r^i) \ll write_k(inc_r^k)$ requires $t_i \xrightarrow[SSG]{*} t_k$, i.e., that $t_k$ is ordered later than $t_i$. Since FrameSweep stores only a single version of each $inc_{r,d}$ on each $r$, a $read_j(inc_{r,d}^i)$ implies that $t_i$ was the latest instant – in real time – to write $inc_{r,d}$. The only write of $inc_{r,d}$ is in Listing 2.9 Line 25 of `updateDeps`, which in turn is executed only in Line 10 of `reevaluate`. Since, following Corollary 3, real time order of reevaluations matches *SSG* order, $t_i$ must also be the latest instant in *SSG* order that wrote $inc_{r,d}$. Thus, any $t_k$ ordered later than $t_i$ in the *SSG* cannot have executed a $write_k(inc_r^k)$ yet, thus this construction rule never applies.
- $read_j(v_r^j)$ in Listing 5.3 Line 118f. of `Event.readSynchronized` and Line 123f. of `Signal.after`: These reads correspond to $t_j$ reading $v_r^j$ from $write_j(v_r^j)$ written by itself. The *MVSG* construction rules apply only to instants reading values of other instants, so they do not apply in these cases.
- $read_j(v_r^i)$ executed in Listing 5.3 Line 132 of `Signal.beforeSynchronized`:
  - $t_i \xrightarrow[MVSG]{} t_j$ for the instant $t_i$ that wrote the value $v_r^i$ returned by the read. The value $v_r^i$ is chosen from an instant $t_i$ in the set `preds` computed in Line 128ff. This set is constructed such that it contains only instants $t_i$ with $t_i \xrightarrow[SSG]{*} t_j$, hence the *MVSG* edge is covered.
  - $t_j \xrightarrow[MVSG]{} t_k$ for every $write_k(v_r^k) \in H$ with $write_i(v_r^i) \ll write_k(v_r^k)$. The only write of $v_r$ is in Listing 5.2 Line 81 of `reevOut`. Following Lemma 5, any such $t_k$ must therefore have had a placeholder in the node history of $r$, which has been replaced by a written version by the $write_k(v_r^k)$. We show that, as a result, either $t_j \xrightarrow[SSG]{*} t_k$ or $t_k \xrightarrow[SSG]{*} t_j$, for every location where `Signal.beforeSynchronized` is executed:
    - In Line 141 of `before`, the preceding Line 139 created a node version for $t_j$ on $r$. Thus, both $t_k$ and $t_j$ have a node version on $r$, and thus Lemma 3 implies that either $t_j \xrightarrow[SSG]{*} t_k$ or $t_k \xrightarrow[SSG]{*} t_j$.
    - In Line 126 of `Signal.readSynchronized` called from Line 137 of `after`, the preceding Line 134 created a node version for $t_j$ on $r$. Thus, both $t_k$ and $t_j$ have a node version on $r$, and thus Lemma 3 implies that either $t_j \xrightarrow[SSG]{*} t_k$ or $t_k \xrightarrow[SSG]{*} t_j$.
    - In Line 126 of `Signal.readSynchronized` called from Line 114 of `depend`, $t_j$ may not have a version on $r$. To execute `depend`, $t_j$ must be reevaluating some derived reactive $d$, meaning Lemma 4 implies that $t_j$ has a version on $d$. Further, this branch is executed only on the condition that edge $r \xrightarrow[DG]{} d$ exists, discovered by some instant $t_x$. Because $t_x$ can have discovered $r \xrightarrow[DG]{} d$ only during its own reevaluation of $d$, but $t_j$ is currently reevaluating $d$, Corollary 3 implies that $t_x \xrightarrow[SSG]{*} t_j$. Further, following Lemma 1, $t_x$ must have a version on $r$ from executing this discovery, so following Lemma 3, it is either

$t_x \xrightarrow[SSG]{*} t_k$ or $t_k \xrightarrow[SSG]{*} t_x$. In the case of $t_x \xrightarrow[SSG]{*} t_k$, $t_k$ must also have had a placeholder on $d$, created by either own framing traversal or a retrofitting traversal by $t_x$. Thus, both $t_k$ and $t_j$ have a node version on $d$, and thus Lemma 3 implies that either $t_j \xrightarrow[SSG]{*} t_k$ or $t_k \xrightarrow[SSG]{*} t_j$. In the case of $t_k \xrightarrow[SSG]{*} t_x$, $t_k \xrightarrow[SSG]{*} t_j$ follows by transitivity with $t_x \xrightarrow[SSG]{*} t_j$.

We show by contradiction, that $t_k \xrightarrow[SSG]{*} t_j$ is impossible, meaning $t_j \xrightarrow[SSG]{*} t_k$ must be true, which covers the *MVSG* edge. If $t_k \xrightarrow[SSG]{*} t_j$ was true, then $t_k$ would have been included in set `preds` from Line 128ff. From this set, $t_i$ was selected as the latest instant in Line 131, implying that all other $t_k$ must be ordered $t_k \xrightarrow[SSG]{*} t_i$. Following Definition 5, $write_i(v_r^i) \ll write_k(v_r^k)$ requires $t_i \xrightarrow[SSG]{*} t_k$ in the reverse direction though. Following Lemma 2, $t_k \xrightarrow[SSG]{*} t_i$ thus cannot be the case, as it would close a cycle in the *SSG* in combination with $t_i \xrightarrow[SSG]{*} t_k$.

- $read_j(out^i_{r,d})$ executed whenever a graph traversal recurses to all outgoing dependencies of $r$. Following Lemma 14, (a) $t_j$ has a version on $r$ for any such read, and (b) it occurs either before or while $t_j$ is propagating and has the first placeholder on $r$. Since in all of these cases, either an earlier instant or $t_j$ itself has not completed its propagation traversal over $r$, glitch freedom implies that this earlier instant or $t_j$ itself cannot have completed its propagation traversal over $d$ either. This in turn means, that no instant $t_k$ ordered $t_j \xrightarrow[SSG]{*} t_k$ can have started a reevaluation of $d$ yet.

  - $t_i \xrightarrow[MVSG]{} t_j$ for the instant $t_i$ that wrote the value $out^i_{r,d}$ returned by the read. The only writes of $out_{r,d}$ are in Listing 5.4 Line 150 of `drop` and Line 156 of `discover`. Following Lemma 1, both drop and discovery imply that $t_i$ has a version on $r$, meaning both $t_i$ and $t_j$ have a version on $r$. Following Lemma 3, this implies that it is either $t_i \xrightarrow[SSG]{*} t_j$ or $t_j \xrightarrow[SSG]{*} t_i$. We show that $t_j \xrightarrow[SSG]{*} t_i$ cannot be true for both drop and discovery by contradiction, thus instead $t_i \xrightarrow[SSG]{*} t_j$ must be true, which covers the *MVSG* edge.

    Consider $t_i$ with $t_j \xrightarrow[SSG]{*} t_i$ to have executed a drop of edge $r \xrightarrow[DG]{} d$. Drop and discovery of an edge $r \xrightarrow[DG]{} d$ can be executed by $t_i$ only while reevaluating $d$, and therefore only after having reached the propagation phase. Following Lemma 7, this means that $t_j$ must also have reached the propagation phase, and must be executing one of the propagation traversals. This means the framing phase of $t_j$ executed while edge $r \xrightarrow[DG]{} d$ still existed, and thus must have created a placeholder for $t_j$ on $d$. This in turn means, that $t_i$ cannot have had the first placeholder on $d$ yet, and by Lemma 4 thus cannot have started a reevaluation of $d$ and thus cannot have executed a drop of edge $r \xrightarrow[DG]{} d$.

    Consider $t_i$ with $t_j \xrightarrow[SSG]{*} t_i$ to have executed a discovery of edge $r \xrightarrow[DG]{} d$. Such a discovery is always executed in response to a `depend` read of $r$ during a reevaluation of $d$ while edge $r \xrightarrow[DG]{} d$ did not exist, i.e., the `depend` read was dispatched in Listing 5.3 Line 116 through `after`, defined in Listing 5.3 Line 133ff. Therein, in Line 135, such reads are suspended until all earlier placeholders have been cleared. Seeing as $t_j$ executing its read of $out_{r,d}$ can only happen when either itself or an even earlier instant still has the first placeholder on $r$, this suspension would not have completed yet, and therefore such a discovery cannot have executed yet.

– $t_j \xrightarrow[MVSG]{} t_k$ for every $write_k(out^k_r) \in H$ with $write_i(out^i_r) \ll write_k(out^k_r)$. Following Definition 5, $write_i(out^i_r) \ll write_k(out^k_r)$ requires $t_i \xrightarrow[SSG]{*} t_k$, i.e., that $t_k$ is ordered later than $t_i$. Since FrameSweep stores only a single version of each $out_{r,d}$ on each $r$, a $read_j(out^i_{r,d})$ implies that $t_i$ was the latest instant – in real time – to write $out_{r,d}$. The only writes of $out_{r,d}$ are in Listing 5.4 Line 150 of `drop` and Line 156 of `discover`. Drop and discovery of an edge $r \xrightarrow[DG]{} d$ can be executed by $t_k$ only while reevaluating $d$. Since, following Corollary 3, real time order of reevaluations matches *SSG* order, $t_i$ must also be the latest instant in *SSG* order that wrote $out_{r,d}$. Thus, any $t_k$ ordered later than $t_i$ in the *SSG* cannot have executed a $write_k(out^k_r)$ yet, thus this construction rule never applies.

•

Every execution by instant $t_k$ of a write operation on variable $x$ adds a $write_k(x^k)$ to $H$. From Definition 3, this adds the following edges to the *MVSG*:

- $t_k \xrightarrow[MVSG]{} t_j$ for every $read_j(x^k) \in H$. Since no instant can read $x^k$ before it was written, no such reads can exist yet at the time where $write_k(x^k)$ is executed. Thus this rule never applies.
- $t_k \xrightarrow[MVSG]{} t_i$ for every $read_j(x^i) \in H$ with $write_k(x^k) \ll write_i(x^i)$. Following Definition 5, $write_k(x^k) \ll write_i(x^i)$ is derived from $t_k \xrightarrow[SSG]{*} t_i$, which corresponds to the required *MVSG* edge. Thus, these *MVSG* edges are always covered by *SSG* paths, and need not be addressed individually for each read.
- $t_j \xrightarrow[MVSG]{} t_k$ for every $read_j(x^i) \in H$ with $write_i(x^i) \ll write_k(x^k)$. For each individual write operation, the proofs will show that a corresponding path $t_j \xrightarrow[MVSG]{} t_k$ exists.

FrameSweep's pseudocode contains the following read operations:

- $write_k(inc^k_r)$, executed in Listing 2.9 Line 25 of `updateDeps`:
  - Edges $t_j \xrightarrow[MVSG]{} t_k$ for every previously executed $read_j(inc^i_r)$ with $write_i(inc^i_r) \ll write_k(inc^k_r)$. This write is executed as part of `reevaluate`, meaning $t_k$ is currently reevaluating $r$. Following Corollary 3, all reevaluations of $r$ are executed in order of the *SSG*. The only reads of $inc_r$ are in Line 21 and Line 23 of `updateDeps`, also part of `reevaluate`. Hence any $read_j(inc^i_r)$ must come from an earlier reevaluation, and is thus ordered $t_j \xrightarrow[SSG]{*} t_k$, which covers the requried *MVSG* edge.
- $write_k(v^k_r)$, executed in Listing 5.2 Line 81 of `reevOut`.
  - Edges $t_j \xrightarrow[MVSG]{} t_k$ for every previously executed $read_j(v^i_r)$ with $write_i(v^i_r) \ll write_k(v^k_r)$. Following Lemma 5, to execute `reevOut`, $t_k$ must have the first placeholder on $r$. Following Lemma 13, any $t_j$ must be ordered against $t_k$, i.e., either $t_j \xrightarrow[SSG]{*} t_k$ or $t_k \xrightarrow[SSG]{*} t_j$. Further following Lemma 13, the latter cannot be the case, since such reads would be suspended, waiting for the placeholder by $t_k$ to be removed first. Hence, it must be $t_j \xrightarrow[SSG]{*} t_k$, which covers the required *MVSG* edge.
- $write_k(out^k_r)$, executed in Listing 5.4 Line 150 of `drop` and Line 156 of `discover`.
  - Edges $t_j \xrightarrow[MVSG]{} t_k$ for every previously executed $read_j(out^i_{r,d})$ with $write_i(out^i_{r,d}) \ll write_k(out^k_{r,d})$. Following Lemma 14, $t_j$ must have a version on $r$, as does $t_k$ from Lemma 1. Hence, $t_j$ and $t_k$ must be ordered in the *SSG*, i.e., it is either $t_j \xrightarrow[SSG]{*} t_k$ – which covers the

required *MVSG* edge – or $t_k \xrightarrow[SSG]{*} t_j$. The latter case is possible, and would close a cycle in combination with the required *MVSG* edge, meaning these edges are *not* covered by the *SSG*. We distinguish the following cases, depending on what kind of version $t_j$ has on $r$:

* $t_j$ has an idempotent version on $r$. In this case, $t_j$ may have traversed $r$ with a framing traversal (or an equivalent discovery retrofitting traversal, cf. Lemma 9) and a subsequent no-change propagation traversal (or an equivalent drop retrofitting traversal, cf. Lemma 8). A framing traversal followed by a no-change propagation traversal is by design not user-visible. Therefore, in terms of user-visible operations, the non-serializable $read_j(out^i_{\mathtt{r},\mathtt{d}})$ is equivalent to a serializable $read_j(out^k_{\mathtt{r},\mathtt{d}})$ without any modifications.
* $t_j$ has a placeholder on $r$. In this case, $t_j$ has traversed $r$ with a framing traversal (or an equivalent discovery retrofitting traversal, cf. Lemma 9). This $t_j$ is collected by $t_k$ in the set `retrofitFrame` in Listing 5.4 Line 163ff. of `computeRetrofit`. If $t_k$ is dropping edge $r \xrightarrow[DG]{} d$, it submits a drop retrofitting traversal for $t_j$ in Line 169, which following Lemma 8 is equivalent to a no-change propagation traversal of $t_j$. This no-change propagation traversal negates the previously executed framing (or equivalent discovery retrofitting) traversal in that their combined effects are not user-visible. If $t_k$ is discovering edge $r \xrightarrow[DG]{} d$, it submits a discovery retrofitting traversal for $t_j$ in Line 169 instead, which following Lemma 9 is equivalent to the normal framing traversal of $t_j$ over $d$. In both cases, the non-serializable $read_j(out^i_{\mathtt{r},\mathtt{d}})$ is rewritten into a serializable $read_j(out^k_{\mathtt{r},\mathtt{d}})$.
* $t_j$ has a written version on $r$. In this case, $t_j$ has traversed $r$ with a framing traversal (or an equivalent discovery retrofitting traversal, cf. Lemma 9) and a change propagation traversal, i.e., at least two instances of $read_j(out^i_{\mathtt{r},\mathtt{d}})$ were executed. This $t_j$ is collected by $t_k$ in the set `retrofitChange` in Listing 5.4 Line 160ff. of `computeRetrofit`. If $t_k$ is dropping edge $r \xrightarrow[DG]{} d$, it undoes in Line 173 the effects of the previously received `Changed notify` in Listing 5.2 Line 99, thereby turning the previously executed change propagation into no-change propagation. Further, as necessary, $t_k$ submits a drop retrofitting traversal for $t_j$ to all successors of $d$, which following Lemma 8 is equivalent to a no-change propagation traversal of $t_j$. Again, this no-change propagation traversal negates the previously executed framing (or equivalent discovery retrofitting) traversal in that their combined effects are not user-visible. If $t_k$ is discovering edge $r \xrightarrow[DG]{} d$, it applies in Listing 5.4 Line 173 only the effects of Listing 5.2 Line 99, since this is the only remainder after the combination of a framing traversal (`pending += 1`) and a `Changed notify` (`pending -= 1` and `changes += 1`). Further, as necessary. $t_k$ submits a discovery retrofitting traversal for $t_j$ instead, which following Lemma 9 is equivalent to the normal framing traversal of $t_j$ over $d$. In both cases, all non-serializable $read_j(out^i_{\mathtt{r},\mathtt{d}})$ are rewritten into serializable $read_j(out^k_{\mathtt{r},\mathtt{d}})$.

To show that the replaced $read_j(out^k_{\mathtt{r},\mathtt{d}})$ are indeed serializable, we show that all their required *MVSG* edges are covered by *SSG* edegs:

* $t_k \xrightarrow[MVSG]{} t_j$ is covered by $t_k \xrightarrow[SSG]{*} t_j$ from the precondition above.
* $t_j \xrightarrow[MVSG]{} t_l$ for every $write_l(out^l_{\mathtt{r}}) \in H$ with $write_k(out^k_{\mathtt{r}}) \ll write_l(out^l_{\mathtt{r}})$. Following Definition 5, $write_k(out^k_{\mathtt{r}}) \ll write_l(out^l_{\mathtt{r}})$ is derived from $t_k \xrightarrow[SSG]{*} t_l$, i.e., $t_l$ is ordered later than $t_k$. The only writes of $out_{\mathtt{r},\mathtt{d}}$ are in Listing 5.4 Line 150 of `drop` and Line 156 of `discover`. Both are executed by $t_l$ only while reevaluating $d$. Since, following Corollary 3, reevaluations execute in order of the *SSG* order, and $t_k$ is currently reevaluating $d$, no such $t_l$ can have executed a $write_l(out^l_{\mathtt{r}})$ yet, thus this construction rule never applies.

□

**Theorem 4** (Correctness of FrameSweep)**.** *FrameSweep produces only abort-free strict serializable histories.*

*Proof.* Following Lemma 2, the *SSG* is acyclic. Following Theorem 3, the *SSG* correctly represents the *MVSG* for all histories $H$ that FrameSweep can produce, which is therefore also acyclic. Following Theorem 1 with $\ll_{SSG}$ thus implies that all such $H$ are therefore serializable.

Consider as the linearization point for any instant $t_i$ Listing 5.2 Line 74. This line is executed after instants were started, but before they complete. The previous suspension condition in Line 71ff. ensures that this line is executed for all instants in an order matching the *SSG*. Therefore, FrameSweep ensures linearizability for instants, with their linearization order matching their serialization order, thus strict serializability overall.

Lastly, FrameSweep's pseudocode implementation does not contain any abort or roll-back mechanisms for reevaluations. Since aborts or roll-backs do not occur naturally in programs, FrameSweep's execution is therefore also abort-free. In conclusion, FrameSweep thus provides isolation in the form of abort-free strict serializability for all instants, including their mark-sweep propagation for glitch freedom and liveness. □

# 7 Distribution Engineering and Optimizations

This chapter addresses modifications, additional engineering and optimizations for devising a decentralized – and therefore suitable for distributed applications – implementation of FrameSweep. First, Section 7.1 discusses how FrameSweep extends the way it does graph traversal from Chapter 5 to support traversals over distributed *DG* topologies. Next, Section 7.2 discusses lazy framing, an optimization that removes the need for retrofitting to execute additional graph traversals beyond the framing and propagation phase, and reduces the amount of interactions between concurrent instants in general. Section 7.3 presents FrameSweep's decentralized implementation of the *SSG* functionality, and how node histories from different hosts safely interact with it through more fine-grained synchronization than global mutual exclusion. Section 7.4 discusses garbage collection of obsolete node versions from reactives' node histories and completed instants from the decentralized *SSG*. Finally, Section 7.5 presents FrameSweep's support for extended transactions, which allow imperative threads to execute multiple `s.now` reads atomically together with the execution of an `update(...)` instant.

## 7.1 Distributed Propagation

This section discusses, how FrameSweep adapts the way it executes instants' graph traversals to distributed applications. Figure 7.1 illustrates how interactions between reactives and thread pools differ in the distributed setting. At its core, the figure shows a distributed *DG* topology consisting of two hosts. Its dependency graph is visualized through reactives connected by solid bold edges for local dependency edges, and a hollow bold edge for a remote dependency crossing the network between the two hosts. For a summary of FrameSweep's local implementation, consider for now only the interactions inside the left host A.

Locally, the execution of graph traversals is implemented by reactives pushing `frame` and `notify` tasks during the framing and propagation phase respectively upon each other along their dependency edges indirectly through the thread pool. The dotted arrows between thread pool and reactive *a* visualize the resulting cycle of direct interactions between thread pool and reactives. The thread pool runs ("run") a submitted task by dispatching a worker thread to the affected reactive. During the task's execution, the reactive submits ("submit") new tasks to the thread pool, which increment the associated instants' `activeTasks` counter. Once the task is done ("done"), the worker thread returns to the thread pool. The transfer of user values between reactives during the propagation phase on the other hand is implemented through direct interactions between reactives, visualized as dashed arrows. Submitted `notify` tasks carry only a boolean parameter of whether or not a reactive has changed. Values are then retrieved during ongoing reevaluations by executing read operations `before`, `after` and `depend` directly on other reactives, usually also along dependency edges but in the reverse direction.

In summary, instants' graph traversals in FrameSweep are a back-and-forth process: Push tasks forwards along dependency edges indirectly through interactions with a thread pool that counts the number of active tasks, and (during propagation) pull changed values through direct interactions in usually the reverse direction of dependency edges. Applying this model in distributed applications, where instants affect reactives on multiple hosts and dependency edges cross network boundaries, would incur two significant performance impacts. First, any instant's tasks are spread across multiple host. This would make governing their execution through a single thread pool on one host highly inefficient, as every submitted and completed task would have to be started and report its completion through slow network communication. Second, back-and-forth interaction between reactives would repeated back-and-forth remote communication over each remote dependency edge for the propagation a single change.

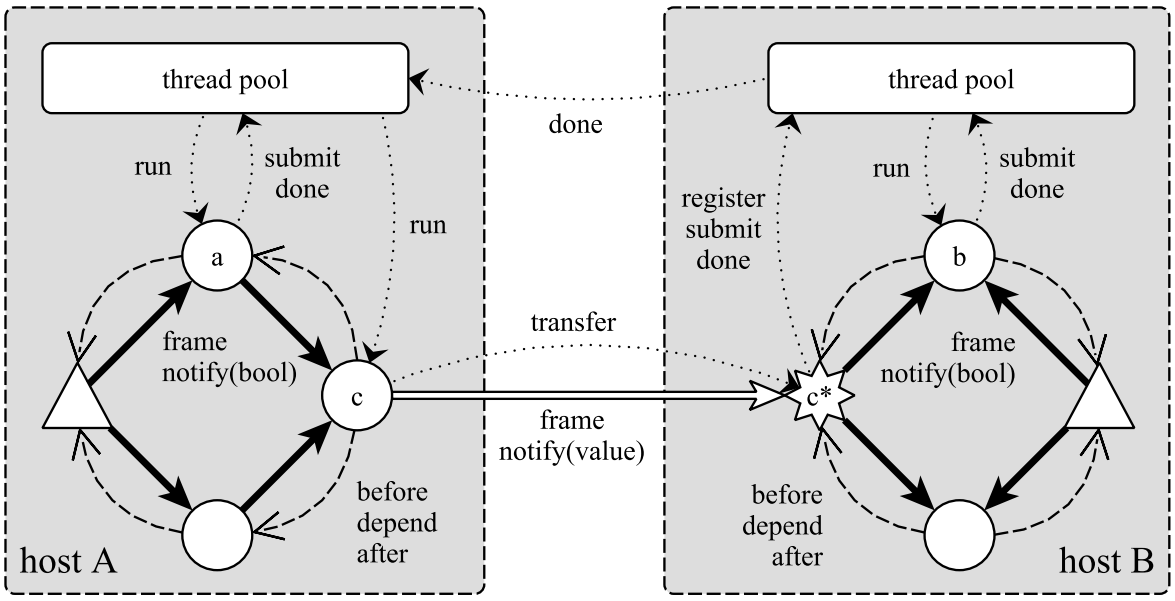



**Figure 7.1:** FrameSweep's Fundamental Approach to Distribution

To minimize the amount of necessary remote communication, FrameSweep implements a different communication scheme for change propagation across remote dependencies. First, FrameSweep introduces *mirror reactives* for reactives that are shared remotely. Local reactives never depend on remote reactives directly. Instead, all remote dependency edges are indirected through a mirror reactive on each remote host. In Figure 7.1, the star-shaped reactive labelled $c^*$ is the mirror reactive for $c$ from host A on host B. The mirror reactive serves as a remote replica of the original reactives' node history. It thereby acts as a gateway for all derived reactives on host B to depend on the remotely shared reactive $c$ from host A while still requiring only a single remote dependency edge. As a first improvement, this allows any number of local derived reactives to engage in back-and-forth interaction with the same remote reactive at the same cost as just a single one.

Second, remotely shared reactives interact with their mirrors through different operations than local reactives on individual hosts during the propagation phase, replacing the back-and-forth interaction with purely forwards push-based propagation. Remote reactives send a `notify` task to all mirror reactives that does not just carry a boolean flag of whether or not a change occurred, but instead directly pushes the newly changed or no value. Upon receiving a notify task with no value, it simply sends out local `notify` tasks with their boolean flag set as `Unchanged`, thus engaging with further local reactives on the remote host regularly through back-and-forth interactions again. Upon receiving a notify task with a changed value, the mirror reactive adds this value to its node history, thereby replicate the original reactive's change without communicating backwards over the remote dependency edge, and then sends out local `notify` tasks with a boolean flag set as `Changed`. This way, the propagation traversal executes equally to the framing traversal over remote dependency edges by pushing tasks forwards, without any reads in the opposite direction.

Third, the run-submit-done cycle of interactions, which is executed during each graph traversal locally once per affected reactive, is coarsened over remote dependency edges so that it executes only once for each remotely affected host, regardless of the actual number of reactives affected within that host. In essence, this makes FrameSweep's graph traversals an instance of diffusing computations [Dijkstra and Scholten, 1980]. Figure 7.1 visualizes this coarsened cycle of interactions in the center of the figure, above the remote dependency edge. Initially equal to local propagation, the thread pool on host A runs ("run") a submitted `frame` or `notify` task for a given instant on the remotely shared reactive $c$. Then differently though, $c$ does not submit further tasks for the instant to the local thread pool, and does not

report itself as done once its reevaluation is complete. Instead, alongside sending a `frame` or `notify` task to its mirror reactive $c^*$, it conceptually transfers ("transfer") its running task to host B. Upon receiving this `frame` or `notify` task, the mirror reactive $c^*$ registers ("register") with the local thread pool that it is executing a task that is actually counted by the thread pool on host A. At this point, the instant has two active tasks counted: one in the thread pool of host A, and one in the thread pool on host B. The active task on host B is the ongoing reevaluation of the mirror reactive $c^*$, and will interact with the thread pool on B as a local graph traversal, submitting and completing tasks normally ("submit" and "done"). The active task counted on host A does not correspond to an actual active task though, but instead stands as a representation of the instant being active – with any number of actual tasks – on host B. When the instant's `activeTasks` counter on host B eventually reaches zero, the host B will report a single "done" to the thread pool on host A, representative for *all* tasks on host B having completed. With this, the instant's `activeTasks` can then also reach zero and thereby also conclude termination of the entire distributed graph traversal.

In summary, these modifications achieve that FrameSweep executes instants over distributed *DG* topologies through two back-and-forth communications over each remote dependency edge. During the framing traversal, a single `frame` task is pushed across, and the remote host eventually reports completion of all subsequently spawned tasks in a single return message. During the propagation traversal, the same happens with a single `notify` task that pushes the remotely shared reactive's changed value eagerly to the remote host, thereby avoiding any backwards communication to retrieve this value

## 7.2 Lazy Framing

Lazy Framing is an optimization that affects several aspects of FrameSweep in the cases where pipeline parallelism is being utilized. Most importantly, it removes the need for retrofitting to execute graph traversals, thereby achieving that FrameSweep is indeed only a two-phase algorithm. Lazy Framing is based on nearly the same observation that forms the basis for the protective shield that enables retrofitting: If an instant $t$ has a placeholder on some reactive $r$, but it is not the first placeholder on that reactive, then all placeholders by $t$ on all reactives $d$ reachable from $r$ (i.e., $r \xrightarrow[DG]{*} d$) are not first placeholders either. This is because all instants $t'$ that have an earlier placeholder on $r$ (i.e., $t' \xrightarrow[SSG]{*} t$), will also have earlier placeholders on all $d$. Moreover, because of glitch freedom allowing placeholders to be removed only in topological order of the *DG*, any of these placeholders on any $d$ can only be removed after the respective placeholders on $r$ were removed first. Therefore, the placeholder by $t$ on $r$ will become the first placeholder on $r$, before any placeholder by $t$ on any $d$ can become the first placeholder on that $d$. In conclusion, this means that any placeholders created by $t$ on any such $d$ are useless while $t$ does not yet have the first placeholder on $r$, because no corresponding reevaluations can be executed yet. The only effect of eagerly creating these useless placeholders is that more placeholders will have to be added, removed or updated by potential later retrofitting executions. The key change implemented by Lazy Framing is therefore to defer the creation of useless placeholders until they may actually be relevant: At any time, only the first placeholder of every reactive is actually propagated to all its successor reactives.

### 7.2.1 Framing

The first key change of Lazy Framing is for the framing phase to terminate early. The framing traversal only recurses to successor reactives if – as before – it visited a reactive for the first time, i.e., newly created a placeholder where there was none before, *and* – newly added – no other placeholder by any earlier instants exist on the reactive. Figure 7.2 visualizes the effects of Lazy Framing, using the same running example setup as FrameSweep's original introduction did in Chapter 5. In Figure 7.2 (a), instant $t_1$ has completed its framing phase on the left-hand side of the graph normally, having created placeholders on

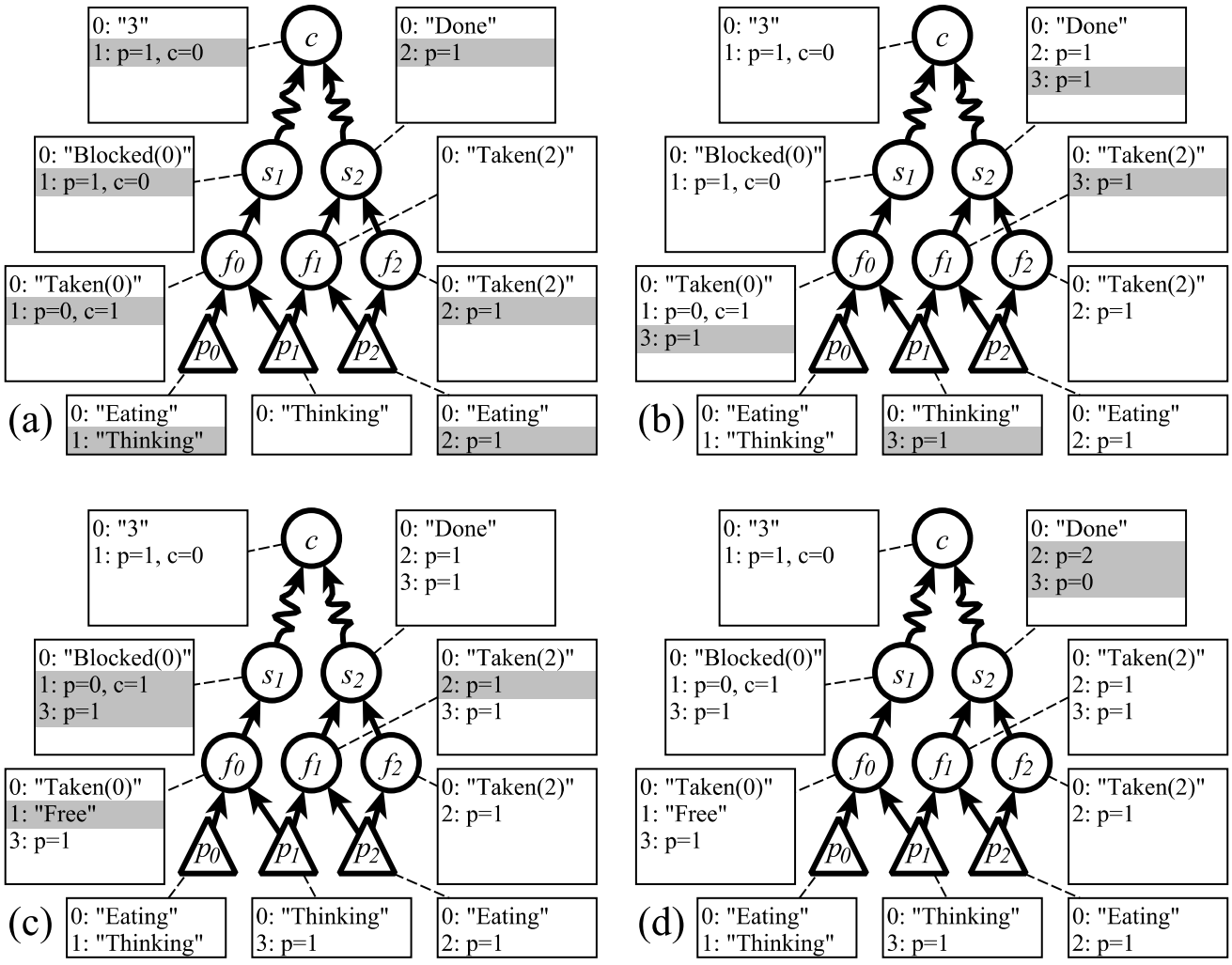



**Figure 7.2:** Lazy Framing Example Trace: Framing and Propagation

$p_0$, $f_0$, $s_1$ and all reached omitted reactives up to $c$. On the right-hand side of the graph, instant $t_2$ is still executing its' framing phase, having created placeholders on $p_2$, $f_2$ and $s_2$. Lazy Framing did not have any effect yet, because neither $t_2$ nor $t_1$ did not encounter any other instants so far. In Figure 7.2 (b), instant $t_3$ has started its framing phase, and already created placeholders on $p_1$, $f_0$, $f_1$ and $s_2$. Here, Lazy Framing does have an effect, because the placeholders that $t_3$ created on $f_1$ and $s_2$ are preceded by earlier placeholders by $t_1$ and $t_2$ respectively. Therefore, the framing traversal of $t_3$ does not recurse to the successor reactives of either of these reactives. In fact, this means that $t_3$ at this point has already completed its entire lazy framing traversal, after having created only four placeholders. Notice however, that since $t_3$ has $t_2$ as a predecessor in *SSG* and $t_2$ has not transitioned to its propagation phase yet, $t_3$ is also not allowed to transition to its propagation phase yet.

The downside of Lazy Framing is that it introduces significant additional complexity to FrameSweep in that each graph traversal requires several additional case distinctions for recursion, as the next step demonstrates. In Figure 7.2 (b), the framing phase of $t_3$ created the first placeholder on $f_1$, and therefore recursed to $s_2$ where it also created a placeholder and ordered $t_2 \xrightarrow[SSG]{*} t_3$. In Figure 7.2 (c), the continued framing traversal of $t_2$ now also created a placeholder on $f_1$. Because of the order $t_2 \xrightarrow[SSG]{*} t_3$, the new placeholder by $t_2$ is ordered earlier than previously created placeholder by $t_3$. This means, $t_2$ usurped the position of first placeholder on $f_1$ from $t_3$. Since $t_3$ then no longer has the first placeholder, the framing traversal by $t_3$ should not have recursed to successor reactives. As a result, the framing traversal of $t_2$ must recurse to the successors of $f_1$ to not only create more placeholders for itself, but to simultaneously undo the framing traversal by $t_3$. As a result, in Figure 7.2 (d) on $s_2$, $t_2$ incremented its own placeholder and simultaneously decremented the placeholder of $t_3$. Because the latter thereby became an idempotent version, the framing traversal by $t_2$ must continue to undo the framing traversal by $t_3$ on the successors of $s_2$ as well.

### 7.2.2 Propagation

During instants' propagation phase, placeholders are replaced either by written or idempotent versions through change or no-change propagation. In all of these cases, one placeholder being replaced gives way to a potentially existing next placeholder to become the first on the reactive in its stead. If that happens, the new first placeholder must be propagated to all successor reactives simultaneously together with the `notify` task that propagates the previous first placeholder's removal. This is demonstrated on the left-hand side of Figure 7.2 (c). The reevaluation of $f_0$ by $t_1$ completed and replaced the respective placeholder with a written `Free`. As a consequence, the placeholder by $t_3$ has become the new first placeholder on $f_1$. Therefore, the `notify` task by $t_1$ must not only propagate the change of $f_1$ by $t_1$, but must simultaneously progress the Lazy Framing traversal of $t_3$. As a result, $t_1$ on $s_1$ has updated its own placeholder from "p=1, c=0" to "p=0, c=1" and simultaneously has created a new placeholder for $t_3$.

This example shows, that Lazy Framing in general does not reduce the cost of framing traversals, but only shifts it onto the propagation phases of preceding instants instead. Still, Lazy Framing can result in actual reduced costs through tertiary effects. Since generally fewer placeholders exist in a given application, finding or creating new versions becomes slightly cheaper because fewer comparisons involving the *SSG* order are necessary. Further, when dynamic dependency drops occur, fewer placeholders are removed whose previous creation had no effects except establishing additional edges in the *SSG* shrinks, i.e., marginally fewer *SSG* edges are created overall. The most notable effect of fewer *SSG* edges is that instants with fewer predecessors in turn have fewer reasons to suspend before phase transitions.

Lastly, Lazy Framing can have beneficial interactions with no-change propagation, resulting in pipelined instants potentially achieving nearly depth-minimal execution. As an example, consider the progression shown in Figure 7.3 (independent of the running example), showing a section of the philosopher application of one philosophers' `filter` Event, `fold` Signal and the first following Signal in the summing-up Signal chain, here labelled `add`. In the initial step (a), instant $t_4$ just emitted `Done`

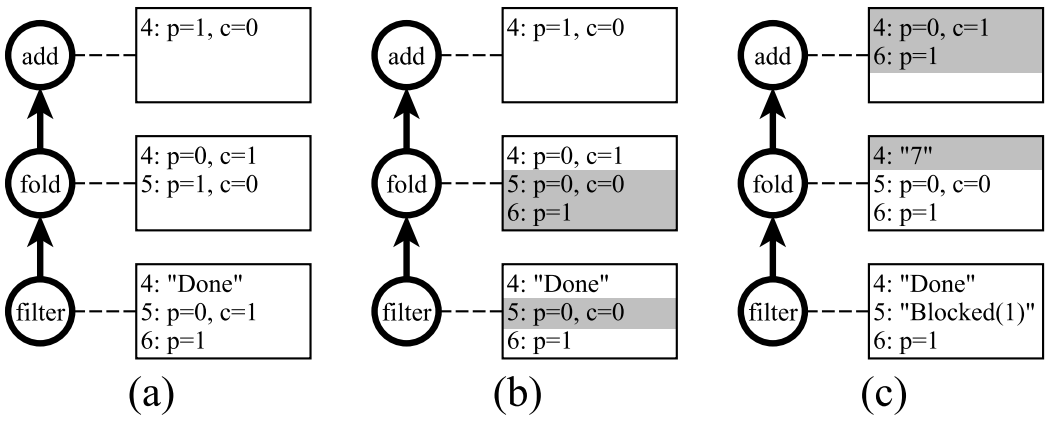



**Figure 7.3:** Depth-Minimal Execution Opportunity with Lazy Framing

from `filter`, and is now reevaluating `fold` (placeholder "4: p=0, c=1"). Instant $t_5$ is pipelined directly behind $t_4$, currently reevaluating `filter`, and having the first placeholder there. Because its placeholder is the first on `filter`, the framing phase of $t_5$ has (been) recursed to create a placeholder on `fold`, too. Its placeholder on `fold` is not the first though, meaning $t_5$ does not have a placeholder on `add` yet due to Lazy Framing.

In step (b), the reevaluation of `filter` by $t_5$ completed but did not emit a value. As a result, the corresponding placeholder was replaced with an idempotent version, and an `Unchanged notify` was submitted for `fold`. This `notify` subsequently replaced the placeholder by $t_5$ on `fold` by an idempotent version, too. Since under Lazy Framing, this previously non-first placeholder was not propagated to successor reactives yet, the no-change propagation by $t_5$ now does not recurse further to `add`, but terminates early. In step (c), $t_4$ finally completed its reevaluation of `fold` and replaced its placeholder by a written version with the incremented user value `7`. The resulting search by $t_4$ for a successor's placeholder that becomes the new first placeholder and that $t_4$ must therefore propagate alongside its `Changed notify` skipped over the idempotent version by $t_5$ and found the placeholder by $t_6$ instead. As a result, $t_4$ on `add` has updated its own placeholder and created a new one for $t_6$. Neither the framing nor the propagation traversal of $t_5$ have ever reached $t_6$ or any further successor reactives. This interaction of pipeline parallelism with lazy framing and no-change propagation therefore resulted in $t_5$ completing without having traversed the full depth of the *DG*. Note though, that it is equally possible for $t_4$ to complete its reevaluations sooner, such that the no-change propagation of $t_5$ never catches up but still propagates no-changes through the entire depth of the *DG*, continuously trailing a few reactives behind the propagation of $t_4$ in the "pipeline". Lazy Framing therefore does not achieve strictly better minimalism for executions, but only opens the possibility better-than-breadth-minimal executions under fortunate concurrency interactions.

### 7.2.3 Retrofitting

The following reasoning proves, why retrofitting under Lazy Framing does not need to traverse the *DG*. Recall the following properties of retrofitting:

- An instant $t$ can add or remove an edge $r \xrightarrow[DG]{} d$ only during its active reevaluation of $d$, i.e., $t$ has a placeholder on $d$ that persists until after retrofitting is completed. In Figure 7.2 (d), $t_1$ has a placeholder and is reevaluating $s_1$ (striped grey background).
- Retrofitting is applied for all instants $t'$ that have a placeholder or written version on $r$ that is ordered later than the version by $t$ itself (i.e., $t \xrightarrow[SSG]{*} t'$, cf. `computeRetrofit` in Listing 5.4 Line 159ff.)

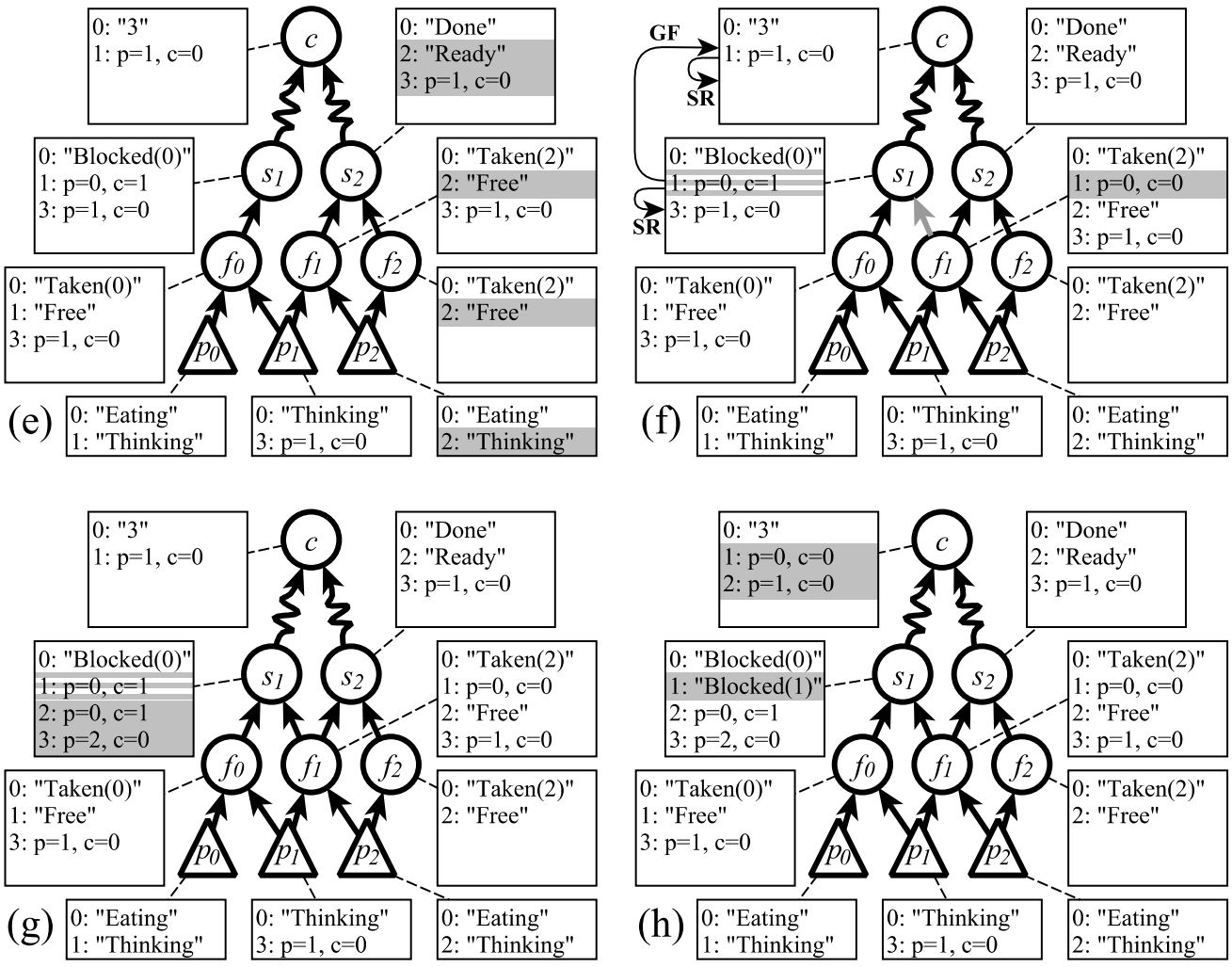



**Figure 7.4:** Lazy Framing Example Trace: Retrofitting

```
1 locally-exclusive procedure Reactive.retrofit(reactive, retrofitChange, retrofitFrame, delta):
2   # delta is +1 or -1
3   foreach (succ ∈ retrofitFrame):
4     execute ensureVersion(reevaluating, succ)
5     increment/decrement reevaluating[succ].pending += delta
6   foreach (succ ∈ retrofitChange):
7     execute ensureVersion(reevaluating, succ)
8     increment/decrement reevaluating[succ].changes += delta

9 # procedure Reactive.reframe(reactive, instant, delta) is no longer needed.
```

**Listing 7.1:** Pseudocode Implementation of Retrofitting under Lazy Framing.

In Figure 7.2 (d), $t_1$ must apply retrofitting for the written `Free` by $t_2$ and the placeholder by $t_3$ on $f_1$.

- Retrofitting undoes – or belatedly executes – the framing traversal of these $t'$ over the added or removed edge, i.e., starting from $d$ (cf. `retrofit` in Line 167ff.) and potentially affecting placeholders on all $d'$ transitively reachable from $d$ (i.e., $d \xrightarrow[DG]{*} d'$, cf. `reframe` in Line 178ff.).

Any placeholder by any such $t'$ on $d$ cannot be the first placeholder there, because the placeholder by $t$ is the first placeholder on $d$, ordered earlier than those of any $t'$. Thus, with Lazy Framing, neither did any previously superfluously executed, nor would any missing framing traversals by any such $t'$ recurse to the successors of $d$, i.e., to any $d'$. Therefore, even if placeholders by any such $t'$ exist on any such $d'$, they cannot have been affected by framing traversals over $d$. Thus, retrofitting only needs to add, remove or update any placeholders directly on $d$, but not on any transitively reachable $d'$, i.e., retrofitting with Lazy Framing does not need to execute any graph traversals.

While the many additional case distinctions for Lazy Framing make the pseudocode of all other scheduling procedures of FrameSweep too complex to be suitable for presentation here, the opposite is true for retrofitting. Listing 7.1 shows the updated implementation of `Reactive.retrofit` under Lazy Framing. For every placeholder that prompted retrofitting (Line 3ff.), `retrofit` ensures that a version exists on the `reevaluating` reactive and decrements/increments its `pending` counter. For every written version that prompted retrofitting (Line 6ff.), `retrofit` instead decrements/increments the `changed` counter. In both cases, the need to check if the version turned from an idempotent version into a placeholder or vice versa and then recurse to all successor reactives is gone. Equally, the entire implementation of the recursive `reframe` traversal (cf. Listing 5.4 Line 178ff.) is no longer needed.

To exemplify retrofitting under Lazy Framing, Figure 7.4 (e) shows the running example after $t_2$ has started, and has progressed both its own propagation traversal and the framing traversal of $t_3$ past $s_2$. This state is equal to the state in Figure 5.7 (f), from when retrofitting was first introduced in Chapter 5, except for fewer placeholders existing on $c$ because of Lazy Framing. Figure 7.4 (e) shows the ongoing reevaluation of $s_1$ by $t_1$ (placeholder with striped gray background) while executing its `depend` read on $f_1$. Still equal to Chapter 5, an idempotent version by $t_1$ on $f_1$ was already created and the reevaluation is now tasked with executing the discovery of dependency edge $f_1 \xrightarrow[DG]{} s_1$ (grey bold edge). The protective shield, visualized by "GF" and "SR", looks smaller, because the region it protects contains fewer placeholders than in Chapter 5. Nevertheless, it still prevents all the same user-visible operations from executing, they merely may be suspended on different sets of placeholders.

Figure 7.4 (g) shows the result of retrofitting with Lazy Framing. The visible changes are identical to the those shown without Lazy Framing in Chapter 5 from Figure 5.6 (f) to Figure 5.7 (g). There however, $t_1$ had to execute further retrofitting for $t_2$ on several omitted nodes, i.e., the shown differences were incomplete, whereas here this is not the case. Figure 7.4 (g) visualizes the complete set of all changes made by retrofitting, omitting none. The application of these changes to the omitted reactives happens

instead as part of the regular progression of all involved instants: While in Figure 7.4 (f), the next placeholder on $s_1$ after $t_1$ belonged to $t_3$, in Figure 7.4 (g) the new placeholder by $t_2$ has taken its place. Figure 7.4 (h) demonstrates this result. The propagation traversal of $t_1$ is complete, with its placeholder on $s_1$ replaced by a written `Blocked(1)` and later no-change propagation having replaced its placeholder on $c$ by an idempotent version. Together with its propagation traversal, $t_1$ has progressed the framing traversal of $t_2$ instead of $t_3$, creating new placeholders for $t_2$ on $c$ and all omitted reactives between $s_1$ and $c$. From here on out, $t_2$ will progress the framing phase of $t_3$ alongside its own propagation, and yet again the entire application will ultimately reach the same final state as Figure 5.7 (j), having upheld all the same semantic guarantees along the entire way.

## 7.3 Parallelizing and Decentralizing the Data Structures of FrameSweep

The simplified implementation of FrameSweep in Chapter 5 relied on extensive mutual exclusion to protect node histories and the *SSG* against concurrent modifications. Most problematic for distributed applications is its reliance on global mutual exclusion while searching or inserting new node versions into any reactives' node history and during instants' phase transitions. Global mutual exclusion requires a centralized lock, which is highly impractical in distributed applications without access to shared memory. Moreover, a central lock very quickly becomes a performance bottleneck, because every instant must acquire it many times, once for each affected reactive. Further, a centralized lock inevitably is a single point of failure for the entire application, which is bad for reliability.

Less problematic, but still potentially causing significant performance issues in distributed applications, is local mutual exclusion per reactive. Creating versions may require new edges to be added to the *SSG*, which is slow in distributed applications because the *SSG* is a global data structure, meaning this process involves slow remote communication. Due to local mutual exclusion, any operation performing such remote synchronization will block all other operations from executing on the reactive for its entire duration. This includes operations such as `depend` reads over existing *DG* edges, that would complete near instantaneously without requiring any remote synchronization because all required versions already exist.

This section discusses a more sophisticated design for the interaction of node histories and the *SSG*, that avoids such performance issues. The central abstraction around which the improved design revolves is the following interface that abstracts the functionality of the *SSG* through three operations with linearizable (atomic) semantics:

- **`existsPath(a` $\xrightarrow[SSG]{*}$ `b)`:** query, if path a $\xrightarrow[SSG]{*}$ b exists.
- **`addAcyclicEdge(a` $\xrightarrow[SSG]{}$ `b)`:** create edge a $\xrightarrow[SSG]{}$ b unless it closes a cycle, i.e., unless `existsPath(b` $\xrightarrow[SSG]{*}$ `a)`.
- **`predecessors(b)`:** retrieve the current list of predecessors a with a $\xrightarrow[SSG]{}$ b.

Section 7.3.1 discusses a non-blocking implementation of searching and creating node versions in reactives' node histories based on `existsPath` and `addAcyclicEdge`. Section 7.3.2 discusses a non-blocking implementation of instants' phase transitions based on `predecessors`. Finally, Section 7.3.3 discusses a concurrent, decentralized and distributed implementation of the *SSG* behind this interface.

### 7.3.1 Non-Blocking Version Creation

FrameSweep's improved implementation of node histories is based on sorted non-blocking linked lists, but is simpler in some and more complex in other aspects. The implementation is simpler because nearly all of the complexity in the implementation of non-blocking linked lists originates from the need

```
1  procedure Reactive.ensureVersion(reactive, instant):
2    execute tailRecursiveCASCreateVersion(reactive, instant, Reactive.head)

3  procedure Reactive.tailRecursiveCASCreateVersion(reactive, instant, reactive[current]):
4    let maybeNext := reactive[current].next
5    if (maybeNext matches Some(reactive[next]) ∧ next == instant):
6      return reactive[next]
7    else if (maybeNext == None ∨
8      (maybeNext matches Some(reactive[next]) ∧
9        (existsPath(instant -*SSG-> next) ∨
10       next.phaseLocked {
11         next.phase <= instant.phase ∧ addAcyclicEdge(instant -SSG-> next)
12       }))):
13     if (addAcyclicEdge(current -SSG-> instant)):
14       create reactive[instant] := (next := maybeNext, pending := 0, changed := 0, value := None)
15       if (atomic-compare-and-set(current.next, maybeNext, reactive[instant]):
16         return reactive[instant]
17       else:
18         tailrecurse tailRecursiveCASCreateVersion(reactive, instant, reactive[current])
19     else:
20       tailrecurse tailRecursiveCASCreateVersion(reactive, instant, Reactive.head)
21   else:
22     tailrecurse tailRecursiveCASCreateVersion(reactive, instant, reactive[next])
```

**Listing 7.2:** (Almost) Non-Blocking Version Creation.

to support deletion of intermediate elements, but node histories only ever grow, meaning support for deletion is unnecessary. A node history is simply a `head` pointer to the oldest node version that is still of potential interest to some executing instant – Section 7.4 will discuss this in more detail in the context of garbage collection. Every node version has a `next` pointer – either `Some(nextVersion)`, or `None` to mark the end of the list – and new versions can be linked into the list with linearizable semantics through a single atomic compare-and-set attempt of this pointer. At the same time, the implementation is more complex, because the *SSG* order that determines the position at which a version can be found or inserted is potentially incomplete. Establishing missing ordering relations through `addAcyclicEdge` executions is more expensive than any other aspect of inserting node versions, because it involves the need for remote synchronization to protect against concurrent executions from possibly other remote hosts. To minimize the number of `addAcyclicEdge` executions, FrameSweep decides on a position speculatively in the presence of missing ordering relations. Once decided, FrameSweep then attempts to establish all missing ordering relations to finalize this position in at most two `addAcyclicEdge` calls: one for all missing predecessor relations, if any, and one for all missing successor relations, if any. This means, inserting a node version has more possibilities to fail due to race conditions than the single compare-and-set instruction of the `next` pointer, all of which must be handled.

Listing 7.2 shows the pseudocode of an alternative non-blocking implementation of `ensureVersion` as a tail-recursive retry loop that traverses the linked list of node versions. The execution starts at the `head` node version (Line 2). On each `current` node version, it retrieves the `maybeNext` node version (Line 4). If the searching `instant` already has a version, it will show up this way, and is returned (Line 5f). Otherwise, the search verifies, if one of the following three conditions are met:

- the search has reached the end of the list (Line 7).
- the order `instant` $\xrightarrow[SSG]{*}$ `next` exists (Line 9).

- the `next` instant is in an earlier phase and the order `instant` $\xrightarrow[SSG]{*}$ `next` was successfully newly established before `next` transitioned its phase (Lines 10 to 12). Note that, since any further following versions' instants are already ordered later than the `next`, this single `addAcyclicEdge` call – if successful – establishes by transitivity, that all following versions' instants are also ordered as successors of the searching `instant`.

If none of these conditions is met, the search recurses onwards in the in the node history (Line 22). In case `next` $\xrightarrow[SSG]{*}$ `instant` is already established, the onwards recursion is necessary and correct because `instant` is ordered later in the node history. In case `next` $\xrightarrow[SSG]{*}$ `instant` is not established yet, the onwards recursion is speculation. Normally, the searching `instant` should try to establish the order `next` $\xrightarrow[SSG]{*}$ `instant`, but there may be further versions for which the same is true. In order to establish all missing predecessor ordering relations in a single `addAcyclicEdge` call, FrameSweep thus recurses further to find the latest of them.

If any of the three conditions *is* met, the searching `instant` has reached the latest position in the history where it can be ordered. Since no own version was found up to this point, none exists, meaning one must be newly inserted between the `current` and `maybeNext` versions. To do so, any previously speculated predecessor ordering relations must first be established. A single `addAcyclicEdge(current` $\xrightarrow[SSG]{}$ `instant)` call will establish all of these by transitivity (Line 13). It may however occur, that this `addAcyclicEdge` call fails, because `current` $\xrightarrow[SSG]{}$ `instant` would close a cycle. This happens when the reverse order `instant` $\xrightarrow[SSG]{}$ `current` was established concurrently, e.g., by a previously speculated predecessor ordering itself as successor of the searching `instant` on a different reactive. The meaning of this failure is that the speculated position turned out to be wrong, and the searching `instant` must be ordered earlier than speculated. To accommodate this case, the search must therefore backtrack to find the correct earlier position. For simplicity, this backtracking here is implemented as a simple restart from the `head` pointer (Line 20).

Once all ordering relations between instants have been established successfully, the search can finally attempt to insert a new version for the parameter `instant` between `current` and `next`. To do so, it instantiates a new version with the `next` pointer set to the `maybeNext` version (Line 14). Finally, the search tries to atomically compare-and-set the `next` pointer of `current` from `maybeNext` to the new version. If that fails, the `next` pointer was changed by racing operations, and therefore the search resumes from the `current` reactive and re-assesses the `next` pointer's new value (Line 14). This ensures linearizable semantics in that from any amount of racing insertion attempts between `next` and `current`, only a single one can succeed, while all others make no changes and re-assess the situation. No insertion can get lost from the pointer update being overwritten by a concurrent insertion. No two unordered instants can insert two consecutive versions without seeing each other and establishing some ordering relation beforehand. No two concurrent tasks of the same instant can simultaneously insert two versions for that same instant.

In summary, this implementation of `ensureVersion` executes all synchronization that is necessary for creating new idempotent versions without relying on any kind of local or global mutual exclusion. FrameSweep still uses local mutual exclusion between concurrent tasks on the same reactive, but only when already inserted node versions will be modified in order to avoid race conditions in relation to which node version is the first placeholder of a given node history. Specifically, local mutual exclusion is applied only to the second half of `frame`, `notify`, `reevOut` and `retrofit`, where they may convert versions between idempotent and placeholder or placeholder and written. Read operations on the other hand (`before`, `after`, `now` and `depend`) execute entirely without any mutual exclusion of other operations on the reactive, since they may require the insertion of an idempotent version, but do not change any versions afterwards. The only synchronization they stems from suspending for placeholders to be removed, but this is implemented through the `park()` and `unpark()` notification mechanism of Java's `LockSupport` API, i.e., also without mutual exclusion.

### 7.3.2 Non-Blocking Phase Transition

The simplified implementation of FrameSweep in Chapter 5 guarded instants' phase transitions with global mutual exclusion to prevent race conditions with two other scheduling aspects. First, `ensureVersion` collected all instants of a smaller phase, and then added an edge to the *SSG* for the earliest one. For this process to be safe, all of these instants had to be prevented from transitioning their phase, which was assured through global mutual exclusion. In the new implementation of `ensureVersion`, the earliest of these instants is always encountered before all the others. Therefore, it is feasible to directly prevent only the earliest instant from transitioning its phase, meaning an instant-specific lock suffices. This is what the `phase-locked` scope in Listing 7.2 Line 10 expresses. It is no longer necessary to prevent all instants in a smaller phase from transitioning while the earliest has not been determined yet, and thus this aspect no longer requires global mutual exclusion.

Second, an instants phase transition may occur only once the following two conditions are met at the same time: The instant's `activeTasks` counter must be zero, i.e., its graph traversal of the current phase must be complete, and all its predecessor instants must already have completed the same phase transition. Since the second condition again involves multiple instants, the simplified implementation of FrameSweep in Chapter 5 also protected it through global mutual exclusion. These conditions remain unchanged, i.e., they still involve multiple instants, and therefore an implementation without global mutual exclusion requires further engineering.

Observe first, that it is insufficient to check the two conditions individually in any order, or even repeatedly alternating, as the following example will show. Consider a propagating instant `t` trying to transition to completion. It has no active tasks, but it has a predecessor instant `p` that is still propagating. First, `t` verifies that its `activeTasks` counter is zero. As part of completing a reevaluation, `p` afterwards submits a subsequent reevaluation of the same reactive for `t`, and then transitions to completed. At that point, `t` verifies its predecessors, and sees that all have completed, but its `activeTasks` counter is no longer at zero due to the newly submitted reevaluation. Next, the restarted propagation traversal of `t` establishes a new predecessor relation to a propagating instant `p'`, and afterwards completes its last task. Now `t` can again verify that its `activeTasks` counter is at zero, but also again has a new propagating predecessor. Therefore, this process can repeat an arbitrary number of times, showing that even observing both conditions to be true in alternating order any number of times does not mean that a phase transition is allowed yet.

Notice though, that this loop can only continue if set of predecessors keeps growing larger. This is the key insight that FrameSweep exploits to implement its phase transition precondition check without any mutual exclusion through the following three step loop. First, retrieve the current set of predecessor instants, and suspend until all of them have completed the planned phase transition. Second, suspend until the `activeTasks` counter has reached zero. Third, retrieve the current set of predecessor instants again, and compare it to the one retrieved in the first step. If the set has changed, repeat from the first step with the new set. If the set is unchanged though, then both conditions of the precondition were met at the same time in the second step, and therefore the instant can execute its phase transition.

### 7.3.3 Decentralized Serialization Graph Tracking

FrameSweep's implementation of the *SSG* adapts a partially dynamic graph algorithm for maintaining the transitive closure of a directed graph under only edge insertions [Italiano, 1986] to its distributed and concurrent environment. In its original form, this algorithm is suitable for use neither under distribution, nor under concurrency. To remedy this, the next section first describes, how FrameSweep adapts the algorithm for use in the distributed setting. The remaining sections then introduce another algorithm for determining connectedness in the form of undirected transitive reachability, also adapt it to FrameSweep's distributed setting, and finally use it to add thread-safety for concurrent modifications to the aforementioned algorithm for directed transitive reachability.

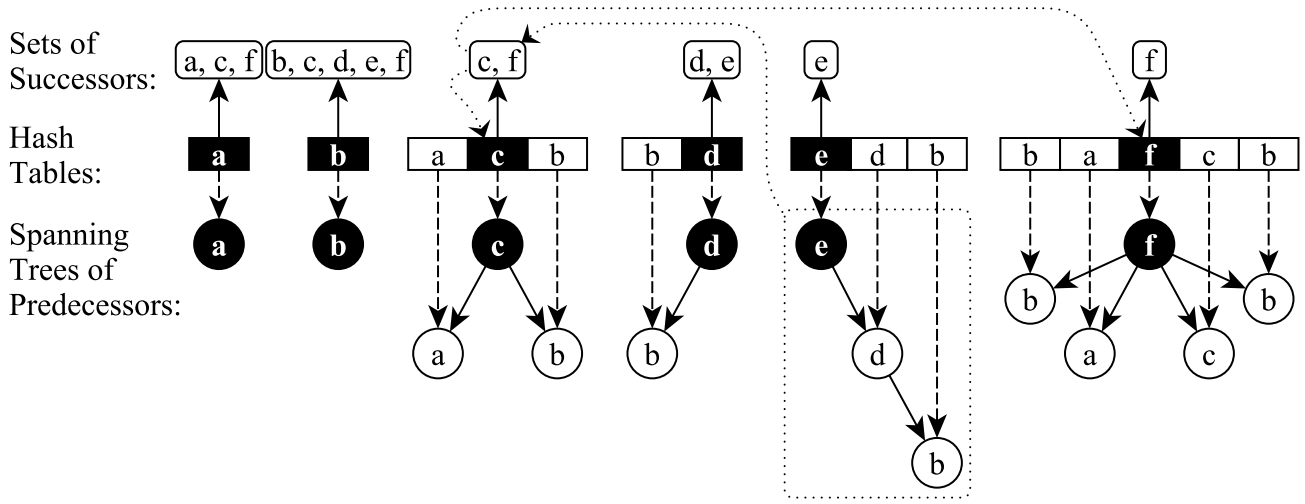



**Figure 7.5:** Transitive Reachability through Hash-Indexed Spanning Trees

### Distributed Transitive Reachability

Each instant `b` stores a spanning tree of all predecessor instants, i.e., all `a` with $a \xrightarrow[SSG]{*} b$. All nodes in the spanning tree of each `b` are indexed in a thread-safe non-blocking hash table, with the instant that they represent as their key. Lastly, each `b` has a set of all successor instants, i.e., all `c` with $b \xrightarrow[SSG]{*} c$.

Figure 7.5 shows an exemplary state of these internal data structures for the *SSG* visualized with black edges in Figure 7.6. The bottom row visualizes the spanning trees of all instants. Every instant is the root of its own spanning tree, the corresponding nodes are highlighted black. Note that because the spanning trees contain predecessors rather than successors from the *SSG*, spanning tree children edges and *SSG* edges point in opposite directions. The middle row visualizes the hash table of the spanning tree nodes. Each node of an instant's spanning tree is indexed through an entry in the instant's hash table that references the node. Each instant's entry in its own hash table is again highlighted black and points to its spanning tree's root. The top row visualizes the successor sets as rounded rectangles. E.g., instant `a` has a node in its own spanning tree and an entry in its own hash table, both highlighted black. It also has a node in the spanning tree and an entry in the hash table of both instants `c` and `f`. Correspondingly, the successor set of `a` contains `a`, `c` and `f`.

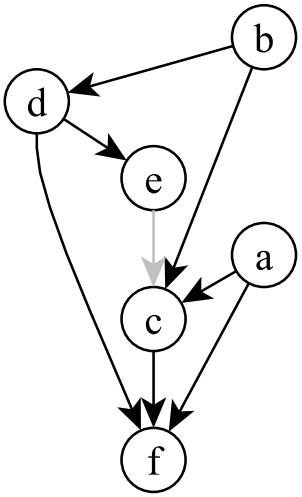



**Figure 7.6:** Example *SSG*

For distribution, the hash-indexed predecessor spanning trees are replicated to remote hosts. E.g., if instant `c` from Figure 7.5 originated on host `A`, but also affected some reactive on host `B`, then host `B` stores a copy of its hash-indexed spanning tree. Further, because the hash-indexed spanning tree of `c` contains instants `a` and `b`, host `B` also has copies of their hash-indexed spanning trees. When additional instants are added to the hash-indexed spanning tree of `c`, this addition is replicated on host `B` too, in order to keep the replica up to date. Removal of instants from these replica is discussed in Section 7.4 in the context of garbage collection.

Having replicas of instants' hash-indexed spanning trees available on each host enables FrameSweep to provide non-blocking constant time implementations for the idempotent operations `existsPath(a` $\xrightarrow[SSG]{*}$ `b)` and `predecessors(b)`. For `existsPath(a` $\xrightarrow[SSG]{*}$ `b)`, FrameSweep simply looks up, whether or

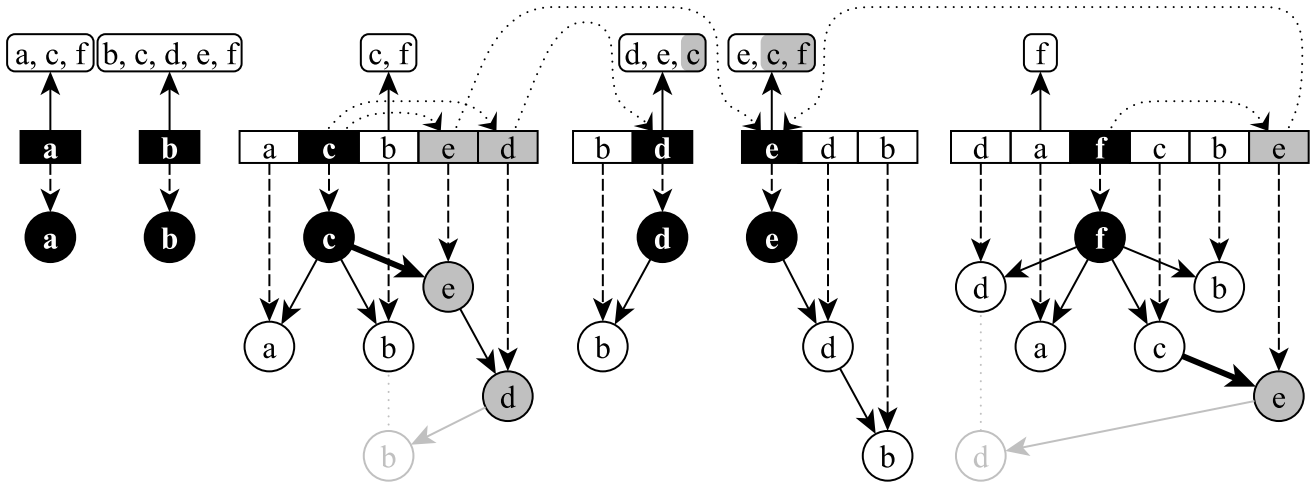



**Figure 7.7:** Pruning and Copying Spanning Subtrees

not the local replica of the hash table of instant `b` associates a spanning tree node with instant `a`. For `predecessors(b)`, FrameSweep simply returns the current contents of the hash table of `b`.

Operation `addAcyclicEdge(a` $\xrightarrow[SSG]{}$ `b)` is significantly more complex, because it is not idempotent and requires remote synchronization to achieve linearizable semantics. The following paragraph concludes the description of the transitive reachability algorithm by discussing the operation `addEdge(a` $\xrightarrow[SSG]{}$ `b)` for updating hash-indexed spanning trees when adding a new edge to the *SSG* as introduced by the original algorithm, i.e., not thread-safe and without avoid cycles. The next subsection then introduces the "Lock-Union-Find" data structure, a partially dynamic graph algorithm for mutual exclusion over connected components (reachability in *un*-directed graphs) under only edge insertions. Finally, the last subsection describes, how `addAcyclicEdge(a` $\xrightarrow[SSG]{}$ `b)` can be implemented correctly by tying together `addEdge(a` $\xrightarrow[SSG]{}$ `b)` and `existsPath(a` $\xrightarrow[SSG]{*}$ `b)` under mutual exclusion provided by "Lock-Union-Find".

As an example for edge insertion, consider `addEdge(e` $\xrightarrow[SSG]{}$ `c)` to the *SSG* from Figure 7.6, which visualizes this edge as a gray arrow. First, a reference to the spanning tree of `e` is sent to all successors of `c`, together with the instruction to attach integrate the tree below `c`. In case these successors reside on a different host, a copy of the spanning tree is sent instead of a reference. In Figure 7.5, the spanning tree of `e` highlighted by a rectangular frame with dotted outline, and dotted arrows visualize the paths along which it is sent to `c` and `f`.

Each instant that receives a spanning tree pointer and an instant below which to attach this tree first retrieves the node in its own spanning tree for that instant. It then recursively attaches each node below the corresponding node in its own spanning tree, starting with the one just looked up. Figure 7.7 shows the result of this process. Instant `c` newly added `e` below its spanning tree node for itself (i.e., the root), then `d` below `e` (spanning tree nodes and hash table entries with grey background). Next, it found that instant `b` was already reachable (the grey dotted line visualizes the redundancy) and therefore ignored it (i.e., the spanning tree node with grey outline was not added). Instant `f` also newly added `e` below its spanning tree node for `c` (grey background), but then found that `d` was already reachable (dotted grey line visualizing the redundancy), and therefore ignored `d` (grey outline) and skipped assessing any of its children, i.e., `b` (not shown), because they are already reachable by transitivity. Drawing this transitivity conclusion is why the algorithm is built on spanning trees, and what enables `addEdge` to execute in amortized linear time [Italiano, 1986]. Finally, each instant appends itself to the set of successors of each newly reachable instant. In Figure 7.7, the dotted arrows show how `f` appended itself to the successor set of the newly reachable `e`, and how `c` did the same for both `d` and `e` (successor set entries with grey background).

```
1  procedure lock(node):
2    let found := find(node)
3    if (atomic-compare-and-set(found.ptr, None, Some(found))):
4      return Some(found)
5    else:
6      return None

7  procedure lockUnion(a, b):
8    let maybeLocked := lock(a)
9    if (maybeLocked matches Some(locked)):
10     let found := find(b)
11     if (found == locked ∨
12             atomic-compare-and-set(found.ptr, None, Some(locked))):
13       return Some(locked)
14     else:
15       execute unlock(locked)
16       return None
17   else:
18     return None

19 procedure unlock(locked):
20   update locked.ptr := None
```

**Listing 7.3:** Pseudocode of Lock-Union-Find.

## Lock-Union-Find Fundamentals

A union-find data structure [Galler and Fisher, 1964] is a partially dynamic graph algorithm that associates a single representative element with each connected component of an undirected graph under only edge insertions. Each node maintains a pointer that is either `None`, indicating that the node itself is the representative of its connected component, or `Some(other)`, in which case the representative for its connected component can be found by following these pointers recursively – this is the `find(node)` operation. If, for two nodes `a` and `b`, `find(a)` and `find(b)` return the same representative, then `a` and `b` are part of the same connected component, i.e., there is some transitive path `a` $\overset{*}{\underset{SSG}{}}$ `b` in the graph. Union-find therefore relates to the hash-indexed spanning trees from the previous section in that it also computes transitive reachability between nodes, but for undirected instead of directed graphs. Inserting a new edge `a` $\underset{SSG}{}$ `b` may unify two previously separate connected components into one single connected component. If `a` and `b` are not part of the same connected component, i.e., have different representatives, then their connected components can be unified into a single common connected component by setting one representative's pointer to the other representative – this is the `union(a` $\underset{SSG}{}$ `b)` operation.

For the purposes of synchronizing `addAcyclicEdge(a` $\underset{SSG}{\longrightarrow}$ `b)` operations, the lock-union-find data structure extends union-find twofold. First, lock-union-find adds a third pointer state: A node whose pointer points to the node itself implies that the node is the representative of its connected component, but the connected component is currently locked. Second, lock-union-find adds operations `lock(node)` and `unlock(locked)`, and replaces `union(a` $\underset{SSG}{}$ `b)` with `lockUnion(a` $\underset{SSG}{}$ `b)`. Listing 7.3 shows simplified pseudocode implementations for these operations. In practice, both implementations are significantly more complex since they perform additional tasks that are not relevant here, e.g., non-blocking path compression for execution in nearly constant time, and non-blocking manual reference counting for distributed garbage collection, cf. Section 7.4.

The `procedure lock(node)`, defined in Line 1ff., tries to lock the connected component that `node` belongs to. It first executes `find(node)` (Line 2) and then tries to lock the returned representative by updating its pointer from `None` to itself through an atomic compare-and-set instruction (Line 3). If

this fails, either because the representative is locked by a different party or because a racing `union` operation concurrently changed its pointer to a different representative, then `lock(node)` returns `None` to indicate an unsuccessful attempt (Line 5f). If it succeeds, the connected component was locked successfully, and `lock` returns the reference to the locked representative of the connected component as `Some(locked)` (Line 4). While locked, the connected component can neither be locked nor unified with other components by other parties until `unlock(locked)` is executed.

The `procedure lockUnion(a` $\overline{SSG}$ `b))`, defined in Line 7ff., tries to establish that both `a` and `b` are part of a single locked connected component. It first tries to lock the representative of the connected component of `a` (Line 8). If that fails, `lockUnion` also fails and returns `None` (Line 17). Otherwise, it retrieves the representative of the connected component of `b` (Line 10). If the representative of `b` is the same as the locked representative of `a` (Line 11), then `a` and `b` are already part of the same connected component that was locked by `lock(a)`. Alternatively, if an atomic compare-and-set attempt of the pointer of the representative of `b` from `None` to the locked representative of `a` succeeds (Line 12), then `a` and `b` were newly unified into a single connected component whose lock is still held from the initial `lock(a)`. In both of these cases, `lockUnion` succeeded and returns – equally to lock – a reference to the locked representative of the unified connected component.

The last compare-and-set attempt can also fail though, again either because the representative of `b` is locked by a different party or because a racing `union` operation concurrently changed its pointer to a different representative. In this case (Line 14), `lockUnion` discards the progress it made up to that point, by unlocking the connected component of `a`, and then also fails, returning `None`. Discarding this progress instead of retrying the unification of both connected components is necessary to avoid the following deadlock. Consider `lockUnion(a` $\overline{SSG}$ `b)` and `lockUnion(b` $\overline{SSG}$ `a)` to execute concurrently. The former successfully locks the connected component of `a`, the latter that of `b`. At that point, neither execution can succeed in unifying the connected components of `a` and `b`, because they both simultaneously and indefinitely block each other's compare-and-set attempts from succeeding. To avoid this situation, both must repeatedly unlock their locked connected components, so that the respective other has a chance to succeed.

### Distributed Lock-Union-Find

To make lock-union-find usable in a distributed and concurrent environment, the only necessary modification is to allow the use of remote references in nodes' representative pointers. No further synchronization is required to deal with any concurrency issues, because the code as presented already implements linearizable semantics:

- For `lock(node)`, this is because all its functionality is implemented by the single atomic compare-and-set instruction. If a different thread concurrently locks a connected components representative, or unifies it under a different representative, then the compare-and-set attempt will fail, and therefore `lock` will fail, without having caused any changes.

- For `unlock(locked)`, this is because it is not subject to any race conditions. Only a single thread can lock a given connected component, and only that single thread is allowed to execute `unlock` for its representative. Both `lock` and `lockUnion` cannot make any changes to the pointer while it is locked, because their atomic compare-and-set operations can only succeed while the pointer is `Null`.

- For `lockUnion(a` $\overline{SSG}$ `b)`, linearizable semantics become apparent when realizing that the unification of two separate connected components essentially requires *both* to be locked beforehand. For the parameter `a` the connected component is clearly locked by the explicit `lock(a)` call. For the parameter `b` on the other hand, this is not the case, since its representative is compare-and-set directly from unlocked to the other representative. This is semantically equivalent to updating it in

```
1  procedure addAcyclicEdge(a ⟶_SSG b):
2    let maybeLocked := lockUnion(a ⟷_SSG b)
3    if (maybeLocked matches Some(locked)):
4      try {
5        if (existsPath(a ⟶*_SSG b)):
6          return true
7        else if (existsPath(b ⟶*_SSG a)):
8          return false
9        else:
10         let changes := addEdge(a ⟶_SSG b)
11         return true
12     } finally { execute unlock(locked) }
13   else if (existsPath(a ⟶*_SSG b)):
14     return true
15   else if (existsPath(b ⟶*_SSG a)):
16     return false
17   else:
18     tailrecurse addAcyclicEdge(a ⟶_SSG b)
```

**Listing 7.4:** Pseudocode of "Add Acyclic Edge".

two steps the following way though: Instead of compare-and-set from unlocked to the other representative, compare-and-set it from unlocked to locked. This succeeds and fails in the exact same cases as the direct compare-and-set would. If it succeeds, both connected components are locked and no other thread can change either representatives' pointer. Then, the executing thread can safely update the pointer from locked to the other representative, achieving the same result as if the compare-and-set had updated it to that value directly. This shows that for any given connected component, only a single thread can successfully execute `lockUnion` with any other instant at any given point in time, while all other concurrent invocations may at most lock and then again unlock some components, leaving no lasting changes.´

In conclusion, lock-union-find therefore is a partially dynamic graph algorithm that implements mutual exclusion over connected component of distributed graphs under only edge additions in a thread-safe and fully decentralized fashion.

### Thread-Safe Reachability Changes through Lock-Union-Find

Concluding the implementation of decentralized Serialization Graph Tracking, this section presents the distributed, linearizable and fully decentralized implementation of `addAcyclicEdge(a` $\longrightarrow_{SSG}$ `b)`. It ties together `existsPath(a` $\overset{*}{\longrightarrow}_{SSG}$ `b)` and `addEdge(a` $\longrightarrow_{SSG}$ `b)` from the directed transitive reachability of instants under the mutual exclusion over connected components, i.e., *un*directed reachability, of lock-union-find. The key concept behind the implementation is, that any thread may only establish a *directed* edge `a` $\longrightarrow_{SSG}$ `b` in the data structures for transitive reachability, if it holds the lock for the connected component in which the two instants have unified through establishing the *undirected* edge `a` $\overline{SSG}$ `b`. This means, after a `lockUnion(a` $\overline{SSG}$ `b)` call succeeded, the both the directed and undirected transitive reachability of both instants `a` and `b` cannot be modified by concurrent threads. As a consequence, the result of any `existsPath` query involving either `a` or `b` becomes reliable, and `addEdge(a` $\longrightarrow_{SSG}$ `b)` can

execute without any danger of race conditions from other concurrent `addAcyclicEdge` calls. This way, `addAcyclicEdge(a` $\xrightarrow[SSG]{}$ `b)` achieves linearizable semantics overall.

Listing 7.4 shows the corresponding pseudocode. An execution of `addAcyclicEdge(a` $\xrightarrow[SSG]{}$ `b)` first attempts to `lockUnion(a` $\overline{\xrightarrow[SSG]{}}$ `b)` (Line 2). If `lockUnion` succeeds (Line 3ff), `addAcyclicEdge` verifies that the insertion of edge `a` $\xrightarrow[SSG]{}$ `b` is neither redundant (path `a` $\xrightarrow[SSG]{*}$ `b` already exists, Line 5) nor illegal due to closing a cycle (reverse path `b` $\xrightarrow[SSG]{*}$ `a` already exists, Line 7). Then, `addAcyclicEdge` executes `addEdge(a` $\xrightarrow[SSG]{*}$ `b)` (Line 10) and afterwards unlocks the connected component again (Line 12). If `lockUnion` fails, this means that other threads are concurrently executing `addAcyclicEdge` operations within the same connected component. These may affect the reachability of `a` and/or `b` directly or even transitively. In that case, `addAcyclicEdge` therefore verifies optimistically, i.e., through non-blocking `existsPath` executions without locking anything, if the planned insertion of edge `a` $\xrightarrow[SSG]{}$ `b` has become either redundant (Line 13) or illegal (Line 15). Only if still neither of is the case, `addAcyclicEdge` loops back to retrying `lockUnion` again.

In summary, once `addAcyclicEdge(a` $\xrightarrow[SSG]{}$ `b)` returns, `a` and `b` are guaranteed to be ordered against each other. They may be ordered in either way, and the return value indicates, which way is the case. If the call returns `true`, then it either established `a` $\xrightarrow[SSG]{*}$ `b` itself, or some other thread did so concurrently. If it returns `false`, then some other thread concurrently established the reverse order `b` $\xrightarrow[SSG]{*}$ `a` instead.

## 7.4 Garbage Collection

Three aspects of FrameSweep require manual maintenance to facilitate garbage collection and prevent memory leaks. Instants become obsolete, but are still referenced in the *SSG* data structures for directed reachability (hash-indexed spanning trees) and undirected reachability (lock-union-find representative pointers). Node versions also become obsolete, but are still referenced by node histories. This section addresses each of these aspects.

Instants become obsolete once they transition to completed. FrameSweep enables garbage collection of completed instants from the *SSG* through two aspects. First, when an instant transitions to completed, all replica of its hash-indexed spanning tree are unlinked. Second, the implementation of `existsEdge(a` $\xrightarrow[SSG]{}$ `b)` returns true by default if instant `a` is completed. This means, any framing or propagating instants will always see completed instants as predecessors, without having to actually store them as predecessors. Therefore, no new references to completed instants will be established. Once all instants have completed, that added a now completed instant as a predecessor in their hash-index spanning tree when it was still executing, the instant has become entirely unlinked from the *SSG*.

For lock-union-find representative pointers, automatic garbage collection is prevented because each host references them statically as a substitute for all remote references. Enabling them to be garbage collected therefore requires to unlink these static references once no remote references exist anymore. To determine the absence of remote references reliably, FrameSweep uses reference counting. Reference counting usually requires complex algorithms to detect unreachable cyclic object graphs. This is unnecessary for FrameSweep though, because lock-union-find representative pointers form a tree structure that cannot have cycles. FrameSweep therefore implements this reference counting simply by counting for each instant the number of other instants whose representative pointer point to that instant. Once this counter drops to zero, the its static reference is dropped, and then the local garbage collector can eventually deallocate it.

Node versions generally must be retained even after their associated instant completed, because later instants may still read their value, e.g., through `before`. A version can only be deleted safely, once a written successor version exists that also belongs to a completed instant. FrameSweep implements this

garbage collection of node versions as a by-product of traversing node histories. Any traversal keeps track of the head node version where it started. If the traversal finds another written version by a completed instant, it attempts to compare-and-set the head pointer to this later version. If that succeeds, all node version before the new head pointer are no longer referenced and can be garbage collected. Failures on the other hand are simply ignored, since they indicate that a concurrent operation already progressed the head pointer.

## 7.5 Syntactic Transparency and Explicit Transactions

The implementation of FrameSweep presented in Chapter 5 is syntactically transparent in that all scheduling is implemented inside of the unchanged established API of reactives as introduced in Section 2. As a result, existing REScala code not written with multi-threading in mind, e.g., the code of the philosopher example shown in Section 2, can run without any changes on top of FrameSweep ensuring its correct execution in a multi-threaded environment. Notably, this includes the instantiation of new reactives. When code such as that from Section 2 is executed, it imperatively instructs new reactives to be created one by one. FrameSweep executes one instant for each reactive instantiation, where the reactive is first instantiated and then connected to its dependencies by dynamically inserting all required edges during its initial evaluation. This is possible at any place in the *DG* at any time, i.e., even while concurrent instants are propagating, without such changes having been foreseen or planned during development of the applications. Any number of edges can be added or removed as part of a single instant, which yields consistent semantics for atomically (dis-)connecting new or obsolete nodes or subgraphs. Retrofitting automatically (de-)schedules superfluous or missing reevaluations of newly (dis-)connected nodes as needed.

In some cases though, applications require more extensive synchronization than linearizability for individual imperative interactions. The philosphers' driving thread from Listing 2.8 is one example of this. Due to FrameSweep's syntactic transparency, this code can be reused without modifications on top of FrameSweep, and all imperative `set` instants and `now` reads will appear to execute with linearizable semantics. In order for the philosopher application to behave correctly though, the `sights(idx).now == Ready` condition from Line 3 and the subsequent `phils(idx).set(Eating)` instant from Line 4 must execute as a single linearizable operation together, instead of as two separate ones. In fact, the example execution trace of FrameSweep visualized in Figure 5.4 through Figure 5.7 of Chapter 5 is only possible if this condition and update are executed as two separate linearizable operations, because instant $t_3$ of `phils(1).set(Eating)` is started in Figure 5.6 (e) while `sights(1).now == Ready` is no longer true. The `sights(1).now == Ready` condition must have been evaluated much earlier and independently of the subsequent `phils(1).set(Eating)`, with both `phils(1)` and `phils(2)` having executed their `Eating` updates between the two interactions.

To enable applications to declare multiple imperative interactions as a single atomic unit of work, FrameSweep supports the optional feature of *extended transactions*. The first form of extended transactions supported by FrameSweep are *multiple read* transactions. These bundle together any number of `s.now` operations. As an example, consider the following code, also based on the philosophers example:

```
2  if (transaction{ forks(0).now == Free && forks(1).now == Free })
3    phils(1).set(Eating)
```

**Listing 7.5:** Multiple Read Transaction.

Because the explicit `transaction` scope ensures that both `.now` reads are executed together as a single atomic unit, this condition behaves identically to the single `sights(1).now == Ready` read from the original Listing 2.8, Line 3.

The second form of extended transactions supported by FrameSweep are *read-update-read* transactions. These bundle together a single `update(...)` instant with any number of proceeding and succeeding `s.now` reads. Read-update-read transactions allow programs to express instants with atomic

pre- and post-conditions. They come with one complication related to the fact that the scheduler must know the `declaredInputs` set of possibly changed input reactives before starting to execute the transaction. For an isolated `update(`$\texttt{i}_1$ `->` $\texttt{v}_1$`,` $\texttt{i}_2$ `->` $\texttt{v}_2$`, ...)` call, FrameSweep automatically infers $declaredInputs = \{\texttt{i}_1, \texttt{i}_2, ...\}$ from the call's parameters to retain syntactic transparency in Listing 5.1 Line 28. For a call to `transaction{...}` with an arbitrary user-defined closure as the only parameter though, this is not possible. Instead, FrameSweep requires read-update-read transactions to provide their `declaredInputs` explicitly, as a second parameter to the `transaction` scope. This is slightly inconvenient in that it does introduce some syntactic clutter, but FrameSweep still keeps this to a minimum in that only potentially changed input reactives must be declared, while all other potentially reevaluated derived reactives are still inferred automatically. With read-update-read transactions, the loop iteration from Listing 2.8 can be implemented correctly by grouping together the `sights(idx) == Ready` check and the subsequent `phils(idx).set(Eating)` instant:

```
2  transaction(Set(phils(idx))) {
3    if (sights(idx).now == Ready)
4          phils(idx).set(Eating)
5  }
```

**Listing 7.6:** Thread-Safe Eating Through Read-Update-Read Transactions.

Note that for the case where `sights(idx).now` does not return `Ready`, `phils(1)` is included in the `declaredInputs` for the framing phase, but then not assigned a new value for the propagation phase. This is the reason for the propagation phase to support a discrepancy where some of the `declaredInputs` do not have a new value associated in the `inputChanges` map, handled in Listing 5.2 Line 69.

# 8 Evaluation

The evaluation measures the performance costs of FrameSweep's different features, and puts them in relation against each other and against other scheduling algorithms as a frame of reference. In general, it consists of several comparisons experiments. Each experiment uses a specific set of instants on a specific *DG* topology, emphasizing different features of RP propagation over others. Each experiment then runs its specific workload with FrameSweep and – as far as available – other propagation algorithms, measuring the time each algorithm takes to complete said workload. The evaluation then draws conclusions by comparing the results within each experiment, as well as across multiple experiments.

In distributed applications, remote communication is orders of magnitude slower than local computations. Therefore, performance measurements over distributed *DG* topologies will always be dominated by the longest path of sequential remote communication required to complete a given workload, overshadowing any differences in CPU processing cost. The evaluation is thus is split into two different classes of experiments for measuring two different cost associated with the different features of FrameSweep. Section 8.1 evaluates the performance of a variant of FrameSweep optimized for non-distributed topologies, quantifying associated CPU processing cost and showing that FrameSweep, despite its comparatively complex implementation, provides better performance and scalability. Section 8.2 evaluates FrameSweep's performance on distributed topologies, quantifying the associated amount of sequential remote communication. Lastly, Section 8.3 quantifies, how much effort is required to parallelize the *existing*, non-distributed "Universe" RP application from REScala's example corpus, and the effect this has on the application's performance.

Another desirable evaluation approach involves measuring the effect of using FrameSweep in real complex applications beyond the Universe example, particularly in distributed applications. Even evaluating the effects of FrameSweep on multiple complex applications though will not yield any guarantees that the results generalize to other/future applications. Yet more critically, since FrameSweep is the first RP implementation to enable RP with synchronous propagation semantics in distributed applications, no applications exist yet that use such a library. Build such an application (or even multiple ones) for the sole purpose of this evaluation would unavoidably introduce significant bias. Therefore, the evaluation does not investigate this direction, but follows only the above-mentioned microbenchmarks approach to outline a set of rules that does allow users to generally estimate the impact that given features of any applications will have on their performance.

## 8.1 Comparing Local Performance

Using non-distributed *DG* topologies, we evaluate the effect of the following factors on the performance of FrameSweep: relation between cost of user computations and synchronization overhead, level of thread contention, *DG* topology, and cost of dynamic edge changes and handling of their conflicts (retrofitting). In Section 8.1.1, we evaluate, which topologies FrameSweep can parallelize, and how costly user computations must be for this to be efficient. The experiments in Section 8.1.2 investigate scalability as a function of overhead, which itself is a function of contention, in benchmarks with very cheap user computations, i.e., where scheduling overhead dominates execution costs. Section 8.1.3 analyzes the performance costs related to dynamic edge changes.

The non-distributed evaluation compares the following scheduling approaches:

- A variant of FrameSweep optimized for the local setting, differing from the distributed implementation in two aspects: Most significantly, the thread pool was removed, because it causes a lot of

context switching between different threads for executing even an individual instant, which has significant negative performance impacts in the absence of network communication. Instead, the submitted tasks of each instant are executed inside of the respective `update(...)` call by the calling imperative thread itself. Secondly, the decentralized *SSG* implementation uses a global lock instead of lock-union-find for mutual exclusion. This was done because the way that distributed FrameSweep uses lock-union-find for mutual exclusion is not well-optimized yet (discussed in more detail in Section 8.2), and quickly became a performance bottleneck in high contention scenarios.

- G-Lock implements global locking by wrapping the execution of each instant with a global `synchronized{..}` scope. This prohibits concurrent executions, but thereby trivially fulfils our correctness for all applications.

- Handcrafted refers to application-tailored manual locking implementations, which we include whenever feasible. In Handcrafted implementations, no synchronization is integrated into the RP runtime. Instead, the imperative user code is changed to manually acquire and release locks before and after executing any `now` or `update(...)`. Handcrafted solutions have little overhead, but often require significant effort to be devised individually for each application, and are fragile in that they do not generalize to other applications and easily break when parts of the application are changed. Note that more elaborate Handcrafted solutions become possible when also acquiring and releasing locks inside of reactives' user computations (this would, e.g., enable some cases of pipeline parallelism). But, such solutions are much more complex to explain, and extremify above-mentioned negative aspects so much that we consider them unreasonable and hence do not include them in the evaluation.

- STM-RP is an integrated RP scheduler that stores all reactives' variables in ScalaSTM [Bronson et al., 2010], a library-based off-the-shelf software-transactional memory. Each instant's execution is wrapped in an `atomic{...}` scope and executed optimistically, aborting and restarting upon concurrency conflicts. STM-RP thus does not fulfil our correctness: it provides strict serializability, but is not abort-free. It is therefore applicable only to applications without side-effects; for compatibility, all our benchmarks adhere to this.

All non-distributed experiments measure throughput, i.e., number of instants completed over a certain amount of time (more is better), for an increasing number of threads that concurrently execute instants. All experiments run on computers with dual Intel Xeon E5-2670 CPUs for 2x8 cores at 2.6 - 3.3 GHz, using the OpenJDK JMH benchmark harness in the 64-Bit Oracle JRE 8u141 at 1 GB fixed heap under CentOS Linux 7.4. In general, shown results are average measurements of executing each benchmark operation in every configuration for at least 35 seconds (after at least 25 seconds unmeasured warmup), repeated six times on fresh JVM instances each. Every single data point in each of the following chart thus represents at least 3.5 minutes of execution; a data point of, e.g., 50 ops/ms is the average measurement based on the benchmark operation having executed over 10.5 million times.

### 8.1.1 Cost of User Computations and Impact of Topology

Our first set of experiments serves a twofold purpose. First, they analyze, how well FrameSweep can parallelize concurrent instants on different topologies. Second, we analyze, how costly user computations have to be so that scheduling overhead doesn't reduce throughput more than scaling increases it. For the first goal, the experiments use a set of minimalized fundamental topology building blocks. We analyze scalability by measuring throughput across 1 to 16 threads concurrently executing instants on each topology. We sized each of these topologies to contain precisely 16 designated work reactives, so that maximum scalability can be achieved, but only if the framework actually manages to parallelize each topology down to single reactives. For the second goal, through trials on each topology, we pinpointed

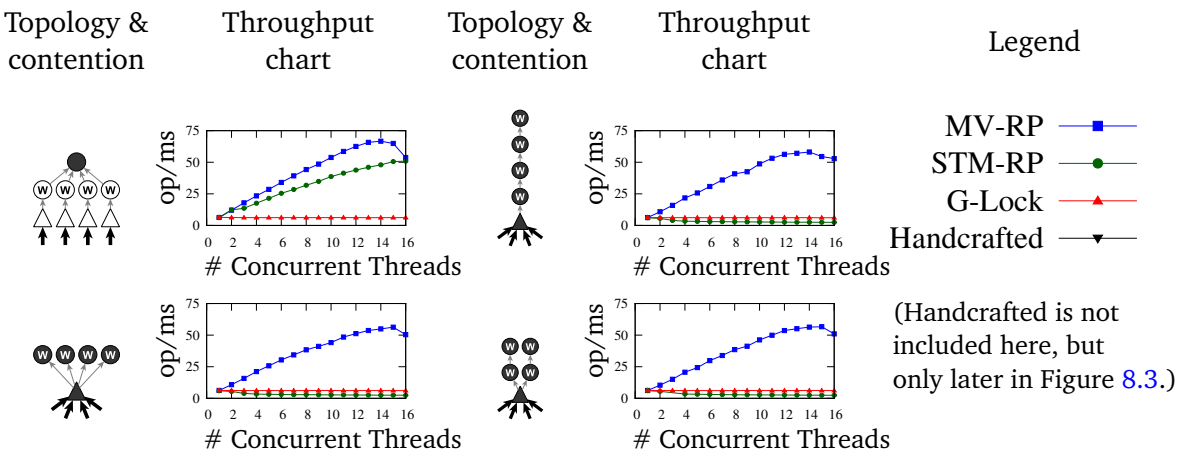



**Figure 8.1:** Scalability across base topologies of instants with approx. 160 $\mu$s user computation time

the minimum amount of work per work reactive, such that FrameSweep provides scalability up to just below 16 threads. We choose the target point just below 16 threads so that the corresponding pinnacle point in the throughput graphs is visible and shows that the selected amount of work is precisely as large as necessary.

Figure 8.1 shows the topologies and resulting throughput graphs. In the topology visualizations, thicker arrows pointing towards input reactives represent active threads admitting changes. The color of reactives visualizes a heatmap of how much contention these resulting instants cause on each reactive, with darker shades representing higher contention. In the top left experiment, all instants perform some uncontended workload before funneling into a maximally contended global bottleneck reactive. For this topology, each instant affects only a single work reactive (marked W). Scalability to just below 16 threads (shown in the throughput graph) is reached at a computational cost (on the hardware we used) of only ca. 160 $\mu$s user computation time per work reactive.

Notice that all topologies in this set of experiments have at least one single-reactive bottleneck for all instants. Achieving scalability thus requires support for pipeline parallelism. FrameSweep features pipeline parallelism by design. Handcrafted solutions cannot provide pipeline parallelism (without managing locks from inside user computations), so no Handcrafted measurements are included. STM-RP can support pipeline parallelism, but only under lucky external circumstances. Such circumstances are given in this first topology (conflicts can only occur for a brief moment at the very end of each instant), but not in the others, thus STM-RP provides scalability only for this first topology.

In the bottom left experiment, the top left topology is reversed so that instants first pass the bottleneck and only afterwards reach the parallelizable work reactives. This way, all reactives in the graph are contended by all threads. To parallelize this topology, multiversion concurrency control is indispensable: to concurrently recompute all work reactives in different threads, all threads' written values on the topology's input reactives must be available to be read simultaneously. The scalability shown in this experiment's throughput graph was achieved with each work reactive performing only ca. 10 $\mu$s of computations, which adds up to again ca. 160 $\mu$s for each entire instant.

In the top right topology, the work reactives are arranged in a "chain" topology, and the bottom right "grid" topology is a combination of the previous two experiments, with work reactives arranged in 4 chains of 4 reactives length each. The throughput scalability shown for both the chain and grid topologies was also achieved with ca. 10 $\mu$s of computations per reactive, i.e., ca. 160 $\mu$s per instant. All topologies in this set of experiments thus have their inflection point at just below 16 threads at the same amount of computations per instant. The reason for this is as follows. The inflection point in these charts occurs, when the capability of the scheduler to order instants in the *SSG* reaches its limit and becomes a bottleneck for throughput. This bottleneck *is* dependent on the topology, but only indirectly, in that it depends on how often instants need to be ordered in which numbers, which is a consequence of the *DG* topology. It is independent though, of how the cost of reevaluations is distributed across the topology. The topologies in this set of experiments all have the same inflection point because they differ

only in the second aspect, but not in the first (in each topology, every instant conflicts with all others upon reaching the bottleneck reactive). We evaluate the effect of differences in the first aspect in the next set of experiments.

Summarizing this set of experiments, we conclude that FrameSweep with multiversions and pipeline parallelism can parallelize reevaluations down to individual reactives, regardless of the *DG* topology. The necessary computational cost of user computations per instant to overcome the overhead cost of scheduling up to almost 16 threads is only ca. 160 $\mu$s (on our hardware, and assuming that work is spread evenly across at least 16 reactives). This cost can mostly be considered an upper bound, because – as the following experiments will also show – the bottleneck becomes much less restrictive when there is less contention between concurrent instants.

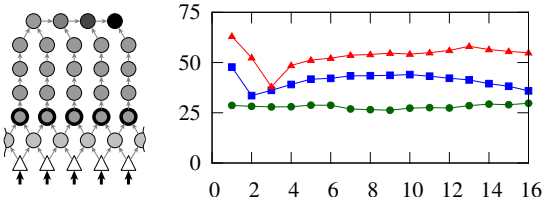
### 8.1.2 Very Cheap User Computations

Next we analyze scheduling overhead and its impact on scalability in more detail. To do so efficiently, we use an application with cheap and fast instants, so that the scheduling overhead dominates the cost of executions and is therefore heavily emphasized in the measurements. Our philosophers application from Section 2 serves well for this, since the majority of its reactives executes only a single integer addition and equality check, or even less. The operation (op) that we use as a benchmark always executes two instants: First, the Read-Update-Read transaction from Listing 7.6 is repeated until the if-condition passes and the `phils(idx).set(Eating)` instant is executed once. Afterwards, a second instant changes the philosopher back to `Thinking`.

Three aspects of FrameSweep cause overhead: the framing phase, operations on reactives must search for their correct placement in the node histories, and instant ordering relations must be recorded in the *SSG*. Clearly, these aspects easily dominate the reactives' user computations of the philosopher application in execution cost. Moreover, the latter two aspects are more expensive under higher contention (node histories contain more elements and more ordering relations between instants are created). Contention is therefore the most influential factor on scheduling overhead, and thus also on scalability if instants are cheap. We thus run different configurations of philosophers that produce different amounts of contention, to evaluate this entire space.

#### Extreme Contention

Figure 8.2 on the left shows the topology of the philosphers application from Section 2. This original configuration is a fairly extreme worst-case in terms of contention. All `forks` are contended by two threads, all `sights`, Events, and the folding Signal `counts` by three, and the summing-up Signal chain (`totalCount`) successively funnels all 16 threads into a single reactive bottleneck. This bottleneck again means, we cannot devise a Handcrafted solution better than global locking (without managing locks from inside user computations). Thus, we compare only G-Lock, FrameSweep and STM-RP.

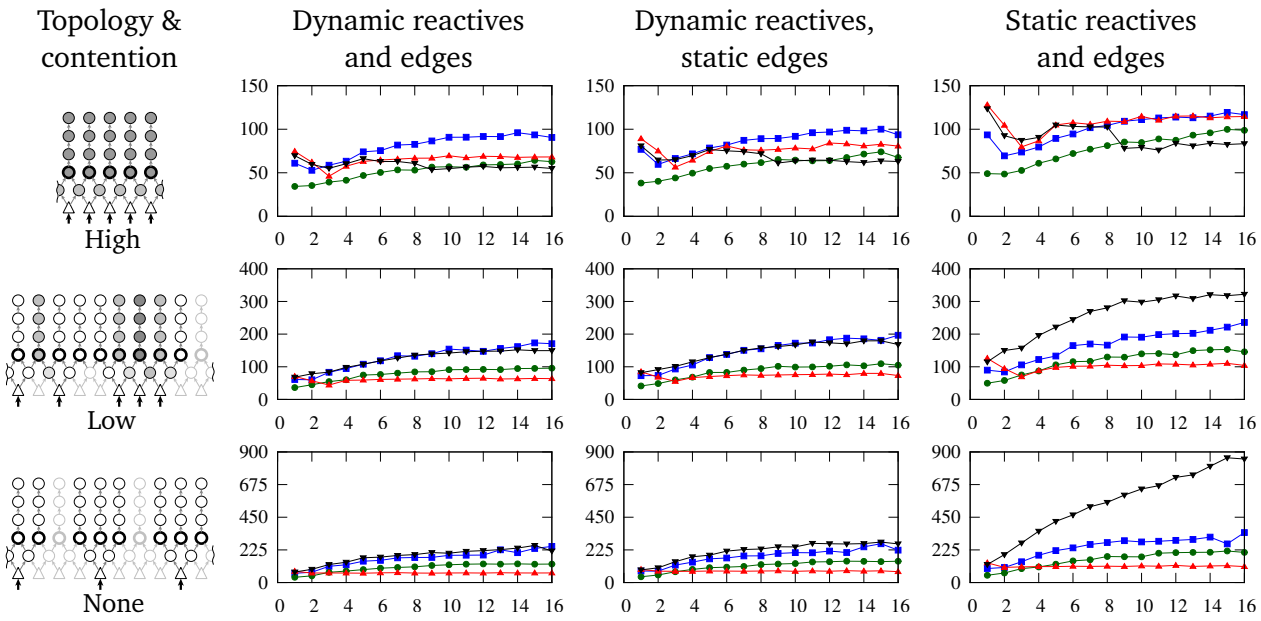



Axes' units and lines' labels as in Figure 8.1

**Figure 8.2:** Extreme contention

The chart in the middle of Figure 8.2 shows the throughput graph of 1 to 16 threads running on a table of 16 philosophers.[1] First, consider the data points for one thread. The throughput of G-Lock represents synchronization-free single-threaded performance of instants, as the overhead of acquiring an uncontended global lock before each change propagation is negligible. With about 75 op/ms (recall, each op is two instants plus two `.now` reads in single-threaded measurements), this corresponds to each

[1] When there are more philosophers than threads, philosophers are distributed to threads round-robin and each thread for each measured op picks a random philosopher from its assigned pool (e.g., thread #2 of 5 randomly chooses `phils(2)`, `phils(7)` or `phils(12)`).




Axes' units and lines' labels as in Figure 8.1.

**Figure 8.3:** Throughput for Different Configurations of the Philosophers Application

instant executing in about 6.5 $\mu$s time, i.e., instants are indeed very cheap. Hence, the overhead cost of the other schedulers in a single-threaded environment stands out clearly: application performance drops by 25% with FrameSweep, and by 55% with STM-RP.

Next, consider the configuration with two threads. G-Lock still executes the same workload, but has become noticeably slower because the global lock is now contended by multiple threads, i.e., the CPU has to engage its synchronization mechanisms, which makes the lock acquisition more expensive. For FrameSweep, this situation is more complex. With pipeline parallelism, FrameSweep allows very fine-grained parallelization of RP applications (shown in detail in Section 8.1.1). With only two threads executing on 16 philosophers, contention is still low enough that they can execute most of their workload (including scheduling overhead) concurrently. Nevertheless, FrameSweep also becomes noticeably slower, i.e., its scheduling overhead increased significantly. Aside from also having some contention on the reactives' locks, this is mainly caused by its initially expensive serializability mechanisms (large histories, instant ordering) now having to engage.

For three and more threads, the throughput of G-Lock and STM-RP remains constant, with STM-RP around 50% lower than G-Lock. For G-Lock this is expected (it prohibits concurrency), for STM-RP this is because it cannot parallelize instants under high contention of the application. FrameSweep is successful in parallelizing the application's instants, indicated by its throughput initially increasing up to around 9 threads. This shows that while initially engaging its serializability mechanisms significantly increased its overhead, adding on more threads afterwards has less of an impact. Beyond 9 threads, the potential for parallelization becomes increasingly saturated while the overhead continues to grow, and thus the throughput decreases again.

Overall, in a setting with extreme contention and very cheap instants the performance of FrameSweep remains between 5% and 30% lower than G-Lock, i.e., FrameSweep fails to improve performance despite successfully parallelizing instants. This is to be expected because in this setting the synchronization overhead clearly dominates the cost of what can be parallelized. On the other hand, the overhead does not make FrameSweep unreasonable to use. We consider this scenario an example at the lower bound for the usability of FrameSweep. In the following experiments in this subsection, we show that lowering contention allows FrameSweep to succeed in scaling performance, even if instants remain cheap and synchronization overhead still dominates the cost of each instant's execution.

### High Contention

First, by removing the summing-up signal chain (i.e., remove Line 37 in Section 2), we remove the most extremely contended reactives, including the bottleneck, from the application. The remaining topology still induces a high amount of contention with most reactives still being contended by three threads, but no longer contains reactives contended by more than three threads. Its contention heatmap is labelled "High" in the first row, left-most column of Figure 8.3.

Without the bottleneck, we can devise a Handcrafted synchronization: For each `phils(i)`, a `java.util.concurrent.locks.ReentrantLock` referenced as `lock(i)` is added. In order to update a given `phils(i)`, the executing thread must hold the philosopher's own `lock(i)`, and both neighbors' `lock(i-1)` and `lock(i+1)` (out-of-bounds indices wrap around). This way, the only interaction that can occur between concurrent instants are racing read operations of some shared `forks`, which is harmless since reads are idempotent and commutative. Threads simply block to acquire needed locks, ordered by their indices to prevent deadlocks. E.g., to update `phils(0)`, locks `lock(0), lock(1), lock(15)` are acquired in that order. Similar to G-Lock, this Handcrafted approach has almost no overhead (three locks per instant, instead of one), but permits parallelization.

The left-most chart in the first row (second column) of Figure 8.3 shows throughputs again on a table of 16 philosophers. Without the summing-up chain, throughput of G-Lock has increased by about 20%, but follows the same shape as in Figure 8.2. Handcrafted performs similar to G-Lock and does not achieve any scalability. This is because threads queue up while blocking to acquire their needed locks, and then transitively waiting even for threads with non-interacting instants[2]. This shows that such simplistic locking is unsuitable for scenarios with high contention. Adding more sophisticated mechanisms for dealing with such issues, however, is usually not reasonable, as it would require to invest yet more effort into a fragile synchronization that cannot be applied to other topologies and breaks under changes.

For FrameSweep, removing the summing-up-chain reduced single-threaded overhead to 20%, for STM-RP it is still at 55%. Further, despite the still very high amount of contention causing significant overhead, FrameSweep already does manage to scale throughput, surpassing the performance of G-Lock and Handcrafted from three threads onwards. This is due to pipeline parallelism providing more fine-grained mutual exclusion than Handcrafted, blocking threads only on individual reactives and only as long as necessary, thus threads do not queue up needlessly. STM-RP also achieves speed-ups, but does not manage to outperform G-Lock even at 16 threads.

### Low Contention

To investigate low contention scenarios, we spread threads more thinly by using a larger table of 64 philosophers. The corresponding topology, with a contention heatmap for an exemplary thread distribution snapshot, is labelled "Low" in Figure 8.3 (second row, left-most column). The left-most chart in that row (second column) shows the experiment's results. G-Lock is the same as under high contention, which is expected since the workload per thread is the same. Contention is low enough that Handcrafted no longer forms excessive queues and can scale across all 16 threads. The throughput of FrameSweep scales about twice as well as under high contention for large thread counts. Even under low contention, FrameSweep still scales better than Handcrafted, from its 20% lower single-threaded throughput to outperforming Handcrafted by about 10% at 16 threads. STM-RP also manages to scale above G-Lock from three threads onwards, but still achieves only approximately 60% of the performance of FrameSweep and Handcrafted.

### No Contention

As a final variation, we use a fixed placement of one thread on every fourth philosopher. This allows all threads to execute concurrently without their instants ever interacting, resulting in zero contention.

[2] The sudden drop in throughput from 8 to 9 threads, which becomes more exacerbated in later experiments, is caused by queueing times becoming long enough to cross the threshold where `ReentrantLock` moves from spinning until the lock becomes available to de-scheduling the requesting thread instead, which leads to a sudden increase in overhead.

The corresponding topology is labelled "None" in the last row of Figure 8.3. Reactives not reevaluated by any thread are faded out through grey outlines, to visualize the conflict avoidance. The corresponding throughput graph is shown next to it (third row, second column). G-Lock is again as before, but all other schedulers can now scale freely, without their differing concurrency management approaches having any effect. Their relative differences are thus equal across all thread counts (modulo some measurement noise): FrameSweep 20% below Handcrafted, STM-RP 55%.

Comparing this contention-free spacing to the previous experiments, we can now estimate the cost of contention for each scheduling approach on this application. At 16 threads, low contention costs about 35% performance when using Handcrafted, 25% when using FrameSweep, and 15% when using STM-RP. High contention respectively costs about 75% performance under Handcrafted, 60% under FrameSweep, and 45% under STM-RP. Extreme contention is not comparable due to its different workload (the removed summing-up chain).

**Summary**

To recap this set of experiments, we have shown that even for instants of only few computations under high contention, FrameSweep is capable of providing scalability with multi-threading. STM-RP is least affected by contention, but has significantly more overhead, resulting in performance consistently and significantly lower than FrameSweep and Handcrafted, in addition to its weaker guarantees (not abort-free). FrameSweep manages to outperform Handcrafted under low contention and even more so under high contention (despite higher single-threaded overhead). Considering in addition that FrameSweep is usable out-of-the-box and applicable to all topologies, whereas Handcrafted must be manually developed and maintained for each application, FrameSweep is clearly the best choice here.

### 8.1.3 Cost of Dynamic Dependency Changes and Retrofitting

To evaluate the scheduling overhead for executing dynamic dependency changes, as well as for handling the conflicts they produce (i.e., retrofitting), we modify the behavior of reactives that perform dynamic dependency changes in the philosophers application, i.e., all `sights` reactives. For visualization, these reactives are highlighted through bold outlines in the topology overviews of Figure 8.3. They account for ca. 23% of executed reevaluations, with close to 10% of all reevaluations actually executing a dynamic dependency change.

First, we changed the behavior of `sights` such that they always access both forks they depend on, but are still declared as dynamic reactives. The results for these experiments are shown in the second-to-right column of Figure 8.3, labelled "dynamic reactives, static edges". This way, no dynamic dependency changes occur, but the remaining workload of reevaluating these reactives stays nearly the same in that the RP framework still must collect, which dependencies were accessed, and check if any changes occurred compared to the previous reevaluation. For the high and low contention configurations (first and second row), this removes both dynamic dependency changes and their according retrofitting. For the contention-free configuration (third row), the lack of interaction between concurrent instants means that the previously executed dynamic dependency changes never required any retrofitting, and thus only dynamic dependency changes but no retrofitting was removed in this configuration. Considering the contention-free configurations first, we see that all scheduling approaches show an insignificant increase in performance across all threads. In particular, this includes G-Lock, which implies that this change cannot be attributed to any operation-specific scheduling overhead, as G-Lock has no such thing. From this we conclude that dynamic dependency changes have no noticeable cost overhead in any of the scheduling approaches. Considering the low and high contention configurations, we observe the same changes, meaning the handling of conflicts of dynamic dependency changes (i.e., retrofitting in case of FrameSweep) on top of executing the changes themselves also has no noticeable impact on performance.

Concluding the philosophers experiments, we consider a final variation of the `sights` reactives, where they declare both forks as their static set of dependencies at initialization, and the RP framework thus

no longer collects and compares their accessed dependencies during each reevaluation. This removes a significant chunk of their reevaluation costs, but this chunk consists only of thread-local computations that do not involve the schedulers, i.e., scheduling overhead remains unchanged. The results are shown in the right-most column of Figure 8.3, labelled "static reactives and edges". As expected, with less computations to be executed under the same scheduling overhead, we observe noticeable increases in throughput compared to the previous set of experiments (second-to-right column) across all scheduling approaches, but the schedulers' overhead is emphasized stronger, placing them relatively further apart. In particular in the contention-free setup, the throughput of Handcrafted is much higher than FrameSweep or STM-RP, because it has the least overhead (just three locks per instant). Comparing this to the low contention scenario though, shows that this advantage quickly diminishes, and at high contention even disappears entirely.

**Summary**

To recap the results from this set of experiments, both the cost of scheduling for dynamic dependency changes and of handling the conflicts they cause (i.e., retrofitting) are negligible, in particular compared to the significant cost of using dynamic dependencies in the first place.

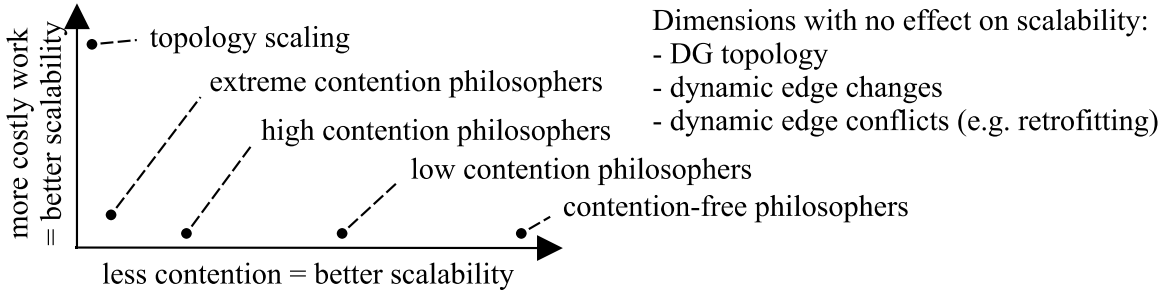



**Figure 8.4:** Effects of Dimensions of RP applications on FrameSweep Scalability

### 8.1.4 Summary of Non-Distributed Experiments

The evaluation of FrameSweep on non-distributed *DG* topologies lead to the following conclusions. First, the topology of the dependency graph, and whether or not the application uses and possibly has conflicts from dynamic edge changes, does not affect the scalability provided by FrameSweep. The only two dimensions that have an impact are conflict density, which depends on how many threads are spread how densely across a given topology, and the cost of user computations during reevaluations. Figure 8.4 visualizes these dimensions and intuitively positions the experiments we conducted in this space. This shows that FrameSweep will be beneficial to use for most applications except those that execute instants of only very cheap computations under extreme thread contention. Even for those worst-case scenarios though, FrameSweep is feasible to use with only some overhead over global locking. It is also worth noting that despite some optimization, FrameSweep is still a very young implementation with several more potential optimizations not yet explored or even discovered. Thus, even though the evaluation already shows mostly favorable results, FrameSweep still has potential for future performance improvements.

## 8.2 Evaluating Distribution Characteristics

The distributed experiments differ from the non-distributed experiments in the previous section in two ways. First, the distributed evaluation only characterizes the cost of FrameSweep to provide its properties, but does not compare it to alternative approaches. This is because – as the current results will

show – FrameSweep's distributed implementation isn't optimally realized yet, while the integration of any alternative approaches is more complex than in local applications and therefore would detract from completing FrameSweep's implementation. Second, in order to better visualize cost rather than comparing performance, experiments measure duration rather than throughput, i.e., time per instants instead of instants per time. To measure this cost as cleanly as possible, all experiments of the distributed evaluation execute with a simulated message delivery delay of 24ms, so that when additionally accounting for local processing times, one round-trip communication from one host to another and back takes approximately 50ms.

Section 8.2.1 evaluates, to which degree FrameSweep is able to parallelize the remote communication involved in instants' regular traversals of distributed *DG* topologies, and to what amount of round-trip times the non-parallelizable remote communications adds up. Section 8.2.2 evaluates the increase in round-trip time cost caused by the need for remote synchronization if instants conflict in different locations of distributed *DG* topologies. Section 8.2.3 summarizes the current state of FrameSweep's distributed evaluation, and outlines future work in terms of FrameSweep's implementation, comparison against alternative approaches, and further experiment designs.

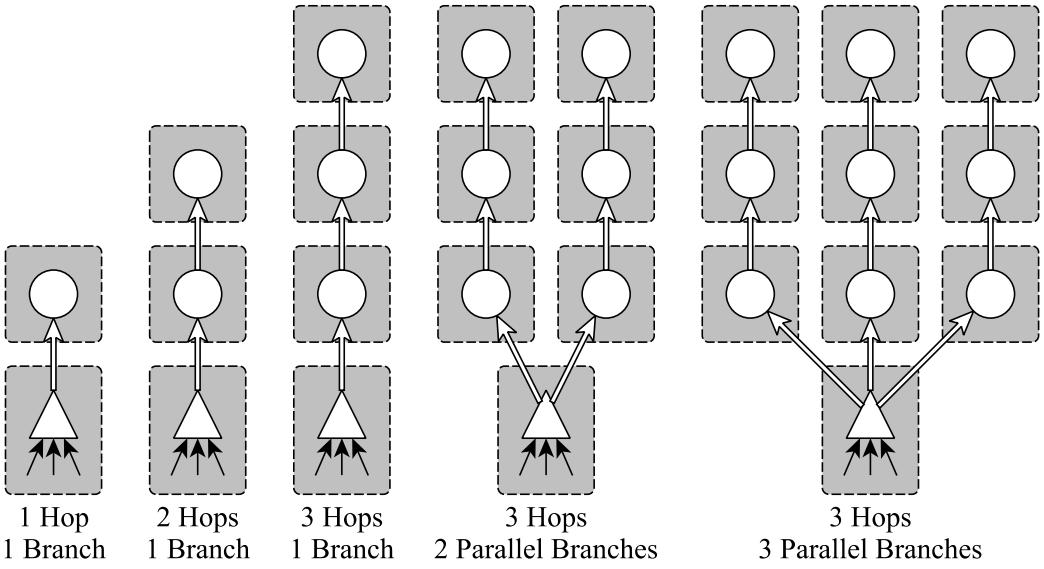



**Figure 8.5:** A Scalable Distributed Topology

### 8.2.1 Distributed Propagation

The first distributed experiment measures the most fundamental aspect of distributed propagation, which is the number of round-trip times that the propagation of changes takes over a given amount of network hops. Figure 8.5 visualizes a distributed *DG* topology that is scalable in both the number of sequential

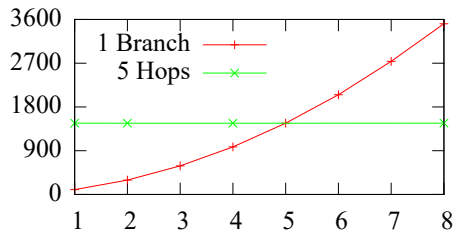



**Figure 8.6:** Single Thread Scaling

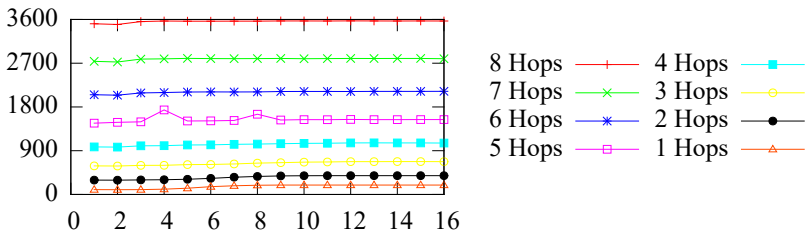



**Figure 8.7:** Multi-Threading Scaling

hops per branch, and the number of parallel branches. Figure 8.6 shows two representative curves of how the time it takes instants to propagate through the topology evolves when scaling the topology along these two dimensions. The green curve "5 Hops" corresponds to a configuration of five sequential hops per branch growing from one to eight parallel branches. The green curve demonstrates, that FrameSweep's support for concurrent reevaluations per instant works flawlessly. The time it takes each instant to complete is equal, regardless of the number of parallel branches.

The red curve "1 Branch" corresponds to a configuration of a single branch growing in length from one up to eight sequential network hops. The first measurement on this curve lies at 98ms, showing that propagating an instant over a single hop takes precisely two round-trip times. This corresponds exactly to the two phases of FrameSweep. One round-trip for transferring a `frame` task and reporting its completion, and a second round-trip for transferring a `notify` task and reporting its completion. The measurements for two and more hops on the red curve show though, that the implementation of FrameSweep does not yet properly implement the communication scheme presented in Section 7.1. FrameSweep implements the replication of instants as a static spanning tree, with each replica being associated with precisely one remote host as its origin. If an instant is newly replicated from a host A to host B where it was previously unknown, then host B statically records the instant as originating from host A. This is incompatible with the model of diffusing computations described in Section 7.1. If an instant sends a task from host C to host B, then once all tasks on B are completed, B should report this back to C. If the instant was previously propagated to host B from a different host A though, then B has recorded A as its origin, and will register its active tasks and report their completion through A instead. This causes a quadratic increase of communication time, which is what the red curve in Figure 8.6 shows. FrameSweep lacks support for hosts to switch between or track multiple origins for each instant. Once this is added though, which requires only further engineering but no further theoretical advances, round-trip times should grow only linearly, at a rate of two round-trip times per sequential network hop (in line with the measurement for a single hop).

The next experiment investigates the effects of multi-threading and pipeline parallelism. Figure 8.7 shows the impact of having up to 16 threads admit instants concurrently to the topology from Figure 8.5 scaled to a single branch of one through eight sequential network hops. Regardless of the number of sequential network hops, the time that each instant takes to complete remains nearly constant, modulo some noise. With more threads executing more instants within the same amount of time, this corresponds to throughput (number of instants per time, as measured for the local experiments) increasing linearly. This experiment therefore demonstrates, that FrameSweep's pipeline parallelism not only enables instants to propagate in parallel even inside mutually affected regions of the *DG* topology, but also enables them to propagate in parallel over shared slow network links.

**Summary**

The time required for instants to complete basic distributed change propagation is unaffected by both dimensions of concurrency. Any number of instants can propagate from the same input reactive concurrently, and instants can propagate over arbitrarily many parallel branches of independent reactives, both without any effect on their execution times. Execution time *is* affected by the number of network hops that an instant has to complete though. While this is expected and unavoidable, FrameSweep's implementation is incomplete, resulting in execution times increasing quadratically rather than linearly with the number of successive network hops.

### 8.2.2 Distributed Synchronization

The previous section evaluated the effect on execution time under concurrency only for concurrent instants admitted to the same input reactive, i.e., which ordered themselves in the *SSG* directly on their origin host without any need for remote synchronization. To measure the amount of remote communication required to for conflict do involve remote synchronization, the topology in Figure 8.8 shows

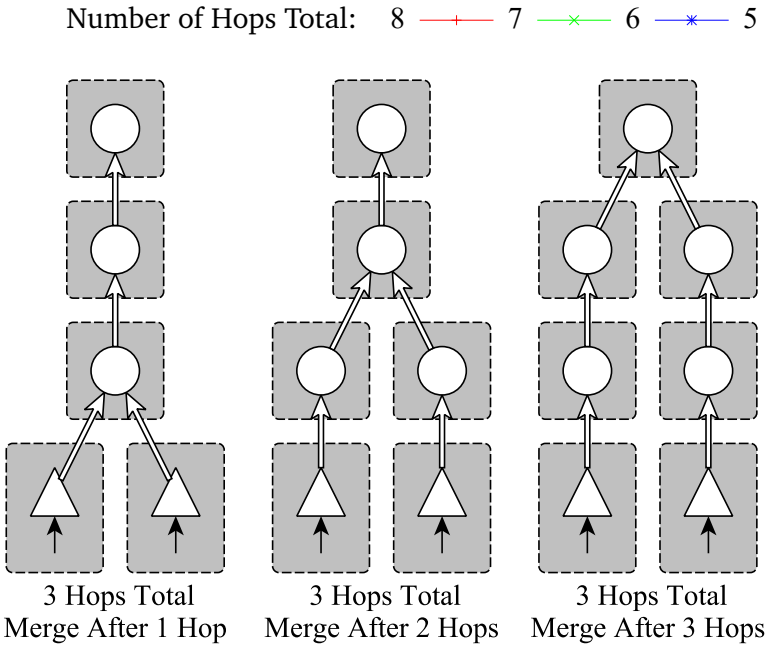



**Figure 8.8:** Different Conflict Distance Topologies

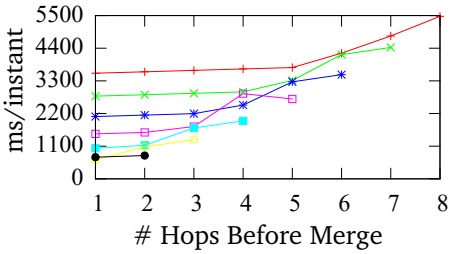



**Figure 8.9:** Total Running Times

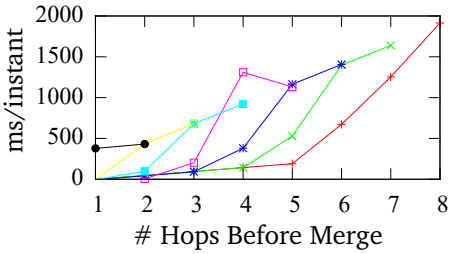



**Figure 8.10:** Added Running Times

a distributed *DG* topology that is scalable again along two dimensions. As in the previous, the total number of hops that each instant has to traverse is configurable, to control the length of instants' regular propagation. By choosing a fixed number of total length, a baseline for the execution time of instant's basic propagation can be selected. Differently from the previous experiments though, instants do not all start from the same input reactive, but each instant starts on its own independent branch, and all branches merge into a single common branch only at some later point. Figure 8.8 shows three examples of this topology, each with a total count of three hops. In the left example, the merge is placed immediately after the first hop, meaning the conflict occurs one network hop away from both instants' origin hosts. In the middle example, the merge is placed after two hops, and in the right example after three hops, i.e., on the last host of each instant's propagation. By varying the length of independent branches, i.e., the number of hops before the merge, the distance between instants' origins and conflicts can be controlled. Increasing this distance makes conflict resolution more costly, since it requires remote synchronization (the `lockUnion` execution) to execute across more hops. Thus, varying the length of independent branches within a fixed total hop count allows to measure the effect of distance on conflict resolution independently from the total propagation time.

Consider two instants propagating over an instance of this topology. The first instant to arrive at the merge host will continue its framing traversal unhindered. The second instant must pause its framing traversal to first establish synchronization between its own and the other instants' origin host, and then order itself later than the first instant in the *SSG*. When this order eventually is established, one of two cases can apply. If the first instant still has a placeholder on the merge host, the second instant's framing phase is immediately complete because of lazy framing (cf. Section 7.2), and the second instant can begin its propagation phase as soon as this information has reached its origin host. If the first instant has begun propagating and already removed its placeholder on the merge host though, lazy framing has no effect and the second instant must continue its framing traversal through the entire depth of the topology. Because FrameSweep's current implementation executes distributed *DG* traversals with quadratic complexity, this will make a significant difference in the second instants' execution time. If both threads continuously admit a new instant as soon as their respective previous instant is complete (as

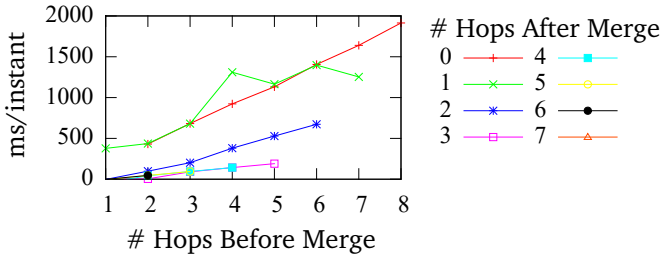



**Figure 8.11:** Added Running Times Re-Categorized

they did in previous experiments with concurrency), the relative timing of this interaction between both threads' instants will continuously shift, causing significant non-deterministic variance in measurements.

To measure execution time as deterministically as possible, this experiment therefore executes instants differently than the previous experiments. Each measurement is taken by starting two concurrent instants, one from each input, at nearly the same time. Execution time is measured as the time it takes until both instants are complete, and the next measurement with a new pair of instants is only started afterwards. This way, the timing of the two instants' interaction will always be the same. Moreover, the instant to arrive at the at the merge host second will always be the instant that executes all remote synchronization, and will always order itself later than the other instant in the *SSG* and therefore also complete later. Thus, each measurement will always correspond to the execution time of that instant which actually executed the remote synchronization.

Figure 8.9 shows one curve for each total number of hops from one through eight, with data points showing instants' measured average execution time for each possible conflict distance within that total hop count. E.g., the blue line with star points shows the execution time of instants on the topology with a total length of six hops, with branches merging after one through six hops. All curves show a similar shape: while the merge distance lies in the first half of the total length, execution time slowly increases linearly, but once the merge distance lies further back than half the number of total hops, the increase is much steeper. Figure 8.10 shows this effect more clearly, by presenting only the execution time that is added to each instant by the conflict resolution, on top of its regular propagation. This is achieved by subtracting from each measurement the baseline measured in Figure 8.6 that it takes for an instant to propagate across a *DG* topology of the respective total hop count. E.g., from each measurement on the red line of Figure 8.9, for eight hops total length, the baseline measurement is approximately 3500ms, from the eight hops data point on the red line of Figure 8.6.

The slow linear increase of the curves in Figure 8.10 before their inflection point corresponds to the case where the first instant still has a placeholder on the merge host when the second instant finishes establishing an order between the two. The execution time slowly increases in this case, because the second instant may transition from its framing phase to its propagation phase only after the first instant did so. A greater distance to the merge host means that the two instants' origin hosts are further apart from each other, and thus it takes slightly longer for the first instant to communicate its phase transition to the second instant's origin host, thereby increasingly delaying the second instant's own transition.

The steep increase of the curves in Figure 8.10 after their inflection point corresponds to the case where the first instant has already removed its placeholder on the merge host when the second instant finishes establishing an order between the two. Since in this case, the second instant must additionally execute its framing traversal over the remaining depth of the topology before it can start its propagation traversal, the framing traversal's quadratic execution time interferes with the measurements of the synchronization overhead. The execution time of the synchronization overhead hidden within this quadratic increase is only linear though. Figure 8.11 demonstrates this by re-categorizing the exact same measurements into curves based on how many hops follow the merge, rather than the total number of hops. The measurements from where the merge was followed by one further network hop (green line with X markers) show some noise. For the cases where the merge at the very end of the topology (red line with plus markers), or two or three hops before the last node (blue line with asterisk markers and red line

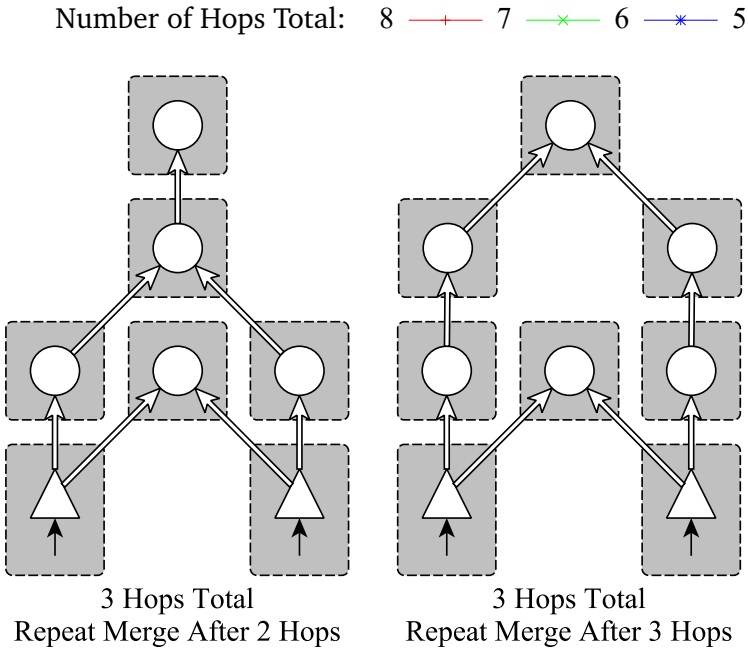



**Figure 8.12:** Repeat Conflict Distances Topologies

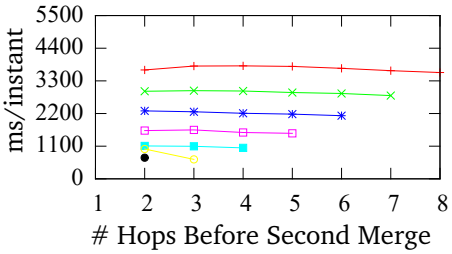



**Figure 8.13:** Total Running Times

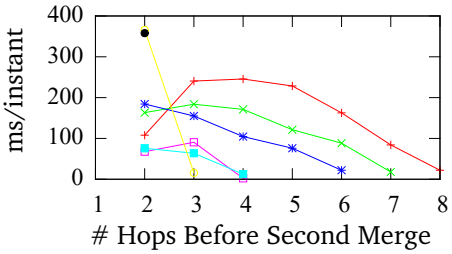



**Figure 8.14:** Added Running Times

with hollow square markers), the execution time overhead from remote synchronization again grows only linearly when additional nodes are added before the merge and the topology after the merge is left unchanged.

Figure 8.12 shows a slightly modified topology from the previous experiment, designed to evaluate the effects of repeated conflicts. As in the previous topology, two input reactives initially propagate over two independent branches that merge after a certain number of hops. Additionally though, a dedicated host always merges both input reactives directly after only a single hop, so that for the instants' interaction on the later merge host, an order between the two instants has already been established. Figure 8.12 shows the measured total execution times, and Figure 8.14 the computed overhead based on Figure 8.6 as the baseline measurement for each total number of hops. The lines consistently show, that there is some overhead in overall execution time, but that this overhead shrinks with increased distance to the later merge host.

The reason for this overhead to exist is because each instant's framing phase traverses to the dedicated single-hop merge and over its independent branch concurrently. This means, while the single-hop merge host establishes an order between the two instants, their framing traversals over their independent branches concurrently traverse further down the *DG* topology. As a result, they may interact on the later merge host before the established order from the single hop merge host has been replicated all the way up to the later merge host. If that happens, the second instant's framing traversal still pauses briefly on the later merge node in an attempt to establish an order, and will continue once the replication of the already-established order has caught up.

The reason for this overhead to shrink with increasing distance to the later merge host is the quadratic execution time of FrameSweep's graph traversals. The order replication traverses through the *DG* topology in linear time. Meanwhile, the concurrently executing framing traversal slows down quadratically the further down the *DG* topology it gets. This means, the further back the merge host lies, the further will the order replication have caught up to the framing traversal, thereby shortening the framing traversal's pause, and thus reducing the overhead cost again.

### 8.2.3 Summary and Future Work

The experiment on distributed topologies have shown, that FrameSweep's distributed graph traversals still support all dimensions of concurrency. In particular, pipeline parallelism – once conflicts have been resolved – and breadth of the *DG* topology do not impact the execution time of instants at all. Added execution time due to remote synchronization needs for safely establishing orders between concurrent instants in the *SSG* on top of regular propagation execution time grows linearly with number of network hops between the host where instants interact and the hosts where they originate from. Graph traversals for framing and propagation are not implemented correctly yet, with their execution time growing quadratically with the number of subsequent network hops, rather than linear.

The experiments have shown, that the quadratic time graph of graph traversals interferes with and complicates FrameSweep's performance analysis. As such, we have not implemented further experiments yet, since optimizations, especially a proper implementation of graph traversals with linear execution time, take priority. The same applies to integrating alternative scheduling approaches into the existing experiments. For a comprehensive evaluation of distributed synchronization, both should be addressed.

Further experiments should be implemented in particular in regards to FrameSweep's behavior on distributed topologies under contention. Both the remote synchronization for *SSG* order establishment, as well as the remotely synchronized phase transitions between instants are at risk of quickly forming performance bottlenecks once many concurrent instants interact with each other. Thus, dedicated experiments should be designed and implemented to that stress both of these factors. Lastly, a comprehensive experiment to evaluate the overall performance of FrameSweep in distributed applications, without stressing any single factor in particular, is desirable.

In terms of alternative approaches for comparison, as in the local setting, no prior works exist that offer the same guarantees as FrameSweep. Still, several options exist that offer similar but weaker properties.

- G-Lock could also be applied to distributed applications, although with certain additional restrictions. First, it requires a central host to manage the global lock. Second, it should still be combined with a decentralized propagation algorithm, e.g., mark-sweep instead of REScala's default height-based priority queue.
- STM-RP may be applicable if a suitable reusable implementation of software transactional memory can be found.
- DREAM (cf. Section 4.3.4), configured for Atomic consistency, can be used to run all experiments that do not use dynamic dependencies.
- Similarly, a distributed adaptation of the propagation algorithm of Elm (cf. Section 4.3.2) may be applicable to all experiments that do not use dynamic dependencies. Further, equal to G-Lock, it would require a central host to initiate the propagation of each instant from all input reactives of the entire application.

## 8.3 Parallelizing Existing Applications

This final experiment evaluates two research questions: First, how much effort and code changes are necessary to migrate an existing single-threaded RP application into a multi-threaded environment safely. Second, can this migration improve the application's performance. The experiment uses "Universe", an existing REScala case study, because it is one of the largest applications in REScala's example corpus and has a structure that makes it easy to introduce multi-threading in its imperative parts. Universe is a local application without distribution. This is both necessary, since no suitable distributed applications exist yet, and better-suited for this experiment, because it allows to demonstrate the syntactic transparency of FrameSweep in full effect.

For the first question, the experiment showed that no changes to the RP code of the application were necessary at all. Only the application's imperative code needed modifications, and these served only to introduce multi-threading, not to facilitate the integration of FrameSweep: "Universe" implements a simulation of an ecosystem with plants and animals, which repeatedly executes two phases. First, a list of tasks for animals and plants to move, feed, grow etc. is populated by one single instant. Due to executing only a single instant, this first phase is not parallelizable with concurrent instants. Second, all the generated tasks from that list – each of which is again an instant – are executed. This second phase is parallelizable by executing all of these instants at the same time. To achieve this parallelization, the only change to the application was to use a parallel drop-in replacement from the standard Scala collections library for the list of tasks. This parallel list submits all tasks to a thread pool for concurrent execution instead of running them one after the other. No further changes were necessary to execute the resulting multi-threaded application on top of FrameSweep.

For the second question, Figure 8.15 shows the throughput (instants per amount of time, higher is better) under up to 16 worker threads. Since the application is only a simulation, it does not use side-effects, and we were able to include STM-RP in this experiment. We do not include Handcrafted results, as we deemed too large the effort of not only precisely analyzing and understanding the possible topologies of the application's dependency graph, but then also trying to find, how – or if at all – manual synchronization is possible. As a rough intuition though, the application executes a large number of instants across a very wide graph, meaning even at 16 threads there is still very little contention during each second phase. Hence, STM-RP is equally capable as FrameSweep in scaling the application's throughput through multi-threading. Throughput increases less quickly under higher numbers of worker threads because only each iteration's second phase is parallelized, but first phase execution times remain constant. Further noteworthy is the application's unusually heavy use of creating and removing reactives when animals and plants are spawned and killed. This gives STM-RP a slight advantage over FrameSweep, because the data structures of STM-RP on the reactives have a smaller memory footprint than those of FrameSweep, and are thus faster to instantiate.

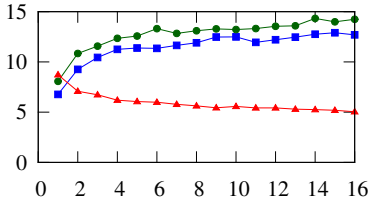



**Figure 8.15:** Universe Throughput

In conclusion, both research questions have positive answers. The necessary effort for migrating to safe multi-threaded execution is minuscule, and performance increased noticeably.

# 9 Conclusion

This dissertation has presented a complete and detailed model of Reactive Programming. Through analysis of the model's interaction with concurrency and distribution, the dissertation derived a formal definition of correctness for synchronous propagation semantics in Distributed Reactive Programming. A comprehensive survey has classified related prior works in regards to how their features and consistency guarantees relate to those of Distributed Reactive Programming.

This dissertation has presented FrameSweep, the first algorithm to provide synchronous propagation semantics for concurrent change propagation over dynamic distributed dataflow graphs. The correctness of FrameSweep's semantics has been formally proven, as well as empirically validated, alongside its syntactic transparency. The dissertation discussed solutions for the practical engineering concerns in regards to decentralizing the implementation of FrameSweep. The combination of syntactic transparency with support for dynamic changes of the dependency graph topology enables arbitrary modular composition of applications implemented through Distributed Reactive Programming. Any applications' dataflow graphs can be merged and split, even dynamically at run-time, without the applications' developers having to design the application with such composition in mind. As a result, using FrameSweep as an example, this dissertation has proven, that the design of distributed and multi-threaded interactive applications can be improved through Distributed Reactive Programming with all the proven benefits (improved code quality, program comprehension and maintainability) that traditional Reactive Programming brings for single-threaded local interactive applications.

Through empirical benchmarks, FrameSweep's performance and scalability for local applications was shown to compare favorably over global locking and software transactional memory, except for applications that consist only of few small reactive computations while being subject to heavy thread contention. Empirical benchmarks on distributed topologies have shown that FrameSweep's distributed implementation is not optimal yet. Correspondingly, completion, further optimization, and more extensive performance evaluation of FrameSweep's distributed implementation is are obvious next steps for future work.

Further important avenues for future works are:

- An integration of FrameSweep into a Distributed Reactive Programming language implementation that has an established syntax for specifying the distribution of dependency graph topologies, such as ScalaLoci [Weisenburger et al., 2018].
- Integration of techniques for detecting and tolerating partial failures of distributed applications, such as automatically persisting accumulated state and restoring it after crashes [Mogk et al., 2018].
- Developing concepts for defining limits to the scope of synchronous propagation. For instance, a state update on a server may result in changes propagating to many clients, but completion of the server's state update should not be transactionally bound to this change being processed by all clients.
- An integration with (transactional) databases. Many distributed applications use databases. In such applications, change propagations will change the contents of the database. In response, the results of queries will change. This gap could be bridged, in that queries return live data views [Mitschke et al., 2014] which update as part of the same change propagation that updates the contents of the database.

# Bibliography


[Abadi et al., 2005] Abadi, D. J., Ahmad, Y., Balazinska, M., Cherniack, M., hyon Hwang, J., Lindner, W., Maskey, A. S., Rasin, E., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. (2005). The design of the borealis stream processing engine. In *In CIDR*, pages 277–289.

[Acar, 2005] Acar, U. A. (2005). *Self-adjusting Computation*. PhD thesis, Pittsburgh, PA, USA. AAI3166271.

[Acar and Chen, 2013] Acar, U. A. and Chen, Y. (2013). Streaming big data with self-adjusting computation. In *Proceedings of the 2013 Workshop on Data Driven Functional Programming*, DDFP '13, pages 15–18, New York, NY, USA. ACM.

[Agha, 1986] Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA.

[Bainomugisha et al., 2013] Bainomugisha, E., Carreton, A. L., Cutsem, T. v., Mostinckx, S., and Meuter, W. d. (2013). A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34.

[Bernstein and Goodman, 1981] Bernstein, P. A. and Goodman, N. (1981). Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221.

[Bernstein et al., 1986] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1986). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Berry and Gonthier, 1992] Berry, G. and Gonthier, G. (1992). The Esterel synchronous programming language: design, semantics, implementation. *SCP*, 19(2).

[Bhatotia et al., 2015] Bhatotia, P., Fonseca, P., Acar, U. A., Brandenburg, B. B., and Rodrigues, R. (2015). ithreads: A threading library for parallel incremental computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 645–659, New York, NY, USA. ACM.

[Bhatotia et al., 2011] Bhatotia, P., Wieder, A., Rodrigues, R., Acar, U. A., and Pasquin, R. (2011). Incoop: Mapreduce for incremental computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA. ACM.

[Bonér et al., 2014] Bonér, J., Farley, D., Kuhn, R., and Thompson, M. (2014). The reactive manifesto.

[Boniol and Adelantado, 1993] Boniol, F. and Adelantado, M. (1993). Programming distributed reactive systems: A strong and weak synchronous coupling. In Schiper, A., editor, *Distributed Algorithms*, pages 294–308, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Boudol, 2004] Boudol, G. (2004). Ulm: A core programming model for global computing. In Schmidt, D., editor, *Programming Languages and Systems*, pages 234–248, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Boussinot, 1991] Boussinot, F. (1991). Reactive c: An extension of c to program reactive systems. *Softw. Pract. Exper.*, 21(4):401–428.

[Boussinot and de Simone, 1996] Boussinot, F. and de Simone, R. (1996). The sl synchronous language. *IEEE Transactions on Software Engineering*, 22(4):256–266.

[Bronson et al., 2010] Bronson, N. G., Chafi, H., and Olukotun, K. (2010). CCSTM: A library-based STM for Scala. In *The First Annual Scala Workshop at Scala Days*.

[Burckhardt and Coppieters, 2018] Burckhardt, S. and Coppieters, T. (2018). Reactive caching for composed services: Polling at the speed of push. *Proc. ACM Program. Lang.*, 2(OOPSLA):152:1–152:28.

[Burckhardt et al., 2011] Burckhardt, S., Leijen, D., Sadowski, C., Yi, J., and Ball, T. (2011). Two for the price of one: A model for parallel and incremental computation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 427–444, New York, NY, USA. ACM.

[Carbone et al., 2015] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4).

[Caspi et al., 1994] Caspi, P., Girault, A., and Pilaud, D. (1994). Distributing reactive systems. In *Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94*, Las Vegas, USA. ISCA.

[Caspi et al., 1987] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. (1987). LUSTRE: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 178–188, New York, NY, USA. ACM.

[Cooper and Krishnamurthi, 2006] Cooper, G. H. and Krishnamurthi, S. (2006). Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 294–308, Berlin, Heidelberg. Springer-Verlag.

[Cugola and Margara, 2012] Cugola, G. and Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62.

[Czaplicki and Chong, 2013] Czaplicki, E. and Chong, S. (2013). Asynchronous functional reactive programming for guis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, New York, NY, USA. ACM.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

[Dijkstra and Scholten, 1980] Dijkstra, E. W. and Scholten, C. S. (1980). Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4.

[Drechsler et al., 2018] Drechsler, J., Mogk, R., Salvaneschi, G., and Mezini, M. (2018). Thread-safe reactive programming. *Proc. ACM Program. Lang.*, 2(OOPSLA).

[Drechsler and Salvaneschi, 2014] Drechsler, J. and Salvaneschi, G. (2014). Optimizing distributed rescala. REBLS'14.

[Drechsler et al., 2014] Drechsler, J., Salvaneschi, G., Mogk, R., and Mezini, M. (2014). Distributed REScala: An update algorithm for distributed reactive programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 361–376, New York, NY, USA. ACM.

[Elliott and Hudak, 1997] Elliott, C. and Hudak, P. (1997). Functional reactive animation. *SIGPLAN Not.*, 32(8):263–273.

[Eugster et al., 2003] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131.

[Galler and Fisher, 1964] Galler, B. A. and Fisher, M. J. (1964). An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Gautier et al., 1987] Gautier, T., Le Guernic, P., and Besnard, L. (1987). SIGNAL: A declarative language for synchronous programming of real-time systems. In *FPCA*, pages 257–277, London, UK. Springer-Verlag.

[Halbwachs and Mandel, 2006] Halbwachs, N. and Mandel, L. (2006). Simulation and verification of asynchronous systems by means of a synchronous model. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 3–14.

[Hammer et al., 2007] Hammer, M., Acar, U. A., Rajagopalan, M., and Ghuloum, A. (2007). A proposal for parallel self-adjusting computation. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 3–9, New York, NY, USA. ACM.

[Han and Zhang, 2015] Han, Z. and Zhang, Y. (2015). Spark: A big data processing platform based on memory computing. In *2015 Seventh International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, pages 172–176.

[Herlihy and Wing, 1990] Herlihy, M. P. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Italiano, 1986] Italiano, G. (1986). Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273 – 281.

[Kahn, 1962] Kahn, A. B. (1962). Topological sorting of large networks. *Commun. ACM*, 5(11):558–562.

[Liberty and Betts, 2011] Liberty, J. and Betts, P. (2011). *Programming Reactive Extensions and LINQ*. Apress, Berkely, CA, USA, 1st edition.

[Lombide Carreton et al., 2010] Lombide Carreton, A., Mostinckx, S., Van Cutsem, T., and De Meuter, W. (2010). Loosely-coupled distributed reactive programming in mobile ad hoc networks. In Vitek, J., editor, *Objects, Models, Components, Patterns*, pages 41–60, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Maier and Odersky, 2012] Maier, I. and Odersky, M. (2012). Deprecating the Observer Pattern with Scala.react. Technical report.

[Maier and Odersky, 2013] Maier, I. and Odersky, M. (2013). Higher-order reactive programming with incremental lists. In Castagna, G., editor, *ECOOP 2013 – Object-Oriented Programming*, pages 707–731, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Margara and Salvaneschi, 2014] Margara, A. and Salvaneschi, G. (2014). We have a dream: Distributed reactive programming with consistency guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 142–153, New York, NY, USA. ACM.

[Margara and Salvaneschi, 2018] Margara, A. and Salvaneschi, G. (2018). On the semantics of distributed reactive programming: The cost of consistency. *IEEE Transactions on Software Engineering*, 44(7):689–711.

[Meijer, 2010] Meijer, E. (2010). Reactive extensions (rx): Curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 11:1–11:1, New York, NY, USA. ACM.

[Meyerovich et al., 2009] Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A., and Krishnamurthi, S. (2009). Flapjax: A programming language for ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1–20, New York, NY, USA. ACM.

[Mitschke et al., 2014] Mitschke, R., Erdweg, S., Köhler, M., Mezini, M., and Salvaneschi, G. (2014). I3ql: Language-integrated live data views. *SIGPLAN Not.*, 49(10):417–432.

[Mogk et al., 2018] Mogk, R., Baumgärtner, L., Salvaneschi, G., Freisleben, B., and Mezini, M. (2018). Fault-tolerant Distributed Reactive Programming. In Millstein, T., editor, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:26, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Myter et al., 2016] Myter, F., Coppieters, T., Scholliers, C., and De Meuter, W. (2016). I now pronounce you reactive and consistent: Handling distributed and replicated state in reactive programming. In *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2016, pages 1–8, New York, NY, USA. ACM.

[Proença and Baquero, 2017] Proença, J. and Baquero, C. (2017). Quality-aware reactive programming for the internet of things. In Dastani, M. and Sirjani, M., editors, *Fundamentals of Software Engineering*, pages 180–195, Cham. Springer International Publishing.

[Prokopec et al., 2014] Prokopec, A., Haller, P., and Odersky, M. (2014). Containers and aggregates, mutators and isolates for reactive programming. In *Proceedings of the Fifth Annual Scala Workshop*, SCALA '14, pages 51–61, New York, NY, USA. ACM.

[Reynders et al., 2014] Reynders, B., Devriese, D., and Piessens, F. (2014). Multi-tier functional reactive programming for the web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 55–68, New York, NY, USA. ACM.

[Salvaneschi et al., 2014a] Salvaneschi, G., Amann, S., Proksch, S., and Mezini, M. (2014a). An empirical study on program comprehension with reactive programming. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 564–575, New York, NY, USA. ACM.

[Salvaneschi et al., 2013] Salvaneschi, G., Drechsler, J., and Mezini, M. (2013). Towards distributed reactive programming. In De Nicola, R. and Julien, C., editors, *Coordination Models and Languages*, pages 226–235, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Salvaneschi et al., 2014b] Salvaneschi, G., Hintz, G., and Mezini, M. (2014b). Rescala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 25–36, New York, NY, USA. ACM.

[Sant'Anna et al., 2013] Sant'Anna, F., Rodriguez, N., Ierusalimschy, R., Landsiedel, O., and Tsigas, P. (2013). Safe system-level concurrency on resource-constrained nodes. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 11:1–11:14, New York, NY, USA. ACM.

[Sawada and Watanabe, 2016] Sawada, K. and Watanabe, T. (2016). Emfrp: A functional reactive programming language for small-scale embedded systems. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 36–44, New York, NY, USA. ACM.

[Weisenburger et al., 2018] Weisenburger, P., Köhler, M., and Salvaneschi, G. (2018). Distributed system development with scalaloci. *Proc. ACM Program. Lang.*, 2(OOPSLA).

[Wu et al., 2006] Wu, E., Diao, Y., and Rizvi, S. (2006). High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 407–418, New York, NY, USA. ACM.

[Yip et al., 2016] Yip, E., Girault, A., Roop, P. S., and Biglari-Abhari, M. (2016). The forec synchronous deterministic parallel programming language for multicores. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, volume 00, pages 297–304.