Synthesis-Aided Development of Distributed Programs

by

Ivan Kuraj

S.B., Software Engineering, University of Belgrade, 2010
S.M., Computer Science, École polytechnique fédérale de Lausanne, 2013

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2024

Authored by:	Ivan Kuraj Department of Electrical Engineering and Computer Science January 26, 2024
Certified by:	Armando Solar-Lezama Distinguished Professor of Computing Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Synthesis-Aided Development of Distributed Programs

by

Ivan Kuraj

Submitted to the Department of Electrical Engineering and Computer Science on January 26, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Despite many advances in programming models and frameworks, writing distributed programs remains hard. Even when the underlying logic is inherently sequential and simple, addressing distributed aspects results in complex cross-cutting code that undermines such simplicity. While the sequential computation model of programs represents a simple and natural form for expressing functionality, corresponding distributed implementations need to break this model. One of the most challenging aspects that impede achieving separation of concerns, significantly increases the difficulty of reasoning about distributed programs and, subsequently, complicates their implementation is the consistency model.

This thesis examines the possibility of using the sequential model for writing distributed programs, characterizes the requirements for making that possible, and presents a synthesis approach that allows programmers to automatically generate distributed implementations from behaviors given as sequential programs and orthogonal specifications of distributed aspects. The end result is a programming system in which programmers define sequential behaviors and separately specify data allocation, reactivity, the underlying network with orthogonal specifications, as well as integrity, as a set of high-level semantic properties. Given such specifications, the system automatically finds an optimal consistency model needed to maintain the given integrity and emits low-level message-passing implementations. The system combines two novel techniques into a two-step process: first, it statically infers optimal consistency requirements for executions of bounded sets of operations, and then, it uses the inferred requirements to parameterize a new distributed protocol to relax operation reordering at run time when it is safe to do so. We demonstrate the system's expressiveness and examine run-time performance impact on benchmarks from prior work, as well as new benchmarks.

Thesis supervisor: Armando Solar-Lezama
Title: Distinguished Professor of Computing

Acknowledgments

I would like to express my sincere gratitude to my advisor, Armando, for the unwavering support and encouragement throughout the years, fostering my motivation to delve into this research topic. During hard times, Armando approached challenges with a positive perspective. His encouragement served as a constant driving force, urging me to keep moving forward. Engaging in conversations with Armando proved invaluable, leading to either a productive brainstorming session filled with new ideas, the development of a clear mental model for explaining existing concepts, or a renewed focus on the path forward. Beyond academia, many of Armando's stories resonated deeply with me, giving me a multitude of useful perspectives. I am truly grateful to Armando for his belief in me, the freedom he gave me to explore my own ideas, and the guidance that has been instrumental in navigating and completing this long journey.

The Computer Aided Programming group and the entire Programming Languages and Software Engineering community in CSAIL have been my welcoming home at MIT. My interactions with Adam Chlipala and Martin Rinard have profoundly shaped my perspectives on verified software and the potential of programming languages research. I am grateful for the camaraderie and friendships with fellow graduate students and lab mates Aleks Milicevic, Eunsuk Kang, Sasa Misailovic, Nadia Polikarpova, Jeevana Priya Inala, Kuat Yessenov, Shachar Itzhaky, Harshad Kasture, Anurag Mukkara, Alaa Khaddaj, and many others. They provided me with a warm atmosphere and a never-ending source of inspiring discussions. It was a joy and a privilege to share their company. I will always truly miss our "coffee train" getaways.

I would also like to thank Daniel Jackson for the warm welcome to MIT and the Software Design Group. A significant portion of my Ph.D. journey unfolded within this group, filled with pleasant moments and invaluable learning experiences.

I want to thank my parents, family, and my wife Danica for their unwavering love and support. Danica became the brightly shining light that provided me with the strength needed to persevere on this journey. I also want to thank Marija for being kind and supportive of me and my academic work, even during our challenging times. To them, I would like to apologize for using my academic pursuit as an excuse for my absence on numerous occasions. I will be eternally grateful for their understanding and patience.

I would like to thank the NSF for funding my work (through the grants #1438969, #1918839, #2217064, #1918889, #1665282), and allowing me to have the freedom to pursue my ideas, as well as the Technology, Innovation and Partnerships directorate at NSF for giving me the opportunity to explore entrepreneurship through the I-Corps program (grant #2028103).

Contents

Title page 1

Abstract 3

Acknowledgments 5

List of Figures 11

List of Tables 13

1 Introduction 15

1.1 Overview of the Approach . . . 16

1.1.1 SyCoord . . . 17

1.1.2 PEEPCO . . . 19

1.1.3 System Integration . . . 20

1.2 Illustrative Example . . . 21

1.2.1 Integrity of Distributed Programs . . . 22

1.2.2 Consistency Optimization . . . 26

1.2.3 New Point in the Design Space of Consistency Optimization . . . 30

1.2.4 Extensions of the Main Approach . . . 32

1.2.5 Implementation and Benchmarks . . . 33

1.3 Overview . . . 34

1.3.1 Discussion and Limitations . . . 35

1.3.2 Scientific Journey . . . 36

2 PeepCo: Batch-Based Consistency Optimization 38

2.1 Motivation . . . 41

2.2 Batch-Based Consistency . . . 44

2.2.1 Semantics of Batch-Based Consistency . . . 44

2.2.2 Batched Operation Execution . . . 46
2.2.3 Consistency Constraints and Lattice . . . 47
2.2.4 PeepCommit Protocol . . . 49
2.2.5 Integrity-Driven Consistency Strengthening . . . 52
2.3 Partial Replication . . . 55
2.3.1 Motivation . . . 56
2.3.2 Finer Grained Batch-Based Consistency . . . 56
2.4 Implementation . . . 59
2.5 Evaluation . . . 60
2.6 Discussion . . . 65

3 SyCoord: Optimizing Consistency for Partially Replicated Data-Centric Distributed Applications 67
3.1 SyCoord's Data Store Model . . . 71
3.2 Coordination Synthesis in SyCoord . . . 74
3.3 Cause-Effect Framework . . . 77
3.3.1 Executions in Partial Replication . . . 78
3.3.2 Elements of Inference . . . 80
3.3.3 Cause-Effect Graph Instrumentation . . . 82
3.3.4 Consistency Constraints Inference . . . 84
3.4 Conflict-Driven Propagation . . . 87
3.5 SyCoord Extensions . . . 92
3.6 Implementation . . . 94
3.6.1 Components of SyCoord . . . 95
3.7 Evaluation . . . 96
3.7.1 Inference Benchmarks . . . 97
3.7.2 Data Store Run-Time Performance . . . 99
3.7.3 Game Demo Use Case . . . 102
3.8 Discussion . . . 105

4 EdgeC: Aspect-Oriented Language for Reactive Distributed Applications at the Edge 107
4.1 Overview . . . 109
4.2 EdgeC Compilation . . . 116
4.3 Language Semantics . . . 118
4.4 Evaluation . . . 119
4.5 Extensions . . . 120

4.6 Discussion 122

5 Related Work **123**

5.1 Consistency Optimization 124

5.2 Distributed Programming Languages 127

6 Conclusion **131**

References **133**

List of Figures

1.1 Bank account class . . . 21
1.2 Distribution . . . 21
1.3 Execution . . . 22
1.4 Message-passing code . . . 22
1.5 Safe execution . . . 22
1.6 PEEPCO specification . . . 23
1.7 Bank with history . . . 24
1.8 Distribution . . . 24
1.9 Two withdrawals . . . 26
1.10 Replication . . . 26
1.11 True and false violating trace . . . 27
1.12 Backends . . . 27
1.13 Architecture overview of the overall synthesis system . . . 34

2.1 Bank account in PEEPCO . . . 41
2.2 Execution snapshot . . . 41
2.3 Specialized model . . . 41
2.4 PEEPCO specification . . . 43
2.5 Run-time calls . . . 43
2.6 Causal ordering trace . . . 43
2.7 Bank account . . . 56
2.8 Distributed execution . . . 56
2.9 Specification . . . 56
2.10 Execution . . . 57
2.11 Redundant conflict . . . 57
2.12 No-conflict scalability . . . 62
2.13 Run-time performance in full replication. For (a)–(c) and (e)–(f), higher is better. For (d), lower is better. . . . 63

2.14 Sensitivity to contention and batch size 65

3.1 Components in SyCoord. 70
3.2 Bank account class definition. 71
3.3 Partial replication data store. 72
3.4 Bank account data store in SyCoord. 73
3.5 Execution diagrams for the bank account, in full replication (a, b) and partial replication (c, d, e). Red dots denote invariant violations. Lines between nodes denote inter-node message passing, where dashed lines denote effect propagation. 75
3.6 A DPG and a cause-effect graph for the bank data store. 79
3.7 Semantics of distributed executions 80
3.8 Instrumented execution. 84
3.9 Synthesis rules 85
3.10 Additional methods. 92
3.11 Changed specifications. 92
3.12 Instrumented execution: liveness. 93
3.13 Extensions of core SyCoord. 94
3.14 Abstract syntax of SyCoord's Scala DSL 95
3.15 SyCoord data store performance. Y-axis: throughput (op/s), left; latency (ms), right. X-axis: number of nodes. "Relative" shows performance comparison of two data stores emitted by SyCoord, while "off-the-shelf" shows comparison of the emitted data store with an off-the-shelf data store. The dashed lines represent SyCoord (red for throughput, green for latency) and solid lines represent results employing strong consistency in the compared approach (blue for throughput, yellow for latency). 100
3.16 Game state 103
3.17 Naive simulation of the game usecase 103
3.18 Consistency optimized run of the game 104

4.1 Application overview, its logical "sliced view", and the architectures it can capture 110
4.2 Distributed reactive game 111
4.3 Network model specifications 115
4.4 EdgeC compiler 116
4.5 Performance evaluation 120

List of Tables

2.1 Inference statistics after 15 minutes. Legend: $|O|$ - number of methods; int. - number of integrity properties (safe/congruent); $|\mathcal{C}|$ - number of found constraints; B_o, B_c, B_e, B_s - maximal explored bound, default, without consistency, effect, and symmetry pruning, respectively. 61

3.1 Benchmarks and SyCoord results. For each benchmark, we report: the number of invariants $\#I$ (total/live); sufficient search bound b $\#B$; cumulative size of invariant *Spec*ification (in AST nodes); the number of discovered pairwise conistency relations $\#C$; the number of minutes SyCoord runs until exploring bound $b + 1$ (T-*all*), without pruning (T-*npr*), without causality-driven exploration (T-*ncd*), and symmetry breaking (T-*nsb*). Times are given in minutes, where "-" denotes timeout after 10 minutes. 97

Chapter 1

Introduction

Distributed reactive programs such as web-facing data-centric applications, online multi-user services, and data stores constitute an important category of software systems [I. Zhang et al., 2016; Viennot et al., 2015]. Such a program generally consists of a set of compute nodes which must communicate with each other to update their state in response to external stimuli. Some of these applications must also combine incomplete inputs from multiple sources, promptly respond to asynchronous user interactions, and process protocol messages needed to maintain application-level integrity [Chlipala, 2015]. With traditional programming methodologies, design decisions about distributed aspects, such as data distribution, reactivity, and consistency need to be woven together with the application logic. As a result, implementations become complex even when the underlying logic is conceptually simple, and exploring different design choices is tedious because small changes to how data is distributed or how communication is orchestrated require cutting through multiple layers of code.

One of the aspects that significantly increase the difficulty of reasoning about distributed programs and, subsequently, complicate their implementation, is the consistency model. Consistency models for distributed programs are widely studied in programming languages, databases, and systems [Lesani, Bell, and Chlipala, 2016; Sovran et al., 2011; Bailis, Fekete, et al., 2014; I. Zhang et al., 2016]. Traditionally, programmers have relied on strong consistency guarantees, such as serializability [Papadimitriou, 1979], to preserve integrity of their distributed applications, which masks the intricacies of distributed execution over data allocated across a network of nodes, under unpredictable communication delays and concurrency, at the expense of runtime performance [Brewer, 2012]. Modern data-driven distributed applications, however, often sacrifice consistency in return for performance [DeCandia et al., 2007]. This design choice exposes programmers to the possibility of inconsistent states in their application that might violate the integrity. This trade-off means that the choice of consistency model can crucially impact the practical properties and feasibility of modern applications [Kleppmann et al., 2019]. Weaker consistency mechanisms offer higher performance but can cause integrity violations if used naively, while a stronger consistency than necessary can lead to a significant loss of performance. Programmers are faced with the challenge of balancing integrity and performance when choosing the right consistency model given the requirements of their applications [Sivaramakrishnan, Kaki, and Jagannathan, 2015]. Moreover, implementing the resulting application tightly couples the application logic and distributed aspects, and any change to either of the two has to propagate changes in the consistency model through a costly development cycle [Viennot et al., 2015; Stonebraker, Madden, et al., 2007; I. Zhang et al., 2016; Kaki, Priya, et al., 2019]. Automating this choice and the subsequent implementation remains a challenge.

In this thesis we focus on an automated approach to consistency optimization and synthesis of the resulting distributed program implementations. The approach provides an end-to-end bounded-verification strategy for inferring consistency requirements and emitting efficient implementations, expressive for various input programs and data distribution specifications. The synthesized implementations can use standard unbounded consistency protocols based on the inferred requirements that prevent all encountered violating traces during exploration, as well as generalize to unseen traces. The approach also supports a new run-time protocol that ensures correctness on unbounded executions while performing consistency optimization based on the results of bounded static analysis. We first present the consistency optimization approach and then describe extensions that allow programmers to specify additional distributed aspects.

1.1 Overview of the Approach

The main consistency optimization approach consists of two systems for automated consistency optimization and synthesis of distributed programs that effectively capture two evolution points of the development of the overall approach.

The first system, SyCoord [Kuraj and Solar-Lezama, 2020; Kuraj, Solar-Lezama, and Polikarpova, 2022], introduces a static analysis approach that identifies bounded execution traces that can violate integrity, which are then generalized into consistency protocols used at run time. The static analysis supports checking integrity of traces on concrete inputs given by programmers and is expressive for a large class of input programs, as well as data distribution schemes. SyCoord synthesizes message-passing implementations that leverage the statically identified protocols and provide speedups at run time, compared to the strong consistency baseline. Due to boundedness, however, the synthesized implementations do not provide strong guarantees for cases where an unbounded number of operations might start executing at the same time.

The second system, implemented in the tool PEEPCO [Kuraj, Feser, et al., 2023], builds on top of SyCoord and introduces a new strategy for run-time assisted consistency optimization that guarantees correctness for unbounded executions. It introduces batch-based consistency, a new approach for consistency optimization that allows programmers to specialize consistency with application-level integrity properties and implement applications that safely optimize consistency at run time for any concrete values that might occur at run time, based on results of a bounded symbolic exploration.

We explain the two systems in more detail below. In addition, we present an extension to the programming model adopted by SyCoord and PeepCo, and extend it with rich types that allow programmers to specify properties pertaining to the location of data and computation, as well as reactivity.

1.1.1 SyCoord

The goal of the first system is to provide an automated approach for bounded consistency reasoning and optimization inference in a model that supports expressiveness for practical distributed programs. Expressive models that allow controlling distributed aspects at finer granularity pose new challenges for reasoning and development of practical distributed programs, which go beyond existing cloud-based approaches [Alur, E. Berger, et al., 2016; I. Zhang et al., 2016]. Design choices for distributed programs that developers often face involve customizing data allocation schemes, adjusting the program to the underlying network model, and handling the consistency and reactivity requirements of behaviors. The expressiveness of such requirements goes beyond the prior work [Kaki, Earanky, et al., 2018; Sivaramakrishnan, Kaki, and Jagannathan, 2015; Houshmand and Lesani, 2019], and the developers have no choice but to explore design decisions by writing full implementations, significantly different for every combination of design choices made. Prototyping and searching for adequate implementations becomes cumbersome, in spite of some aspects, like the core logic and integrity constraints, being fixed. An approach that achieves such expressiveness through orthogonal specifications for behaviors, data allocation, reactivity, and data consistency, without forcing programmers to reason about low-level implementations of distributed programs, could improve the usability of automated consistency optimization in practice.

SyCoord introduces a new bounded inference strategy that finds requirements on the ordering of operations by leveraging bounded reasoning, capable of capturing expressive execution models [Kuraj, Solar-Lezama, and Polikarpova, 2022]. It relies on an analysis algorithm for consistency inference and provides a framework for transforming the found results into standard consistency protocols to be used at run time. The algorithm discovers the consistency requirements of bounded execution traces of operations derived from sequential objects at compile time, and it can leverage both explicit and SMT-based concolic-style reasoning to check the specified integrity. The algorithm is efficient; it incrementally builds a set of consistency requirements and uses them to aggressively prune the set of execution traces to explore, in addition to using traditional symmetry-breaking techniques. This allows SyCoord to explore traces up to non-trivial bounds, even when the integrity invariants allow many operation reorderings.

We use the SyCoord approach in an end-to-end bounded-verification programming system that supports specifications of additional distributed aspects and synthesizes implementations of the intended distributed programs by incorporating optimal consistency protocols in message-passing implementations [Kuraj and Solar-Lezama, 2020]. In addition to full replication (where all nodes replicate the application state), we extend the system to support *partial replication*, where users specify how certain parts of the application data are split among nodes, while other parts are replicated [Belaramani et al., 2006]. Partial replication is a requirement for modern distributed applications, which rely on sharding and fine-grained allocation on the backend to achieve security and performance [Stonebraker, Madden, et al., 2007]. It also enables "local-first" applications on the front-end [Kleppmann et al., 2019]. The expressiveness of bounded verification allows SyCoord to support distributed programs that exhibit transactional behaviors and achieve speedups over the strongly consistency baseline.

Limitations

The main limitation of the SyCoord approach is its inability to provide soundness guarantees in the general case, where the bound on the number of operations cannot be guaranteed. (Some usecases, such as a multiplayer back-end, where only a limited number of operations are issued by each player at any given time, indeed fall into the class of programs for which SyCoord can optimize consistency and keep strong guarantees at run time.) In principle, the extent of SyCoord's verification depends on the checker with which it is parametrized. A stronger checker could be paired with SyCoord's consistency reasoning to detect consistency violations fast, but verify classes of unbounded traces for which no violating trace could be found, achieving soundness at run time. However, verifying expressive application-specific properties of distributed programs, with a high degree of automation remains an open challenge [Lesani, Bell, and Chlipala, 2016; Wilcox et al., 2015]. The exploration of expressiveness in SyCoord lead us to develop a new run-time approach that can guarantee soundness based on bounded analysis results but also use the dynamic information at run time to recognize additional consistency optimization opportunities.

1.1.2 Peepco

The goal of the second system is to address the shortcomings of SyCoord and obtain a strategy that can provide strong guarantees at run time. Prior work has introduced the idea of using static analysis in an end-to-end approach to identify when operations are allowed to safely execute under weaker notions of consistency to optimize performance [Sivaramakrishnan, Kaki, and Jagannathan, 2015; Houshmand and Lesani, 2019; Kaki, Earanky, et al., 2018; Gotsman et al., 2016; Kaki, Priya, et al., 2019; Bailis, Fekete, et al., 2014; C. Li, Leitão, et al., 2014]. However, these strategies are conservative and miss optimization opportunities based on the dynamic context of the operation currently being executed. While certain properties, such as commutativity and $\mathcal{I}$-confluence [Bailis, Fekete, et al., 2014], hold inductively, on any reordering of unbounded sets of specific operations, to guarantee integrity in the presence of diverse sets of operations in the worst case, there is no choice but to issue operations under strong consistency. This, however, forces convergence of states across nodes and limits the set of interleavings allowed in the system, and thus performance. The inability to reason about global states a distributed program can exhibit at run time limits the extent to which developers can optimize their distributed applications while guaranteeing the desired level of integrity.

Peepco presents an automated approach that uses symbolic reasoning over executions of bounded numbers of operations to derive efficient run-time checks that can be used to decide the optimal consistency at run time, for a given set of concurrent operations. We define optimal consistency as the weakest consistency (which allows the largest set of different execution traces) needed to maintain a set of semantic properties given by the programmers. The approach is implemented with a two-step process: we statically infer optimal consistency requirements for executions of bounded sets of operations, and then, use the inferred requirements to parameterize a new distributed protocol to relax operation reordering at run time when it is safe to do so. Our approach provides two key advantages over the purely static approach. First, we can exploit fine-grained optimization opportunities that only occur in specific situations at run time. Existing techniques must revert to strong consistency whenever there exists some interleaving of operations that violates integrity, even in cases where those operations do not get invoked together at runtime. Second, the verification burden of our approach is smaller than other approaches that must verify that the chosen consistency level maintains integrity for *all* possible inputs and unbounded interleavings. Instead, our approach checks that integrity is maintained for bounded executions and uses this information, together with the dynamic information obtained from the run-time checks, to perform optimization, safely and profitably.

We present a new consistency model, implemented with a new blocking run-time protocol that uses static information about consistency requirements between operations, as well as the agreement between nodes in the system, to attain information needed for applying consistency optimizations at run time and recognize more optimization opportunities when compared to the state of the art. To better facilitate our consistency optimization strategy, we introduce *batch-based consistency*, which, given a set of semantic properties, defines a specialized consistency model. We implement the approach in PEEPCO. PEEPCO is a consistency optimizer that takes the definition of a sequential object, data allocation specification, and integrity properties defined as predicates on the object state, and synthesizes a message-passing implementation that respects the allocation specification and executes object methods as distributed operations. Our evaluation shows our approach can bring significant performance benefits over the state of the art, especially in cases where operations might conflict only with relatively smaller portions of other operations, under high-contention workloads, exhibiting throughput and response time speedups across various benchmarks, when compared to prior work approaches, as well as when compared to a strongly consistent baseline.

1.1.3 System Integration

This thesis centers on optimizing consistency using PEEPCO and SYCOORD. Additionally, we showcase the integration of these approaches into a comprehensive system for developing distributed programs. While these systems form the central part of consistency optimization in development, they specifically introduce static inference of consistency constraints and the runtime protocol, which constitute only a segment of the entire system. To illustrate and assess these approaches, we constructed supplementary components essential for achieving end-to-end programming in the adopted model. These components can also function independently to develop distributed programs without managing consistency through integrity. When necessary, programmers can activate the consistency optimization segment of the system pipeline by incorporating integrity specifications in their programs. We detail the requisite components, both at the frontend and the backend, and outline the design and implementation of the overall end-to-end programming system. This system design supports the integration of SYCOORD and PEEPCO, along with extensions to handle additional distributed aspects.

From this point onward, we will use PEEPCO to denote both the entire system and the two-step consistency optimization approach with runtime protocol support, making clarifications as necessary.

```
class Account(b: Int) {
 var bal: Int = b; // current balance
 var max: Int = 50;
 var last = None;

 def withdraw(x: Int): Boolean = {
  if (x > max || x > bal) return false;
  bal -= x; last = Some(x); return true; }
 def setMax(v: Int) { max = v; last = None }
}
```

Figure 1.1: Bank account class

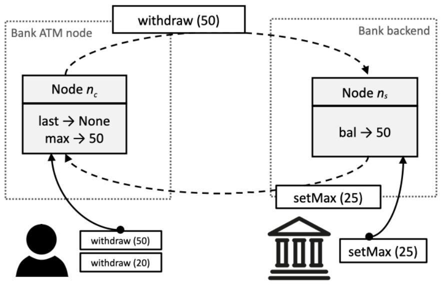


Figure 1.2: Distribution

1.2 Illustrative Example

In this section, we introduce the key functionalities of the approach and its extensions through a concrete example. We will describe functionality of both of the proposed systems, and the programming model extensions, within one general system, PEEPCO, and defer details of the particular approaches to specific chapters that expand on the same motivating example. While SyCoord shares the programming model and most DSL (domain-specific language) constructs with PeepCo, programmers have the flexibility to use a special keyword to designate the consistency optimization backend for generating final implementations. Initially, we showcase the capabilities of consistency optimization in PEEPCO, followed by a brief discussion of extensions to the programming model.

We consider implementing a simple but illustrative distributed bank account in PEEPCO. A bank object supports operations like deposits and withdrawals while storing the current balance and a history of performed methods. The case study was used in prior work on reasoning about consistency [Gotsman et al., 2016; Sivaramakrishnan, Kaki, and Jagannathan, 2015; Kaki, Earanky, et al., 2018]. We adapt and expand the example to demonstrate consistency optimization based on end-to-end integrity properties when a data allocation scheme is specified at a finer granularity. While this motivating example is simple, it shows what challenges arise in consistency optimization given end-to-end integrity properties, when data is replicated and split across nodes. (Expressive data allocation schemes are common in modern applications, both at the backend [Stonebraker, Madden, et al., 2007] and front-end [Kleppmann et al., 2019].)

Let us assume we want to implement the banking application in Figure 1.1 and distribute it across ATMs (clients) and a bank server (backend; Figure 1.2, nodes n_c and n_s). In this application, the bank would like to make sure each account holder is allowed to withdraw only within some specific amount assigned to the ATM. The application consists of a single distributed `Account` object that maintains the state of a user's account. The `Account` object

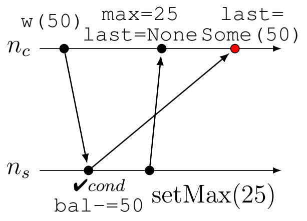
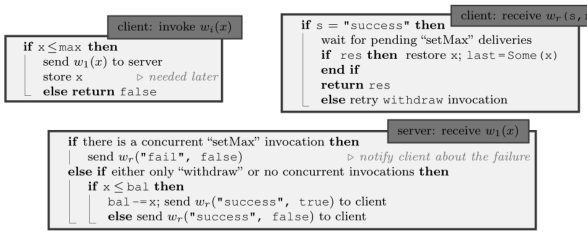


Figure 1.3: Execution



```
client: invoke $w_i(x)$
if x ≤ max then
    send $w_1(x)$ to server
    store x                ▷ needed later
else return false
```

```
client: receive $w_r$(s,res)
if s = "success" then
    wait for pending “setMax” deliveries
    if res then restore x; last = Some(x)
    end if
    return res
else retry withdraw invocation
```

```
server: receive $w_1(x)$
if there is a concurrent “setMax” invocation then
    send $w_r$("fail", false)                ▷ notify client about the failure
else if either only “withdraw” or no concurrent invocations then
    if x ≤ bal then
        bal -= x; send $w_r$("success", true) to client
        else send $w_r$("success", false) to client
```

Figure 1.4: Message-passing code

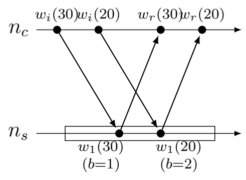


Figure 1.5: Safe execution

maintains the withdrawal limit max, the current balance bal, and the amount withdrawn last. We want to ensure the following integrity properties are maintained: no overdrafts (thus withdraw modifies bal only if it is big enough) and the last recorded withdraw should not be greater than max. In the following, we will focus on the latter, revisiting overdraft later.

1.2.1 Integrity of Distributed Programs

Assuming a client-server architecture, if developers want to distribute the bank account, the first question they are faced with is that of data allocation. We assume the ATM shows the last transaction to the user, so we allocate last and max on the ATM, and bal on the bank server.

Distributed execution. During distributed execution of withdraw, due to data allocation, the **if** condition checks data allocated on both nodes. When withdraw gets invoked on the ATM, to preserve semantics of the sequential code, the execution blocks and communicates with the backend node that contains the rest of the needed state. Without any ordering constraints, the application might exhibit an execution that violates our integrity property (Figure 1.3, with the initial state of Account(50)). Here, setMax is invoked concurrently with withdraw. withdraw read the initial value of max and writes a value to last that is greater than the current max, due to delayed message delivery (red dots designate integrity violation). We can prevent this with a straightforward solution, by enforcing strongly consistent executions for all invocations across the two nodes; however, this would prevent executing any concurrent invocations, and hurt performance. (Note that, while some systems support strong consistency with highly concurrency executions, with techniques such as sequencing, we assume the general case in a distributed setting, where this entails coordination across the nodes in the system [Bailis, Fekete, et al., 2014; Gray and Lamport, 2006; Thomson et al., 2012].) In this example, however, *e.g.,* multiple withdrawals and setMax invocations from ATM nodes can be issued concurrently and be serialized only on the server (without requiring global coordination), as long as updates from operations are not delivered in different orders on any n_c and n_s.

```
withObject { case (a: Account) =>
 withNodes { case (c: Node,
 s: Node) =>
  allocate((a.max, a.last), c)
  allocate(a.bal, s)
  ensuring(safe(lastOK(a)))
}}
```

Figure 1.6: PEEPCO specification

Message-passing implementation. We can prevent the integrity violation and avoid forcing coordination by ordering an appropriate part of `withdraw` when implementing it as a message-passing program. (A message-passing program allows programmers to carefully control the possible execution interleavings.) Here, one valid execution trace with two concurrent invocations (concurrency bound b is 2), which preserves integrity, is given in Figure 1.5. It checks for "ordering conflicts" on the server. In the generated implementation, `withdraw` is implemented using three message handlers: w_i, w_1 and w_r; we show the pseudocode of the implementation in Figure 1.4. The `withdraw` handler w_i runs when `withdraw` is invoked on the client, w_1 when the request is received on the server, and w_r when the client receives the response. Note that w_1 serves two purposes: it modifies the balance per the withdrawal, and it ensures that no concurrent invocation of `setMax` has been issued (and can write "out of order"), which maintains the application invariants.

A full manual message-passing implementation of the example consists of six message handlers, while the programmers must reason about every way that these message handlers can interleave to ensure that application invariants are maintained. Moreover, w_1 checks if there is no concurrent `setMax` invocation across the system—such a check involves distributed data requires coordination in the general case, which programmers have to recognize manually [Lamport, 1978a]. (We present how PEEPCO's protocol generalizes to any number of nodes in Chapter 2.) PEEPCO automatically identifies this potential violation and ensures that state changes of `withdraw` and `setMax` are delivered in the same order on both n_s and n_c, avoiding coordination when possible, allowing a higher degree of concurrent executions. (This goes beyond static pessimistic consistency approaches, in which strong consistency and coordination would need to be used due to the worst case scenario of concurrent executions with `setMax` [Sivaramakrishnan, Kaki, and Jagannathan, 2015].)

Next, we show how programmers can use PEEPCO to automatically generate distributed object implementations that are correct-by-construction and take advantage of application semantics and allocation to allow concurrency, in cases where it cannot violate integrity properties.

Invariant-based consistency. To use PEEPCO, programmers specify the application's

```
class Account(...) { // code as before
 var hist: List[(String, Int)] = ...

 def deposit(x: Int) = {
  bal += x; hist += ("d", x) }
 def withdraw(x: Int): Boolean = {
  if (x > max || x > bal) return false;
  bal -= x; last = Some(x);
  hist += ("w", x);
  return true;
}}
```

Figure 1.7: Bank with history

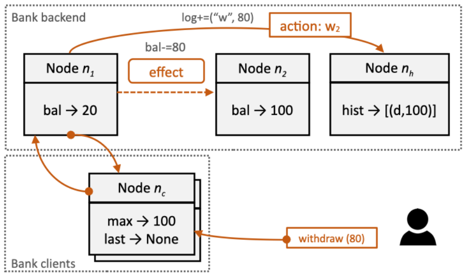


Figure 1.8: Distribution

integrity with invariants. PEEPCO will allow reorderings of operations to optimize consistency and improve performance as long as these invariants are maintained. The main insight behind PEEPCO is that, for many practical properties, as long as the integrity properties are not violated, the implemented system can allow many different reorderings of operations and thus avoid using costly protocols for total order. We write the specification in Figure 1.6, and provide it to PEEPCO along with the bank object definition (Figure 1.1). Given the application description and invariant specification as input, PEEPCO generates an implementation that takes advantage of the consistency optimizations discused previously, while preserving the application's integrity.

The specification in Figure 1.6 describes two key properties of the banking application: allocation of fields to nodes in the system and the invariant that expresses the integrity properties discussed above. First, it specifies the objects to be distributed: here, instances of the class `Account`. The `withNodes` construct specifies the nodes in the system. Here, the two nodes captured in the specification are a server and a client. Object fields are assigned to nodes using **`allocate`**. We place `last` and `max` on the client, and `bal` on the server. Invariants are specified using **`ensuring`**. Our invariant is expressed with *safe(p)*, which is a function that takes a Boolean argument p, representing the integrity check on the state of the system that can reference replicated state across multiple nodes. For this application, we specify a Boolean predicate method `lastOK`:

```
def lastOK(a: Account)} =
  a.last match {
    case Some(amt) => amt < a.max
    case None => true
  }
```

We then use the predicate to define *safety* [Lamport, 1977-03]—it must hold at every point in the execution trace to maintain integrity:

Data replication and splitting. We now show how PEEPCO's DSL can be used to extend our distributed application and change the way the data is distributed. We make the following application changes: we add a deposit operation, a history of operations hist, and update withdraw to account for hist (Figure 1.7). Next, we replicate balance onto multiple nodes (for availability) and store the history (hist) on a separate "audit" node. (If hist is large and non-critical, replicating it could hurt performance [Stonebraker, Madden, et al., 2007].) To change the data distribution strategy, in PEEPCO, programmers only need to change the specification, without changing the rest of the program.

A withdraw operation in this new configuration is shown in Figure 1.8. Due to replication, whenever an operation modifies replicated state, it generates an *effect* that contains that modification of the replicated state, which is propagated to all other replicas (containing a copy of the replicated state). Let's consider how PEEPCO distributes withdraw in this replication scheme. An execution trace for the invocation where x is 40 is shown in Figure 1.9. After the if condition check passes on the replica, updating its balance, the execution communicates with n_h to update hist on n_h, and additionally, concurrently, propagates an effect to the other backend replica (bal-=40). This introduces the possibility of a new kind of invariant violation: if withdraw is executed concurrently on two replicas n_1 and n_2, both checks pass, their update effects get passed on to the other replicas, and the account gets overdrafted (red dots in Figure 1.9). While both effects pass the balance check locally at replicas, cummulatively they subtract more from the balance than it is available. To prevent this, we define a no-overdraft integrity property with a predicate that uses bal and the sum of entries in hist (where the strings denote the operation type in hist):

```
def noOverdraft(a: Account) = {
  0 <= a.bal && 0 <= a.hist.fold(0)({
    case (res, ("d", x)) => res + x
    case (res, ("w", x)) => res - x
  })
}
```

We add this predicate as a safety invariant to our new specification. This specification achieves the previously described allocation scheme: a NodeSet is a group of replicas n and a Node which only has one instance in the system. We replicate bal onto the group n and assign hist to the history node nh. Then, we specify our new safety invariant using **ensuring**. With (executable) predicates, programmers need not learn a new specification notation, but can use the full expressive power of the programming language (here, the higher-order function fold[1]), to specify complex properties that are used to optimize consistency. (When using symbolic verification, however, the checked properties need to fall within the decidable fragment supported by the underlying SMT solver.) This goes beyond prior work that requires

[1]When programmers use PEEPCO with symbolic reasoning, without concrete inputs, the expressiveness is limited by the underlying used SMT solver.

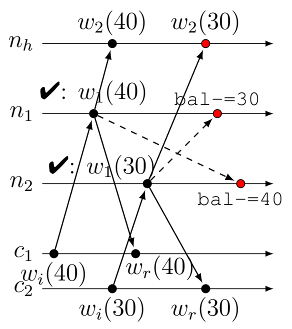


Figure 1.9: Two withdrawals

```
withNodes { case (n: NodeRegion,
 nh: Node) =>
 withStore { case (a: Account) =>
  replicate(a.bal, n)
  allocate(a.hist, nh)
  ensuring(safe(noOverdraft(a)))
 }
}
```

Figure 1.10: Replication

specifying low-level contracts that operations need to follow [Sivaramakrishnan, Kaki, and Jagannathan, 2015].

Next, we will show how PEEPCO optimizes consistency and prevents executions that might violate integrity.

1.2.2 Consistency Optimization

Consistency inference in PEEPCO and SYCOORD consists of two steps: static-analysis and run-time implementation generation. The first step is consistency inference, a compile-time step that splits operations based on data distribution and infers consistency constraints for operations. The second step then emits a message-passing implementation, based on the results of the first step, that can utilize two types of strategies. These strategies dictate the distributed protocol usage at run time, which in turn leverage inferred constraints to relax consistency:

1. in PEEPCO, a new *PeepCommit* run-time protocol uses agreement between nodes to obtain dynamic information and optimize consistency when it is safe to do so

2. in SYCOORD, the implementation consists of a set of standard unbounded consistency protocols that can only guarantee integrity if the number of concurrent operations is within the explored bound

The overview of the system at run time is shown in Figure 1.12. Operations are triggering with an external, or an internal, condition-based trigger (condition-based triggers of behaviors in our model is described in Chapter 4). Emitted message-handlers interact with the run-time protocols to invoke the application logic, such that they implement the ordering as prescribed by the inferred consistency constraints. We describe the two choices of protocol run-time strategies, *i.e.,* backends for the emitted implementations, subsequently.

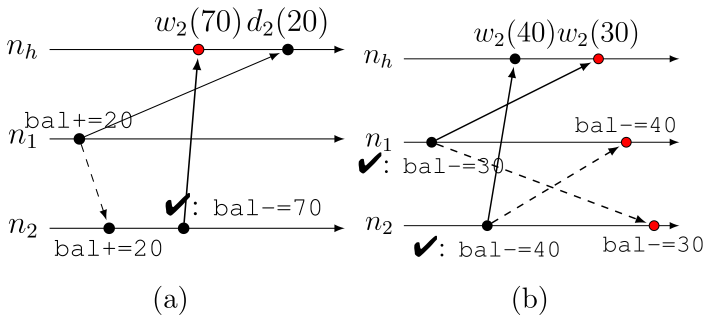


Figure 1.11: True and false violating trace

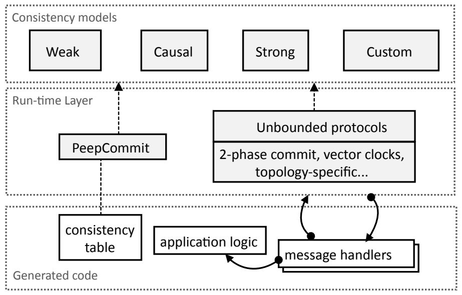


Figure 1.12: Backends

Consistency Inference. PEEPCO infers consistency constraints by exploring different operation invocations up to a bound. The inference procedure then generalizes results obtained from a bounded set of traces. We will describe the concolic strategy PEEPCO can employ, but PEEPCO also offers both symbolic (described Chapter 2) and concrete exploration (which we describe as a part of the general inference algorithm in Chapter 3). PEEPCO starts with a concrete initial state and concrete arguments to the `withdraw` calls, but checks viability of traces for any given values (by means of SMT solving) and infers consistency requirements that hold for any concrete values (and, also, unseen traces). The initial state can be provided by the user, or PEEPCO can automatically generate one from the given invariants.

Let us consider the case when two clients concurrently invoke `withdraw`, where the bank account's balance is initialized to `50`. PEEPCO splits `withdraw` into *actions* (parts of the overall method) w_i, w_1, w_2, w_r (preserving its sequential semantics), as shown in Figure 1.9. w_i, w_1, w_r actions correspond to the ones we described before: invoking action w_i, action checking `bal` and modifying it on a replica w_1, and response on the client w_r. w_2 is a new action that adds an entry to the history log on n_{hist}. We say that an action produces an *effect*, such as `bal-=40`, if the action modifies a replicated variable (as in computation models adopted in prior work [Sivaramakrishnan, Kaki, and Jagannathan, 2015]). PEEPCO analyzes possible interleavings between the actions.

One possible interleaving of concurrent operations is depicted in Figure 1.9. When `withdraw` is invoked, after checking the condition and executing the withdrawal, that effect is propagated to the other replica that contains `bal`. The overdraft scenario discussed previously, of delivering effects of two concurrent withdrawals, leads to `bal=-20` on replicas, thus an invariant violation (depicted with red dots in Figure 1.9). Once PEEPCO finds this trace, with the `noOverdraft` invariant violated, it analyzes the trace to determine the cause of the violation. Here, the effects from the independent concurrent `withdraw` invocations are observed in different orders on n_1 and n_2. The safety invariant `noOverdraft` is thus violated: while individual withdrawal amounts are lower than `bal`, `bal` becomes negative when both are subtracted (`70` in total, as described earlier). Since these methods are not causally related (the two `withdraw` invocations originated on two separate replicas), after analyzing the interaction of operations and identifying the reordering, PEEPCO infers that `withdraw` actions and their effects on n must be strongly consistent (preventing generating the concurrent effects `bal-=40` and `bal-=30`). It adds this constraint to its set of *consistency constraints*, which describe the consistency relationships between actions on particular nodes. This prevents the found source of inconsistency for the violating trace but also for other traces that exhibit such reordering, for all possible concrete values.

PEEPCO continues exploring executions from the same initial state until it discovers the trace in Figure 1.11*(a)* (which only shows nodes that are relevant to the violation). In this trace, the causally dependent (according to the definition of causal order [Lamport, 1978b]) effects of `deposit(20)` and `withdraw(70)` are reordered on the receiving replica. Specifically, `withdraw` was delivered without the delivery of the causally dependent `deposit`. Note that while the integrity constraint gets restored on n_h after delivery of $d_2(20)$, the overall trace violates safety. To eliminate this bad reordering, PEEPCO adds a constraint that `withdraw` causally depends on `deposit`. PEEPCO supports standard strong, causal, and weak consistency models as well as extensions for new models (described in more detail in Chapter 2 and Chapter 3). PEEPCO's analysis terminates when it has explored all traces up to a given bound (or a timeout) and verified they cannot violate the invariant, or they can be pruned as infeasible due to the inferred consistency constraints. Verification confirms integrity for any concrete value, by checking if the safety predicate always holds, *i.e.,* at each step, after executing the trace symbolically, with an SMT solver.

Agreement-Based Run Time Protocol. PEEPCO using the PeepCommit strategy executes operations in batches. It uses the statically inferred constraints to parametrize the run-time checker with a "consistency table", which is used at runtime by the PeepCommit protocol to dynamically relax consistency. Intuitively, the table maps sets of operations to the consistency requirements for those operations. Specifically: 1) if given operations can be executed concurrently within a batch, and, 2) under which consistency operations need to execute if added to the batch.

In the synthesized implementation, `withdraw`, started on node n_1, first checks if none of the operations that might be conflicting, started executing on n_1. If the operations cannot conflict, based on the table, it starts an agreement protocol (akin to two-phase commit) on the replicas in n, acting as the coordinator. It proposes a new `withdraw` to be executed. If all participants agree that `withdraw` can be added for the execution, the withdrawal gets executed from the node that originated the call. A withdrawal, however, will not be allowed to be executed concurrently in the given batch, if the batch contains another invocation of withdrawal. Therefore, for multiple `withdraw` that are invoked concurrently, PEEPCO enforces total order. In particular, given the found constraints between `withdraw` and `deposit`, one withdrawal and multiple concurrent `deposit` invocations are permitted to execute in the same batch, as long as all withdraw effects are executed causally relative to witnessed deposits. Figure 1.12 captures the interaction in this emitted scenario with the *PeepCommit* protocol. The protocol gets invoked from the generated message handlers. Every operation invocation invokes the protocol, which, after checking the consistency table and performing the agreement, calls the appropriate messages handlers in case the operation was accepted for the execution.

Bounded Consistency Optimization Run-time. Alternatively, the SYCOORD backend can synthesize distributed implementations that leverage unbounded protocols, based on the two found constraints. PEEPCO uses its repository of message-passing consistency protocols to enforce each of the discovered consistency constraints between operations: two-phase commit (2PC) ([Gray and Lamport, 2006]) and vector-clock dependencies ([Fidge, 1987]), for strong and causal consistency, respectively. SYCOORD handles the two mentioned consistency levels, in addition to issuing operations without any ordering constraints (weak consistency). (The support for expressing consistency levels and integrating new protocols is described in Chapter 3). In our example, the synthesized implementation uses 2PC to enforce strong consistency between withdrawals on n_1 and n_2 and additionally makes sure all actions from causally related deposits are executed before delivering withdrawals (on both the replicas n_1, n_2, and the history node n_{hist}). As weak and causal consistency do not affect other invoked operations in the system, the correctness cannot be be guaranteed if the number of operations that gets invoked concurrently in the system becomes greater than the largest explored bound. These consistency constraints are fixed, regardless of the number and type of operations running concurrently in the system, thus only guaranteeing integrity if concurrent operation calls can be bounded by some other means.

Performance of the emitted implementations. Given the input program, PEEPCO infers the needed consistency requirements and synthesizes an appropriate message-passing implementation. In the case of SYCOORD, the implementation, while not verified for arbitrary bounds, is correct for any number of concurrently invoked operations in the system, and the system preserves integrity. For our running example, PEEPCO's inference completes within 15 minutes covering all combinations of up to 7 concurrent operations that are sufficient for significant performance gains at run time. The synthesized code achieves more than 2.5x speedup (for 960 clients) and more than 4x speedup (on a microbenchmark with 12 clients), in terms of throughput, when compared to issuing operations with strong consistency, in the case of Peepco and SyCoord respectively. (We give more details about the evaluation of both systems in their respective chapters.)

1.2.3 New Point in the Design Space of Consistency Optimization

While prior work presented approaches that can leverage static analysis for consistency optimization, they either force programmers to reason about low-level operation effects [Sivaramakrishnan, Kaki, and Jagannathan, 2015], identify operation parts requiring specific consistency levels and annotate them manually [Milano and Andrew C. Myers, 2018], or rely on a worst-case analysis that might miss fine-grained optimizations at run time, due to both the dynamic information as well as the granularity of the partially replicated data [Houshmand and Lesani, 2019]. In our approach, programmers write end-to-end behaviors in the sequential model, specifying integrity with semantic properties given as predicates in the same language without requiring programmers to learn new constructs. Peepco automatically reasons about fine-grained consistency requirements of different combinations of the object methods that can occur at run time. In addition, our approach supports the expressiveness of partial replication, where portions of data might be split or replicated across the selected nodes in the system, and allows incorporating additional specifications into the supported programming model, such as information about the underlying network model (described in Chapter 4).

Challenges. To perform consistency optimization in a programming model under partial replication, Peepco: (1) splits operations based on data distribution, potentially examining different ways of splitting; (2) infers consistency constraints of individual operation parts; (3) incorporates the inferred information into the message-passing implementations and the run-time protocol. In addition to addressing the challenge of achieving consistency optimization based on results of bounded reasoning with a new protocol, one of the main challenges in Peepco is to achieve scalable and sound bounded consistency inference that allows effective and sound consistency optimization inference, expressive for various invocation scenarios that might occur at run time. Consistency inference in partial replication exhibits combinatorial blowup in terms of the traces to explore, as well as consistency levels, to consider, which makes full exploration practically challenging even for small scopes.

To handle the expressiveness of the distributed computation and finer granularity of consistency requirements, Peepco exhaustively explores bounded traces for the given data allocation under different invocation scenarios. Such granularity of inference, on the level of actions and nodes, goes beyond coarse-grained, single-node reasoning in prior work in the full-replication model [Sivaramakrishnan, Kaki, and Jagannathan, 2015; Kaki, Earanky, et al., 2018]. Moreover, approaches that analyze interactions between pairs of operations statically are not applicable here as some consistency requirements can only be exhibited after executing a specific combination of operations [Houshmand and Lesani, 2019]. Due to the partial replication, PEEPCO needs to discover that different actions that belong to the same method might have different consistency requirements. For example, PEEPCO determines that the effects of `withdraw` and `deposit` require causal consistency on both the replicas n and the history node n_h. However, while any two w_1 actions require strong consistency on replicas n, the corresponding w_2 actions ($w_2(10)$ and $w_2(80)$) require no coordination (on n_h): the reason is that a w_2 action can only be issued after the corresponding w_1 action has performed the overdraft check, due to previous discovery that w_1 actions have to be strongly consistent (as shown in (Figure 1.11*(b)*). PEEPCO's reasoning not only preserves the given invariants in granularity of the partial replication, but discovers *consistency requirement inter-dependencies*, which can achieve additional speedups relative to a coarse-grained operation-based consistency optimization.

A naive approach of exploring all possible traces, analyzing conflicts that lead to invariants being violated, and including all resulting constraints would not result in a useful solution. The returned solution would be too strong. Inferring conflicts from all traces which violate an invariant might lead to incorrect solutions: some traces might exhibit an anomaly we call *false conflicts*. Such traces are not feasible due to inter dependencies with other discovered consistency requirements and thus cannot lead to the discovery of new consistency requirements when explored. For example, reordering w_2 from different withdrawals on n_h, causes a violation (Figure 1.11*(b)*), but this is an example of a redundant conflict: a conflict that is possible when considered in isolation, but not possible when enforcing other consistency requirements, inferred from the rest of the (possibly yet unexplored) traces. Namely, the `withdraw` actions dispatched on n require strong consistency. As a consequence, the corresponding w_2 actions require no coordination on n_h, as they can only be issued after the overdraft checks are performed, under total order. Thus, while a strong consistency requirement can be discovered on n_h if such a trace is explored first, it should not be included in the final solution. Incorporating consistency constraints from redundant conflicts results in a solution that is overly strict: in this case, requiring strong consistency on n_h. An approach that relies on exploration of traces inevitably has to use a strategy that avoids discovering redundant conflicts, in order to guarantee finding the minimal set of consistency requirements.

Design decisions. Our approach is motivated by achieving separation of concerns in the input language. To the best of our knowledge, PEEPCO is the first approach that allows writing sequential operations and invariants in the same, general-purpose language, and specifying data allocation (at the granularity of partial replication), timing constraints, and the network model, decoupled from one another. Such *decoupling* allows programmers to easily change different aspects of the intended program and allows programmers to easily debug and tune consistency by making changes to different parts of their input program, separately. For example, moving `hist` to a separate node or any changes to the invariants only requires a local change to the configuration, A method can be added without changing existing invariants, and an invariant and specifications can be added without changing methods or the rest of the specification. This is despite the fact that any of these changes influences the set of actions and feasible or valid interleavings, which may significantly affect the required operation orderings at run time, and lead to completely different consistency mechanisms and synthesized implementations. Various changes to the same input programs and specifications from the running example retain the property that even for relatively short inference times, generated implementations achieve significant performance improvements in both variants of the run-time consistency optimization protocols. PEEPCO can also be used to interactively discover and show traces of violating executions (minimized to graphs that show conflicting actions; see Chapter 3), helping programmers in debugging consistency choices. PEEPCO's aim is to fit into the traditional development process, where programmers start by considering the operations, then proceed to reason and test the behaviors of those operations in the sequential model, and finally start thinking about other intended aspects of their programs, here, pertaining to distribution.

1.2.4 Extensions of the Main Approach

We now present extensions of the main approach that allow programmers to strengthen the desired integrity, specify reactivity of their distributed behaviors, and adjust the computation based on the underlying network the program is deployed on.

Convergence. In the running bank example, we might want to make sure no money can be created. For example, we might want to state the predicate `a.bal == sum(a.hist)`, which says that the balance should equal the sum of entries in the history. However, this predicate might only hold *periodically* on every trace (akin to "always eventually" [Pnueli, 1981]). It cannot be expressed as a safety invariant, as it cannot hold at every point. (It inevitably has to be violated during operation execution.) To allow for cases like this, PEEPCO supports convergence invariants which state that the system will eventually reach a state that satisfies this form of integrity, after some bounded set of operations finishes executing. Convergence invariants directly affect the resulting consistency model in case of the PeepCommit backend (described in Section 2.2), which makes sure that any execution restores convergence at the end of any batch of operations.

Network model. Peepco allows developers to model the network, specifying the cost of communication as well as execution on individual nodes. While some approaches allow mapping node computation to physical machines, these existing facilities become insufficient in many cases where the program itself and shape of its behavior need to be directly tailored according to the specific network at hand. (Peepco's design is influenced by the *end-to-end argument* [Saltzer, Reed, and Clark, 1984].) Given such a specification, Peepco then searches for the optimal shape of computation of each method, potentially allocating more computation to "stronger" nodes and changing how the computation propagates over the network. In the current example, since it is left unspecified, Peepco assumes the default network model, which specifies uniform shape, with costs of execution and communication being equal. Developers can specify custom network models by defining directed weighted graphs, which Peepco then uses to compute the cost and rank different shapes of behaviors (i.e. allocations of different parts of the computation to different nodes). Different network models can be globally set at run time, dictating the choices of computation shapes when methods are invoked. This extension is described in Chapter 4.

Sharding and specialized protocols. Peepco allows entity-based sharding [Baker et al., 2011], as well as consistency protocols that depend on the underlying network and data allocation. Programmers can specify sharding for certain supported data types, similarly to how replication is expressed, while Peepco automatically emits functions that choose the appropriate node of the shard at run time. In our example, programmers can decide to shard `hist`, as it might hold large amounts of data. Network-specific protocols allow emitting specialized protocols that are applicable in specific data allocation schemes and behavior shapes. A violating write to `last` in our example can be prevented by a specialized protocol for preventing "write skew", which does not need require costly strong consistency. It guarantees the needed ordering of actions of `withdraw` and `setMax`, prescribed by the inferred consistency constraints, while providing additional performance compared to general protocols (*e.g.,* by avoiding coordination, as described before). We describe the specialized protocols in Section 2.3 and Chapter 3, and the support for sharding in Chapter 3.

1.2.5 Implementation and Benchmarks

The final version of our system, Peepco, which includes both consistency strategies, is implemented in approximately 33k lines of Scala code, where 26k lines comprise the compiler and 7k lines comprise the run-time that is used by the generated programs. Symbolic and concolic checking in consistency inference is achieved by discharging formulas that describe execution traces to the Z3 SMT solver. The inputs to the system are sequential programs

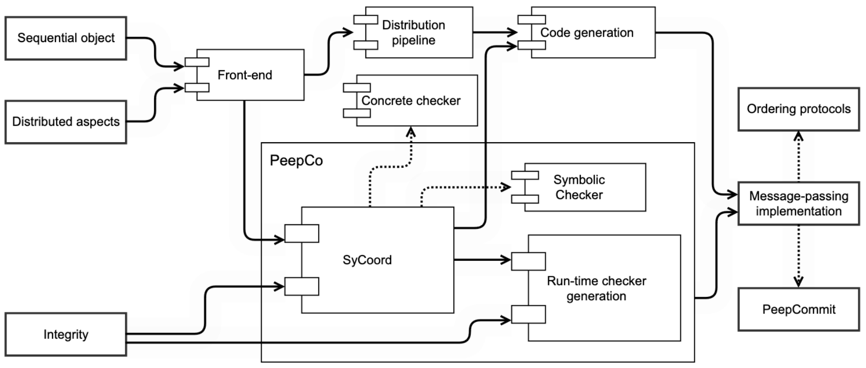


Figure 1.13: Architecture overview of the overall synthesis system

written in Scala and specifications written in Peepco's domain-specific language (DSL) embedded in Scala. The system outputs final implementations as message-passing programs in Scala that use networking facilities provided by the Akka framework [*Akka – actor toolkit and runtime, http://akka.io/* 2023], and are deployed together with the PEEPCO's run-time, on a Java Virtual Machine (JVM). Message sending and receiving, in the generated code and distributed protocols uses TCP-based communication channels.

We evaluated consistency inference in SyCoord and PeepCo on 28 and 18 benchmarks, respectively, that included conflict-replicated data types (CRDTs), models of relational databases, and application usecases that leverage different data replication and splitting schemes. In the case of SyCoord, the benchmark suite includes 1 benchmark that does not have a straightforward encoding in SMT logic and thus cannot be supported by PeepCo. We evaluated run time performance, in terms of throughput and operation response times, of emitted implementations in the case of consistency optimization backends SyCoord and PeepCo, as well as the EdgeC version of our system in which programmers specify the intended consistency per operation. For our our run-time performance evaluation, we picked representative case studies that were considered in prior work. We include includes run 4 case studies in under different settings in the case of PEEPCO, and a single case study (representing different variants of the running example) in the case of SYCOORD, and two case studies in the case of EdgeC. Our evaluation includes performance comparison of the emitted implementations to our manually developed baselines, as well as off-the-shelf data stores.

1.3 Overview

The system architecture for our end-to-end synthesis-aided programming system is shown in Figure 1.13. While PEEPCO and SyCoord represent two systems that capture two different evolution points in the development of the overall system for optimizing consistency of distributed programs, they make up two pipelines within the system. PEEPCO builds on top of the static analysis introduced for SYCOORD and instantiates it with a symbolic checker to enable reasoning about bounded execution traces symbolically. Programmers can choose different pipelines when interacting with the overall system: for PEEPCO, the pipeline will invoke the static analysis using the symbolic SMT-based checker, and in addition to emitting the implementation code, PEEPCO emits a "consistency table" that is used to parameterize the run-time protocol according to the inference results; for SYCOORD, the pipeline will invoke the static analysis using the concrete checker, requiring programmers to also provide concrete input values, while the code generator emits final implementations that use unbounded protocols. To support end-to-end programming with both of the mentioned consistency optimization pipelines, we have designed a programming model and a domain specific language, and implemented the code generation in the backend. We extend the system to support specifying reactivity and underlying network models by providing additional specification elements in the system's DSL and introduce additional processing elements, such as the cost analysis for computing best shapes of behaviors according to the underlying network models, to the system's pipeline.

This dissertation is organized as follows. We first present PEEPCO's full, unbounded, approach for guaranteeing sound executions at run time, by introducing its computation models and consistency specialization strategy, the run-time PeepCommit protocol, and the symbolic consistency requirements inference procedure in Chapter 2. We then present SYCOORD, expand on the consistency inference procedure, characterize its expressiveness, and describe the inference process using a concrete bounded checker, as well as the backend for emitting implementations that leverage unbounded protocols to guarantee consistency up to a specific concurrency bound in Chapter 3. In Chapter 4, we present EdgeC and explain the overall system design for supporting additional aspects such as reactivity, and implementation of the end-to-end programming approach that supports integrating consistency optimization procedures, as well the extensions for specifying the underlying network models. Chapter 5 compares the programming model, inference procedure, and the consistency optimization approach embedded in PEEPCO to approaches presented in prior work, as well as alternatives the programmers can choose for development of distributed programs.

1.3.1 Discussion and Limitations

Our approach can provide significant gains in performance in cases where the given operations of the object exhibit many conflicts in the worst case and executing subsets of those operations exhibits scenarios where consistency can be optimized. While the dynamic aspect of our new protocol allows it to apply such performance optimizations, it does this at the cost of paying for additional communication whenever an operation (that is not read-only) gets invoked. For applications that have a lot of operations that modify the state, but exhibit no conflicts, developers could identify the non-conflicting operations a priori and issue them without such additional costs.

In this thesis we demonstrate that integrity can be effective in eliminating bad executions by enforcing optimal consistency within the batch-based consistency model. However, to ensure the integrity properties are sufficient for the given program, for some classes of programs programmers might need to manually inspect the resulting executions. If, *e.g.,* effects of the operations simply assign new values to variables, instead of producing non-destructive effects, state updates in the system might get effectively nullified at run time. (This might occur as a result of overwriting a value that was updated by an effect produced concurrently.) Such executions would not be allowed by strong consistency, but might still preserve the given integrity, and thus be eligible in our model. Expressiveness of the supported integrity properties might not be sufficient to fully specify such programs, and programmers might need to resort to other means to ensure that their intended behaviors account for all the operation effects. Adding a node that contains the whole state and asserting equality between portions of the state with states on other nodes can alleviate the problem, albeit slowing down the inference.

Programmers that need to leverage partial replication need to be aware of the data flow that arises due to operation splitting, and check if that adheres to their security needs. Moreover, extending the approach with new protocols needs to be done with care (potentially, as library extensions maintained by domain experts), as any discrepancy between a newly introduced consistency model and the run-time guarantees of the protocol can lead to violating the overall correctness of the system and its guarantees.

1.3.2 Scientific Journey

The overall motivation for an end-to-end programming system came from the work on analyzing different programming models and frameworks, as well as its follow up work that motivated the programming model that leverages sequential semantics for distributed behaviors and discussed different features of modern distributed programs that such a programming system should support [Kuraj and Solar-Lezama, 2017; Kuraj and Jackson, 2016].

EdgeC represents the first iteration of our end-to-end programming system for writing distributed programs, which highlighted the need for consistency optimization, and resulted in a prototype that introduced multiple foundational components of the overall system. We presented the system design of an end-to-end programming system that supports specifying network models and computation costs, as well as consistency of different operations, and generates final implementations of distributed programs that employ consistency protocols based on the chosen operation consistency [Kuraj and Solar-Lezama, 2020].

The work on the second system, SyCoord, started in 2018 and concluded in May 2021. We presented a static bounded inference that supports application-level invariants and uses concolic and concrete exploration to find consistency requirements based on the safety specified by those invariants in [Kuraj, Solar-Lezama, and Polikarpova, 2022].

The last version in the system's evolution, Peepco, was motivated by achieving soundness for unbounded executions, and the work on it started in 2021 and concluded in November 2023 [Kuraj, Feser, et al., 2023]. PEEPCO added a new run-time protocol that is capable of reusing operation ordering requirements found by a SyCoord-based, purely symbolic, procedure, to perform optimizations at run time, while guaranteeing integrity through the use of distributed agreement. To that end, the work also introduces a new consistency strategy that parametrizes consistency models, which force bounded periodic convergence, based on the given integrity properties.

Some ideas and figures in this thesis have appeared previously in the aforementioned publications.

Chapter 2

PeepCo: Batch-Based Consistency Optimization

Programmers traditionally relied on strong consistency to preserve data integrity, which masks the intricacies of distributed execution over data allocated across a network of nodes, under unpredictable communication delays and concurrency, at the expense of runtime performance [Brewer, 2012]. Weaker consistency mechanisms offer higher performance but can cause application-level integrity violations if used naively [Sovran et al., 2011; Bailis, Fekete, et al., 2014]. Programmers are faced with the challenge of balancing integrity and performance when choosing the right consistency model given the requirements of their applications [Sivaramakrishnan, Kaki, and Jagannathan, 2015]. Moreover, implementing the resulting application tightly couples the application logic and distributed aspects, and any change to either of the two has to propagate changes in the consistency model through a costly development cycle [Viennot et al., 2015; Stonebraker, Madden, et al., 2007; I. Zhang et al., 2016; Kaki, Priya, et al., 2019]. Automating this choice and the subsequent implementation, while attaining high-level integrity provided by strong consistency, remains a challenge.

As mentioned in Chapter 1, while prior work has presented static analysis approaches for consistency optimization, avoiding conservative analysis at static time remains a challenge [Sivaramakrishnan, Kaki, and Jagannathan, 2015; Houshmand and Lesani, 2019; Kaki, Earanky, et al., 2018; Gotsman et al., 2016; Kaki, Priya, et al., 2019; Bailis, Fekete, et al., 2014]. The conservative nature of such analyses implies that they might miss certain optimization opportunities based on the dynamic context of the operation currently being executed. In this chapter, we present an automated approach that uses symbolic reasoning over executions of bounded number of operations to derive efficient run-time checks that can be used to decide the optimal consistency model at run time, for a given set of concurrent operations. We define the right consistency as the weakest consistency needed to maintain a set of semantic properties given by the programmers. Our approach provides two key advantages over the purely static approach. First, we can exploit fine-grained optimization opportunities that only occur in specific situations at run time. Existing techniques must revert to strong consistency whenever there exists some interleaving of operations that violates integrity, even in cases where those operations do not get invoked together at runtime. Second, the verification burden of our approach is smaller than other approaches that must verify that the chosen consistency level maintains integrity for *all* possible inputs and unbounded interleavings. Instead, our approach checks that integrity is maintained for bounded executions and uses this information, together with the dynamic information obtained from the run-time checks, to perform optimization, safely and profitably.

To better facilitate our consistency optimization strategy, we introduce *batch-based consistency*, which, given a set of semantic properties, defines a specialized consistency model. Batch-based consistency is parameterized over two kinds of application-specific integrity properties, *safety* and *congruence*, and generates a specialized consistency model that restricts the states the system can go through. A safety property must hold in every state. A congruence property must be restored periodically after a finite number of state changes. By specifying the two kinds of integrity properties, programmers specify the desired level of consistency, picking a point between eventual and strong consistency. This allows the programmers to limit the allowed discrepancy of states between nodes at any point, but it also weakens the strict notion of state convergence (which prescribes all nodes have to see the effects of distributed operations in the same order, as if the effects were produced by a sequential execution), opening up further opportunities for relaxing orderings of operations at run time. Consistency optimization can then optimize consistency at run time, as long as the execution maintains the integrity prescribed by the model specified by the programmers.

Batch-based consistency optimization. Our approach, *batch-based consistency optimization*, is implemented in the PEEPCO tool. PEEPCO is a consistency optimizer that takes the definition of a sequential object, data allocation specification, and integrity properties defined as predicates on the object state, synthesizing a message-passing implementation that respects the allocation specification and executes object methods as distributed operations. PEEPCO infers consistency requirements from the specification and synthesizes a distributed application by allocating data, splitting and allocating computation, and integrating the results into our new run-time protocol, to implement the batch-based consistency model with the specified integrity.

Our approach relies on a statically constructed oracle that determines if an operation can safely execute given dynamic information about the operations running in the distributed system. Our strategy relies on the insight that if we have such an oracle, we can leverage standard techniques for distributed agreement [Gray and Lamport, 2006] to relax consistency. Once all nodes agree on the operations that can be executed concurrently, their execution ordering can be relaxed to allow any of those the oracle deems safe. The oracle is then used to implement batched execution: given the set of running operations already in a batch, a new operation is allowed to execute in the same batch only if: 1) the oracle allows it to run concurrently, and 2) it follows execution order with respect to other operations in the batch, as specified by the oracle.

The strategy uses a new run-time distributed algorithm—*PeepCommit*—which operates in batches and uses the oracle to add as many *non-conflicting* operations to a batch as possible, falling back to strong consistency when the batch might contain conflicting operations. Our protocol is blocking and thus not resilient to process crashes (similar to "two-phase commit," which can use a write-ahead log for fault-tolerance [Gray and Lamport, 2006]). All nodes in the system consult the same oracle locally while maintaining safety and restoring congruence at the end of each batch. The correctness and performance of the strategy directly depend on the set of relaxed orderings known to the oracle. To construct the oracle, we introduce *integrity-driven consistency strengthening*, a new SMT-based bounded symbolic analysis algorithm for consistency inference for batch-based consistency. (*Integrity-driven consistency strengthening* represents an instantiated strategy for consistency requirements inference, which we present later, in Chapter 3.) The algorithm discovers the consistency requirements of bounded execution traces of operations derived from sequential objects at compile time. Our algorithm is efficient; it incrementally builds a set of consistency requirements and uses them to aggressively prune the set of execution traces to explore, in addition to using traditional symmetry-breaking techniques. This allows us to explore traces up to non trivial bounds, even when the integrity invariants allow many operation reorderings. While the bounded nature of our analysis means that it can miss optimization opportunities that arise at larger bounds than the one explored, it will not lead to unsound executions.

In addition to full replication (where all nodes replicate the application state), we extend our approach to support *partial replication*, where users specify how certain parts of the application data are split among nodes, while other parts are replicated [Belaramani et al., 2006]. Partial replication is a requirement for modern distributed applications, which rely on sharding and fine-grained allocation on the backend to achieve security and performance [Stonebraker, Madden, et al., 2007]. It also enables "local-first" applications on the front-end [Kleppmann et al., 2019]. Partial replication introduces new opportunities for optimization, which PEEPCO can take advantage of.

Contributions. In summary, this chapter makes the following contributions:

- *Batch-based consistency* (Section 2.2), a new approach for specializing consistency models.

```
var bal: Int = 0
var hist: List[(String, Int)] = []

def deposit(amt: Nat) {
 bal += amt; hist += ("d", amt) }

def withdraw(amt: Nat) {
 if (amt ≤ bal) {
  bal -= amt; hist += ("w", -amt) }}
```

Figure 2.1: Bank account in PEEPCO

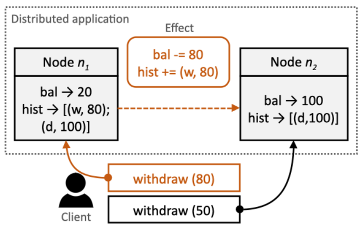


Figure 2.2: Execution snapshot

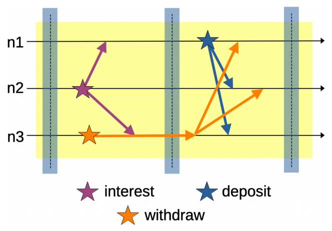


Figure 2.3: Specialized model

- A strategy for consistency optimization in batch-based consistency, consisting of 1. the *PeepCommit* protocol (Subsection 2.2.4), a new round-based distributed algorithm that relaxes consistency constraints within bounded sets of concurrently invoked operations, while maintaining integrity, and 2. the *integrity-driven consistency strengthening* algorithm (Subsection 2.2.5), an efficient, symbolic, bounded, static analysis that determines the weakest set of consistency constraints under which operations in the given set are safe to reorder.

- An extension of the strategy that allows for consistency optimization when data is replicated only partially (Section 2.3).

- An implementation of PEEPCO as a Scala DSL that allows programmers to orthogonally specify behavior, data allocation, and integrity, and an evaluation (Section 2.5) of PEEPCO on benchmarks, both new and drawn from prior work, that compares the performance of the resulting distributed implementations to relevant baselines, as well as identifies conditions under which PEEPCO can achieve significant speedups.

2.1 Motivation

We now show how developers use PEEPCO to synthesize a simple, but illustrative, replicated application ([Sivaramakrishnan, Kaki, and Jagannathan, 2015]) and achieve performance and desired integrity in batch-based consistency. We will reuse and expand upon the motivating example introduced in Chapter 1.

Distributed Model. We consider a bank account object (Figure 2.1), which we have adopted from [Sivaramakrishnan, Kaki, and Jagannathan, 2015], that has a balance `bal` and an ordered history `hist` that tracks performed operations, reflecting the value in `bal` when reduced by the `fld` fold function. (`fld` simply folds over the list, applying its operations in order, starting from 0.) The object has three methods: `deposit`, `withdraw`, and `interest`. The methods modify `bal` accordingly and record operations by adding entries to `hist`. In addition, withdraw performs a check to avoid overdrafting the account. We define interest subsequently.

We deploy the bank account as a distributed application where the object is *replicated* onto the nodes n_1 and n_2. Let us consider execution when PEEPCO's consistency optimization is not applied. Clients invoke methods, which then get dispatched to arbitrary nodes. Figure 2.2 shows a client that calls withdraw twice: the first is dispatched to n_1 and the second to n_2. After a replica receives a call, it produces an *effect* (a state update function, closed over the parameters), applies the effect locally, and propagates it to other replicas over the network.[1] Here, starting with the state bal:100 and hist:[(d,100)], withdraw(80) is dispatched to n_1, the overdraft check passes, so n_1 produces and applies the effect bal-=80 locally, resulting in bal:20. Figure 2.2 shows this intermediate state where the effect was sent to n_2 (shown with a dashed line). If the effect bal-=80 reaches n_2 before the second call withdraw(50) is issued, n_2's bal is updated to 20, before the overdraft check of the call fails, producing no effects. However, if withdraw(50) is issued concurrently, its check passes and a second effect bal-=50 will be applied at n_2 and sent to n_1. When both effects are applied, on n_2, the account becomes overdrafted (state bal:-30, hist:[(w,-50);(w,-80)]). While both effects pass the local overdraft check, cumulatively they subtract more from bal than is available. The overdrafted state is caused by reordering due to concurrent executions, and we want to avoid it at all times, on all nodes.

Now we consider the interest operation, which sets bal to a percentage x of the current bal value: bal * x / 100. A concurrent invocation of interest with deposit or withdraw cannot cause an overdraft but might result in different values of bal on different replicas, as the methods are not commutative. After all methods finish their execution, we would like bal, and the value reflected by hist, to be the same across all nodes.

To eliminate such executions, we could use strong consistency [Harding et al., 2017; Abadi, 2012]. Prior approaches for optimizing consistency in the eventual consistency model [Sivaramakrishnan, Kaki, and Jagannathan, 2015; Kaki, Earanky, et al., 2018] require replicas to converge to equal values and thus would force strong consistency to prevent reordering in hist. Static consistency optimization approaches [Kaki, Earanky, et al., 2018; Sivaramakrishnan, Kaki, and Jagannathan, 2015; Houshmand and Lesani, 2019] need to assign strong consistency to all non-read method calls to ensure equal copies of bal, due to interest not being commutative with any of the other two methods. However, this eliminates concurrent executions even when concurrency is safe, *e.g.,* multiple deposits or

[1] We adopt the operation-centric model of replication with effect propagation [Burckhardt et al., 2014; C. Li, Leitão, et al., 2014; Shapiro et al., 2011a; Sivaramakrishnan, Kaki, and Jagannathan, 2015; Kaki, Earanky, et al., 2018; Kaki, Priya, et al., 2019].

```
replicate({bal, hist}, N)

safe ∀n ∈ N. def i1 =
 bal[n] ≥ 0 && fld(hist[n]) ≥ 0

congr ∀na, nb ∈ N. def i2 =
 bal[na] == bal[nb] &&
  fld(hist[na]) == fld(hist[nb])
```

Figure 2.4: PEEPCO specification

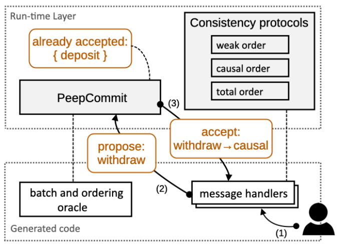


Figure 2.5: Run-time calls

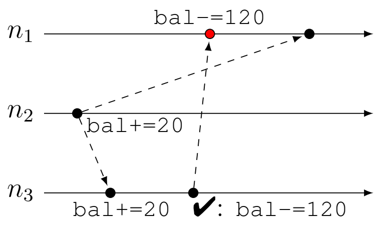


Figure 2.6: Causal ordering trace

calls to `interest` in a row. In contrast to prior work, PEEPCO specializes the consistency model to allow safe concurrent executions by dynamically optimizing consistency, falling back to strong consistency only when necessary.

Batch-Based Consistency. We specify integrity properties (Figure 2.4) with PEEPCO's Scala embedded DSL. First, we replicate `bal` and `hist` on all nodes in the system (variable **N**). Then, we define a *safety* property: `bal` and the fold of `hist` are always non-negative. The syntax `bal[n]` refers to the local copy of `bal` on node `n`. Finally, we define a *congruence* property: copies of `bal` and folds of `hist` (starting from 0) are equal in congruent states. Note that this property requires equality only between all replicated `bal`, but not `hist`.

Using these properties, PEEPCO generates a distributed program that executes methods in batches, according to the specified integrity. Methods within a batch may be executed concurrently as long as all safety properties are maintained and congruence is restored at the end of the batch. Figure 2.3 shows an execution trace; points on the trace where safety properties hold are highlighted in yellow and points where congruence properties hold are highlighted in blue. Method calls being dispatched are depicted with a star (they do not immediately start executing), with brown, red, and green designating `interest`, `withdraw`, and `deposit`, respectively. PEEPCO prevents concurrent executions of withdrawals, due to safety, and concurrent `withdraw` and `interest` calls, due to convergence. In the trace, `withdraw` and `interest` are invoked concurrently, but the `withdraw` is deferred to the second batch, where it executes concurrently with a deposit. Due to congruence not requiring equality between replicated `hist`, the system will allow interleaving executions of multiple `deposit` and `interest`. Any reordering between these operations cannot violate congruence due to commuting between themselves.

Batch-based consistency is implemented using PeepCommit, a new run-time protocol for batched execution. Figure 2.5 shows the interaction between the generated code and PeepCommit in the second batch, where `withdraw` is invoked after the node had already seen `deposit`. Upon invoking `withdraw`, the node invokes PeepCommit to check with the other nodes whether `withdraw` can be added to the current batch, and with the oracle under which order it needs to execute with respect to `deposit` that is already in the batch. After all nodes agree to add `withdraw` into the batch, the oracle mandates `withdraw` execute under causal ordering (which means it can be delivered on other nodes, only after all causally-dependent operations [Lamport, 1978b]). Message handlers generated by PEEPCO then execute it locally and propagate its effect to other nodes, using consistency protocols provided by PEEPCO's run-time.

PEEPCO obtains the oracle by symbolically exploring all bounded execution traces, based on the given sequential object and the specification. When PEEPCO detects an integrity violation, such as in the overdraft scenario in Figure 2.2, it analyses the trace and concludes the bad trace can be prevented by disallowing concurrent executions and enforcing strong consistency (total order) of `withdraw` and their effects. PEEPCO continues exploring new traces, pruning all those that concurrently invoke `withdraw`. It then finds another violation of safety (Figure 2.6) and concludes `withdraw` effects need to be delivered in the causal order with respect to `deposit` in the same batch [Lamport, 1978b] (as, *e.g.,* `withdraw` effect `-120` can depend on the particular amount deposited already, `20`). The violation can be prevented by enforcing strong consistency between the two methods, but that would be more restrictive. PEEPCO supports weak, causal, and strong consistency orders. Once PEEPCO explores all traces for a bounded set of operations, it synthesizes an oracle that is used by PeepCommit.

2.2 Batch-Based Consistency

In this section, we define batch-based consistency. In batch-based consistency, consistency is defined relative to an integrity specification, which asserts application-specific invariants. We show how our consistency optimization procedure produces a distributed object implementation that leverages a run time protocol, to allow methods to be executed concurrently and ordered differently on different nodes, whenever such reordering cannot violate the integrity specification, which increases application performance.

2.2.1 Semantics of Batch-Based Consistency

Batch-based consistency is parameterized by an *integrity specification* $\mathcal{I} = (\mathcal{I}_S, \mathcal{I}_C)$, where $\mathcal{I}_S, \mathcal{I}_C$ are two sets of integrity invariants: safety $\mathcal{I}_S$ and congruence $\mathcal{I}_C$. The invariants are evaluated on a *global state* σ of the system. Global state captures *local states* at each node $n \in \mathsf{Node}$ in the system. A special set of node identifiers Node is the set of all nodes in the system. A global state $\sigma : \mathsf{State}$ maps nodes to local states $\mathsf{LocalState}$, $\mathsf{State} = \mathsf{Node} \mapsto \mathsf{LocalState}$.

A local state $s : \mathsf{LocalState}$ maps variables Id to values, $\mathsf{LocalState} = \mathsf{Id} \mapsto \mathsf{Val}$. (We will use state to refer to the global state and specify locality when needed.) In our running example, the account balance `bal` is replicated, so each of $\sigma[n_1]$ and $\sigma[n_2]$ contains a copy of `bal` and `hist` and initially $\sigma = [n_1 \to [bal \to 0, hist \to \texttt{Nil}], n_2 \to [bal \to 0, hist \to \texttt{Nil}]]$.

An *integrity invariant* is a first-order formula of the form:

$$\forall \overline{n} \in \mathsf{Node}^k.\ c(\overline{n}) \implies p(\Omega(\overline{n})) \text{ where } \Omega(\overline{n}) = \sigma[n_1], \ldots, \sigma[n_k],\ \sigma \in \mathsf{State}$$

where $\overline{n} \in \mathsf{Node}^k$ specifies a sequence, of length k, of node identifiers $\overline{n}$ quantified over all nodes in the system Node. c is a conjunction of equalities and inequalities over node identifiers $\overline{n}$ which defines combinations of node identifiers that are used to construct parameters $\Omega(\overline{n})$ for the predicate p. p is a quantifier-free predicate. In our running example convergence invariant `i2` has $c = \texttt{true}$, but we could add the condition c that requires $n_a \neq n_b$ to avoid trivial comparisons of the same variables, *e.g.,* $\sigma[n_a][bal] = \sigma[n_a][bal]$. We use the notation $\sigma \models I$ to mean "I holds in state σ."

We define a *state trace* τ to be a sequence of states $\sigma_1, \ldots, \sigma_n$, where $\sigma_i \in \mathsf{State}$. The trace characterizes an execution in which a node n_x in the system goes exactly through the sequence of states $\sigma_1[n_x], \ldots, \sigma_n[n_x]$. Next, we define a "satisfies" relation between traces τ and different kinds of invariants I.

Definition 1 (Safety invariant). *I is a* safety *invariant on a trace τ (denoted $\tau \models_s I$) if it holds on every state in τ,* i.e., *$\tau \models_s I \equiv \forall \sigma \in \tau.\ \sigma \models I$.*

Definition 2 (Congruence invariant). *I is a* congruence *invariant on a trace $\tau = \sigma_1, \ldots, \sigma_n$ (denoted $\tau \models_c I$) if for every σ_i, I holds on σ_i or some other state that follows σ_i in the trace,* i.e., *$\tau \models_c I \equiv \forall 1 \leq i \leq n.\ \exists i \leq j \leq n.\ \sigma_j \models I$.*

We lift safety and congruence to sets of invariants: a trace τ satisfies a set of safety invariants $\mathcal{I}_S$ (resp. congruence $\mathcal{I}_C$) iff $\tau \models_s I$ (resp. $\tau \models_c I$) for every $I \in \mathcal{I}_S$ (resp. $I \in \mathcal{I}_C$). For any point in the trace, congruence must be restored at some point after (it holds "always eventually" [Pnueli, 1981]). $C_{\mathcal{I}_C}(\tau)$ denotes the set of congruent states of τ; $\sigma \in C_{\mathcal{I}_C}(\tau)$ iff $\sigma \models \mathcal{I}_C$.

An integrity specification $\mathcal{I} = (\mathcal{I}_S, \mathcal{I}_C)$ holds on a trace τ (denoted $\tau \models \mathcal{I}$) if $\tau \models_s \mathcal{I}_S$ and $\tau \models_c \mathcal{I}_C$. An integrity specification holds on a *system* if it holds on every trace that the system exhibits. A system or trace where an integrity specification holds is "correct." In our example, a correct system exhibits only traces where `bal` are always non-negative and replicas periodically restore a state with values of `bal` and folded `hist` equal across replicas.

Operation Execution. The definitions of integrity given so far in this section do not account for operation (or method) execution. For example, strong consistency models such as linearizability guarantee that, if an operation is considered to be executed, there exists a linearizability point where the operation's effects are accounted for on every node [Herlihy and Wing, 1990]. However, eventual consistency is less strict. An arbitrary number of operations might finish executing, while nodes might never reach a congruent state (replicas having the same state) [Shapiro et al., 2011b]. (This scenario can occur *e.g.,* if operations keep being invoked in the system indefinitely often.) Models of batch-based consistency account for places where the invoked operations are considered to be finished and require convergence to be restored regardless of the number of other operations that might be invoked concurrently.

Given a trace τ and an operation call o, we define two state indexes: its invocation state $start_\tau(o)$ and its finish state $finish_\tau(o)$. A operation call o *executed* in τ if $1 \leq start_\tau(o) \leq finish_\tau(o) \leq n$. Operation calls are unique and represent operation types being invoked at specific points in time. For a set of operation calls O, the finish state is the finish state of the last call, *i.e.,* $finish_\tau(O) = l$, where $\forall o \in O.finish_\tau(o) \leq l$. On a particular node n_x, a call o is considered finished if all of the execution effects of o on n_x are executed on n_x. In Figure 2.6, at the end of the given execution, the call `deposit(20)` (of type `deposit`) has finished, as all of its effects were propagated and executed on all nodes, while `withdraw(120)` has not.

We now constrain states in which congruence has to be restored. We say a trace τ is B($\mathcal{I}$)-consistent for operations O if and only if τ is safe and congruent, and O finished in a congruent state σ_l, *i.e.,* $finish_\tau(O) = l$ and $\sigma_l \in C_{\mathcal{I}_C}(\tau)$.

Definition 3 (TC($\mathcal{I}$)-consistency)**.** *A system is B($\mathcal{I}$)-consistent if for any set of executed operations O, every trace that executes O is B($\mathcal{I}$)-consistent.*

This definition prevents any B($\mathcal{I}$)-consistent system from executing operations, *i.e.,* making progress, if it cannot guarantee that the system will start converging to a congruent state. This disallows nodes from arbitrarily diverging in their state and demands communication between all nodes, thus a slow node or network partition can prevent progress. Peepco guarantees B($\mathcal{I}$)-consistent executions. In turn, this leads to sacrificing availability under network partitions in favor of consistency, following the impossibility results from [Fischer, N. A. Lynch, and Paterson, 1985; Gilbert and N. Lynch, 2002; Gilbert and N. Lynch, 2012].

2.2.2 Batched Operation Execution

We define rules for allowing operations to execute concurrently. To that end, we define the notion of *batched execution.* A *batch numbering* B^τ partitions a trace τ into contiguous sequences of states and assigns every state an increasing batch number, *i.e.,* $B^\tau(\sigma_{i+1}) \geq B^\tau(\sigma_i)$ and $B^\tau(\sigma_{i+1}) \leq B^\tau(\sigma_i)+1$. For operations O, we then define *operation mapping* $B^\tau_O = O \mapsto N$, such that for an operation call $o \in O$, the batch of o is k, $B^\tau_O(o) = k$.

Definition 4. *A trace $\tau = \sigma_0, \ldots, \sigma_n$ is a batched execution of operations O iff there exists a pair (B^τ, B^τ_O) such that the given batch numbers defined by B^τ, B^τ_O assigns every operation o a batch within τ,* i.e., $\forall o \in O.\ 1 \leq B^\tau_O(o) \leq n$, *and o is fully executed in some batch k,* i.e., $start_\tau(o) = i \wedge finish_\tau(o) = j$, *where σ_i, σ_j are assigned the batch k by B^τ.*

An operation belongs to exactly one batch. The size of the k-th batch is $|\{o | B^\tau_O(o) = k\}|$. Strong consistency corresponds to batches of size 1. Eventual consistency, in the worst case, might exhibit an infinite trace with only one batch.

We now define the notion of valid bounded executions between operations in the same batch on subtraces and use it as the basis for defining *batched B($\mathcal{I}$)-consistent executions*. Given a trace τ, $\mathcal{I}$, and O, let $\mathsf{CanBatch}_\mathcal{I} : 2^O \mapsto \mathbb{B}$, where 2^O denotes all subsets of O, be a mapping that prescribes whether for a given subset $O' \subseteq O$, τ executes O, and the subtrace $\tau' = \sigma_{start_\tau(O')}, \ldots, \sigma_{finish_\tau(O')}$ is B($\mathcal{I}$)-consistent.

Theorem 2.2.1. *A trace τ exhibits a batched B($\mathcal{I}$)-consistent execution of a finite set of operations O iff there exists $\mathsf{CanBatch}_\mathcal{I}$ and a batched execution defined by (B^τ, B^τ_O), such that all operations executed in the batch i, $O_i = \{o \mid B^\tau_O(O) = i \wedge o \in O\}$, can belong in a same batch according to $\mathsf{CanBatch}_\mathcal{I}$,* i.e., $\forall i.\ \mathsf{CanBatch}_\mathcal{I}(O_i)$.

The theorem holds as it relies on $\mathsf{CanBatch}_\mathcal{I}$, which allows only safe executions within any batch and restores congruence at the end of any batch, thus assuring B($\mathcal{I}$)-consistency of the entire trace τ.

Progress and Congruent Reads. To capture progress, we put a finite bound on all batches, *i.e.,* $\forall i.\ 0 < |B_i|$. Due to operations being finite, a system can map a finite number of states to a batch before starting a new one. Therefore, any batch eventually finishes, executing one or more operations concurrently. To read state, Batch-Based Consistency models support two special types of read operations. Safe reads return a local state from a node at any point. Congruent reads return a congruent state snapshot across all nodes.

Given $\mathcal{I}$ and a set of operations O, any system generated by PEEPCO implements batched B($\mathcal{I}$)-consistent executions of O. It does this by first inferring $\mathsf{CanBatch}_\mathcal{I}$ for O, and then, executing concurrent operations from O in batches, as prescribed by $\mathsf{CanBatch}_\mathcal{I}$. Such a system supports safe and congruent reads, and satisfies progress. PEEPCO implements congruent reads by executing them at the end of the current batch, collecting the state from all the needed nodes.

2.2.3 Consistency Constraints and Lattice

PEEPCO implements $\mathsf{CanBatch}_\mathcal{I}$ by identifying bad traces and inferring requirements on the order of operations that would prevent those traces. We follow prior work [Kaki, Earanky, et al., 2018; Sivaramakrishnan, Kaki, and Jagannathan, 2015] and characterize the space of consistency models as a finite partially ordered lattice, defined over standard notions of visibility and program order between executed operations, where the partial order denotes the relative strength of the considered models [Viotti and Vukolić, 2016; Sivaramakrishnan, Kaki, and Jagannathan, 2015]. We define a *consistency constraint* $c(t_1, t_2)$ as a predicate on traces that captures rules about the ordering of any operations of types t_1, t_2. Each element of the lattice is a set of constraints $\mathcal{C}$. PEEPCO supports predicates for three consistency levels: weak, causal, and strong consistency. (We add a new level in Section 2.3.) We define a "weaker than" relation $\sqsubset$ between sets of consistency constraints: for any operation types t_1, t_2, $\{weak(t_1, t_2)\} \sqsubset \{causal(t_1, t_2)\} \sqsubset \{strong(t_1, t_2)\}$. In our example, the set $\{strong(\texttt{withdrawal}, \texttt{deposit})\}$ prevents the given violation, but it is not the weakest that does so, as the causal ordering prevents it as well.

Definition 5 (Feasibility). *A trace τ is* feasible *under $\mathcal{C}$, denoted $\tau \models_{cons} \mathcal{C}$, iff there exists τ' such that $\mathcal{C}$ is true for τ++τ'.*

Effectively, when a trace is infeasible under a constraint c, no matter how the rest of the pending steps are ordered, c will not be satisfied when all operations finish executing. For example, in Figure 2.6, the trace becomes infeasible under $causal(\texttt{withdraw}, \texttt{deposit})$ right after executing `bal-=120` on n_1, as $w(120)$ cannot witness a causally dependent $d(20)$.

Sets of constraints $\mathcal{C}$ form a bounded lattice $(2^{\mathcal{C}} \cup \top, \sqsubset)$ with $\bot = \emptyset$, and $\top$, which contains the strongest constraint for any two operations. $\bot$ and $\top$ make all, and none, of the traces feasible, respectively, *i.e.,* $\forall \tau . \tau \models_{cons} \bot \wedge \tau \not\models_{cons} \top$. Given a set of traces Ω and $\mathcal{I}$, we say $\mathcal{C}$ is correct, or $\mathcal{C}$ is a solution, if $\forall \tau \in \Omega.\ \tau \models_{cons} \mathcal{C}_s \implies \tau \models \mathcal{I}$. Additionally, $\mathcal{C} = \mathcal{C}_1 \sqcap \mathcal{C}_2$ is correct for $\Omega = \Omega_1 \cup \Omega_2$, if $\mathcal{C}_1$, $\mathcal{C}_2$ are correct for Ω_1, Ω_2, respectively. If $\mathcal{C}$ is correct, then any stronger $\mathcal{C}'$ (*i.e.,* $\mathcal{C} \sqsubset \mathcal{C}'$) is also correct. We say that $\mathcal{C}$ is *optimal* if there is no correct $\mathcal{C}'$ such that $\mathcal{C}' \sqsubset \mathcal{C}$.

Note that the only correct $\mathcal{C}$ might be $\top$, thus no trace, correct under $\mathcal{I}$, is feasible. We define $\mathcal{C}_S^O$ as a set of strong constraints between all operations in O. (Intuitively, if $\mathcal{C}_S^O$ is not correct, integrity cannot be fixed by consistency choices.)

Definition 6 (Well-formedness). *We say invariants $\mathcal{I}$ are well-formed if there exists $\mathcal{C}$ that is correct, such that $\mathcal{C} \sqsubseteq \mathcal{C}_S^O$.*

To obtain a correct $\mathcal{C}$ for a set of operations, it is sufficient to explore all their traces and unify their solutions with $\sqcap$. We define a solution map that contains a correct $\mathcal{C}$ per size of the considered set of operations. A *solution map*, of some order k, is a function $\mathcal{M}_k$, that maps each integer $i \leq k$ to a solution $\mathcal{C}$ for all traces that execute subsets of O of size

i, *i.e.*, $\mathcal{M}_k(i) = \mathcal{C}$, where $i \leq k$, and $\mathcal{C}$ is correct for $\{\tau \mid O' \subseteq O, |O'| = i, \tau \text{ executed } O'\}$. For an *optimal* solution, $\mathcal{C}$ is optimal for all traces exhibited by all O' where $|O'| \leq k$. In Figure 2.2, a solution with order $k = 2$ is used, as it concurrently executes two operation calls.

Definition 7. *A system implements B($\mathcal{I}$)-consistent k-optimal execution, if it is B($\mathcal{I}$)-consistent, batched, and for any batch O_b, it exhibits a trace τ that executes O_b if and only if $\tau \models_{cons} \mathcal{M}_k(|O_b|)$.*

We define an additional property that captures most practical systems, which states that adding new operations cannot make the optimal solution smaller (more permissive).

Definition 8 (Monotonicity)**.** *Invariants $\mathcal{I}$ are* monotonic *iff* $O_1 \subseteq O_2 \Rightarrow \mathcal{C}^{O_1}_{opt} \sqsubseteq \mathcal{C}^{O_2}_{opt}$.

For non-monotonic invariants, a violation with fewer operations might not be exhibited with more operations. This is possible for congruence: *e.g.*, , if `i1` was a congruence property, interleavings of `withdraw(70)` and `withdraw(20)` would become permissible, when `deposit(x)` is added, if $x \geq 90$. For well-formed $\mathcal{I}$ PEEPCO produces B($\mathcal{I}$)-consistent, while for well-formed and monotonic $\mathcal{I}$, PEEPCO produces B($\mathcal{I}$)-consistent k-optimal executions.

Given a bound k and a solution $\mathcal{M}_k$, PEEPCO derives two predicates that are used at run time: 1) `canBatch`, which answers if the given operations B form a valid batch; it is implemented by checking if no two operations in B have a strong consistency constraint, *i.e.*, $\nexists o, o' \in B.\ strong(o, o') \in \mathcal{M}_k(|B|)$, and thus, executing them concurrently is safe; 2) `needCausal`, which answers if the given operation o needs to be executed in the causal ordering with respect to any operation in the given set B, *i.e.*, $\exists o' \in B.\ causal(o, o') \in \mathcal{M}_k(|B \cup \{o\}|)$.

2.2.4 PeepCommit Protocol

In this section we present PeepCommit, a distributed blocking protocol for batch-based consistency that is parameterized with predicates for concurrent batching. The protocol executes operations concurrently in batches, where each batch includes as many non-conflicting operations as possible.

Key Idea. For each new operation call, all nodes vote on whether the operation can be added to the current batch. Nodes store operations they agreed on and can only vote "yes" for an operation if it does not conflict with any of the operations already agreed upon. If an operation is not agreed upon by all nodes, it aborts and can be reissued later. This ensures that no conflicting operations are ever simultaneously accepted (similar in operation to *two-phase commit* [Gray and Lamport, 2006]). Once an operation is accepted, it is broadcast

Algorithm 1 B($\mathcal{I}$)-consistent Delivery (PeepCommit protocol)

```
     PeepCommitObject
       request: call(C)
       response: ret(C, V) | aborted(C)
     Params:
       canBatch: Set[O] → Bool
       needCausal: (O, Set[O]) → Bool
       k: Int                                  ▷ bound
     Using:
       rb: ReliableBroadcast
       cb: CausalBroadcast
       self: Node
     State:
       σ: Σ = σ_0
       r: Int = 0                              ▷ round index
       B: Set[C] = ∅                           ▷ batch calls
       E: Set[C] = ∅                           ▷ executed calls
       votes: C → MSet[Bool] = C ↦ ∅           ▷ vote multisets
C_1  request: call(c)                          ▷ operation call
C_2    broadcast pr(c, r)
P_1  receive: pr(c, r')
P_2    noConflict ← canBatch({c} ∪ ops(B))
P_3    if r = r' ∧ |B|<k ∧ noConflict then
P_4      reply vote(true, c)
P_5      B ← B ∪ {c}
P_6    else
P_7      reply vote(false, c)
Q_1  receive: vote(v, c)
Q_2    votes(c) ← votes(c) ∪ {v}
Q_3    if |votes(c)| = |Node| then
Q_4      if false ∉ votes(c) then              ▷ all voted yes
Q_5        if needCausal(c, ops(B)) then
Q_6          causal-order broadcast execute(c)
           else broadcast execute(c)
Q_7      else
Q_8        broadcast cancel(c)
Q_9        issue response aborted(c)
R_1  receive: execute(c) on rb | cb
R_2    σ ← update(σ, c)
R_3    v ← retv(σ, c)
R_4    if node(c) = self then                  ▷ c issued here
R_5      issue response ret(c, v)
R_6    E ← E ∪ {c}
R_7    checkDone()
S_1  receive: cancel(c)
S_2    B ← B \ {c}
S_3    checkDone()
F_1  fun checkDone()
F_2    if B ⊆ E ∧ E ≠ ∅ then
F_3      r ← r +1                              ▷ node enters next round
F_4      B ← ∅
F_5      E ← ∅
```

to be executed to other nodes, taking into account the order that preserves integrity. This process interleaves execution and proposals of new operations and continues while the batch is not full, or until some node decides to end the batch, which happens if all operations agreed upon finish executing on that node.

Protocol. Our protocol description (Algorithm 1) details the events that can occur in the system and describes how nodes respond to each event [Cachin, Guerraoui, and L. Rodrigues, 2011; Houshmand and Lesani, 2019]. The protocol is parameterized over: 1. `canBatch`, which determines which operations (type `O`) can conflict, 2. `needCausal`, which determines which operations need causal delivery, and 3. `k`, the maximum batch size. Internally, it uses reliable and causal broadcast primitives (`rb` and `cb`) for communication, as described subsequently. The protocol stores the following state on each node: the initial application state δ_0, a batch number r, the set of agreed operations B, the set of executed operations E, and a mapping `votes` from calls to the multisets of received votes from other nodes.

The protocol starts when an operation call is issued. For every new call request `call(c)` (line C_1), a proposal `pr(c, r)` for the call `c` in round `r` is broadcast. Unless otherwise specified, messages are communicated reliably, using *rb*. (Our implementation achieves this using TCP.) When `pr(c, r')` is delivered (line P_1), the receiving node votes on whether to add the call `c` to the batch. The call can only be added if the batch `B` is not full and the operation of the call `op(c)` does not conflict with any of the previously added calls to `B` (line P_2). This is ensured by `canBatch`, which is true if a set of operations is safe to execute concurrently. In addition, only calls belonging to the round the receiving node is in, can be added (line P_3); this ensures that nodes receive all votes reflecting the needed batch even when they are slow in advancing to the next round. The node votes by sending a reply `vote(c, v)`, where `v` is Boolean. On a "yes" vote, it adds `c` to the batch.

Upon receipt of a `vote(c, v)` (line Q_1), the node adds the vote to the set of all votes for the call `c`. Once a vote is received from every node (including self, line Q_3), the call is accepted if every vote is "yes" and aborted otherwise. To execute the call, an `execute(c)` message is broadcast. `needCausal` determines whether that broadcast needs causal ordering[2]. Otherwise, regular broadcast is used and the call might be interleaved with other calls that are accepted in the same batch. While other nodes already have the operation in `B`, broadcasting the operation is necessary to ensure the right order of delivery, which determines the order of execution. When a call does not get a "yes" vote from every node, its issuer sends `aborted(c)` to the caller and sends a `cancel` notification to notify other nodes that the call is not part of the batch. Upon receipt of an `execute(c)` message on either `rb` or `cb`, the node executes the operation `c`, which updates its state and produces a return value `v` (lines R_2–R_3). When the node is the one that originally received `call(c)`, it sends a return message `ret(c, v)` to the caller (line R_5). The node then adds the call to the set of executed calls `E` and checks if it can proceed to the next round by calling `checkDone`. Upon receipt of `cancel(c)` (line S_1), the node removes `c` from its batch set `B`, regardless of its vote on `c`. Afterward, it calls `checkDone` to check if the condition to proceed to the next round is met. Function `checkDone` (lines F_1–F_5) checks if all calls that the node agreed to execute in the current batch `B` have been executed. At least one operation needs to be executed, to prevent empty batches in cases where all the proposed calls get rejected. If all calls have been executed, the node advances to the next round. It increments the round index `r` and resets the batch `B` and executed `E` sets.

Protocol Properties. PeepCommit maintains the following properties:

1. *Validity*: During any batch, only non-conflicting operations are executed concurrently, and they are executed in the right order.

2. *Agreement*: For every round, nodes agree on the same batch of operations.

3. *Termination*: Every operation call eventually completes or aborts, exactly once.

[2]PEEPCO uses the standard causal broadcast abstraction *cb* to deliver messages in causal order, which ensures that messages are only delivered after any causally preceding messages. [Cachin, Guerraoui, and L. Rodrigues, 2011].

The protocol maintains an invariant that at any point in time, on every node, `B` contains operations that are allowed to execute concurrently, according to `canBatch`. Validity follows from this invariant and the fact that to execute an operation concurrently, every node has to vote "yes" on it, only after adding it to `B` locally, which is only possible if it's non-conflicting. Once any node finishes the batch and proceeds to the next round, no more operations can be added to the same batch (as the node will keep rejecting their proposal). Agreement thus follows, as if a node votes "yes" for an operation (and adds it to its `B`), it will eventually get notified about its execution or cancellation, and will arrive at the same final `B` as other nodes and end the batch as well. Termination follows from the fact that the outcome of the operation is known after the voting is done, and the operation is executed locally on the issuing node. We discuss PeepCommit's starvation in Section 2.4.

Correctness. Given a solution $\mathcal{M}_k$ for operations O, PeepCommit implements B($\mathcal{I}$)-consistent live batched executions. If $\mathcal{M}_k$ is k-optimal, executions are k-optimal.

The correctness follows from *Validity* and *Agreement* of PeepCommit when `canBatch` and `needCausal` are derived from $\mathcal{M}$, as operations maintain safety, and congruence is restored at the point where all nodes advance to the next round. Note that new operations cannot be accepted to be executed, until all nodes have advanced, thus reaching a congruent state. Progress in the batch-based consistency follows from *Termination* and the fact that only a finite set of operations (up to k) can be executed before the congruence property is restored. Optimality then simply follows from the optimality of $\mathcal{M}_k$.

Total order optimization. PeepCommit can be extended to allow multiple operations that need strong consistency in the same batch. Executed operations E can be discarded when checking this conflict: canBatch$(\{c\} \cup (ops(B) \setminus ops(E)))$ on line C_2, as total order between a c and all in E is ensured.

2.2.5 Integrity-Driven Consistency Strengthening

In this section we present *integrity-driven consistency strengthening*, which infers consistency constraints used by PeepCommit. The goal of the algorithm is to symbolically identify constraints on the ordering of bounded sets of operations (akin to [Kaki, Earanky, et al., 2018]), support operation splitting under partial replication (see Section 2.3), and prune for scalability.

The idea of the algorithm is to incrementally build a consistency solution by accumulating constraints identified for traces that violate integrity. It explores all traces that execute an increasing number of concurrent operations; when it finds a trace that violates integrity, it identifies the weakest constraint that makes the trace infeasible. Found constraints on smaller

Algorithm 2 INFER (Integrity-Driven Consistency Strengthening)

```
Require: nodes $N$, operations $O$, integrity invariants ($\mathcal{I}_S$, $\mathcal{I}_C$), bound $k$
Ensure: consistency solution $\mathcal{M}$ of order $k$
1: $S_i \leftarrow \emptyset$, for $1 \le i \le k$, $\mathcal{C}_0 \leftarrow \bot$, $b \leftarrow 1$, $\Omega_1 \leftarrow \{([], Init, \mathcal{I}_C \wedge \mathcal{I}_S) \mid Init \in \mathsf{Comb}(O, N, 1)\}$
2: while $b \le k$ do                                          ▷ explore worklists up to size k
3:   if $\Omega_b \ne \emptyset$ then                           ▷ all traces up to b are done
4:     $(\tau_e, Next, pc) \leftarrow \mathsf{Choose}(\Omega_b); \Omega_b \leftarrow \Omega_b \setminus \{(\tau_e, Next, pc)\}$
5:     if $\tau_e \not\models_{cons} \mathcal{C}_b$ then continue                 ▷ prune
6:     $f_{valid} \leftarrow \bigwedge_{I \in \mathcal{I}_S} eval(\tau_e) \wedge p \models I$       ▷ computed by SMT solver
7:     if $Next \ne \emptyset$ then                          ▷ some operations did not finish
8:       $\Omega_b \leftarrow \Omega_b \cup \{(\tau_e +\!\!+ [s], Next \cup \texttt{getNext}(s, pc) \setminus \{s\}, pc \wedge c) \mid c \in \texttt{getCond}(s), s \in Next\}$
9:     else
10:      $f_{valid} \leftarrow f_{valid} \wedge (\bigwedge_{I \in \mathcal{I}_C} eval(\tau_e) \wedge p \models I)$     ▷ computed by SMT solver
11:    if $\neg f_{valid}$ then                             ▷ trace $\tau_e$ can violate invariants
12:      $S_b \leftarrow (\mathcal{C}, \tau_e) \cup S_b$, s.t. $\mathcal{C}$ is minimal and $\tau_e \not\models_{cons} \mathcal{C}$
13:      $\mathcal{C}_b \leftarrow \mathsf{Merge}(S_b) \sqcap \mathcal{C}_b$
14:      if $\mathcal{C}_b = \top$ return $b' < b$ ? $\{(b' \rightarrow \mathcal{C}_{b-1})\}$ : $\top$   ▷ no solution for bounds $b' > b$
15:  else
16:    $b \leftarrow b + 1$, $\mathcal{C}_b \leftarrow \mathcal{C}_{b-1}$, $\Omega_b \leftarrow \{([], Init, \mathcal{I}_C \wedge \mathcal{I}_S) \mid Init \in \mathsf{Comb}(O, N, b)\}$
17: return $b' < k$ ? $\{(b' \rightarrow \mathcal{C}_k)\}$ : $\top$                    ▷ solution of order k
```

traces are also used for pruning larger unexplored traces. Execution traces are checked fully symbolically with an SMT solver. We follow the standard approach of exploring programs with conditionals, and we capture distributed semantics with execution steps, or *actions*, representing node-local computation [Flanagan and Godefroid, 2005]. PEEPCO transforms methods into DAGs of actions, together with conditions that enable their execution in case of branching, accounting for the propagation of effects. PEEPCO supports recursion only within actions locally.

Algorithm. The algorithm (INFER) is given in Algorithm 2. INFER takes nodes N, execution step, or *action*, graphs O, integrity invariants $\mathcal{I}$, and a maximum bound k. A bound reflects the number of operation calls in the system. It returns a solution of order k, which contains the weakest set of constraints that prevent all violations for any number of concurrent operations, up to bound k. INFER maintains a solution $\mathcal{C}_b$ for each explored bound b and starts by initializing each solution with the bottom element $\bot$, the empty set of constraints, which allow any trace. INFER maintains a worklist Ω_b for every bound b. Worklist items are tuples $(\tau_e, Next, pc)$ of a trace of actions τ_e, pending actions to execute $Next$ and a path condition pc.

The algorithm initializes a worklist Ω_1 for exploring single operations invoked in isolation. $\mathsf{Comb}(O, N, b)$ returns the set of initial steps for all different execution scenarios in bound b, covering all combinations of methods in O, dispatched to different nodes in the system. In our example, for $b = 2$, *Init* will contain starting actions for two `deposit` calls dispatched to separate nodes, as well as to the same node. INFER then creates new worklist items for each combination of starting actions. Each new worklist item has an empty trace, and a path condition that reflects starting from a safe and congruent state. All worklists are processed in the order of increasing b, until the max bound k is explored. If the worklist Ω_b is nonempty, INFER chooses an item to process with Choose (line 4). While Choose can make arbitrary choices, PEEPCO implements particular optimized strategies; see Section 2.4. If any of the constraints from the previous bound $\mathcal{C}_{b-1}$ makes the trace infeasible, INFER prunes it (line 5). Next, INFER symbolically checks that the states reachable by executing τ_e, under the current path condition p, are safe, for all safety invariants (line 6). `eval` produces a symbolic formula from a trace, encoding the semantics of a sequential execution of its actions. Then, INFER checks if the trace is completed. If there are pending actions to execute, *Next*, for each such action s, INFER produces a new item (lines 8). A new item is created where s is added as the last action in the trace, updated path condition in case s was a conditional, and newly enabled actions. `getNext` (s, pc) returns the actions that are enabled after executing s, under the path condition pc. If s has conditionals, for every branch of s, INFER adds the condition c (returned by `getCond`) that enables that branch to the new path condition. In our example, `getNext(withdraw, true)` produces two conditions, $\texttt{amt} \leq \texttt{bal}, \texttt{amt} > \texttt{bal}$. The path of the else branch produces a no-op action. If there are no more actions to execute (line 10), *i.e.,* the trace τ_e finished all its actions, INFER symbolically checks if congruence is restored. This check implies the specific point congruence is checked (at the batch end, per batch-based consistency; Subsection 2.2.2). If τ_e violates safety or congruence, INFER infers the weakest $\mathcal{C}$ that prevents τ_e, by traversing and checking the consistency lattice. $\mathcal{C}$ is then added to S_b along with τ_e, tracking all violating traces and consistency constraints that prevent them, as pairs. INFER then unifies constraints in S_b with Merge to produce the weakest solution for the violating traces in S_b, as described subsequently. INFER then updates the solution $\mathcal{C}_b$ by computing join, $\sqcap$ (line 13). Joining two lattice elements ensures we get the weakest consistency constraints that prevent traces prevented by both elements (as defined in Subsection 2.2.3). If $\mathcal{C}_b = \top$, INFER returns without a solution, as the given integrity is unsatisfiable.

If Ω_b is empty (line 3) INFER proceeds to the next bound and populates the worklist Ω_b with new items using Comb. It also propagates already discovered constraints C_{b-1}, to prevent all violating traces from prior bounds. After all worklists up to bound B have been explored (line 17), INFER returns $\mathcal{C}_k$ as the k-solution.

Merging Constraints. Merge unifies constraints in S_b by simply joining them ($\sqcap$). We describe a modified Merge for partial replication in Section 2.3.

Soundness, Completeness & Optimality. Infer is *sound*—if it returns a solution, it admits only traces that satisfy integrity. All traces are either checked or pruned, based on the least upper bound of the weakest elements that prevent all found violating traces. For well-formed invariants Infer is *complete*—if a solution exists in the lattice, Infer will return it. As the algorithm checks all traces, for well-formed invariants at least strong consistency will be returned. Otherwise, the algorithm reports unsatisfiability. For monotonic invariants, the solution is *optimal*—Infer returns the weakest solution. Optimality relies on the fact that the algorithm correctly identifies the weakest set of constraints that prevent individual traces (line 12), merges them with $\sqcap$, and prunes traces that would be infeasible under those constraints. Due to monotonicity, the orderings that caused consistency violations cannot be made viable by adding new operations, thus the solution for larger bounds cannot be made weaker.

Relationship with SyCoord's consistency inference framework. *Integrity-driven consistency strengthening* instantiates a general framework for consistency inference, which we present in Chapter 3. Notably, the algorithm presented here differs from the concrete and concolic instantiations presented in Chapter 3 in two ways. First, *integrity-driven consistency strengthening* uses a symbolic checker to check bounded execution traces and thus guarantees integrity for arbitrary values without requiring concrete inputs from the programmer. This comes at the cost of slower inference, in cases where efficient evaluation on concrete values might lead to discovering important conflicts sooner. (Our evaluation, however, shows the two strategies lead to comparable scalability.) Second, *integrity-driven consistency strengthening* leverages a simplified strategy for consistency constraint merging to efficiently find optimal consistency requirements in the context of the adopted model, *i.e.,* batch-based consistency. Specifically, here, given some operations O, for a violating trace τ, if more than one constraint prevents it, let $\mathcal{C}_1$ and $\mathcal{C}_2$ be the constraints that prevent τ, while the constraints cannot be ordered, *i.e.,* $\mathcal{C}_1 \not\sqsupseteq \mathcal{C}_2$ and $\mathcal{C}_2 \not\sqsupseteq \mathcal{C}_1$, an optimal solution $\mathcal{C}_o$ is allowed to contain (and the result of the algorithm always produces) $\mathcal{C}_o \sqsupseteq \mathcal{C}_1 \sqcap \mathcal{C}_2$. However, in the case that $\mathcal{C}_1$ and $\mathcal{C}_2$ are not ordered in the consistency lattice, while both $\mathcal{C}_1$ and $\mathcal{C}_1$ are correct consistency constraints, only one might comprise the final optimal solution $\mathcal{C}'_o$, *i.e.,* $\mathcal{C}'_o \sqsubset \mathcal{C}_1 \sqcap \mathcal{C}_2$. Chapter 3 describes a fine-grained strategy of obtaining optimal solutions $\mathcal{C}_o$ in such cases, which maintains and propagates multiple candidate solutions.

2.3 Partial Replication

In this section, we describe the extension of our approach for *partial replication* to allow data to be distributed across nodes [Belaramani et al., 2006]. We show how developers can allocate

```
var max = 100; var last = None

def withdraw(amt: Nat) {
 if (amt ≤ max && amt ≤ bal) {
  bal -= amt; hist += ("w", -amt)
  last = Some(amt) }}

def set(x: Int) {
 max = bal * x / 100;
 last = None }
```

Figure 2.7: Bank account

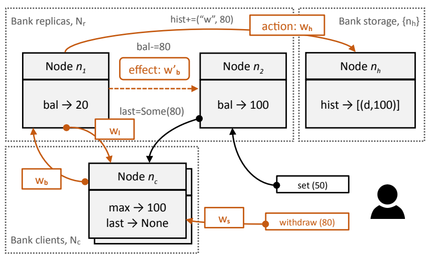


Figure 2.8: Distributed execution

```
replicate(bal, Nr)
allocate(hist, nh)
distribute({last, max}, Nc)

safe ∀n ∈N. def i1...
congr ∀na, nb ∈N. def i2...

safe ∀n ∈Nc. def i3 =
 last[n].isEmpty ||
   last[n].get ≤ max[n]
```

Figure 2.9: Specification

data to specific nodes, treat methods as distributed transactions, and leverage additional consistency optimization opportunities.

2.3.1 Motivation

We extend the banking application (from Section 2.1) to restrict the amount that can be withdrawn and to optionally track the amount last withdrawn. New code is given in Figure 2.7. We modify `withdraw` to additionally check if the amount is within the limit `max` and add the method `set` that updates `max` as a percentage of `bal`. The intended data allocation is shown in Figure 2.8 and is achieved with the specification shown in Figure 2.9. We declare three node sets, N_r (replicas), N_c (clients), and N_h (storage), and use PEEPCO to specify the data distribution. `bal` is replicated on N_r, and `hist` is *allocated* to N_h. *allocate* restricts allocation to a single node, so N_h is a singleton set[3], and *distribute* assigns each client node in N_c its own copy of `last` and `max`. We adapt our existing integrity properties for the new data allocation and add an additional safety property (`i3`): the `last` withdraw should not be greater than `max`, on all client nodes N_c.

2.3.2 Finer Grained Batch-Based Consistency

Partial replication changes the distributed execution semantics, enables additional protocol optimizations, and changes the inference process. In this section, we sketch out these changes.

Distributed Execution. The key change to distributed execution is that methods access fields that are only stored on certain nodes, so PEEPCO splits them into *actions* that execute on the node where the needed data is located. Unlike methods, actions have control-flow dependencies on other actions (Figure 2.8). For `withdraw` invoked with the argument of 80, the starting action w_s is dispatched to any client node (N_c) and checks if `amt` $\leq$ `max`. That check passes (as $80 \leq 100$), and it sends w_b to either of the replicas n_1 or n_2 (N_r). Then, w_b checks `amt` $\leq$ `bal` and then updates `bal`, propagates the effect w'_b (`bal-=80`) to n_2, sends

[3]Storing big data structures like transaction histories on every replica is seldom useful or practical [Stonebraker, Madden, et al., 2007].

w_h to the history node n_h, and w_l back to the N_c node that started `withdraw`. w_h updates the history, and w_l sets `last=Some(80)` on the client. `set` executes two actions. The first action is dispatched to n_2 (not labeled in Figure 2.8) and reads its copy of `bal`. The second action is dispatched to n_c (the one that executed `withdraw`), where it updates `max` to the computed value and resets `last`.

Protocol. In contrast to full replication, where any conflicting method can be dispatched to any node, in partial replication some nodes might never get a conflicting action. PEEPCO takes advantage of that fact to minimize consensus invocations. PEEPCO infers two statically fixed disjoint sets of nodes for a given program: *consensus nodes* N_{cns} and *non-consensus nodes* N_{nc}. Non-consensus nodes N_{nc} do not participate in consensus and only listen for decisions from N_{cns}. The N_{cns} nodes must include any node on which a conflicting action executes. In our example, only `withdraw` has a conflicting action w_b, which executes on $N_r = \{n_1, n_2\}$. As other methods are also dispatched on N_r, we let $N_{cns} = N_r$ (encompassed with a dotted line in Figure 2.10).

We depict an execution of the extended protocol in Figure 2.10, where solid dots represent action execution and empty dots represent protocol events. Nodes in N_{cns} exchange votes as before (message {pr}). They agree on batches and notify N_{nc}, when: 1) a method is accepted into a batch and propagating next actions (dashed, unlabeled, lines in Figure 2.10; line Q_5 of Algorithm 1), and 2) the batch is finalized, with a new message {fin} (dashed lines). Here, w_b invokes the consensus. After accepting w_b into the batch, n_1 informs n_h and n_c that their actions can be executed.

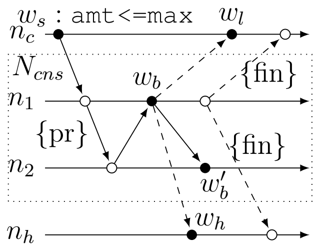


Figure 2.10: Execution

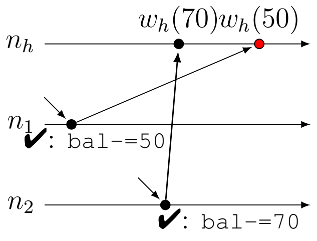


Figure 2.11: Redundant conflict

N_{nc} can make progress if they know which batch they are in and every action carries the batch number the method belongs to. When an N_{nc} node receives an action belonging to a batch the node is currently in, it delivers and executes the action in the order prescribed by the oracle. If the batch numbers do not match, the delivery of the action is delayed until the node reaches the needed batch. After executing an action, the N_{nc} node checks if it executed all actions of methods in the current batch and can proceed to the next batch (similarly as in line F_2 of Algorithm 1). Here, after executing w_h, n_h receives the control message {fin} that confirms `withdraw` is the only accepted method, after which it proceeds to the next batch.

PEEPCO executes read-only actions that precede conflicting actions without invoking consensus. Here, w_s is read-only: it simply performs the check and spawns w_b, which in turn invokes consensus. w_s, however, constrains the order of `set` calls, as shown subsequently.

In the general case, methods may have: 1) multiple conflicting actions and 2) actions preceding conflicting ones that modify the state. Actions in either of the groups cannot avoid consensus. To handle the general case, PEEPCO infers sets of actions, per method, that need to go through consensus (akin to transaction chains [Y. Zhang et al., 2013; Breitbart, Garcia-Molina, and Silberschatz, 2010]) and sets N_{cns} accordingly.

Inference. Inferring consistency constraints for actions is largely the same as for methods. In our example, the effects of `withdraw` and `deposit` actions on replicas N_r have the same consistency requirements as before, in the granularity of actions. However, when used for actions, INFER (Algorithm 2) can produce constraints that are too strong, because of a phenomenon that we call a *redundant conflict*. Unlike methods, actions depend on control flow, and their consistency requirements might depend on the requirements of the actions on which they depend. For example, reordering w_h from two concurrent w_b actions on n_h causes a violation (red dot in Figure 2.11). INFER discovers this trace τ_r when exploring traces. However, w_b requires strong consistency, which means that there will never be two concurrent w_bs, nor the trace τ_r, so there is no need for additional consistency requirements on w_h. The weakest constraint that prevents τ_r and the constraint of strong consistency between w_bs are not ordered by the consistency lattice. The former is an unnecessary constraint.

We modify INFER by modifying Merge to remove constraints that are inferred from infeasible traces. Instead of taking the least upper bound of all inferred constraints immediately (line 13 of Algorithm 2), $\mathsf{Merge}(S_b, \mathcal{C})$ defers this. Only at a point where all different orderings of preceding actions are explored, it includes new constraints into the solution for the current bound $\mathcal{C}_b$. It iteratively removes constraints $\mathcal{C}$ based on the traces τ_e they were inferred from. They are stored as pairs $(\mathcal{C}, \tau_e)$ in S_b (line 12 of Algorithm 2). If there exists a constraint $\mathcal{C}' \in S_b$, under which τ_e is infeasible due to control-flow dependencies, $\mathcal{C}$ is ignored and not included when joining the constraints and updating the solution of the current bound $\mathcal{C}_b$. As pruning takes place only after all non-redundant conflicts within a bound are merged, INFER maintains soundness and optimality defined in Subsection 2.2.5.

Constraints. Consider our example, where `set(50)` is invoked concurrently with `withdraw(80)` that started on n_c. The first action of `withdraw`, w_s, reads the initial value of `max` (100) and invokes w_b on n_1 (Figure 2.8). At that point, concurrently, `set` collects `bal` on n_2 and updates `max` on the same n_c. Afterwards, `withdraw` replies to n_c to execute w_l. However, `set` completed, setting `max` to `bal*x/100` based on a version of `bal` that did not witness the effect of `withdraw`. `last: Some(80)` thus holds a value greater than `max: 50`, violating `i3` (akin to a "write skew" anomaly [Berenson et al., 1995]). While strong consistency would prevent this violation, it would require unnecessary coordination between N_c and N_r. We extend PEEPCO with a new type of consistency constraint, *delay*, which forces a delay on `set`, locally on n_c, until all w_ls of pending `withdraws` finish or are aborted. This avoids conflicts and allows PEEPCO to batch the two together.

2.4 Implementation

PEEPCO is implemented in approximately 29k lines of Scala (22k for static analysis and 7k for the runtime). We use Leon [Kneuss et al., 2013] for lexing and parsing and Leon's verification conditions for data structures with recursive predicates. We use Z3 [De Moura and Bjørner, 2008] to discharge SMT queries. PEEPCO's networking stack is implemented using Akka [*Akka – actor toolkit and runtime, http://akka.io/* 2023] (on top of TCP-based channels). The input to PEEPCO is a Scala source file containing a library of functions and data constructors, along with the integrity and data distribution specifications.

Inference. We implement INFER (Algorithm 2) as a parallel worklist procedure and incorporate additional strategies for pruning into Choose:

- *Effect-driven pruning* examines variable reads and writes to prune traces that cannot change the invariants [Flanagan and Godefroid, 2005].

- *Symmetry breaking* avoids reordering identical effects from calls of the same method [Emerson and Sistla, 1996].

Protocol. PeepCommit can exhibit starvation when there are repeated aborts. To remedy this PEEPCO employs an optimization strategy for breaking ties during voting that favors calls with many retries, as well as with fewer conflicts according to $\mathsf{CanBatch}_{\mathcal{I}}$.

Correctness. The correctness of ensuring the batch-based execution by our end-to-end approach depends on: 1) the correctness properties of PeepCommit, 2) the correctness of integrity-driven consistency strengthening, 3) the correctness of ensuring that the delivery order of operations respects the consistency levels in the consistency lattice. In previous sections we provided correctness arguments for 1) and 2). Correctness of ensuring the right consistency orders are implemented relies on the implementation of all the supported consistency levels, besides strong consistency (ensured by the protocol), and weak consistency (ensured by reliable delivery property of the underlying links). Specifically, the correctness of our overall implementation relies on the implementation of: casual order (which we implement by tracking operation dependencies, as given in [Cachin, Guerraoui, and L. Rodrigues, 2011]) and write skew (which we implement by storing executed operations until their execution terminates).

2.5 Evaluation

In this section, we evaluate our implementation and illustrate the performance benefits of consistency optimization in batch-based consistency. We also evaluate PEEPCO's sensitivity to different workloads and the impacts of the explored bound at run-time.

Inference Benchmarks. We consider the following CRDTs and replicated applications introduced in prior work: *LWWRegister* [Shapiro et al., 2011b]: a last-write-wins register, where reads return the value of the latest write; *NNCounter* [Houshmand and Lesani, 2019]: a non-negative counter with increment and decrement; *Bank account* [Sivaramakrishnan, Kaki, and Jagannathan, 2015]: our motivating example; *Courseware* [Gotsman et al., 2016]: a database model for course enrollments; *Microblog* [Kaki, Earanky, et al., 2018]: a Twitter-like messaging application; *Auction* [Gotsman et al., 2016]: an auction with bidding and closing; and *Payroll* [Bailis, Fekete, et al., 2014]: an employee database.

We also consider a new use case: *DivGSet(x)*, a modification of the grow-only set from [Shapiro et al., 2011b] that can only be optimized by bounding the number of operations at run time. It supports insertions and limits the divergence of the set copies across replicas. A safety property bounds the difference in the set size between replicas by x. PEEPCO maintains safety by only allowing up to x concurrent `add` operations.

Some of our use cases are relational—they model databases with integrity constraints. We fully replicate relational use cases. We specify the integrity properties that are studied in prior work, including uniqueness, referential integrity (one-to-many and many-to-many relations between two tables), and the integrity of table rows [Kaki, Earanky, et al., 2018; Houshmand and Lesani, 2019]. For example, *Courseware* is a database modeling a school, which stores information about students and courses, and tracks students registering and enrolling in courses. It requires referential integrity between the course and student tables and allows adding and removing courses. *Payroll* stores employee and department relations, where employees and departments can be added and removed and their salaries updated. In addition to referential integrity between the tables, salaries have a row-integrity constraint.

We include the transactional use case TPC-E, which is an elaborate database benchmark that emulates a brokerage [Kaki, Priya, et al., 2019; Kaki, Earanky, et al., 2018]. We model "Trade-Result", a transaction that emulates completing a stock market trade. We representing the tables—`Trade`, `Holding`, `Broker`, and `Holding_Summary`—as fields, and distribute them across separate nodes. We assert aggregate equality on trades and commissions, as well as holding quantity across tables [TPC, 2010a].

We include partial replication use cases as variants of the partial bank account from Section 2.3. We use the specified data distribution and vary the methods of the object. In this

Table 2.1: Inference statistics after 15 minutes. Legend: $|O|$ - number of methods; int. - number of integrity properties (safe/congruent); $|\mathcal{C}|$ - number of found constraints; B_o, B_c, B_e, B_s - maximal explored bound, default, without consistency, effect, and symmetry pruning, respectively.

data distribution	*benchmark*	$\lvert O\rvert$	*int.*	$\lvert\mathcal{C}\rvert$	*max. bound*			
					B_o	B_c	B_e	B_s
full rep.	LWWRegister	2	1/1	1	7	6	6	4
full rep.	NNCounter	2	1/1	1	7	3	5	7
full rep.	Bank(w)	2	1/1	1	8	8	5	8
full rep.	Bank(w,d)	3	2/2	2	7	3	2	3
full rep.	Bank(w,d,i)	4	2/2	4	7	4	3	4
full rep.	Courseware	6	3/1	3	5	4	3	4
full rep.	Microblog	2	1/1	1	5	3	4	3
full rep.	Auction	3	1/1	2	6	4	5	5
full rep.	Payroll	7	3/1	2	3	3	3	3
full rep.	DivGSet(3)	2	1/1	1	8	5	8	5
distributed	Bank(w)	3	2/1	0	8	8	7	8
distributed	Bank(w,d)	4	2/1	1	7	6	4	5
distributed	Bank(w,d,i)	5	2/1	3	5	3	5	5
distributed	TPC-E	1	2/1	1	5	4	3	4
partial rep.	Bank(w)	3	1/2	1	8	8	4	8
partial rep.	Bank(w,d)	4	2/2	3	7	5	2	4
partial rep.	Bank(w,d,i)	5	2/2	7	6	4	2	4
partial rep.	Bank(w,d,s)	5	3/2	4	8	5	2	4

benchmark, when `set` is included, PEEPCO discovers and prevents the write skew anomaly between methods.

Inference Platform. Our experiments were performed on a machine with an Intel Xeon E5-2699 v2 with 24 2.2GHz cores and 48GB of RAM. We used OpenJDK 18.0.1 with one JVM thread per core.

Inference Results. We measured the maximal bounds PEEPCO explores, stopping after 15 minutes or exploring bound 8, and show the results in Table 2.1. For each benchmark, we report the data allocation scheme, the number of considered methods, and the number of safety and congruence invariants. For *Bank*, we evaluate multiple sets of methods; `Bank(O)` includes only the methods O. We use shorthand w,d,i,s to stand for `withdraw`, `deposit`, `interest`, and `set`, respectively. Each benchmark includes one safe read method that reads the state from any node in case of full replication and an additional congruent read in case of distribution and partial replication. For each benchmark, we report inference results: the number of constraints comprising the final solution and the maximal bound explored.

We give results for INFER (B_o) and three ablations, in which we turn off an optimization individually: B_c, which disables pruning based on found consistency constraints; B_e, which disables effect-driven pruning; and B_s, which disables symmetry breaking. The results, shown in the right-most columns of Table 2.1, indicate that while consistency-based pruning is the most effective in general, symmetry breaking has a significant impact when methods do not exhibit many violations. Effect-based pruning has visible impacts in the case of distribution and partial replication, as the number of execution steps that don't affect the checked property grows. To investigate limits of bounded exploration, we ran PEEPCO on *Bank(w,d,i,s)*, in partial replication, for 12 hours, and found that the maximum explored bound explored was 10.

We evaluate how inference scales with the number of steps to explore in INFER. We generated variants of the *NNCounter* benchmark, to contain multiple identical increment methods, where *exp n* designates n identical methods. We measured time to infer a particular bound, with a timeout of 480 seconds. We report results in Figure 2.12. We then extended the variants *exp n* to partial replication by including data distributed across different nodes and not including it in any integrity properties. We added two additional actions to each of the counter increment methods. The times to reach the same bounds in partial replication variants were within 10% of *exp n* (not shown). The results suggests that while PEEPCO suffers from exponential blowup, additional actions and longer method traces, do not pose scalability issues and run time performance penalties in the partial replication case of PEEPCO's protocol. Moreover, scalability issues are likely caused by inability to prune operations that do not exhibit violations, and would be otherwise safe to add to batches.

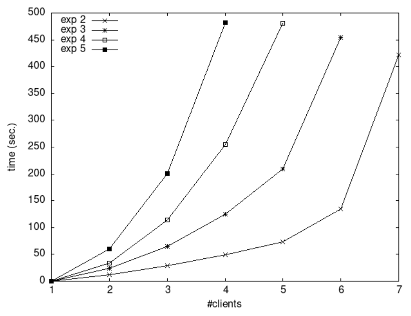
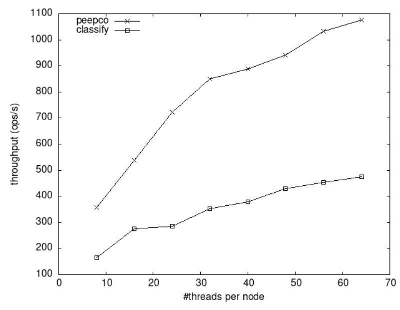


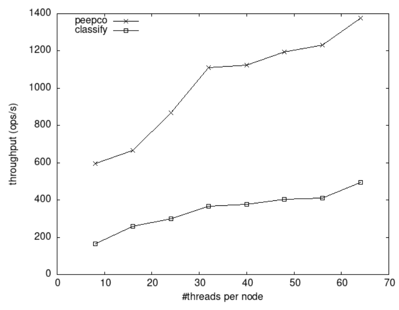
Figure 2.12: No-conflict scalability

Run-Time Platform. For our run-time performance evaluation, we deploy the distributed applications produced by PEEPCO on a commodity OpenStack cluster. Each cluster node has an Intel Xeon E5-2630 with 8 cores at 2.20GHz, 8GB of RAM, and runs OpenJDK 1.8.0. The inter-node latency is 0.5ms on average and we introduce a uniform delay of 40-60ms on each link at the network layer. Our client workloads model the YCSB benchmark in the high contention case for "hot records," where very few entries are accessed most of the time [B. F. Cooper et al., 2010]. The benchmark uniformly chooses an object from an array of size 1000, using a Zipfian distribution with $\theta = 0.5$ [B. F. Cooper et al., 2010]. We spawn between 8 and 64 client threads on 15 nodes. Each experiment runs for 180 seconds with 10 seconds of warmup.

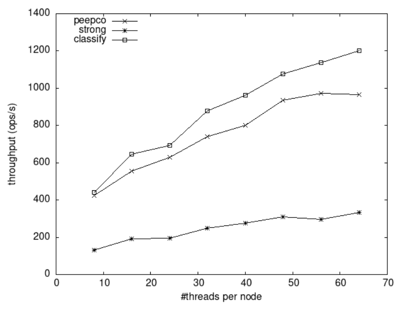
Run-Time Results. We compared the throughput of PEEPCO with two baselines: strong consistency and a fully static consistency optimization approach we modeled (*Classify*).



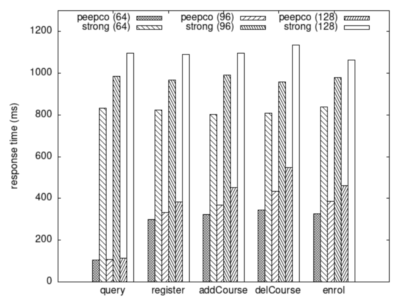
(a) Bank(w,d,i), uniform



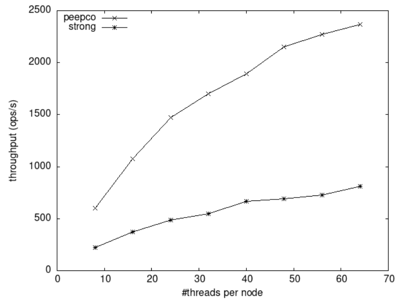
(b) Bank(w,d,i), skewed



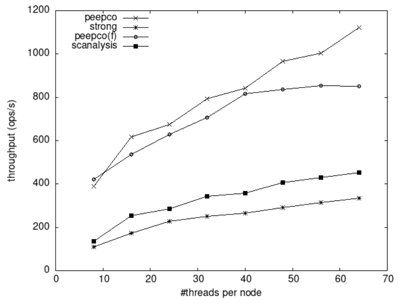
(c) Bank(w,d), uniform



(d) Courseware, uniform



(e) Courseware, uniform



(f) Bank(w,d,s), uniform, partial

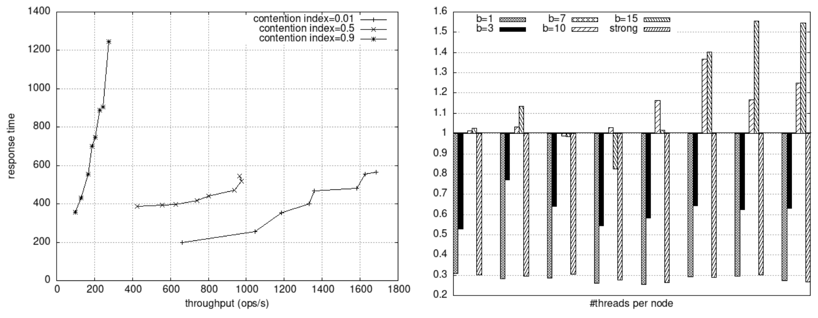
Figure 2.13: Run-time performance in full replication. For (a)–(c) and (e)–(f), higher is better. For (d), lower is better.

The strongly consistent baseline is implemented using two-phase commit (2PC) to ensure total order between operations [Gray and Lamport, 2006]. *Classify* chooses the optimal consistency level for a method but does not use dynamic information. It uses the same reliable and causal broadcast abstractions as PEEPCO to implement weak and causal consistency, and falls back to using 2PC for operations that require strong consistency. PEEPCO uses the inference results after 15 minutes of search (reported in Table 2.1). We measured throughput (Figure 2.13) on the *Bank* use case, fully replicated on 8 cluster nodes. Throughput is measured as operations completed per second. We performed three experiments. In the first experiment, *Bank(w,d,i), uniform* (Figure 2.13a), clients choose one of the four methods (three write methods, and a safe read) uniformly. PEEPCO achieves significantly more throughput than *Classify*—over 2.2× for 64 clients on each node. PEEPCO allows non-conflicting method calls at run time to proceed concurrently, which are assigned strong consistency by *Classify* and need to invoke 2PC in the worst case. Going through a round-trip communication to ensure total order in the worst case is expensive, and fast reads do not alleviate the issue. We also run the same experiment, on *Bank(w,d), uniform* (Figure 2.13b). (This is equivalent to the motivating example examined in [Sivaramakrishnan, Kaki, and Jagannathan, 2015; Houshmand and Lesani, 2019].) Classify outperforms PEEPCO and demonstrates better scalability, as deposits are executed without requiring an agreement phase and potential rollback due to conflicts in other operations in the batch, as well as its size. In our third experiment, *Bank(w,d,i), skewed* (Figure 2.13c), we only require congruence, and clients invoke `interest` only 5% of the time (and `withdraw` and `deposit` 35%). PEEPCO's speedup increases to 2.6× because the only method that requires total order (interest) is present infrequently causing fewer aborts when other two methods are invoked.

We examined response times on *Courseware*, deployed in the same way. We measure the time between issuing the method call and receiving a response of success and report the average (Figure 2.13d). PEEPCO's response time is significantly lower than that of the strong consistency baseline. The read-only method `query` has the lowest response time because it does not require a round-trip. The other methods have significantly higher response times, with `delCourse` dominating because it conflicts with `addCourse` and `enrol`, each of which has only one pairwise conflict. We also measure throughput for *Courseware* (Figure 2.13e) and find that it is higher than for *Bank* (∼1.5× speedup improvement, for 960 clients), due to the lower number of conflicts, relative to the executed operations in the uniform workload.

To demonstrate the benefits of partial replication, we measured throughput (Figure 2.13f) of the partially replicated *Bank(w,d,s)* use case, and compared PEEPCO to *SC-analysis*, an approach that models the analysis used in transaction chopping [Shasha, Llirbat, et al., 1995], as well as fully replicated PeepCommit, *peepco(f)*. *SC-analysis* runs optimized methods with weak consistency, but otherwise defaults to strong consistency (similar to Classify). We spawned 5 nodes as replicas of the balance and 9 nodes as the client nodes (N_r and N_c, respectively, as specified in Section 2.3). PEEPCO partially synchronizes actions on N_r which leads to fewer aborts and achieves better throughput (up to 30%) than full-replication PeepCommit, particularly as the number of clients grows. PEEPCO achieves a speed-up of 2.5× over SC-analysis because SC-analysis requires consensus for all methods except for the safe read.

To measure PEEPCO's sensitivity to contention, we varied the θ parameter of the Zipfian distribution [B. F. Cooper et al., 2010], using 0.01, 0.5, and 0.9 to emulate low, medium, and high contention, last being the highest severity. PEEPCO's batches fill faster when contention is higher, reducing performance (Figure 2.14a). We examined the effect of the batch size b on the cumulative throughput on the write-only, LWW register benchmark, which uses only writes (Figure 2.14b). We report the speedup relative to PEEPCO with b=7, for multiple batch sizes and varying numbers of clients. PEEPCO with b=7 achieves significant speedups relative to smaller batches. Larger batches provide speedups as the number of clients grows, up to 1.5x, due to more concurrent methods being able to saturate larger batches. We compared b=1 (no consistency optimization performed) to the strong consistency baseline,



(a) Bank(w,d), uniform (b) LWW reg., write-only

Figure 2.14: Sensitivity to contention and batch size

showing a small overhead (throughput within 5%). We also compare PEEPCO, b=1, on the LWW benchmark, to GridGain [*GridGain data store benchmark* n.d.], transactionally writing to registers, and found PEEPCO performs up to 20% better with 64 clients per node (not shown here).

2.6 Discussion

Our approach embodied in PEEPCO explores a new point in the consistency optimization design space, by combining static analysis results with information about operation executions at run time. In contrast to prior work in static analysis, it allows alleviating forcing strong consistency for the worst case scenario, even when it rarely occurs at run time. Our approach supports both replication and distribution and fits the high expressiveness needs of our overall end-to-end programming system. In addition, PEEPCO offers a safe, straightforward way to specialize a consistency model based on application requirements. This allows programmers to fine-tune their consistency requirements, and change different parts of the input specification while specializing the consistency model based on the overall application's needs.

Expressiveness and limitations. We implemented our approach in the tool PEEPCO, which provides a pipeline, in our overall system, that achieves consistency optimization with unbounded guarantees of the specified integrity. The tool uses the front-end DSL to achieve the shown expressiveness and the code synthesizer of the overall system, and adds to its JVM-based run-time. Two important limitations of the approach include scalability of the inference and round-trip communication costs for invoking methods. When using PEEPCO, programmers can split their overall programs and rely on the modularity of the specification, where different modules execute independently and rely on the safe and congruent interface for accessing their state, thus achieve better degree of scalability and potentially the extent of consistency optimization. Our evaluation shows the approach, when implemented by employing aggressive symmetry-breaking optimizations, can bring significant performance benefits over the state of the art, especially in cases where operations might conflict only with relatively smaller portions of other operations. Unlike traditional replicated systems, Peepco provides access to congruent snapshots that do not delay other operations. While the approach only allows optimizations within the bounds that can be explored statically, it can provide a higher degree of optimization when programmers allow the static analysis to run longer.

Chapter 3

SyCoord: Optimizing Consistency for Partially Replicated Data-Centric Distributed Applications

Inferring consistency requirements is a crucial part of the PEEPCO's consistency optimization process. The process, however, can also play an important part in the development and debugging of data-centric distributed applications. A big class of data-centric applications are developed on top of existing data stores that automatically handle data storage and provide programmers with means to choose the consistency model. The goal of this chapter is to generalize the consistency inference algorithm from Subsection 2.2.5 and present an approach for generating implementations that allow programmers not to synthesize programs that guarantee the given integrity when the number of concurrently invoked operations can be bounded, and also use such programs to examine and test behaviors under existing consistency choices. In this chapter we present and characterize a consistency inference framework that can be instantiated with different concrete search procedures, giving programmers the ability to instantiate a specialized inference procedure that is potentially more scalable for the particular classes of distributed programs at hand, compared to the general inference from Chapter 2.

Programming with data stores. Distributed data stores aim to give programmers an illusion of a single global state against which different nodes can perform computation [DeCandia et al., 2007]. In practice, however, because the data is distributed among many nodes, the programmer needs to choose a consistency model that is sufficiently strong to ensure correct behavior of the application but still allow the application to run efficiently. While some stores provide ways to control consistency, as described in Chapter 1, determining an optimal consistency model for a given set of operations that operate on the data store requires reasoning about the possible interactions that the consistency model may allow and making sure the application satisfies the overall integrity [Stonebraker, Madden, et al., 2007; Bailis, Fekete, et al., 2014; Shasha and Snir, 1988].

In contrast to existing data stores, replicated data-centric applications often require higher levels of expressiveness and flexibility for customizing the application based on the given replication, consistency, and topology requirements. Existing data stores that support partial replication offer only a limited set of consistency choices. Fine-tuning consistency under complex data replication schemes becomes challenging [Kallman et al., 2008; *Apache Ignite allocation modes* 2020; *Cassandra replication factor* 2020]. Programmers who require fine-grained control over data distribution and want to exploit properties specific to the given application and the intended replication schemes, may have no choice but to reason about consistency manually and implement a *custom integrated replicated application* in terms of low-level message passing and protocols [Thomson et al., 2012; Viennot et al., 2015; Belaramani et al., 2006; Stonebraker, Madden, et al., 2007]. The discovery and integration of consistency protocols with such complex data allocation schemes is often the most challenging part of the development process of distributed data-centric applications [Stonebraker, Madden, et al., 2007; Kallman et al., 2008; Belaramani et al., 2006; Alvaro, Conway, and Joseph M. Hellerstein, 2012]. In this chapter, we adopt the partial replication model introduced in Chapter 2 that allows the needed expressiveness in the presence of replicated data.

Consistency optimization with data stores. Consistency optimization remains a challenge, as even verification techniques that support expressive data models need to strike a balance between guarantees and automation. While prior work introduced expressive specification and verification techniques [Gotsman et al., 2016; Burckhardt et al., 2014; Wilcox et al., 2015; Lesani, Bell, and Chlipala, 2016], checking application-specific properties of distributed applications with a high degree of automation remains an open challenge. Prior work has exploited conditions such as commutativity and confluence under which a set of operations satisfy the given integrity properties [Bailis, Fekete, et al., 2014] and an automated approach for its inference. Full verification of distributed applications is challenging in part because of the need to find and specify suitably strong inductive invariants that justify integrity in the context of the fine-granular actions performed by the application. Such invariants capture deep semantic properties and are thus difficult to extract automatically [Kaki, Earanky, et al., 2018].

While some modern distributed data stores provide ways to relax strong consistency guarantees, some consistency optimization opportunities leverage low-level interactions and communication specific to the data and operation [Alur, E. D. Berger, et al., 2016]. Without more expressive reasoning robust for models such as partial replication, analyzing can lead to overapproximation, and defaulting to strong consistency to ensure correct behavior in the worst case. The tension between the need for expressive high-level invariants and the requirement for reasoning about the possible interactions between operations in the system greatly complicates the connection between high-level correctness arguments and possible consistency optimizations. In this chapter we characterize a model-driven perspective on the problem of consistency inference, which leverages seminal ideas from concurrency analysis [Shasha and Snir, 1988; Shasha, Llirbat, et al., 1995], and combines them with bounded model-checking techniques to achieve a new degree of expressiveness [Kneuss et al., 2013; Solar-Lezama, Rabbah, et al., 2005; Emerson and Sistla, 1996].

SyCoord's inference approach. They key insight behind SyCoord is that certain principles behind concurrency analysis can be extended and applied for automated consistency inference in an expressive programming model, given means to efficiently produce and check finite execution traces. SyCoord presents an automated bounded strategy for consistency inference in partial replication, based on high-level invariants. It uses concrete execution to drive exploration of different possible bounded traces in the system. The strategy can solely rely on the concrete inputs to check whether the refined set of paths satisfies the invariants, or use symbolic reasoning to verify for all concrete values. The approach considers traces incrementally in the number of different operation invocations in the system, finding consistency optimizations on a case-by-case basis. The results are then generalized into consistency protocols that prevent the explored traces, but also other unexplored traces at run time. Lastly, the approach synthesizes an implementation that employs optimized consistency protocols at run time, in case the optimized case can be detected, backtracking to strong consistency otherwise. If the number of concurrently invoked operations never exceeds the checked bound, the integrity given by the invariants is guaranteed.

SyCoord's tool. SyCoord infers the necessary consistency requirements and synthesizes a partially replicated application that implements the found requirements, for the given set of objects and operations. The interface to SyCoord is similar to that of Peepco. The programmers express *objects* and their *operations* with sequential code, oblivious to data distribution, replication, and behavior concurrency. Separately, they describe the *configuration*, i.e. how data is to be distributed and replicated across the system, as well as *invariants*, end-to-end properties that must be maintained during distributed execution. Given *test inputs* for object states and method parameters, and a scope of exploration, SyCoord infers the optimal consistency model and synthesizes a distributed implementation that preserves the invariants on all explored executions within the bound. SyCoord can be parametrized with new and custom consistency levels and protocols, supporting protocols that depend on the particular topology at hand.

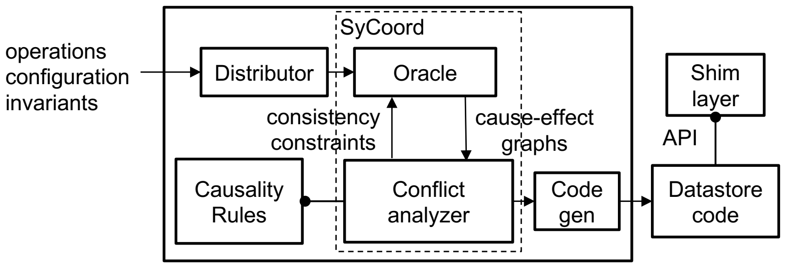


Figure 3.1: Components in SyCoord.

Figure 3.1 gives an overview of SyCoord, its components, and its integration with the overall system. Given the specification, the *distributor* component derives all possible distributed programs from the given sequential operations, by allocating parts of executions to different nodes in the system. *Model checker* uses test inputs to explore bounded concrete executions of the derived distributed programs and discovers execution traces which violate an invariant. *Causality analyzer* then analyzes found violating traces to infer consistency requirements. Finally, *code generator* translates the solution in terms of found requirements into an executable code. SyCoord chooses the optimal set of protocols from the *rule repository*, which the resulting prototype calls at run time, from the shim layer. SyCoord is parametrizable in terms of the distributor, rule repository, and code generation, to customize operation splitting, and supported consistency levels and protocols. We describe components in more detail in their corresponding sections.

Limitations. The main limitation of SyCoord is its inability to provide soundness guarantees in the general case, where the the number of concurrently invoked operations might exceed the explored bound. However, in all of our benchmarks, SyCoord finds the consistency model that indeed generalizes to the unbounded cases. Moreover, when the verification of invariants in the inference is driven by the concrete inputs provided by the programmer, guarantees of found optimizations depend on the quality of the given inputs. In principle, the extent of SyCoord's verification depends on the checker with which it is parametrized. A stronger checker could be paired with SyCoord's consistency reasoning to detect consistency violations fast, but verify classes of unbounded traces for which no violating trace could be found, achieving soundness for any concrete values at run time.

Contributions. Overall, we make the following contributions in this chapter:

- A specification language embedded in Scala that decouples descriptions of functionality, specifications of partial data replication, and topology.

- The abstract *cause-effect framework* that defines rules of consistency inference in the partial replication model.

- *Conflict-driven propagation*, a general bounded inference algorithm that supports extensions for topology-specific protocols and liveness.
- Implementation and experimental evaluation of SyCoord's expressiveness, as well as performance speedups of the resulting data store prototypes, on existing and new benchmarks.

3.1 SyCoord's Data Store Model

In this section, we illustrate the execution model of the considered partially replicated data stores by turning a sequential object into a simple data store. While our example follows the one given in Chapter 2, for completeness, as well as to present SyCoord-specific language constructs (needed for the inference with concrete test values), we present it fully in the context of SyCoord here.

We consider implementing a bank account that supports deposits and withdrawals, and stores the current balance and a history of performed methods (a motivating example introduced in Section 2.1, adopted from [Sivaramakrishnan, Kaki, and Jagannathan, 2015]). Firstly, in SyCoord, we define the application logic of the bank account as a conventional Scala class, shown in Figure 3.2. (We will add specifications later.) We would like to avoid overdrafting the account, hence `withdraw` checks and modifies `balance` only if the balance is big enough.

```
class Account(b: Int, h: List[(String, Int)]) {
 var bal: Int = b // current balance
 var hist: List[(String, Int)] = h // history

 def deposit(x: Int) = {
  bal += x; hist += ("D", x) }
 def withdraw(x: Int) = {
  if (x <= bal) {bal -= x; hist += ("W", x) }}
}
```

Figure 3.2: Bank account class definition.

Fundamentally, to implement the data store, the functionality given in sequential code (Figure 3.2) is not sufficient. Figure 3.3 shows the intended store configuration for the bank account (in the middle of executing a withdraw method), where nodes n_1, n_2 store the `bal` field, and node n_{hist} stores the `hist` field. This store is in *partial replication* because while the `bal` field is replicated, it is not the case that every node stores the same set of fields. The programmers need to consider many orthogonal aspects of the distributed data store, such as data allocation and replication. In SyCoord, programmers can provide orthogonal specifications, in addition to the functionality: Figure 3.4 shows what programmers write (explained subsequently), to achieve a distributed data store with the described configuration. (We describe the syntax of the language in the following sections.)

In SyCoord, programmers specify data replication by providing a data store configuration in a Scala DSL, separately from the sequential code of the behaviors. As a distributed data store, we might deploy the bank account with the `bal` field *replicated* onto multiple nodes (for availability). As before, `hist` might be large and non-critical, thus replicating it makes little sense, and we would like to allocate it on a single, dedicated "audit" node.[1] This is achieved with a specification given in Figure 3.4. In general, a data store consists of a set of *nodes*. This is expressed in the first part of the store configuration from Figure 3.4. The `withNodes` construct (line 1) specifies these nodes: here, a single node `nHist` and a dynamically resizable set of nodes `n` onto which the balance will be replicated. Each node stores a set of *fields*. In the next line of the specification (line 2), programmers specify store objects to be considered: here, corresponding to the class `Account`. **`allocate`** then describes the replication strategy by connecting fields in the object to the declared nodes in the configuration (lines 3–4). SyCoord allows specifying partial data replication, *i.e.,* arbitrary assignment of fields to nodes. The invariants that need to hold during execution are given with **`ensuring`** (line 5) and the test values and scope of exploration with `withScope` (line 6–8). We explain the implications of specifying invariants and scope in the next section (Section 3.2), after explaining the data store model, and give the full SyCoord's specification DSL in Section 3.6. `states` and `invocations` specify initial state values for the objects and operations to consider in the inference. `choose` chooses values for parameters to the operation invocations, from the given tuple `vs`.

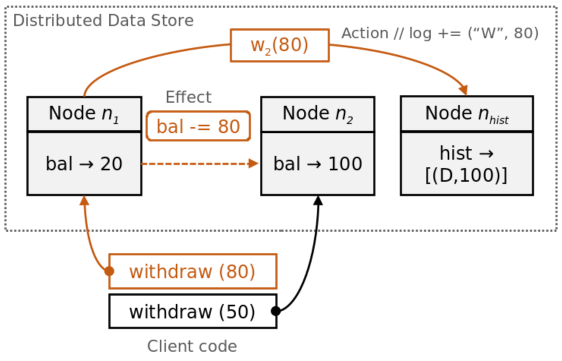


Figure 3.3: Partial replication data store.

[1]Separate replication schemes for `hist` and `bal` demonstrate additional complexity of partial replication and fine-grained consistency reasoning in the common example.

```
withNodes { case (n: NodeRegion, nHist: Singleton) =>
 withStore { case (a: Account) =>
  allocate(a.bal, n, replicate)
  allocate(a.hist, nHist) // don't replicate
  ensuring(safe(noOverdraft(a))) // invariant
  withScope(2) { val vs = (80, 50, 100)
   states(Account(0,Nil), Account(100,List(("D",100))))
   invocations(withdraw(choose(vs)), deposit(choose(vs))) }
}}
```

Figure 3.4: Bank account data store in SyCoord.

Implications of Fine-grained Replication. At run time, clients interact with the store by invoking *operations* (*i.e.,* methods of a given data store object) that might freely perform computation and change the object state. (We use the term "operation" to designate a method, when it's clear from the context.) In a fully replicated store, method invocations might be dispatched to an arbitrary replica, where they can be fully executed to produce an *effect* (a state update), which are then propagated to other replicas. A single method might need to access fields distributed over several nodes, as is the case here with `withdraw`, which updates both `bal` and `hist`, located on different nodes. Figure 3.3 shows a client that invokes two `withdraw` operations, of which the first one happens to be dispatched to n_1, and the second one to n_2. The figure shows an intermediate execution state, just before invoking the second method; we assume that the initial state of both n_1 and n_2 is `bal -> 100`. The execution model of SyCoord is more involved than that of fully replicated systems such as Sivaramakrishnan, Kaki, and Jagannathan [2015]. In the running example, SyCoord splits the `withdraw` method into two distinct operation parts, which we call *actions*: w_1 that operates on `bal` and produces an effect, and w_2 that modifies `hist`. (While oblivious to particular strategy of splitting methods into actions, SyCoord uses a heuristic-based strategy that minimizes communication; see Section 3.4.) When the `withdraw(80)` method is dispatched to n_1, the node executes only the $w_1(80)$ action; in this case, the overdraft check passes, so the action produces an effect `bal-=80`. This effect is applied locally at n_1, resulting in state `bal -> 20`, and also propagated to n_2 (shown with a dashed line in Figure 3.3), where it is yet to be applied. In addition, after applying the effect locally, the node n_1 communicates with n_{hist} for further execution of the action $w_2(80)$ (a solid line in Figure 3.3). While programmers simply write sequential code, SyCoord splits methods into actions, and ensures that actions and their effects get propagated to appropriate nodes. This involves analyzing the control and data flow of methods: *e.g.,* SyCoord makes sure that the action w_2 is issued only after w_1 has completed, because of the control-flow dependency between the two (as w_2 depends on the overdraft check, which gets executed together with

w_1).

The Need for Coordination. Let's consider the intermediate state in Figure 3.3, and how the execution might proceed from here. If the effect `bal-=80` reaches n_2 *before* the second withdrawal, no overdrafting occurs: by the time `withdraw(50)` is issued, n_2's `bal` will have been updated to 20, and the overdraft check will fail, producing no effect. If, however, these events happen in the opposite order, the execution of `withdraw(50)` on n_2 will start in a state `bal -> 100`, and the overdraft check of `withdraw(50)` will succeed, and the effect `bal-=80` will be applied afterwards, overdrafting the account (`bal -> -30`). This is a classical example of how executing methods concurrently without coordination between nodes can cause unexpected behaviours and violate application-level integrity constraints.

To eliminate in this scenario, the programmers need to enforce *strong consistency* (*i.e.,* execution in the same order on every node). However, enforcing strong consistency is prohibitively expensive [Harding et al., 2017; Abadi, 2012], and we would like to avoid enforcing it whenever possible: for example, when issuing many deposits in a row. Solutions that are based on efficient replicated datatypes [I. Zhang et al., 2016; Shapiro et al., 2011b] do not help here, as expressive invariants might not be easily expressible in terms of methods on the needed datatypes. As shown before, allowing programmers to state the high-level integrity invariants and automatically obtain the weakest consistency that guarantees preservation of the invariants, can alleviate significant effort in development of efficient distributed applications. With partial replication, consistency inference needs to *examine all the possible graphs of relationships* between individual operation parts on different types of nodes in the system. This poses new challenges and goes beyond prior work in coordination synthesis that limits expressiveness to full replication (where all the nodes in the system are identical and operations execute atomically) [Kaki, Earanky, et al., 2018; Sivaramakrishnan, Kaki, and Jagannathan, 2015; Houshmand and Lesani, 2019]. In the next section we demonstrate how SyCoord does that.

3.2 Coordination Synthesis in SyCoord

Recall that in our example store, we would like to ensure that the account is never overdrafted, concretely, that `bal` is non-negative and the entries in the `hist` also add up to a non-negative number. It is straightforward to specify this property using an executable predicate, expressed as regular Scala code:

```
def noOverdraft(a: Account) = { 0 <= a.bal && 0 <= sum(a) }
```

where `sum` is a function that folds over the history:

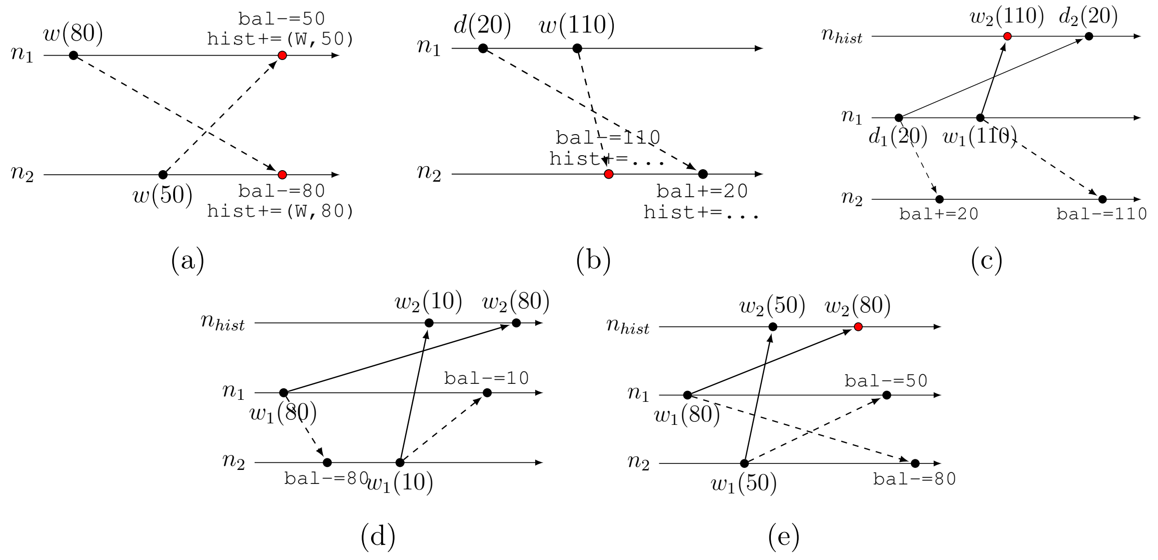


Figure 3.5: Execution diagrams for the bank account, in full replication (a, b) and partial replication (c, d, e). Red dots denote invariant violations. Lines between nodes denote inter-node message passing, where dashed lines denote effect propagation.

```
def sum(a: Account) = a.hist.fold(0)({
  case (res, ("D", x)) => res + x
  case (res, ("W", x)) => res - x })
```

Specifying Safety in SyCoord. In SyCoord, the predicate can be used directly to specify a *safety* invariant [Alford et al., 1985; Alpern and Schneider, 1985], a property that should hold at any point during the execution of any operation. Programmers add an **`ensuring`** clause to the specification (line 5 in Figure 3.4). (SyCoord also supports *liveness* invariants for expressing convergence over time, *e.g.,* `a.bal == sum(a)`; see Section 3.5. Liveness is the same as *congruence* introduced in Chapter 2, but does not entail the same run-time guarantees.) Given test values to drive the exploration, programmers can invoke SyCoord. Within `withScope` programmers specify values for the object state with the first set of parameters. (Multiple `withScope` and sets of invocations can be specified.)

Tools presented in prior work can handle some aspects of optimizing consistency in this example. These tools either force users to reason about low-level method effects [Sivaramakrishnan, Kaki, and Jagannathan, 2015], limit replication to the full model [Houshmand and Lesani, 2019], or identify method parts and annotate them manually [Milano and Andrew C. Myers, 2018]. SyCoord achieves the needed expressiveness and performs this automatically. Here, SyCoord explores invocations of `withdraw` and `deposit` with a combination of parameters `80`, `50`, `100`, on an `Account` with starting balance `0` and `100` (and history `Nil` and `List(100)`), within scope 2. SyCoord then reports found requirements and synthesizes executable code for each type of the declared nodes of the distributed program, which invokes consistency mechanisms to maintain the `noOverdraft` invariant.

Inferring Consistency in Full Replication. To illustrate the inner workings of SyCoord, let us first consider a simplified bank account, where all the nodes replicate all fields (as if the specification had `allocate(a.hist, n, replicate)`, in line 4 in our specification Figure 3.4). SyCoord starts exploring the space of possible interleavings of the given methods (all combinations of stated invocations, within a given search bound; see Section 3.6), until it finds one where `noOverdraft(a)` evaluates to `false`. The overdraft scenario of reordering effects of concurrent `withdraw` invocations causes an invariant violation (depicted in Figure 3.5a). Once SyCoord has detected the violation, it analyses its trace to determine the root cause of the violation. (SyCoord, similarly to prior work, leverages strengthening based on a consistency inference lattice, which is exposed through this analysis; see Section 3.3.) In the running example, the interleaving observes the effects of `withdraw` methods in different order on nodes n_1 and n_2. Since these methods are not causally related (by the standard definition of causality [Lamport, 1978b]), SyCoord infers that this pair of methods must be issued with strong consistency. It adds this relation to its current set of *consistency constraints*, which encode consistency requirements between particular methods on particular nodes.

Our analysis is based on graphs of effects, inspired by conflict analysis in concurrency [Shasha and Snir, 1988; Shasha, Llirbat, et al., 1995], and supports standard consistency levels, including weak, causal, and strong consistency. Then, SyCoord proceeds to explore more executions that are still feasible under the new constraints and finds another violating execution shown in Figure 3.5b. In this case, the root cause is re-ordering effects of `deposit` and a following causally related `withdraw`; to eliminate this interleaving, SyCoord adds a causal consistency constraint for this pair of methods, as they are causally related. Afterwards, there are no more feasible executions that violate the invariant, and the exploration terminates.

SyCoord reports the results and uses the repository to synthesize a data store that uses 2PC ([Gray and Lamport, 2006]) and vector clocks ([Fidge, 1987]) to enforce strong consistency between withdrawals on n_1 and n_2, and causal delivery of deposits before withdrawals on n_{hist}. The consistency optimization in the prototype achieves speedups of more than 2x relative to using strong consistency (in the specified full replication), while maintaining the invariants (see Subsection 3.7.2).

Consistency under Partial Replication. Let us consider what happens if the user decides to move `hist` onto a separate node, as specified in Figure 3.4. In general, such a change to the store configuration might require a completely different set of consistency protocols, and hence would be error-prone to check manually. Instead, SyCoord's configuration language is *compositional*: moving the history to a separate node requires a small, local change to the allocation statements. Given this configuration, SyCoord splits each bank account method into two *actions*: d_1, d_2 for `deposit` and w_1, w_2 for `withdraw`. SyCoord then proceeds by analyzing interleavings between the individual *actions* (and their effects) rather than full *methods*. SyCoord considers all possible method splits, considering all nodes the method can start (be issued) at, and chooses those that minimize communication. (Note that splitting strategy is parametrizable in SyCoord.)

Due to the partial replication, SyCoord might discover that different actions that belong to the same method might have different consistency requirements. Afterwards, SyCoord determines that any pair of d_2 and w_2 actions requires causal consistency (corresponding to causal ordering of their previous actions d_1 and w_1) on n_{hist} and that any two w_1 actions require strong consistency (on replicas n_1 and n_2). (The causes of these violations are analogous to the full-replication case; the first one is depicted in Figure 3.5c.) However, while any two w_1 actions require strong consistency, the corresponding w_2 actions ($w_2(10)$ and $w_2(80)$) require no coordination (on n_{hist}): the reason is that a w_2 action can only be issued after the corresponding w_1 action has performed the overdraft check, due to the previous discovery that w_1 actions have to be strongly consistent (see Figure 3.5d). SyCoord's reasoning not only preserves the given invariants in granularity of the partial replication, but discovers *consistency-requirement inter-dependencies*, which can achieve additional speedups: more than 3x, relative to using strong consistency (see Subsection 3.7.2).

Challenges. As mentioned in Chapter 1, consistency inference in partial replication exhibits combinatorial blowup in terms of the interleavings to explore, which makes full exploration infeasible even for small scopes. To achieve a scalable search, inference needs to prune the space explored, without missing relevant interleavings. Moreover, the strategy of exploration is crucial. Inferring conflicts from all traces which violate an invariant leads to incorrect solutions: some traces might exhibit *false conflicts*, as they are not feasible due to inter-dependencies with other discovered consistency requirements. In our example, a false conflict occurs if naively exploring reordering of $w(80)$ and $w(50)$ on n_{hist}, as given in Figure 3.5e. While this causes a violation of `noOverdraft`, the trace is not feasible as strong consistency has to be enforced between actions w_1 on n_1 and n_2.

3.3 Cause-Effect Framework

This section characterizes Peepco's inference procedure with an abstract non-deterministic *cause-effect framework* that defines rules for exploring execution traces and inferring consistency constraints from them, capturing the interaction of the model checker and causality analyzer from Figure 3.1. It relies on *cause-effect graphs*, a graph data structure that captures causality effects and allows inference of expressive consistency requirements. The framework characterizes how to build cause-effect graphs from traces by instrumentation, infer consistency constraints, and merge constraints inferred over different traces, while allowing *pruning for scalability without losing correctness* due to false (redundant) conflicts.

The framework is organized around two main processes. *Exploration* explores traces to capture relevant information in terms of cause-effect graphs. *Inference* infers consistency constraints from the captured information. We formalize both processes with one set of non-deterministic rules that can get applied in any order, incrementally discovering new traces, analyzing them and inferring consistency constraints. They can interchange in arbitrary order in taking steps, thus allowing different strategies for exploring traces and inferring consistency requirements. A fixed point of applying exploration and inference rules then produces the final result, for a given bound. The abstract nature of the framework allows instantiating the search with custom consistency protocols, and strategies of exploration, analysis and pruning; we demonstrate its extensibility in Section 3.5.

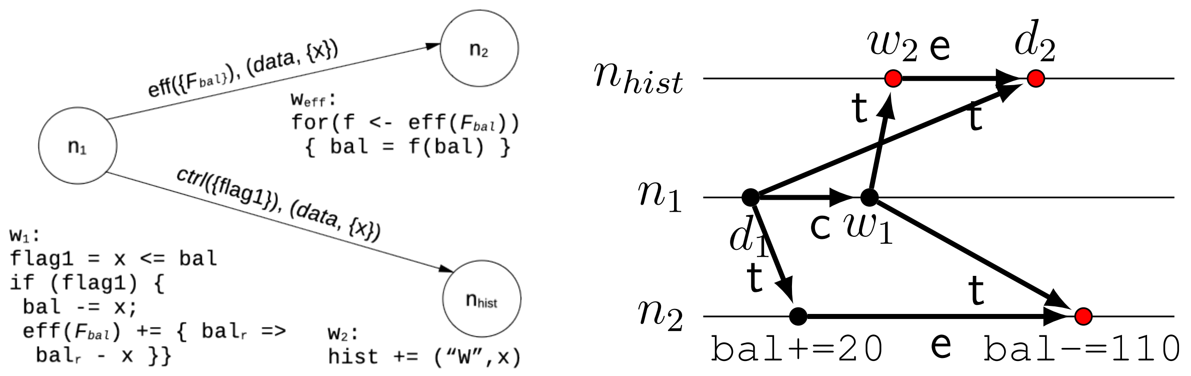
SyCoord instantiates the framework with a concrete strategy and a concrete set of consistency protocols in the *conflict-driven propagation* algorithm (Section 3.4).

3.3.1 Executions in Partial Replication

Our synthesis procedure relies on exploring interleavings of distributed executions through concrete evaluation. We define the dynamic semantics in terms of *distributed program graphs (DPG)* (inspired by [Sarkar, 1998]). A DPG represents a distributed execution of a (sequential) method as a graph. Note that due to replicated data, a method can access data at any replica and thus have multiple DPGs.

Node-local execution. We parameterize distributed execution with a host language λ_H. (Our implementation of SyCoord uses that of Scala.) The semantics is parametric *w.r.t.* the exact syntax of expressions e and local semantics of λ_H. λ_H defines a relation (akin to the standard relation in big-step operational semantics) $\langle s, e\rangle \rightarrow \langle s', \gamma\rangle$, where $s \in S$ and $s' \in S$ are the initial and final states (mappings from object instance *fields* to values, $S : \sigma\colon \mathsf{Field} \mapsto \mathsf{Val}$), e is an expression, and $\gamma : \mathsf{Id} \mapsto \mathsf{Val}$ is the environment containing results. (We capture results with a map to explicitly capture values that might need to get propagated to other nodes.) A *distributed store* σ maps nodes to states: $\sigma \in \Sigma\colon \mathsf{Node} \rightarrow \sigma$; *e.g.,* in our running example, the balance of the account object *a.bal* is replicated, thus both $\sigma[n_1]$ and $\sigma[n_2]$ contain `bal`.

DPGs. A DPG g is a static representation of a distributed execution of some operation



```
w1:
flag1 = x <= bal
if (flag1) {
 bal -= x;
 eff(F_bal) += { bal_r =>
  bal_r - x }}
```

```
w_eff:
for(f <- eff(F_bal))
 { bal = f(bal) }
```

```
w2:
hist += ("W",x)
```

Figure 3.6: A DPG and a cause-effect graph for the bank data store.

(method call) *op* across the store. It maps method parts to nodes and encodes the structure of distributed communication. Vertices v in the DPG are pairs (n, e) of a data store node n and a λ_H expression e, representing parts of the method. Edges $v \bullet\!\!\rightarrow_l v'$ reflect program order, where the label l captures standard control- and data-flow dependencies of v''s expression, as well as effect propagation. Labels can be of the form (Ctrl(c), d) or (Eff(S), d), where d captures data-flow dependencies, Ctrl(c) values of c of the control-flow, and Eff(S) the effect propagation, computed based on values of replicated fields S. (Note that effects, by default, effects capture all replicated values, while we present language extensions later that allow programmers to control this.) As described in the previous section, SyCoord splits the method `withdraw` into two parts, plus the effect: its DPG that starts at the node n_1 is shown in Figure 3.6. Because `bal` is replicated, we have an effect dependency edge between (w_1, n) and its effect on n_2. SyCoord uses the standard approach of producing DPGs and closures for distributed execution, adopted to support replicated sets of data and effect propagation [Andrew D Birrell and Nelson, 1984a; Epstein, Black, and Peyton-Jones, 2011]; see Section 3.6.

Distributed semantics. A distributed execution executes a set of DPGs G over a distributed store σ. A distributed *configuration* c is a pair $\langle \sigma, A \rangle$ of a store σ, and a set A of *actions* to execute. An *action* is a DPG vertex, paired with an environment $((n, e), \gamma)$, representing a point in execution of the operation part e, at node n, with values in γ. For a starting configuration $c_0 = \langle \sigma_0, A_0 \rangle$, σ_0 reflects the starting store state, and A_0 contains all entry point nodes $\mathsf{Fst}(w)$ of DPGs in $w \in G$ (first nodes in their topological orders), coupled with the initial environment γ_0 containing parameter values, *i.e.,* $\{\mathsf{Fst}(w) | w\} \times \{\gamma_0\}$. From the specification, values and the scope, SyCoord builds a set of configurations c_0 to explore; see Section 3.6.

Figure 3.7 defines distributed execution as a small-step evaluation relation between distributed configurations $\langle \sigma, A \rangle \rightarrow_e \langle \sigma', A' \rangle$. E-Step explores one step in the execution:

$$\textbf{Distributed execution} \quad \boxed{\langle \sigma, A \rangle \rightarrow_e \langle \sigma', A' \rangle}$$

$$\textsc{E-Step}\frac{\begin{array}{c} a = (v, \gamma) \quad v = (n, e) \quad \langle \sigma[n] \cup \gamma, e \rangle \rightarrow \langle \sigma', \gamma' \rangle \\ A' = \{(v', \gamma' \cap d) | v \bullet\!\!\rightarrow_l v', l = (\text{ctr}, c, d), \gamma'[c] = \text{true}\} \end{array}}{\langle \sigma, \{a\} \uplus A \rangle \rightarrow_e \langle \sigma[n \mapsto \sigma'], A \cup A' \rangle}$$

$$\textsc{E-Effect}\frac{\begin{array}{c} a = (v, \gamma) \quad v = (n, e) \quad \langle \sigma[n] \cup \gamma, e \rangle \rightarrow \langle \sigma', \gamma' \rangle \\ A_{\text{e}} = \{(v', (\gamma' \cap d) \cup \gamma'[\text{F}_S]) | v \bullet\!\!\rightarrow_l (n', e'), l = (\text{eff}, S, d), S \subseteq \text{dom}(\sigma[n'])\} \end{array}}{\langle \sigma, \{a\} \uplus A \rangle \rightarrow_e \langle \sigma[n \mapsto \sigma'], A \cup A_{\text{e}} \rangle}$$

Figure 3.7: Semantics of distributed executions

it non-deterministically picks an action a from A and evaluates its expression on the given node. It uses the results of the evaluation γ' to update the node state and include new enabled actions to explore, together with new environments containing values for all data dependencies $(\sigma' \cap d)$. It enables actions A' that depend on the control-flow A', by consulting the edge label with Ctrl(c) and adding only actions which control-flow dependencies $\gamma'[c]$ evaluate to true. E-Effect performs a similar step but picks an edge to create its effect propagation for a field. The new actions are those containing effects A_{eff}, for all replicated fields, where the dependency Eff(S) checks if fields S at node n (*i.e.,* the domain of $\sigma[n]$) have been modified, and if so, includes the values for the replicated fields into the passed environment. We write $\gamma'[\text{F}_S]$ to select values of all fields in S from γ'. Note that only the state of one field can get updated, to model stores that cannot update multiple rows atomically. In Figure 3.6, E-Step executes w_1 on n_1, enabling the effect propagation eff(F_{bal}), which E-Effect applies for the state change `bal=bal-110` on n_2.

Distributed execution. A *distributed execution trace* a τ is a sequence of configurations $c_0, \ldots, c_n$ such that $c_{i-1} \rightarrow_e c_i$ for all $i \in [1, n]$. Figure 3.6 presents a τ of length 6, where two DPGs corresponding to `withdraw` and `deposit` calls, are fully executed.

3.3.2 Elements of Inference

SyCoord operates by solving *coordination-synthesis problems*, collecting *consistency constraints* as candidate solutions (which are then generalized into consistency protocol choices). For a given program, invariants, test harness, and a bound, SyCoord produces a set of synthesis problems (it combines DPGs, store values, and call parameters, as explained in Section 3.6) that needs to be solved up to a certain bound. The best found candidate solution that solves all problems is then returned as the result of consistency inference. (SyCoord produces a synthesis problem for every combination of method calls, DPG variants based on replication, store values, and method-call parameters; as explained in Section 3.6).

To define all needed elements for consistency inference in SyCoord, we reintroduce some fundamental notions from Chapter 2 and redefine them in the context of the general inference framework of SyCoord.

Invariants. An *invariant* I is an executable predicate, *i.e.,* a Boolean λ_H expression. Invariants are evaluated against sets of states called "global snapshots". A global snapshot is a state constructed from a union of states at some nodes, such that all combinations of replicated values are considered. (This definition expands the notion of global snapshot from Section 2.2.) The set of all global snapshots $\mathsf{global}(\sigma)$ is defined as an ordered map-join of states of all nodes:
$\left\{ \bigcup^{*}_{n \leftarrow \{n_1,...,n_k\}} \sigma(n) | (n_1, ..., n_k) \in \text{perm}(1..k) \right\}$
where $\text{perm}(S)$ returns permutations as sequences (not sets) of the set S. We write $\bigcup^{*}$ to designate ordered union of maps over the results, where $a \cup^{*} b$ represents $a \cup b$, but whenever $(x, x_a) \in a \wedge (x, x_b) \in b$, the result contains pair from the second operand, *i.e.,* if $x_a \neq x_b$, $(x, x_b) \notin a \cup^{*} b$.

This is to model a general client that would execute an invariant with a load-balancer that, for an accessed field, chooses a random replica that contains it. For a safety invariant I, a trace τ satisfies I, written $\tau \models I$, if, for its every configuration $c_i = (\sigma_i, A_i)$, I evaluates to true in all its global snapshots $\mathsf{global}(\sigma_i)$ (for all sets of concrete test values); *e.g.,* in Figure 3.5b, there are two global snapshots (states at both replicas, $\{\sigma(n_1), \sigma(n_2)\}$), while the snapshot $\sigma(n_2)$ violates the invariant `noOverdraft`.

Consistency constraints. The framework is parameterized by a set of *consistency levels* C_L, which are identifiers that encode consistency models. A level encodes relationships of certain actions and nodes in a distributed execution. A *consistency constraint* encodes a predicate on execution traces τ, that contains a *consistency level.* These levels encode a particular consistency mechanism at run time.

A consistency constraint is a tuple (l, o_1, o_2, n), where l is a consistency level, which serves as an identifier for different consistency models. The level l effectively describes the relationship, namely ordering, of actions of o_2 and o_1, on node n. There exists a special level: $l = \mathit{weak}$, weak consistency, that allows any trace regardless of how o_1 and o_2 are executed. Checking if a constraint holds entails checking whether actions of o_2 are executed in a certain order with respect to actions of o_1, according to the consistency designated by l, on node n. We say that a trace τ is feasible under a constraint, $\tau \models (l, o_1, o_2, n)$ if the relationships between all actions of o_1 and o_2, on node n, in τ, are in accordance with l. *e.g.,* the trace in Figure 3.5b is not feasible under the constraint $(\text{causal}, w, d, n_2)$, as the effect of $w(110)$ was not executed in the causal order with respect to $d(20)$ on n_2. Feasibility is extended straightforwardly to sets of constraints $\mathcal{C}$.

Consistency Lattice and Solutions. A candidate solution is a set of constraints. $\mathcal{C}_s$ is a *solution* to $\mathcal{S}$ iff for all traces τ starting from $S.c$, if $\tau \models \mathcal{C}_s$ then $\tau \models \mathcal{I}$, as defined next. We define a "weaker than" relation $\sqsubset$ between consistency levels and consistency constraints. We instantiate SYCOORD with three levels: weak, causal, and strong consistency and the lattice is depicted in Figure 3.13a. Consistency constraints then form a lattice given the consistency level: *e.g.,* weak $\sqsubset$ causal $\sqsubset$ strong consistency. We define a "weaker than" relation between solutions $\sqsubset$ and a lattice of candidate solutions $(2^{\mathcal{C}}, \sqsubset)$ (where $\bot$ and $\top$ represent no consistency constraints, and strongest levels for any two operations and a node, respectively). Consequently, for two consistency constraints, $\{(l_1, o_1, o_2, n)\} \sqsubset \{(l_2, o_1, o_2, n)\}$ if $l_1 \sqsubset l_2$. Therefore, sets of constraints form a lattice $(2^{\mathcal{C}}, \sqsubset)$. If $\mathcal{C}_s$ is a solution to $\mathcal{S}$, then any $\mathcal{C}'$, such that $\mathcal{C}_s \sqsubset \mathcal{C}'$ is also a solution. SYCOORD finds a weakest $\mathcal{C}_s$ such that all for explored traces τ, if $\tau \models \mathcal{C}_s$ then $\tau \models \mathcal{I}$. In Figure 3.5b, a set $\mathcal{C} = \{(\text{strong}, w, d, n_2)\}$ prevents the given violation, but it is not the weakest that does so, as a weaker $\{(\text{causal}, w, d, n_2)\} \sqsubset \mathcal{C}$ prevents it as well. We set $\top$ to be a candidate solution that forces all actions to be *strongly consistent* with each other.

Well-formedness. A synthesis problem does not have a solution if any trace under the strongest candidate solution $\top$ violates $\mathcal{I}$. Note that there might not be such a set of constraints and invariant violations cannot be prevented with consistency choices. Given a consistency lattice and its top element $\top$, $\mathcal{S}$ has no solution if there exists $\tau \in e(\mathcal{S})$, such that $\tau \models \top$ and $\tau \not\models \mathcal{I}$. We refer to $\mathcal{S}$ as *well-formed* if $\top$ is a solution, and otherwise *ill-formed*. In an ill-formed problem, the invariant violation cannot be prevented with consistency choices.

3.3.3 Cause-Effect Graph Instrumentation

Instrumented execution extends the semantics of distributed execution. It collects information about consistency requirements by building *cause-effect graphs.*

Cause-effect graphs contain information about the given execution trace, pertaining to consistency, and allow efficient inference of consistency requirements (inspired by static concurrency analysis [Shasha and Snir, 1988]). The key property of cause-effect graphs is that they capture sufficient information to allow writing different graph analyses, in the form of graph traversals, of cause-effect graphs, to infer a broad class of requirements and inference locally, potentially for incomplete traces, right after the conflict is found. For a given trace τ over DPGs G, a *cause-effect* graph $\mathcal{G} = \langle V, E \rangle$ is a graph whose vertices are actions of G ($V \subseteq A$) and edges $v \xrightarrow{\mathsf{l}} v'$ capture the shape of the execution and embed additional information through the label $l \in L$. Our framework captures three types of edges:

- $v \overset{\mathsf{t}}{\leftrightarrow} v'$ if $v \bullet\!\!\rightarrow_l v'$ exists in some DPG in G (intuitively: a "sends a message" to b, *i.e.,* v

and v' executed in the program order on different nodes); for example, in Figure 3.6, $w_1 \overset{\mathsf{t}}{\longleftrightarrow} w_2$ because they are sequentially composed in the program, and $w_1 \overset{\mathsf{t}}{\longleftrightarrow}$ `bal-=110` because the latter is an effect of the former.

- $v \xrightarrow{\mathsf{e}} v'$ if v' executed after v on some node n (intuitively, a is visible to b, on node n); *e.g.,* in Figure 3.6, $d_1 \xrightarrow{\mathsf{e}} w_1$ and $w_2 \xrightarrow{\mathsf{e}} d_2$.

- $v \xrightarrow{\mathsf{c}} v'$ if v' is the entry point of a dpg w that is executed after v on some node n, *i.e.,* $v' = \mathsf{first}(w)$ for some $w \in G$, where first contains the first action of w. (This edge designates a "happens before" relation between two operations and implies $v \xrightarrow{\mathsf{e}} v'$.) For example, in Figure 3.6, $d_1 \xrightarrow{\mathsf{c}} w_1$. However, there is no c-edge between w_2 and d_2, since d_2 is not the first action of `deposit`.

The nature of labels allows adding them incrementally to the graph to allow eager inference of requirements. Figure 3.6 shows a cause-effect graph for an execution of the running example. We have $d_1 \xrightarrow{\mathsf{c}} w_1$; however, there is no c-edge between w_2 and d_2, since d_2 is not a starting action of the `deposit` call (*i.e.,* its DPG).

Instrumented execution. An instrumented configuration, which extends regular configurations, is a tuple $\omega = \langle \sigma, A, \mathcal{G}, F, P \rangle$, where $\langle \sigma, A \rangle$ is a distributed configuration; $\mathcal{G}$ is a cause-effect graph constructed up to ω; $F \subseteq \mathcal{I}$ is a set of failed invariants at ω; and $P \subseteq \mathcal{I}$ is a set of pending invariants at ω that remain to be checked. The instrumented step relation $\rightarrow_i$ relates instrumented configurations $\omega \rightarrow_i \omega'$, and formalizes incremental construction of the cause-effect graphs and the discovery of violating traces. Cause-effect graphs enables efficient exploration of interleavings *without having to store and examine entire executions.* Synthesis rules, explained subsequently, then build on top of this relation.

The relation is defined in Figure 3.8. Note that the rule relies on the uninstrumented rules E-Step and E-Effect, thus covering both action and effect execution, while capturing instrumentation information. The rule I-Step performs an "execution step": it executes action a (through $\rightarrow_e$), adds the action and two types of new edges to the cause-effect graph. $last_{\mathcal{G}}$ is an auxiliary function that returnes the last action executed on the given node, based on the cause-effect graph $\mathcal{G}$. first is an auxiliary function that returns true if an action is the first of its DPG. One set of edges captures visibility of a from the last action executed on n (returned by $last_{\mathcal{G}}(n)$, in the current cause-effect graph $\mathcal{G}$), and has either the e or c label, depending on whether a is the first action of its DPG (*i.e.,* causally dependent). The other set represents forward t edges from a to each of its next actions a'. (These forward edges are "dangling"; target actions will be added to the graph later.) For every new expanded configuration, all invariants are pending (and $F = \emptyset$). I-Safe picks a safety invariant and checks if it evaluates to true in all current global snapshots. In case any evaluation fails, it

Instrumentation $\boxed{\langle \sigma, A, \mathcal{G}, F, P \rangle \rightarrow_i \langle \sigma', A', \mathcal{G}', F', P' \rangle}$

$$\textsc{I-Step}\ \frac{\begin{gathered}\langle \sigma, \{a\} \uplus A \rangle \rightarrow_e \langle \sigma', A' \uplus A \rangle \quad \ell = \mathsf{first}(a)\ ?\ \mathsf{c} : \mathsf{e} \\ E' = E \cup \{\mathrm{last}_{\langle V,E \rangle}(n) \xrightarrow{\ell} a\} \cup \{a \overset{\mathsf{t}}{\leftrightarrow} a' | a' \in A'\}\end{gathered}}{\langle \sigma, a \uplus A, \langle V, E \rangle, F, P \rangle \rightarrow_i \langle \sigma', A' \cup A, \langle V \cup \{a\}, E' \rangle, \emptyset, \mathcal{I} \rangle}$$

$$\textsc{I-Safe}\ \frac{\begin{gathered}\mathsf{safe}(e) \in \mathcal{I} \quad R = \{r \mid \forall s \in \mathsf{global}(\sigma). \langle s, e \rangle \rightarrow r\} \quad \mathrm{good} = \bigwedge_{r \in R} \\ F' = \mathrm{good}\ ?\ F : F \cup \mathsf{safe}(e)\end{gathered}}{\langle \sigma, A, \mathcal{G}, F, P \uplus \mathsf{safe}(e) \rangle \rightarrow_i \langle \sigma, A, \mathcal{G}, F', P \rangle}$$

Figure 3.8: Instrumented execution.

adds it to F', which represents the set of failed invariants of the configuration. Figure 3.6 shows a cause-effect graph after several applications of the rules, where three of the I-Safe applications discovered violations (shown in red) of `noOverdraft`.

3.3.4 Consistency Constraints Inference

Inference uses the derived cause-effect graphs from instrumented executions to infer consistency constraints. It infers constraints for each step of the execution by checking invariants, analyzing their cause-effect graphs to determine the right constraint that would prevent invariant violations. Intuitively, the rules infer constraints for steps that violated any invariants and propagate back and merge the inferred constraints from explored configuration expansions back to the starting configuration. The set of constraints at the starting configuration, where no invariant is failed, represents the solution.

Graph Analysis

$\mathsf{Analyze}\colon \mathcal{G} \rightarrow \mathcal{I} \rightarrow 2^{\mathcal{C}}$ is a function used in the inference process. It takes a cause-effect graph g and an invariant, returning the weakest set of constraints that would prevent the trace encoded in g (*i.e.,* it makes any trace that exhibits g infeasible). Analysis returns the weakest set of consistency constraints that prevent a violation in the cause-effect graph. It is parametrized by the rule repository (in the basic version of SyCoord, rules for checking weak, causal, and strong consistency); *e.g.,* for a trace in Figure 3.6, after executing w_2 $\mathsf{Analyze}$ returns $\{(\mathrm{causal}, w, d, n_{hist})\}$ for the cause-effect graph consisting of d_1, w_1, and w_2, as it is the weakest that would prevent the violation, due to w_2 being delivered out-of-order on n_{hist}.

***Soundness of* Analyze.** Our framework relies on guarantees of $\mathsf{Analyze}$: it should return the least set of constraints that disables the violating action in the $\mathcal{G}$. If $\mathsf{Analyze}(\mathcal{G}, I) = \mathcal{C}$, any execution τ that produces $\mathcal{G}$ according to the instrumented semantics is infeasible under

Sufficient Coordination $\boxed{\langle \sigma, A, \mathcal{G}, F, P\rangle \triangleright \mathcal{C}}$

$$\textsc{Unsat}\frac{\omega.F \neq \emptyset}{\omega \triangleright \top} \qquad \textsc{Disc}\frac{\langle \sigma, A, \mathcal{G}, \{I\} \uplus F, P\rangle \triangleright \mathcal{C}}{\langle \sigma, A, \mathcal{G}, F, P\rangle \triangleright \mathsf{Analyze}(\mathcal{G}, I) \sqcap \mathcal{C}}$$

$$\textsc{End}\frac{}{\langle \sigma, \emptyset, \mathcal{G}, \emptyset, P\rangle \triangleright \top} \qquad \textsc{Coll}\frac{C = \{\mathcal{C}' \mid \omega \to_i \omega', \omega' \triangleright \mathcal{C}'\}}{\omega[F := \cup_{\omega \to_i \omega'} \omega'.F, P := \cup_{\omega \to_i \omega'} \omega'.P] \triangleright \sqcap_{\mathcal{C} \in C} \mathcal{C}}$$

$$\textsc{Wkn}\frac{\mathcal{C} \in 2^{\mathcal{C}} \quad S = \{\mathcal{C}' \mid \omega \to_i \omega', \omega'.\mathcal{G} \models \mathcal{C}, \omega' \triangleright \mathcal{C}'\}}{\omega \triangleright (\sqcap_{\mathcal{C}' \in S} \mathcal{C}') \sqcap \mathcal{C}}$$

Figure 3.9: Synthesis rules

$\mathcal{C}$: $\tau \not\models \mathcal{C}$, and for any other $\mathcal{C}'$, $\tau \not\models \mathcal{C}' \to \mathcal{C}' \sqsupset \mathcal{C}$. In our bank example, returning strong consistency for the trace that exhibits the causal requirement would lead to a sub-optimal solution.

Note that the constraints returned by Analyze are not guaranteed to be part of the solution, *i.e.,* $\mathcal{C} \sqsubset \mathcal{C}_s$ does not necessarily hold. This is because a *false consistency constraint* c_f might be needed to prevent the violation in the current $\mathcal{G}$ but not needed in conjunction with other inferred constraints. In Figure 3.5d, Analyze of the cause-effect graph formed with $w_1(80), w_2(80), w_1(10), w_2(10)$ infers (strong, w, w, n_{hist}); however, as discussed in Section 3.2, this is not in the overall solution. The synthesis framework, however, filters out false constraints in the solution, as explained subsequently.

Inference

The inference infers constraints for each explored trace and merges sets of constrains inferred for multiple traces. We define it with a judgment $\omega \triangleright \mathcal{C}$ ("$\mathcal{C}$ is sufficient coordination for ω"), which says that all (instrumented) executions whose traces expand ω, and are feasible under $\mathcal{C}$, might only violate an invariant in ω.F. The search starts with a configuration $\langle \sigma_0, A_0, \emptyset, \emptyset, \mathcal{I}\rangle$ which represents the initial execution configuration, and every invariant in $\mathcal{I}$ needs to be checked. Using this judgment, we obtain a *solution* $\mathcal{C}$ for all traces to be explored, by deriving $\langle \sigma_0, A_0, \emptyset, \emptyset, \emptyset\rangle \triangleright \mathcal{C}$, which is the starting configuration, with no violated invariants where all invariants were checked. Note that this derivation can occur only after all possible (bounded) traces are explored and checked for the satisfiability of all the invariants.

Inference rules are given in Figure 3.9. They can be applied for any expanded trace, *i.e.,* instrumented configuration ω, thus allowing arbitrary strategies for progressing exploration and inference. However, the only way to infer for the starting configuration is to propagate back constraints inferred for its expansions; this forces either pruning or analyzing all violating traces.

- The rule Unsat says that any configuration ω that has a violated invariant ($\omega.F$ is non empty) requires no consistency constraints ($\top$). Due to the fact that we're inferring configurations where no violation occurs, we will need to strengthen this set.

- DISC discovers consistency constraints from one of the failed invariants I using Analyze to obtain a refined set of constraints under which the current trace is feasible with respect to the invariant. The new constraints are derived by taking the least upper bound $\sqcap$ to make sure that existing violations of that configuration are also prevented. If the exploration came to an end and no invariants are violated, no constraints are needed (END). (Note that invariants might still be pending to be checked by instrumented execution.)

- COLL accumulates constraints from expansions of a configuration ω and computes a least upper bound, which guarantees sufficient coordination for all traces expanded from ω, but accounting for all violated invariants from the expansions. The notation $\omega[F := \cup_{\omega \to_i \omega'} \omega'.F]$ designates a copy of the tuple ω, with the new value for F.

- WKN allows solutions to be discovered while assuming some set of constraints. This rule allows eliminating any potential "false constraints", by incorporating a set of constraints $\mathcal{C}$ into exploration, filtering derived configurations based on $\mathcal{C}$ ($\mathcal{G} \models \mathcal{C}$), if $\mathcal{C}$ is inferred somewhere else. It says that we can expand sub configurations ω', assuming $\mathcal{C}$, as long as we include $\mathcal{C}$ in the solution with $((\sqcap_{\mathcal{C}' \in S} \mathcal{C}') \sqcap \mathcal{C})$.

In the given example, in a violating trace in Figure 3.5d, UNSAT and DISC already inferred $(strong, w, w, n_2)$ from reorderings of w on n_2; then, WKN assumes this to filter out the false constraint between withdrawals w on n_{hist} (identifying a false conflict, as discussed in Section 3.2). Note that while stronger solutions are also inferred for the starting configuration, using WKN we derive the weakest solution as well and then adopt the weakest solution for the starting configuration as the final result.

Correctness. Given a well-formed synthesis problem $\mathcal{S} = \langle\langle\sigma_0, A_0\rangle, \mathcal{I}\rangle$, the set of constraints $\mathcal{C}_s$ that is the least upper bound of the set of all found sets of constrains, *i.e.,* $\mathcal{C}_s = \{\mathcal{C} \mid \langle\sigma_0, A_0, \emptyset, \emptyset, \emptyset\rangle \triangleright \mathcal{C}\}$, is the returned result. Given a set of fully explored traces from $\langle\sigma_0, A_0\rangle$, fixed-point of $\triangleright$, this result is a correct and optimal solution to the synthesis problem.

- **Soundness.** For a solution $\mathcal{C}_s$, and any execution τ, if $\tau \models \mathcal{C}_s$, then $\tau \models \mathcal{I}$.

- **Completeness** The algorithm returns the weakest solution. Given a solution $\mathcal{C}_s$, there is no $\mathcal{C}'$ such that $\mathcal{C}' \sqsubset \mathcal{C}_s$, and $\mathcal{C}'$ is also a solution.

The correctness relies on the fact that any inferred constraint is a result of a violated trace and thus prevents some violation. To "remove" a failed invariant from F, we have to either discover or assume a constraint that would prevent it, while analyzing all traces. Even for a problem that is not well-formed, the inference rules infer unsatisfiability, *i.e.,* $\top$, if Analyze returns it when it detects that the violated trace satisfies strong consistency. The optimality argument relies on the fact that while a stronger solution can be found, a weaker solution will always be found by assuming it with WKN and discovering the stronger solution is not needed.

3.4 Conflict-Driven Propagation

We instantiate the cause-effect framework in the *conflict-driven propagation* procedure. The procedure utilizes an incremental strategy of exploration and inference based on the increasing scoped search in concurrency bounds. Intuitively, it infers the consistency constraints, propagates them from smaller to larger scopes, while using them to prune the search space without losing soundness in the process. It uses a bounded-search oracle to explore traces and a cause-effect analysis instantiated with the core consistency lattice (as shown in Figure 3.1). To limit the search space blowup inherent to model-checking and achieve scalable synthesis, the procedure eagerly prunes exploration of unnecessary interleavings based on previously inferred consistency protocols. It achieves sound propagation of consistency solution candidates with careful propagation of solutions when increasing concurrency bound and *merging* found candidate solutions that eliminates results inferred from false conflicts.

The algorithm. The algorithm of the procedure is given in Algorithm 3. Given starting configurations, initialized from synthesis problems, invariants and the maximum bound to explore, it returns minimal solutions (sets of constraints) for all expanded traces. (Starting configurations are obtained from a SyCoord program; see Section 3.6.) It maintains a *worklist* Ω and explores it in the order of increasing *bound* b, which reflects the number of nodes and explored method invocations (*i.e.,* the number and sizes of DPGs). *Items* in Ω are tuples $(\omega, b, \mathcal{C})$ that consist of instrumented configurations ω, bound b and $\mathcal{C}$, under which the trace ω is expanded. The algorithm starts with bound 0 and assigns the set $\{\bot\}$ to C_b and $C_\omega^\mathcal{C}$, which are sets that contain current best candidate solutions: C_b for a bound b and $C_\omega^\mathcal{C}$ for a trace ω explored under $\mathcal{C}$. As there might be multiple solutions, we keep track of all possible candidates.

We first choose an item from the worklist. Line 5 performs pruning: items are pruned if their traces are not feasible under the constraints they were expanded under $\mathcal{C}$. Lines 6-7, for any invariant violation, infer new constraints by using Analyze and add them to candidates

Algorithm 3 Conflict-driven propagation

```
Input: invariants $\mathcal{I}$, bound $B$, starting configurations $P_i$, $0 < i \leq B$
 1: $\Omega \leftarrow \emptyset$, $b \leftarrow 0$, $C_i \leftarrow \{\bot\}$, $C^{\mathcal{C}}_w \leftarrow \{\bot\}$, for any $i, \mathcal{C}, w$
 2: while $b < B$ do                                              ▷ explore bounds up to B
 3:   if $\Omega \neq \emptyset$ then
 4:     $(\omega, b, \mathcal{C}) \leftarrow \mathsf{choose}(\Omega, b)$; $\Omega \leftarrow \Omega \setminus \{(\omega, b, \mathcal{C})\}$
 5:     if $\mathcal{C} \not\models \omega.\mathcal{G}$ then continue                       ▷ prune this trace
 6:     for $I \in \omega.F$ do                                         ▷ discover constraints
 7:       $C^{\mathcal{C}}_w \leftarrow \{\mathsf{analyze}(\omega.\mathcal{G}, I) \sqcap \mathcal{C}' \mid \mathcal{C}' \in C^{\mathcal{C}}_w\}$
 8:     for $w'$ in $\mathsf{pred}(w)$ do                                   ▷ propagate to root
 9:       $C^{\mathcal{C}}_{w^p} \leftarrow \mathsf{Merge}(w^p, \{\mathcal{C}' \mid \mathcal{C}' \in C^{\mathcal{C}}_{w'}, w^p \rightarrow_i w'\})$
10:     $\Omega \leftarrow \Omega \cup (\{(\omega', b, \mathcal{C}) \mid \omega \rightarrow_i \omega'\})$                  ▷ expand
11:   else                                               ▷ all traces within b are explored
12:     $C_b \leftarrow \mathsf{min}(\{\mathcal{C}' \sqcap \mathcal{C}_b \mid \mathcal{C}' \in C^{\mathcal{C}}_\omega, \mathcal{C}_b \in C_b, \omega \in P_b\})$
13:     $b \leftarrow b + 1$                                          ▷ prepare for next batch
14:     $\Omega \leftarrow \{(\omega, b, \mathcal{C}) \mid \omega \in P_b, \mathcal{C} \in C_{b-1}\}$
15: return $C_B$                                      ▷ set of solutions for the last bound
```

for the given trace, $C^{\mathcal{C}}_\omega$ (by taking the least upper bound, embedding the rule DISCOVER). Lines 8-9 accumulate constraints, for each expanded configuration on the path from the root configuration. pred returns all predecessors of ω (w.r.t. $\rightarrow_i$ expansion). This embeds the COLL as it collects and merges constraints for ω infered from derivations of ω. We collect found candidates from child traces using Merge. Merge merges solutions found for all child configurations, accounting for potential false conflicts: the fact that some solutions ω' might have been inferred from configurations ω' that have traces that would not be feasible under other discovered constraints, already discovered for the parent configuration ω^p. (We define `merge` subsequently.) Afterwards, we expand ω and add items for all expanded traces to the worklist, under the current $\mathcal{C}$. In case there are no more items to explore within the bound b, we refine candidates for the current bound C_b, using the candidate solutions for the root of all starting configurations withing this bound b (embedding WKN). min minimizes the set of candidate solutions, by filtering out candidates if they are subsumed by any other in the set (w.r.t. $\sqsubset$). It performs this check by a pairwise comparison between all candidate solutions that are derived by combining existing candidate solutions for the given bound $\mathcal{C}_b$ and all newly found ones. The pairwise comparison preserves correctness in case there are multiple sets of constraints that belong to the solution. Then, we increment the bound and update the worklist for the next batch, taking new configurations from P with the number of invocations for the new bound b, expanding each under every candidate from C_b; this ensures we propagate all candidate constraints for exploring in the new batch. After all configurations up to B have been explored, C_B contains the solutions.

Merging. Merge merges candidates of child configurations of ω^p, accounting for potential false conflicts. It takes a set of candidate solutions and their traces, and returns a set of candidates C. Merge discards any candidate $\{\mathcal{C} \in C\}$, inferred from a trace τ, if there exists another candidate $\{\mathcal{C}' \in C\}$ under which τ is not feasible. In case some other expansion found a solution under which the offending trace is not feasible, this solution is not needed in the final solution, thus preventing the false-conflict anomaly. Merge then returns all unfiltered candidates.

Correctness. The algorithm maintains *sufficient and minimal* candidate solutions for a each bound explored C_b. C_b contains: 1) all necessary solutions, as all of the candidate sets of constraints returned by Analyze are propagated, only discarding unsatisfiable traces; 2) the minimal set of solutions, as C_b is set only after all traces are explored and the set of all collected candidates is minimized.

Soundness holds due to propagating and exploring all inferred candidates and only pruning (in line 5) based on C_b, from lower bound b, which contain only necessary consistency constraints (guaranteed to make pruned traces infeasible).

Completeness holds due to two reasons: 1) if a solution exists, the trace that exhibits the solution will be examined; 2) any solution will not contain any unnecessary constraints. Property 1) holds, as if the trace is pruned, the solution is already included in the candidate solutions for the current bound. Property 2) holds if all false constraints c_p are filtered out.

By contradiction we will provide correctness argument for property 2), by assuming it contains a false constraint. Let us assume the solution contains some false constraints, *i.e.,* some $\mathcal{C} \in C_b$ is not necessary (redundant). Let $c_p \in \mathcal{C}$ be the unnecessary constraint returned as a part of the final solution $\mathcal{C}$, and c_p is inferred from a trace τ within bound b. As it's a false constraint, the real source of inconsistency in τ can be prevented by $\mathcal{C}' \sqsubset \mathcal{C}$, which is inferred within some bound b'. Note that $\mathcal{C}'$ has to be inferred at some point, due to soundness. We consider the three cases for b':

- If $b' < b$, τ cannot be explored, as it is pruned out by the propagated $\mathcal{C}'$ from the bound b' (line 5).

- If $b' = b$, there exists a trace τ' which exhibits the real source of inconsistency and infers $\mathcal{C}'$. Then, at some point when exploring within bound b, Merge merges $\mathcal{C}'$, therefore discarding it at that point.

- If $b' > b$, $\mathcal{C}'$ needs a larger bound to be discovered. We assume $\mathcal{C}'$ forms the solution $C_{b'} = C'_{b'} \uplus \{\mathcal{C}'\}$ is too strong, where $\mathcal{C} \sqsubseteq \mathcal{C}'$. This can only happen if when exploring bound b, exploration propagates two different unrelated candidate solutions $\mathcal{C}$ and $\mathcal{C}_2$.

However, as both candidates are propagated if $C'_{b'} \uplus \{\mathcal{C}_2\}$ is a weaker solution, this is exactly the solution that will filter out $C_{b'}$ in line 12.

This entails that the algorithm infers the set of true conflicts, through merge, while still propagating constraints for pruning when moving to a larger bound.

Relationship with the Framework. In the procedure, the exploration and inference rules are not applied directly, they are embedded into the bounded search strategy. Notably, it avoids non-determinism of the rule WKN through *merging* (Merge), which uses additional information about the specific points in the exploration to make a decision whether to filter out or propagate any constraints from potential false conflicts. The oracle explores interleavings incrementally structured by bounds, but eagerly prunes based on results from smaller bounds to avoid non-deterministic exploration and additional exploration with propagation of constraints.

Concolic Version of the Algorithm. Our concolic strategy instantiates the given general algorithm, similarly to the algorithm in Chapter 2. Notably, in addition to maintaining a set of concrete values for concrete checking of integrity properties, whenever the strategy encounters a conditional, it checks if both branches of the conditional are already taken. If this is not the case, the strategy will use symbolic reasoning to synthesize the values that would make the program take the branch that was not visited. This allows the algorithm to explore all branches of the program and guarantee soundness for any concrete value at run time.

We give a concolic algorithm (Alg. 4) called Concolic *Conflict-Driven Propagation*, that builds on the cause-effect framework to perform bounded inference of consistency constraints. Given a set of starting configurations P, invariants and a maximum bound B, our algorithm returns a minimal solution (*i.e.,* a set of constraints) for each bound $1 \leq b < B$ that covers all traces up to that bound. It maintains a *worklist* Ω_b for each bound b; worklists are explored in order of increasing b. This bound reflects the number of nodes and explored method invocations (*i.e.,* DPGs). *Items* in Ω are tuples (τ, ω, e) that consist of a trace τ, the last instrumented configuration in the trace ω, and a concrete example e. (The worklist is slightly changed in its structure from the previous algorithm.) In this version of the algorithm, we simplify the propagation of constraint and trace pairs, as well as their merging. (This version, however, can use a similar strategy to further optimize the inference.) A point on the used notation: in the algorithm, we use $\models_{cons}$, $\models_e$, $\models_*$ to denote satisfiability of trace with respect to: consistency constraints, concrete evaluation under the environment of e, and symbolic verification condition (using an SMT solver), respectively.

The algorithm executes as follows. The algorithm starts with bound 0 and assigns the set $\{\bot\}$ to C_b, which is the set that contain current best candidate solutions, for a bound b.

Algorithm 4 Concolic Conflict-Driven Propagation

```
Input: configurations $P_i$ for $1 \leq i < B$, examples $E$ ($|E| \geq 1$), invariants $\mathcal{I}$, bound $B$
Output: consistency constraints $\mathcal{C}_b$ for bounds $1 \leq b < B$
1: function InferConsistency($P, E, \mathcal{I}, B$)
2:   $\Omega_i \leftarrow \emptyset$, $S_i \leftarrow \emptyset$, $\mathcal{C}_i \leftarrow \{\bot\}$ for $1 \leq i < B$, $b \leftarrow 0$      ▷ initial worklist and constraints
3:   while $b < B$ do      ▷ explore bounds up to B
4:     if $\Omega_b = \emptyset$ then      ▷ all traces up to b are done
5:       $C_b \leftarrow \mathsf{Merge}(S_B) \sqcap C_{b-1}$
6:       $b \leftarrow b + 1$, $\Omega_b \leftarrow \{([\omega], \omega, e) \mid \omega \in P_b, e \in E\}$
7:     else
8:       $(\tau, \omega, e) \leftarrow \mathsf{choose}(\Omega_b)$; $\Omega_b \leftarrow \Omega_b \setminus \{(\tau, \omega, e)\}$
9:       if $\tau \not\models_{cons} \mathcal{C}_{b-1}$ then continue      ▷ prune this trace
10:      for $I \in \mathcal{I}, \tau \not\models_e I$ do
11:        $\mathcal{C}_I \leftarrow \mathsf{analyze}(\tau, I)$      ▷ discover new constraints
12:        $S_b \leftarrow (\mathcal{C}_I, \tau) \cup \{(C, \tau') \in S_b \mid \tau' \models_{cons} \mathcal{C}_I\}$      ▷ filter out redundant conflicts
13:      if $\omega.A \neq \emptyset$ then      ▷ exists some next steps in exploration
14:        for $l$ in $\{l \mid v_f \bullet\!\!\rightarrow_l v, (v, \gamma) \in \omega.A\}$ do      ▷ all possible static paths
15:          $\Omega_b \leftarrow \Omega_b \cup \{(\tau \mathbin{+\!\!+} [\omega], \omega', e) \mid \omega \rightarrow_i \omega', \omega'.A = \omega.A \setminus \{(v, \gamma)\}\}$
16:          if $l = (\mathrm{Ctrl}(\{c_v\}), d)$ then      ▷ branch
17:            $f \leftarrow \tau \models_e c_v \mathbin{?} \neg c_v : c_v$
18:            if $\forall e \in E.\ \tau \not\models_e f$ then      ▷ concrete example needed
19:              $r_* \leftarrow \mathsf{checkSAT}(\tau \models_* f)$, $\Omega_b \leftarrow \Omega_b \cup \{(\tau, \omega, e) \mid e \in \mathsf{getModels}(\mathsf{r}_*)\}$
20:      else
21:        $r_* \leftarrow \mathsf{checkVALID}(\tau \models_* \forall I \in \mathcal{I}.I \wedge \mathsf{pathCond}(\tau) \implies I)$      ▷ verify path
22:        if $\neg\mathsf{getAnswer}(\mathsf{r}_*)$ then
23:          $E \leftarrow E \cup \{\mathsf{getModels}(\mathsf{r}_*)\}$
24:   return $\{(b, C_b) \mid 1 \leq b < B\}$      ▷ set of solutions for every bound b
```

Whenever the previous bound is fully explored (which holds true for $b = 0$), the algorithm uses Merge to unify all constraint found in the previous bound. Then it increases the bound and instantiates all items to be explored in that bound. Initially, when $b = 0$, the algorithm increases the bound to 1 and instantiates all items that explore executions of invoking a single operation.

For each bound b, we process the worklist Ω_b until it is empty to produce a constraint set C_b that is sufficient for all traces up to b. To process this worklist, we first choose an item from the current worklist Ω_b. Line 9 performs pruning: Items are pruned if their trace is not feasible under the constraints known from the previous bound. Lines 10-12, for any invariant violation, infer new constraints with Analyze and add them to candidates for the given trace, S_b. Similarly to the previous algorithm, we accumulate constraints and the traces that lead to their discoveries, which are then unified with Merge to obtain solutions that account for redundant conflicts. Afterwards, we expand ω and add items for all expanded

```
def interest(r) = { bal *= r; // inside class
  hist += ("I", r) }
def validHist(a: Account) = a.bal == sum(a)
def sum(a: Account) = a.hist.fold(0)({
  case (res, ("D", x)) => res + x
  case (res, ("W", x)) => res − x
  case (res, ("I", r)) => res * r })
```

Figure 3.10: Additional methods.

```
ensuring(
 safety(noOverdraft(a)), // old: safety
 eventual(validHist(a)) ) // new: liveness
withStore( case (a: Account) =>
 allocate(a.bal, nHist)
 allocate(a.hist, n, replicate)
)
```

Figure 3.11: Changed specifications.

traces to the worklist. Intuitively, if the execution depends on a condition (has a control label), we want to make sure we have a concrete example exhibiting both branches. To that end, we check satisfiability of the branch that was not taken, and add a concrete example if needed (by extracting a model from the solver with getModles). In case there are no more steps to execute, the trace is fully explored and we use the solver to check validity of its path condition. If validity is not confirmed, we extract a concrete example that will be used to exhibit the invariant violation and infer additional consistency constraints. After all configurations up to B have been explored, C_b contains the solution for a bound b.

Similarly to conflict-driven propagation, this version of the algorithm is sound, but the soundness is strengthened to hold for any concrete value. The correctness then follows from the fact that the algorithm explores all branches of the program and leverages symbolic reasoning to check the invariants hold for any concrete value.

3.5 SyCoord Extensions

We introduce liveness invariants and topology-aware protocols as extensions to the core approach in SyCoord, in the context of the running example. For the former, we extend the inference process, while for the latter we add a new consistency level in the consistency lattice together with a protocol at run time (extending the rule repository, shown in Figure 3.1).

Liveness Invariants. Let us reconsider the bank account example from Section 3.1. For simplicity, in this section we will consider the fully-replicated version of the bank account. Imagine that in addition to avoiding overdraft, the user would also like to ensure that after all the transactions have been fully executed, the hist and the current balance hold the same value. Additional code is given in Figure 3.10. We can express this property as an executable predicate `validHist`. There are two things to note about this invariant. First, the `validHist` predicate relates fields `bal` and `hist`, which are stored on different nodes; when checking a predicate like this, SyCoord ensures it holds for every combination (in

Instrumentation (extended) $\boxed{\langle \sigma, A, \mathcal{G}, F, P\rangle \rightarrow_i \langle \sigma', A', \mathcal{G}', F', P'\rangle}$

$$\text{I-LIVE}\ \frac{\mathsf{live}(e) \in \mathcal{I} \quad R = \{r \mid \exists s \in \mathsf{global}(\sigma).\langle s, e\rangle \rightarrow r\} \quad \text{good} = \bigwedge_{r \in R} \qquad F' = \text{good} \mathbin{?} F : F \cup \mathsf{live}(e)}{\langle \sigma, \emptyset, \mathcal{G}, F, P \uplus \mathsf{live}(e)\rangle \rightarrow_i \langle \sigma, \emptyset, \mathcal{G}, F', P\rangle}$$

Figure 3.12: Instrumented execution: liveness.

this case, pair) of nodes from the corresponding regions. Second, unlike `noOverdraft`, this property is a *liveness invariant* [Alpern and Schneider, 1985], which is allowed to be violated during execution but must hold eventually, i.e. after every fully executed prefix of operations. To specify a liveness invariant in SyCoord, the programmer uses the `eventual` construct to extend the **`ensuring`** clause in the configuration from Figure 3.4; the new one is given in Figure 3.11.

Due to the new `interest` operation, similarly as in Chapter 2, SyCoord detects this violation and infers that any pair of `deposit` or `withdraw` and `interest` must be issued with strong consistency.

The support for liveness invariants is formalized by extending the instrumented execution with a new rule, shown in Figure 3.12. I-Live operation is similar to I-Safe, for a liveness invariant; the check only applies at the end of the execution (when $A = \emptyset$). Notably, this extension, besides the syntax, directly mounts to just changing the cause-effect analysis, without modifying any other part. (Liveness invariants are checked for last actions in the operation, which is accounted for when constructing synthesis problems.)

Topology-Aware Protocols. We extend SyCoord's syntax with fine-grained store configuration directives to designate a specific node for executing certain operations, and extend the core lattice to include a "central" element (consistency level), shown Figure 3.13b, which allows achieving *strong consistency without coordination.*

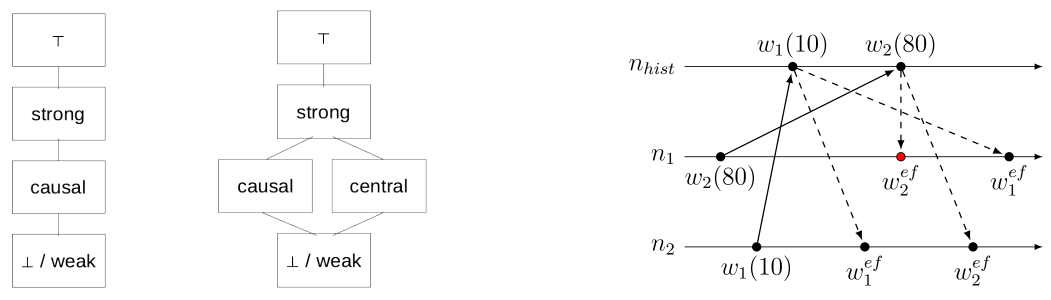
Let's consider a situation where programmers reverse the allocation of balance and the history. Developers change the store configuration as given in Figure 3.11. This makes all operations always dispatch to the same node n_{hist} to modify the balance, after which the action to modify hist gets invoked (lines 3-4 Figure 3.11). Now, all operations, while originating on different nodes, start by modifying `a.bal` on *nHist*. One violating execution is shown in Figure 3.13c. The consistency ordering constraints of conflicting operations remain the same, on the conflicting `hist` variable. However, they can now be enforced with the weaker protocol, without any coordination. SyCoord assigns sequential identifiers to these operations on the "central" node `nHist` and prevents out-of-order delivery to other nodes by simply delivering corresponding effects in their assigned order, thus maintaining total order and strong consistency.

Specifically, we introduce $central(n)$ consistency level into the lattice, for all nodes n. Then, analyze returns $(central(n'), o_1, o_2, n)$ whenever there is a reordering of o_1 and o_2 on some node, and any actions of o_1 and o_2 executed earlier, in program order, on n'. If all conflicts share the same node n' and level $central(n')$, operations can be assigned a unique order index on n' to prevent conflicts on other nodes (akin to the protocol used in Thomson et al. [2012]). This lightweight consistency-enforcement protocol does not require additional coordination between nodes and brings improvements in performance. This extension brings additional performance speedups of up to 4x (see Section 3.7).

Sharding. We extend PEEPCO to supports entity-based sharding [Baker et al., 2011]. Programmers can define static sharding for certain data types similarly to how replication is expressed, providing a set of nodes and a function that control how sharding is implemented at run time. In addition to the sharding data scheme annotation, PEEPCO takes a function f, as a parameter, that identifies the intended shard at run time. The generated implementation invokes f at run time giving it some value of the system's state as the parameter (which lives on the node that performs the sharding, or is communicated to it), the function returns a node identifier from the given set, which represents the specific shard. During consistency inference, PEEPCO translates accesses to sharded data as access to any of the shards, similarly to handling replication, exploring all possible shard accesses regardless of the given function.

3.6 Implementation

In this section we describe the language of SYCOORD, the process of constructing synthesis problems, as well as the backend connection of the synthesized code with the runtime shim layer. We implemented SYCOORD using the three-level lattice with weak, causal, and strong, as well as a custom topology-dependent protocol.



(a) Core (b) Extended (c) Central-ordering

Figure 3.13: Extensions of core SYCOORD.

$$
\begin{array}{llll}
\multicolumn{4}{c}{n : \text{NodeRegion} \quad s : \text{StoreObj} \quad op \in \text{MethodName}} \\
obj & ::= & P(alloc*, opc*, inv*) & \textit{Synthesis object} \\
alloc & ::= & (u_{alloc}, n, mode) & \textit{Data allocation} \\
u_{alloc} & ::= & \text{fieldName}(s) \mid s & \textit{Data unit} \\
mode & ::= & \text{"replicate“} \mid \text{"allocate“} & \textit{Allocation mode} \\
opc & ::= & (op, s, ef*) & \textit{Invocations} \\
ef & ::= & (expr, var*) & \textit{Effects} \\
inv & ::= & \text{safe}(expr) \mid \text{live}(expr) & \textit{Invariants}
\end{array}
$$

Figure 3.14: Abstract syntax of SyCoord's Scala DSL

3.6.1 Components of SyCoord

SyCoord language. SyCoord embeds the language in Scala. Figure 3.14 shows the syntax of the language. Allocation can define a simple assignment of data to nodes (which is the default if omitted, as in line 5 of Figure 3.4), or replication across node regions, on the level of object fields or objects. A method call is tied to a particular store object and also take a set of effects. Effects capture functions used to compute updates (*e.g.,* `(x: Int)=> bal-=x` is the effect computed and propagated in Figure 3.5d). It is defined by, for a given expression, the given set of variables that remain free in the effect. The default effects (if omitted), fully evaluate an expression. To give users flexibility, SyCoord provides the construct `replLocal`($\{x, y, \dots\}$), as annotation of an expression e, which specifies the set of free variables in the effect produced by e. Lastly, SyCoord supports two types of invariants, safety and “liveness”.

DPG construction. The instantiated distributor (Figure 3.1) constructs DPGs that minimize the network communication. As expressions using replicated state can be assigned to any of the replicas, any such assignment belongs to a separate DPG for a method. All replicated assignments of all “load-balanced accesses” of clients (described in Section 3.3) and all possible assignments of subexpressions to nodes, are considered.

Starting configurations. SyCoord derives a set of starting problems, from the surface program. For every test harness, and bound b, which include combinations of 5 scalar and 3 list values, it picks a b-combination from W and assigns values to the store and call parameters. A configuration is created for every combination.

Shim layer. The shim layer is implemented with Akka [*Akka – actor toolkit and runtime, http://akka.io/* 2023], and provides API calls which are invoked on any node to perform a set of actions under weak, causal, and strong consistency. The code generator in SyCoord encodes every action with its method and a history vector. A history vector encodes delivered messages at any given moment (at any node) and is propagated through messages to achieve causal consistency. For strong consistency, we use 2PC transactions. They share a single lock in the voting phase of 2PC.

Worklist strategy and optimizations. SyCoord explores all instantiated problems within a bound in parallel, synchronizing on constraint accumulation before exploring the next bound, instantiating a JVM thread per CPU core. It also incorporates familiar optimizations (from model-checking) in the worklist search:

- *effect-driven pruning* prunes ω' which cannot make steps that could change state relevant for any invariant, by consulting read and write sets of DPG actions
- *symmetry breaking* avoids mirroring traces [Emerson and Sistla, 1996].

Liveness and Topology-Aware Protocols. We add a DSL construct to specify liveness invariants and change the cause-effect analysis, to infer from cause-effect graphs only if they executed all actions. This makes sure that any invariant violations before all operations finish are not considered.

For the topology-aware protocol benchmarks, we extend SyCoord with the "central" level l^c, where $l^c \sqsubset$ strong and weak $\sqsubset l^c$. To infer the "central" consistency, SyCoord checks if the reordered operations are issued through the same designated node. To implement any consistency constraint involving actions o_1 and o_2 with this level, SyCoord emits sequencing operations for all messages that lead to invoking o_1 and o_2, and delivers their message following the assigned sequence.

3.7 Evaluation

We performed an experimental evaluation of SyCoord with the goal of assessing usability and scalability of the proposed inference technique, compared to existing alternatives. This goal materializes into the following questions:

(1) Can specifications in SyCoord capture interesting data stores: 1) from prior work; 2) in new benchmarks?

(2) Is SyCoord scalable to: 1) tackle all benchmarks; 2) support elaborate specifications?

(3) Do features help in inference performance?

(4) Is the synthesized consistency efficient, confirming speedups from prior work in full-replication?

(5) Do the resulting performance trends: 1) translate to partial replication, compared to full-replication, 2) improve with domain-specific protocols?

Group	*Description*	*#I*	*#B*	*Spec*	*#C*	*T-all*	*T-npr*	*T-ncd*	*T-nsb*
Bank example	w / r	1/0	2	21	1	2	2	2	2
	w,d / r	2/0	2	21	2	2	9	4	9
	w,i / r	2/1	2	24	4	2	9	4	3
	w,d,i / r	3/1	2	24	5	2	7	4	-
	w / d	1/0	2	23	1	1	3	3	3
	w,d / d	2/0	2	23	2	2	4	3	3
	w,i / d	2/1	2	27	4	2	3	2	3
	w,d,i / d	3/1	2	27	5	2	3	3	3
	w / m	1/0	2	28	2	2	4	3	2
	w,d / m	2/0	2	28	4	3	-	-	-
	w,i/ r	2/1	2	31	8	2	-	3	3
	w,d,i/ r	3/1	2	31	10	2	-	-	-
Effect list	weak /d	2/1	2	44	0	1	4	7	4
	weak / r	2/1	2	48	0	3	4	8	5
	weak / m	2/1	2	52	0	2	3	-	-
	causal / d	2/1	2	70	6	1	9	4	5
	causal / r	2/1	2	74	6	1	-	3	6
	causal / m	2/1	2	78	18	2	-	3	2
	strong / d	2/1	2	66	6	1	2	2	2
	strong / r	2/1	2	70	6	2	-	3	4
	strong / m	2/1	2	74	18	2	-	3	3
	strong / m,3x3	2/1	2	74	18	5	-	-	-
	strong / m,4op	2/1	2	74	36	3	-	7	8
Usecase	auction / r	2/0	2	76	6	2	-	-	-
	tournament / r	5/0	2	92	20	5	-	-	-
	microblog / r	3/0	2	55	26	2	-	8	-
	tpc-c / r	3/1	2	84	14	3	-	-	-
	move game / r	2/1	3	137	12	4	-	6	-

Table 3.1: Benchmarks and SyCoord results. For each benchmark, we report: the number of invariants #*I* (total/live); sufficient search bound b #*B*; cumulative size of invariant *Spec*ification (in AST nodes); the number of discovered pair-wise conistency relations #*C*; the number of minutes SyCoord runs until exploring bound $b+1$ (*T-all*), without pruning (*T-npr*), without causality-driven exploration (*T-ncd*), and symmetry breaking (*T-nsb*). Times are given in minutes, where "-" denotes timeout after 10 minutes.

3.7.1 Inference Benchmarks

Our benchmark suite consists of inference problems from different domains that are representative of the expressiveness in terms of operations, invariants, and data store configurations. For comparison with prior work, our suite models benchmarks that had been used in the evaluation of their tools [I. Zhang et al., 2016; Sivaramakrishnan, Kaki, and Jagannathan, 2015; Milano and Andrew C. Myers, 2018; Houshmand and Lesani, 2019]. From each of these papers, we picked the most complex challenge (judging by the reported size or inference time), and expressed them in SyCoord. The goal of our benchmark suite is to capture how inference is affected by different types of invariants (safety and liveness), different data store configurations (node structure and replication), and operations (their number and code complexity). We evaluated consistency inference that uses concrete values to check whether invariants are satisfied. Test harnesses included combinations of 5 scalar and 3 list values.

Benchmarks. We divide benchmarks based on the demonstration goal: scalability, complexity of store configurations, and realistic domains.

- The first benchmark set represents the running example and examines how different invariant forms, paired with combinations of commutative and non-commutative operations, node assignments, affect inference, as described subsequently.

- The next set explicitly captures effects and their visibility relations in code, through lists of effects. Whenever an operation effect is received on a node, it adds the effect to the list but also stores all of its causally dependent effects. The invariants are specified such that the given consistency level needs to be assigned to the three operations in the system. The "3x3" designates a benchmark that has 3 node regions with 3 replicas each, and "4op" designates the same benchmark with 2 operations for each operation type.

- The third set captures benchmarks from previous work, including microblog (modelled by Twitter), auction, and a tournament game system, described in [Sivaramakrishnan, Kaki, and Jagannathan, 2015], [Milano and Andrew C. Myers, 2018], [Houshmand and Lesani, 2019]. We modeled the TPC-C benchmark, which includes operations prohibitive to automated reasoning, such as statistics computation (we model only the part related to customers and orders). The move game benchmark represents an interesting scenario of defining consistency. It models a distributed game, where players can make certain moves in the game. While there are certain moves players are allow to make, depending on the sequence of previous moves, coded in a logic that examines the sequence for every move, the same logic optimizes consistency of move issuing. This benchmark had 4 operations and moves, where for each move, there is one valid sequence of one, two, and three moves.

Per *evaluation goal (1)*, our suite includes prior work benchmarks, as well as benchmarks requiring expressiveness that goes beyond prior work.

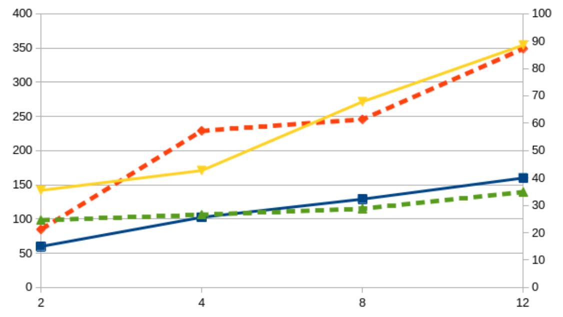
Table 3.1 lists the benchmarks together with benchmark metrics. We group benchmarks according to their usecase. Each benchmark reports the group the benchmark is in, as well as the description of the operations, data scheme, and invariant types. Benchmarks are listed in the form of *bench*/R where *bench* names or describes the benchmark and R denotes data scheme (with r, d, m denoting full replication, distribution, and partial replication, respectively). In the first group benchmarks describe operations of the bank example from the motivation section (with w,d, i, designating withdraw, deposit, and interest, respectively). For each benchmark, we report: the number of invariants $\#I$ (total/live); sufficient search bounds $\#B$; cumulative size of invariant *Spec*ification (in AST nodes); the number of discovered pair-wise consistency relations $\#C$; SyCoord running times to explore a bound b, sufficient to find all constraints that generalize into the unbounded case (*T-all*), as well as ablations. The ablations include running times: without pruning (*T-npr*), causality-driven exploration (*T-ncd*), and symmetry breaking (*T-nsb*). We report full sizes of all invariants, even though in our approach those definitions are reusable between all benchmarks in the same problem domain.

Inference performance. Our setup had a machine with 30 2.2Ghz cores. SyCoord solves all benchmarks, *i.e.,* finds all the needed constraints between operations, in less than 5 minutes, *confirming evaluation goal (2).* It scales well with the number operations and harnesses, while worse with the replication factor, as it increases the number of DPGs and the search space exponentially.

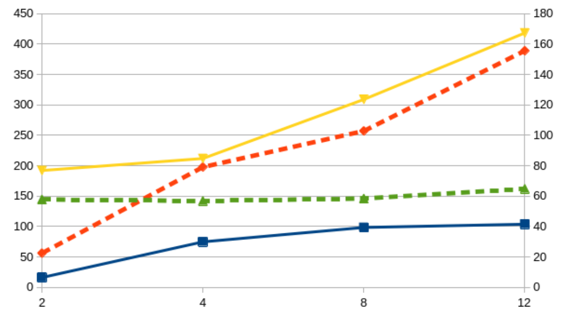
Pruning, based on consistency, brought significant performance benefits in most benchmarks, as they exhibit violations. The slowest benchmarks were those with fewer violations, where main improvements were only due to symmetry breaking and effect-guided optimizations. This confirms the *evaluation goal (3).* The biggest bottleneck is the expression evaluator was called more than 60M times in some benchmarks, suggesting benefits in optimizing the performance of the evaluator.

3.7.2 Data Store Run-Time Performance

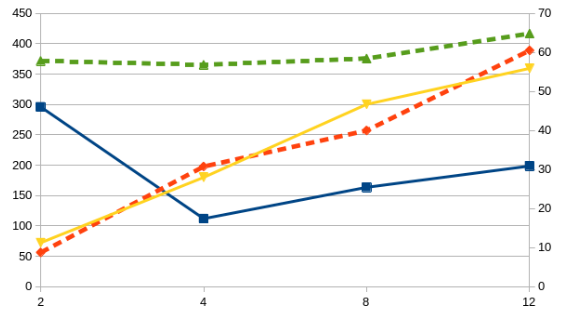
For performance evaluation, we examined relative performance of different consistency protocols discovered, as well as deployment-ready off-the-shelf data stores. One goal is to confirm that the code generation and shim layer of SyCoord behave properly and agree on performance gains already reported in the literature. While prior work established positive effects of different coordination levels in distributed datastores [Sivaramakrishnan, Kaki, and Jagannathan, 2015; I. Zhang et al., 2016], we focus on a more precise evaluation of synthesized consistency levels, in a controlled microbenchmark setting. Additionally, to confirm the practical value of consistency optimization within partial replication, we compare SyCoord to an off-the-shelf datastore, Apache Ignite, capable of both distributed and replicated allocation [*Apache Ignite allocation modes* 2020]. We also compare SyCoord to Apache Ignite under partial replication. (We achieve the same data allocation in partial replication through tweaking Apache Ignite cache affinity values.) We hypothesize that,



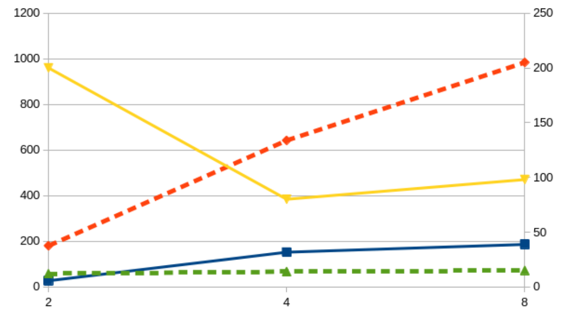
(a) Full replication, relative



(b) Partial replication, relative



(c) Partial, off-the-shelf datastore



(d) Optimization, relative

Figure 3.15: SyCoord data store performance. Y-axis: throughput (op/s), left; latency (ms), right. X-axis: number of nodes. "Relative" shows performance comparison of two data stores emitted by SyCoord, while "off-the-shelf" shows comparison of the emitted data store with an off-the-shelf data store. The dashed lines represent SyCoord (red for throughput, green for latency) and solid lines represent results employing strong consistency in the compared approach (blue for throughput, yellow for latency).

due to the fact that SyCoord leverages standard coordination protocols, scaling trends extrapolate to improvements in larger-scale mixed-workload benchmarks confirmed in prior work. We also examine performance improvements of extending the lattice and leveraging configuration-specific protocols (on the benchmark described in Section 3.5).

Setup. We deployed SyCoord datastores in a distributed cluster, which is composed of 12 virtual nodes, within the same datacenter. We instantiate one VM, hosting both the shim layer and generated code, as well as one Ignite node, on each cluster node. For valid comparison we turned off fault tolerance in Ignite. Clients are instantiated on different VMs, but share nodes with the datastore, and invoke provided operations and block through separate threads. We deployed the cluster on c3.4xlarge OpenStack instances, and measured throughput and latency with the Yardstick distributed benchmark library [*Apache Ignite allocation modes* n.d.]. The average intra-cluster node latency was <10ms. We ran our experiments with 10s warmup and 60s experiment time. We use a smaller, datacenter-local benchmark environment to get more precise performance trends; albeit a smaller and controlled environment, the performance benefits should translate to larger settings and improve with slower links between machines, which occurs across data centers, as confirmed in prior work.

While prior work examined performance gains with optimized consistency of data stores in geo-distributed settings [Sivaramakrishnan, Kaki, and Jagannathan, 2015; I. Zhang et al., 2016], we focus our benchmarks on a controlled microbenchmark setting. We hypothesize, due to the fact that SyCoord leverages standard coordination protocols used in prior work, scaling trends extrapolate to improvements in larger-scale mixed-workload benchmarks confirmed in prior work.

Benchmarks. Our client workload was modeled by the YCSB benchmark (as in prior work), but limited to key size matching the cluster size for more precise comparison with Apache Ignite on conflicts (which uses topology-aware affinity mapping to determine allocation). We evaluate the running bank account example, where clients uniformly choose accounts and operations (modeled by the uniform setup of the YCSB benchmark [B. F. Cooper et al., 2010; Sivaramakrishnan, Kaki, and Jagannathan, 2015]). The datastore included only one bank account per node, thus significantly raising the probability of conflicting operations. The benchmark uniformly chooses from the set of all keys.

Results

Figure 3.15 shows throughput (y-axis, left) and latency of operations (y-axis, right) as the number of data store nodes increase (x-axis). In all figures, dashed lines represent SyCoord (red for throughput, green for latency) and solid lines represent results employing strong consistency in the compared approach (blue for throughput, yellow for latency). We implemented the strong consistency baseline using a two-phase commit protocol [Gray and Lamport, 2006]. For the off-the-shelf data store, we implemented operations as operations under transactional behavior in Apache Ignite. The results across all benchmarks confirm trends of scalability: throughput with consistency optimized scales significantly better than in the strong consistency case, while latencies rise more rapidly, with increasing number of nodes.

Full and partial replication. The full-replicated benchmark corresponds to the trend confirmed in prior work on replicated data store consistency Figure 3.15a, confirming *evaluation goal (4)*. Here, as values are replicated across the whole store, for deposit operations, conflicts exhibit expensive two-phase commits tries and rollbacks, while optimized consistency avoids conflicts with no-coordination protocols.

When moving to partial replication (hist on a separate node and balance on the rest), the trends for SyCoord improve. This is because conflicted operations have to contend for the single hist variable, incurring more conflicts Figure 3.15b. The cummulative speedup raises to more than 4x for 12 nodes.

Off-the-shelf data store and sequencing. Results in Figure 3.15c confirm the SyCoord optimization speedup applies compared to achieving same invariant preservation in the off-the-shelf data store. SyCoord outperforms Ignite on throughput significantly, while sharing similar latencies as the number of nodes grows. Ignite, when used with optimistic transaction execution, does not lock all needed values a priori, thus it exhibits lower latency when no conflicts are present. This is because it avoids double round-trip messaging in non-conflicting accesses. Locking would perform significantly worse, as even with smaller and highly saturated data store, map conflict rates remain relatively low (and thus the number of unnecessary round-trips).

With the “central” consistency extension, SyCoord incurred additional performance improvement, shown in Figure 3.15d, as it further minimizes coordination (by using sequencing through accesses on the single node) while achieving strong consistency. The cumulative speedup is around 5x for 12 nodes. This confirms the *evaluation goal (5)*.

Our evaluation does not account for performance of individual actions, *i.e.,* operation parts after partitioning. While we measure throughput and latencies for whole operations, the data update latency might be smaller with SyCoord for individual actions. Namely, in the partial replication benchmark, we found that hist additions become visible to the store at less than 10% of time required to perform full withdraw or deposit (as no coordination is required).

3.7.3 Game Demo Use Case

To explore practical applications of SyCoord we implemented a multiplayer use case based on the game Slither.io [*Slither.io game* 2023]. In this game, players connect to a game room and control one snake, that can move forward, potentially changing the angle in the process. When a player’s snake collides with the body of any other snake, the player that made the collision loses its snake and its snake is removed from the game. A game state with three snakes in shown in Figure 3.16.

To implement the usecase, we:

1. implemented the basic logic for player-controlled snake movement in Javascript, using Phaser [*Phaser - A fast, fun and free open source HTML5 game framework* 2023]

2. added a new "super move" operation that speeds up the snake for a limited amount of time

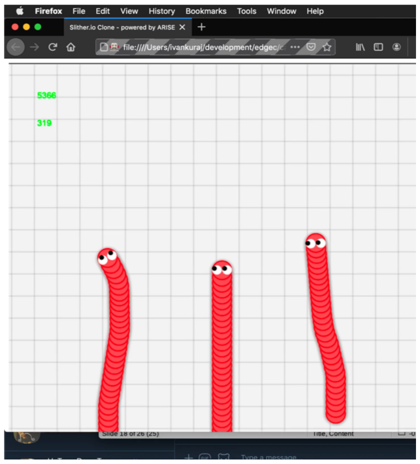


Figure 3.16: Game state

3. used Scala.JS and Akka.JS in our backend to generate JavaScript networking layer on top of WebSockets [*Scala.js* 2023]

We ran our implementation as a two-client simulation, with three snakes, and manually injected delays into the network. The naive implementation renders the state as soon as inputs from other players (clients) are received. The states on the two clients (captured at different times), after running the simulation for a limited duration from the given game state, are shown in Figure 3.17. Due to naively delivering and executing player moves, and the delay, the execution of the super moves early for different snakes, on different clients, can lead to diverging outputs on the two clients.

To utilize SyCoord, we defined a liveness invariant that checks if all clients have the same player-controlled snake objects in the game. Whenever two clients differ in the snakes that are still alive, a violation occurs. We manually extracted concrete state and operation parameter

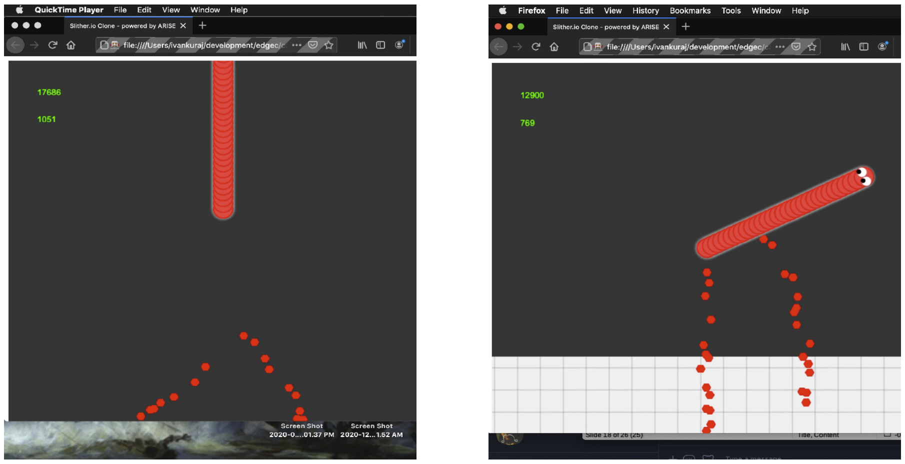


Figure 3.17: Naive simulation of the game usecase

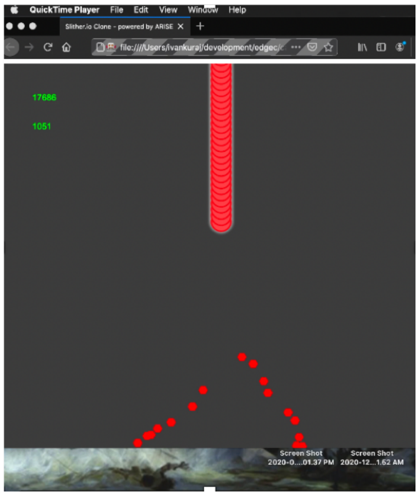


Figure 3.18: Consistency optimized run of the game

values from several scenarios before and after snake collision and fed them to SyCoord. Instead of using the standard oracle, we used the external simulator that determined if the integrity gets violated after a short simulation time. In a few chosen scenarios, both snakes execute the "super move" when close to each other. The goal of the employed consistency analysis was to determine which operations can be safely delivered locally without knowing the information about the invoked operations on the remote client. In order to do this, our simulation executes a normal move in the current direction of the snake, in case the information from the other client is missing. When the information about the move is received, at that point, the simulation invokes the move (potentially a super move).

SyCoord finds that while the normal move can be delivered with delay safely (the snake that is missing information will proceed moving forward slowly), super moves need to be delivered with strong consistency with respect to each other (thus ensuring total order on their delivery). When the execution proceeds with this consistency requirement in place, the simulation starting from the same state shown before (Figure 3.16) finishes without violating the given invariant, as shown in Figure 3.18. Specifically, both clients have the same player-controlled snake rendered as alive. (The position of the snakes differs, however, due to the introduced delays which affect normal moves and thus snake trajectories.)

Our example, while limited, shows the potential for practical application of consistency inference in finding moves in multiplayer games that require careful delivery in order to avoid states that lead to discrepancies and potentially hurting the user experience. Our example was motivated by the developer experience described in a game development report for the game “Halo: Reach” [*I Shot You First: Networking the Gameplay of HALO: REACH* 2023]. Similarly to our super moves, some moves in the Halo game required special consideration on how developers had to deliver and implement game actions. This scenario is akin to identifying strong consistency requirement in our game use case, thus suggesting SyCoord can prove useful in such situations to programmers of multiplayer games, for identifying stronger ordering requirements between game actions and implementing the needed protocols, based on game-specific properties.

3.8 Discussion

SyCoord explores a point in the design space of consistency inference, by providing automation and separation of concerns, support for practical partial replication, and arbitrarily expressive operations and invariants, at the cost of bounded-search and provided test harnesses to drive the inference process. It relies on the idea of the "small-scope hypothesis", and identifies consistency constraints on traces within a small bound and generalizes them to arbitrary executions, leveraging the insights that practical consistency violations can be exhibited with interactions between a small set of operations [Bailis, Fekete, et al., 2014; Jackson, Schechter, and Shlyahter, 2000]. SyCoord finds every consistency requirement, if at least one trace exhibits its violation. Our evaluation found that SyCoord infers all consistency requirements and generalizes to unbounded executions on all our benchmarks, by exploring small scopes, with a small set of concrete values that exercise different paths through the operations.

SyCoord's use in practical development. Due to supporting concrete evaluation, unlike Peepco, SyCoord has additional potential for seamless integration into the modern development process, allowing programmers to re-use sample values from unit tests to prototype and optimize consistency before committing to a distributed solution. The choice of exploration based on concrete values allows exploring unrestricted forms of invariants, using black-box code (*e.g.,* through a library), and specifying specific test cases to explore (which often come with specifications, such as for TPC-C [Bailis, Fekete, et al., 2014; TPC, 2010b]). Moreover, correctness guarantees can be strengthened by leveraging enumerators for efficient exhaustive generation of complex inputs up to a bound [Kuraj, Kuncak, and Jackson, 2015; Kuraj and Kuncak, 2014]. Symbolic specifications can be error-prone, and often cannot capture the needed set of states for practical consistency inference (due to edge cases and inductiveness [Kaki, Earanky, et al., 2018; Houshmand and Lesani, 2019]).

While Peepco leverages the explicit-state model-checking approach, it solves fundamental challenges of inference in partial replication and is oblivious to the engine of exploration; symbolic-value reasoning can be swapped, or integrated in, at the cost of robustness, and operation and invariant expressiveness. For use cases where full correctness guarantees are required, users can rely on existing verification approaches in order to verify results obtained from SyCoord (e.g. [Houshmand and Lesani, 2019; Gotsman et al., 2016]). An external verifier can either be incorporated as a checking oracle (in the model-checker part), or programmers can create test inputs from counter-examples discovered by the verifier and feed them to SyCoord. This separation allows extension of our approach and implementations that fall outside the scope of what can be verified with the current state-of-the-art techniques.

Moreover, in addition to not forcing programmers to learn new language for specifying consistency, SyCoord's input conciseness is comparable to other high-level verification programming systems [Houshmand and Lesani, 2019; Kneuss et al., 2013; Kaki, Earanky, et al., 2018]. SyCoord generates larger code bases, where the generated code size is often dominated by the message passing and protocol code.Note that instead of generating message-passing code, SyCoord could be parametrized to use an off-the-shelf datastore, similar to Sivaramakrishnan, Kaki, and Jagannathan [2015], in cases where choosing consistency in finer granularity might not be needed.

Expressiveness and extensibility. Through the consistency inference lattice employed in the cause-effect analysis, our approach supports arbitrary consistency constraints that are determined by visibility, same transaction (or session order), and happens-before relations, standard notions in consistency analysis. As these relations are directly encoded in the captured cause-effect graphs, in addition to capturing granularity of their occurrence on different nodes, they can be seamlessly used to define standard consistency levels including snapshot isolation and repeatable reads [Terry, Demers, et al., 1994; Thomson et al., 2012]. In order to support synthesis with constraints containing those levels, developers would need to associate proper consistency protocols to be used in the code generator of SyCoord.

By supporting specifying liveness, SyCoord provides the capability that is out of scope of existing approaches to automatic consistency enforcement [Sivaramakrishnan, Kaki, and Jagannathan, 2015; Houshmand and Lesani, 2019]. (Note that our definition of liveness differs the definition used in the literature, and cannot support full range of properties, such as fairness [Pnueli, 1981].) Moreover, this support is easily provided by extending rules used in the inference: during the execution exploration phase, it evaluates a liveness invariant only in the final state of each explored execution. This suggests the possibility of further extensions to allow programming by example strategies, where programmers specify specific traces on which the given integrity properties should be checked. We leave such extensions of the framework for future work. Moreover, SyCoord supports reasoning about the data store configuration and allows supporting realistic configurations, such as banks where banks allow special operations on the account through special, e.g. secured nodes.

Chapter 4

EdgeC: Aspect-Oriented Language for Reactive Distributed Applications at the Edge

Peepco and SyCoord support specifications for data allocation and reactivity aspects with limited flexibility. In this chapter, we present EdgeC, an approach that addresses this limitation and allows programmers to control: 1) fine-grained data access in more elaborate allocation schemes, 2) internal mechanisms for triggering behaviors in the system, and 3) communication patterns exhibited by the resulting implementations. Combining these aspects brings new challenges that require reasoning about the location and timing of computation, as well as the events in the final distributed implementations. These aspects bring forth the necessity for optimizing consistency, for achieving efficient distributed programs. EdgeC, however, is limited to producing efficient implementations for only a limited class of programs, while falling back to either of the two extremes in consistency handling, as described in Chapter 1. When efficiency is needed, it sacrifices integrity for performance, at run time. By combining our approach to consistency optimization, presented in Chapter 2 and Chapter 3, with the support for additional specifications of EdgeC, presented in this chapter, the aim of our end-to-end programming approach is to extend the overall expressiveness for new classes of distributed applications.

Challenges of reactive applications. Distributed, reactive, and interactive applications such as online services, multi-user games or industrial sensor processing systems constitute an important category of software systems [I. Zhang et al., 2016; Viennot et al., 2015]. An application generally consists of a set of compute nodes which must communicate with each other to update their state in response to external stimuli. Some of these applications must also combine incomplete inputs from multiple sources, promptly respond to asynchronous user interactions, or process incoming protocol messages [Reynders, Devriese, and Piessens, 2014; Bykov et al., 2011]. In addition to the complexity of the development of traditional distributed applications, the advent of Internet of Things and "edge computing", brings novel challenges that, unlike traditional cloud computing, require non-uniform architectures with more elaborate distributed requirements, including data and computation allocation, as well as reactivity [Alur, E. D. Berger, et al., 2016].

Edge computing poses new challenges for reactive distributed applications, which go beyond black-box cloud-centric systems [Alur, E. Berger, et al., 2016; I. Zhang et al., 2016]. With traditional programming methodologies, design decisions about distributed aspects, such as data distribution and reactivity, need to be *woven together with the application logic*. As a result, implementations become complex even when the underlying logic is conceptually simple, and the ability to *explore different design choices is limited* because small changes to how data is distributed or how communication is orchestrated require cutting through multiple layers of code. The design choices might involve different network models, data allocation schemes, consistency and reactivity requirements. In such cases, the program developers need to write full implementations, significantly different for every combination of design decisions, making it *difficult to search for an optimal overall design, in spite of core logic being fixed*. The programming complexity further exacerbates when the system needs to operate in *heterogeneous environments*, which involve nodes of various computing capabilities and non-uniform networks.

EdgeC approach. This chapter presents EdgeC[1], a framework that aims to simplify prototyping event-driven reactive distributed programs, amenable for the edge, including non-uniform deployments. EdgeC relies on the insight that distributed systems can remain *sufficiently specified, by separating the application logic and distributed aspects*. Developers write a set of operations, which specify the application logic, together with orthogonal specifications for distributed requirements, *data and computation allocation, reactivity, consistency, and networking concerns*. The system generates low-level implementation of a distributed system which allows invoking the given operations concurrently and reactively, arbitrarily distributed across the system, respecting the sequential semantics of operations according to chosen consistency (akin to distributed transactions). Specifications of behaviors, data/computation allocation, consistency, and network model, can be tweaked and changed separately, while EdgeC handles low-level code automatically. EdgeC thus allows easier *design space exploration of distributed programs*, offloading complex cross-cutting reasoning from developers.

Limitations. This chapter focuses on highlighting the main ideas behind the language

[1]EdgeC is an anagram of initial letters from "**E**vent-driven **d**istributed **g**lobal-view **c**onsistent **e**xecutions".

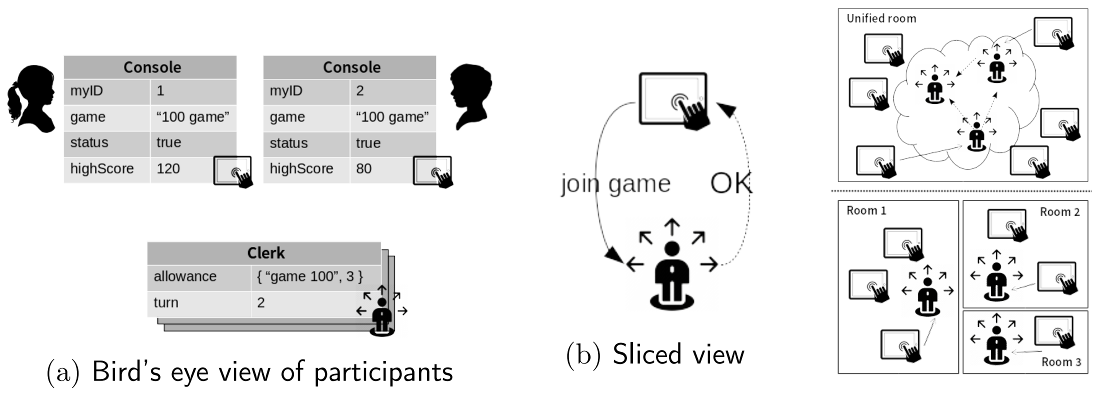
and compiler design, and the adopted programming model in general. Its goal is to provide additional specifications and, in combination with other approaches, achieve a higher level of expressiveness through orthogonal specifications for behaviors and allocation, reactivity, and integrity or data consistency. In turn, EdgeC assumes distributed nodes operate in reliable and trusted environments and does not address aspects related to security and failure-tolerance. We discuss these aspects later as potential avenues for future work. The goal of this chapter is to propose and examine the potential of the programming model in terms of expressiveness of behaviors and performance, without engineering a general and fully optimized full-fledged system.

This chapter makes the following contributions:

- it presents a *novel framework design, which synergizes program analysis and synthesis*, with a run-time, for developing optimized implementations that satisfy specifications across cross-cutting and inter-dependent concerns, applicable in new domains;
- it introduces a new form of specifications for allocation and reactivity aspects of distributed programs;
- it defines requirements and techniques for compilation, and a reference implementation that extends a functional fragment of Scala and generates Scala actor code;
- it presents a prototype implementation of EdgeC and evaluation of its performance.

4.1 Overview

We demonstrate the main ideas behind EdgeC through a tutorial implementing the "100 game" (analyzed as a motivating example in [I. Zhang et al., 2016]). In this simple game, players alternately add a chosen number to the current sum, and the first one to reach 100 wins. Figure 4.1 shows a high-level view of the architecture of the application. The goal is to prototype distributed version(s) of the game, with different requirements in mind. EdgeC allows the developer to start writing operations as sequential code (without distribution in mind, viewing it as "sequential slice", as shown in Figure 4.1b), and orthogonally define distributed aspects that define distributed computation shape, be it standard client-server or more specialized architecture (e.g. ones shown in Figure 4.1c). Through a series of requirements on functionality and distributed concerns, we show how developers use EdgeC to incrementally explore new behaviors and/or distributed specifications, at each point producing a *fully functional reactive application.* EdgeC *synthesizes Scala code and provides*



(a) Bird's eye view of participants (b) Sliced view (c) Architectures

Figure 4.1: Application overview, its logical "sliced view", and the architectures it can capture

a JVM-based runtime, which is then deployed on an interconnected set of machines (see Section 4.2).

Distributed interactive behaviors in EdgeC

The main functionality is making a game move, per player turn, which adds a given number to the overall score. Figure 4.2 shows EdgeC code to implement this behavior. Syntax of the language is based on Scala. Scenarios encapsulate a particular set of distributed behaviors (effectively a sub-system) and are parametrized by node instances that represent physical machines in the system. With the `node` declaration (line 1), developers declare two types of nodes, `Console`, which encapsulates console id (passed as a parameter when a node is created), and `Server`. The `Game` scenario defines the behavior over a given server and a set of console nodes (line 3).

In EdgeC, the core functionality is simply given with sequential code, and distributed aspects are provided on top (through annotations). Developers define data and the `play` function, which takes a parameter and checks if it's the current player's turn: if yes, the score is increased and other variables are updated. The code for `play` is oblivious to any distributed aspects of the computation; it can be reasoned about and tested as any other sequential function (e.g. simple unit testing).

Distributing behaviors. Specifications of distributed aspects characterize how sequential behaviors are mapped onto the distributed system at hand. Firstly, given node instances, here `server` and `consoles` (line 3), each variable is annotated to specify allocation to a node. The annotation *`@loc(n)`* designates the declaration at hand, data or function, is allocated at instance n. With lines 4-7, developers allocate game variables on the server, and `myId` for every console in `consoles` (each console owns a copy), while `notify` can

```
node Server; node Console(cId: Int)

scenario Game(server: Server, consoles: Set[Console]) {
 @loc(server) { var turn: Int = 0; var score: Int = 0;
  var last: Int = -1 }
 @forAll(c in consoles) { @loc(c) var myId: Int = c.cId;
  @loc(c) def notify(r: String) = println(r) }

 @resolve[Console]
 def play(num: Int) = {
  if (turn == myId) {
   score += num; last = myId
   turn = (turn+1)%consoles.size
   return true
  } else return false }

 trigger PlayGame[Console](num: Int)

 triggering(anyOf(consoles) { (c: Console) =>
  when (PlayGame(num)) { bind(c)(play(num)) }
  when (onChange(turn)) { notify("Score is: " + score) }
  when (onTrue(score>100)) { notify("Win: " + last) }
 })
}
```

Figure 4.2: Distributed reactive game

be executed only on a `Console` node. With the construct `forall`, developers allocate one `myId` for each client node and initialize it to `id` of the console (line 6).

In EdgeC, all function invocations (behaviors) need to resolve to specific node instances which determine the actual distributed execution at run time. EdgeC does this at the place of invocation, at compile time, as explained later. While game data is unambiguously bound to the single `server` (through *`@loc`*), developers annotate `play` with `resolve[Console]` which means behavior needs to bind a console instance to be invoked; this is due to accessing `myId`. (EdgeC allows means of binding different consoles to different variables, not shown here.) This in turn means: 1) access to `myId` resolves to an access on the bound console; 2) `play` accesses data spread across different nodes, i.e. server and console. EdgeC automatically splits behaviors into chunks (based on data allocation), analyzes chunk inter-dependencies, and performs the necessary communication needed for distributed execution. Thus, every behavior might be executed chunk-by-chunk, across multiple different nodes; here, console sends request to the server (sending data `num` and `myId`), the server evaluates the whole body, and a response is returned. (EdgeC splits behaviors according to a given network model; by default, it minimizes communication rounds in a uniform model.)

Invoking behaviors. Having defined the behaviors and data allocation, the next step is to define when behaviors should execute, i.e. reactivity. With `trigger`, developers define external events (e.g. interaction with the system). `PlayGame` is a trigger that can occur at `Console` instances; it represents an event that carries `num`. EdgeC generates API stubs for injecting events with parameters (so this can be done e.g. from a UI). The construct `triggering` invokes behaviors in response to triggers. The core of this construct is a `when e {b}` statement, which indicates that `b` should be executed whenever trigger `e` occurred. In this example, since the event `PlayGame` and behavior `play` need to bind to some console, developers first use `anyOf(consoles)` to quantify over all nodes `c` in the set `consoles`, and use it to bind the invocation of `play`. Let's consider line 20: whenever `PlayGame` event is fired, on any quantified console `c` (bound implicitly), the system starts the `play` behavior in the system. The caller, who injected `PlayGame`, then waits for the response.

Resulting distributed behaviors. EdgeC compiles this program, accounting the aspect of allocation, into a message-passing implementation which offers APIs to instantiate nodes and scenarios, as well as to inject events. By default, EdgeC uses the Akka framework [*Akka – actor toolkit and runtime, http://akka.io/* 2023]. Given Akka actors are obtained, and developers construct nodes from actors and start the application with:

```
def init(sa: ActorRef, cas: List[ActorRef]) {
 val cs = cas.zipWithIndex.map({
  case (a,i) =>
   Console(a,i)
 }).toSet
 Game(Server(sa), cs).start() }
```

This creates a console node for each actor in the given list, with the given `cId` argument that represents the initialized field. Interaction is done by injecting events as actor messages. On the console node, developers can inject `PlayGame` with:

```
val console: ActorRef = ...; console ! PlayGame(5)
```

The system performs a client-server communication pattern, by communicating `num` to the server, executing the body of `play` on the node specified by `sa`, retrieves the response, and passes it to the caller (asynchronously through a message). In EdgeC developers can implement the behavior in different patterns, by either changing the given behavior code or orthogonal specifications.

Reactivity

Unlike traditional approaches, handling reactivity is done through a separate specification, without cross-cutting code. When the game is over, i.e. score reaches 100, developers notify players by simply binding a call to `notify` to a trigger `onTrue`; with `onTrue(c)`, whenever the condition `c` (on arbitrary allocated data) goes from false to true, the trigger fires (Figure 4.2, line 22). EdgeC allows implicit resolution – the quantified console `c` is implicitly bound to the call. This achieves the expected behavior of invoking `notify` on all consoles when the game is over. EdgeC makes sure that, at any point in time, if the trigger condition becomes true, the corresponding behavior is invoked. It analyzes the code and checks for all possible places the trigger can fire: in cases it can be statically determined, a call to the behavior is inserted or omitted, otherwise a run time check is emitted. (EdgeC fires at most one declared bound trigger per operation invocation; defined in Section 4.3.) In this case, EdgeC automatically determines a check needs to be inserted after invoking `play`.

However, imagine developers introduced another operation besides `play` (and bound it to nodes), to reset the game:

```
def reset() { score = 0; turn = 0; last = -1 }
```

Then, EdgeC does not insert any checks after `reset`, as analysis confirms `score` cannot become 100 after executing `reset`.

Now, imagine developers want also to notify players of the new score, whenever it changes. In this case, the developers bind an appropriate notify call to the `onChange(e)` trigger, which triggers every time the expression `e` changes (used in Figure 4.2, line 21). EdgeC automatically injects appropriate calls, but also optimizes communication: namely, after the score reaches 100 both triggers are fired but, the server sends only one message to consoles. (A naive event handling would fire two behaviors through two separate messages from the server to consoles, i.e. 2n messages for n consoles.) EdgeC statically analyzes control flow to minimize communication; here, it consolidates messages and emits an additional check (and receive handler on consoles), to cover this case.

Handling consistency

When developing applications with multiple distributed behaviors occurring at different places in the system, developers must often coordinate messages to avoid inconsistent results. Unlike uniform transactional systems, EdgeC handles consistency at different nodes by employing different low-level coordination protocols. It analyzes program semantics, places of invocation, as well as network model to try to optimize and avoid unnecessary coordination.

Our prototype supports strong and weak consistency modes (given as a flag to our prototype compiler; not shown).

Under weak consistency, EdgeC simply executes all behaviors when they get invoked as a sequence of message passing communication across the system (i.e. more specifically a graph, based on data dependencies across nodes), without coordination. For strong consistency, EdgeC analyzes any two behaviors in the system for potential conflicts that could violate serializability. (EdgeC performs cycle graph conflict analysis based on prior work [Shasha, Llirbat, et al., 1995].) In the running example, since the game data is allocated on the `server` node, strong consistency is preserved if all executions (data modifications) on `server` and notification behaviors resulting from those executions on the consoles are observed in the same order on all nodes. Under the Akka implementation, which uses TCP/IP, this is automatically satisfied (as message ordering between two nodes is preserved). However, EdgeC supports backends with weaker assumptions (e.g. UDP or MQTT under low QoS) and implements such ordering automatically.

EdgeC performs pattern matching on the conflicting behaviors and underlying network and applies optimizations, such as ordering, if possible. In case such optimizations are not possible, EdgeC emits a two-phase consistency protocol to ensure strong consistency. In the given example, imagine developers split data between two servers, allocate `turn` on a different server node `server2`, and redefine `reset`:

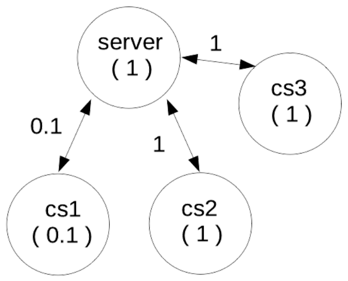
```
scenario Game(server: Server, server2: Server, ...) {
 @loc(server2) var turn: Int = -1
 def reset() { * same as before * }
```

EdgeC emits two-phase commit for all invocations of `play` and `resetTurn` (regardless of places of triggering) to preserve the same ordering of observing two operations on `server` and `server2`, due to potential conflicts.

EdgeC guarantees the chosen level of consistency, however, unlike traditional data management systems, it optimizes distributed behaviors based on network topology and operation semantics, making it more amenable for the edge. The optimization is sound, but incomplete, as EdgeC might fail to recognize a case where optimization is valid and emit a costly consistency protocol.

Adapting implementations with network models

EdgeC allows developers to model the network, specifying a cost of communication as well as execution on individual nodes. Developers specify custom models with `network`. If left unspecified, EdgeC assumes the default uniform network model with costs of execution and



```
val g = Graph[Node]()
g.node(server, 1.0)
val edges = for (c <- consoles) yield {
  g.node(c, 1.0)
  g.edge(c, server, 1.0) } * bidirectional *
g.node(cs1, 0.1); g.edge(cs1, server, 0.1)
initNetwork(g)
```

Figure 4.3: Network model specifications

communication equal to 1 across all nodes and edges. The network model specification takes a directed weighted graph of the network as an argument, as shown in Figure 4.3[2].

In the current example, without network model specification, for `play`, EdgeC generates behavior following the client-server pattern, as mentioned. If the depicted network model is specified, however, since `cs1` has lower cost of computing and connectivity (`0.1`, as depicted on the left), EdgeC will allocate computation on the node, incurring more communication rounds, but still overall lower cost, according to the given model.

Replication and aliasing

EdgeC supports replication as an experimental feature in the prototype, limited to certain forms of scenarios (programs outside the supported class will fail to compile). In the running example, developers might decide to replicate data across multiple server instances (as shown in Figure 4.1). Replication is supported in the EdgeC prototype by providing a special annotation:

```
@replicated(3) node Server
def resolveReplica(c: Console): Int = ...
@resolver(resolveReplica) def play( * as before * )
```

Here, EdgeC replicates `server`, implicitly, across 3 instances. It resolves accesses to replicated variables, e.g. in `play`, based on an externally defined function `resolveReplica`. Modifying server variables under strong consistency now involves 2PC across all server replicas. Replication is supported on the level of a node type; we plan to extend it to apply to specified node groups and specific variables.

Peer-to-peer. Let's assume developers want to fit their application into a p2p model. In this setting, there are only console nodes, while every console has its designated console

[2]Our prototype currently does not use reflection, thus graphs have to be static instances, known at compile time.

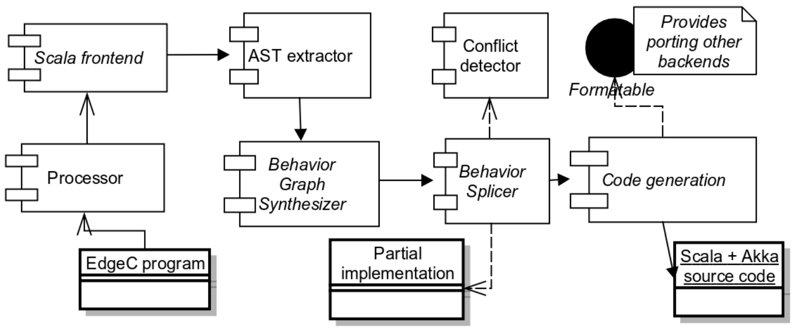


Figure 4.4: EdgeC compiler

node acting as the server. This can be achieved by declaring another scenario, similarly as before, but specifying the server to be one of the console nodes.

```
scenario P2PGame(server: Console, cs: Set[Console]) {
 require(cs.contains(server))
 ... * rest is as before * }
```

Without any other changes to our running example, EdgeC recognizes the precondition to conclude that the `server` is also one of the `console` nodes. The main difference with respect to the previous case is that a console node now stores all the game variables (in addition to `myId`) and `play` is instantiated for two cases: 1) the originating console is also the server, and `play` incurs no communication (as the behavior executes locally on the single console); 2) the originating console is not the server, in which case the behaviors are the same as discussed before.

4.2 EdgeC Compilation

EdgeC contains two parts: the compiler and the runtime. EdgeC employs a novel compiler design which incorporates multiple program analyses. The compiler compiles EdgeC programs to Java bytecode, while the runtime implements communication primitives and consistency protocols (using Akka [*Akka – actor toolkit and runtime, http://akka.io/* 2023]). The runtime exposes APIs which are called by the generated code.

Here, we focus on the main tasks of the compiler. The overview of the compilation is given in Figure 4.4. The core parts are the behavior graph synthesizer and splicer, which analyze behaviors and specifications, and incrementally splice appropriate code into the distributed implementation. The compilation process includes helper components: PEEPCO preprocessor; AST extractor, which extracts behaviors and PEEPCO specifications; implicit resolver, which binds nodes at the places of behavior invocations; and code generator, which transforms the internal representation of programs into bytecode. Our prototype compiler is implemented as a Scala compiler plugin.

High-level compilation loop. The compilation process loops over all triggers, identifying all the places they might trigger, instantiates bound behaviors, and analyzes the current set of behaviors for consistency levels under which the behaviors need to be executed. Effectively, this is done until a fixed point is reached, i.e. no new changes are made to the resulting implementation.

Behavior splitting and allocation. Given an operation invocation and its starting node, a behavior graph designates the shape of a distributed computation: its nodes represent executions of operation chunks on particular nodes, while edges represent data or control-flow dependencies between chunks. For each invoked behavior, the process splits the operation and assigns individual chunks to be executed on particular nodes, producing a behavior graph. This is done with minimizing the overall cost of mapping (usually collocating data with computation). A behavior graph effectively encodes an execution graph: following the topological sort of such a graph, the system can execute the operation, incrementally distributed across the system, while propagating data dependencies according to its edges. (For example, a graph of `play` in the running example, reflects the simple client-server model; it has three nodes, with edges encoding data dependencies for the request and response.) A set of behavior graphs serves as the intermediate representation of the final implementation, and it captures enough information to allow emitting low-level code (and allow optimizations based on the given network model and interactions with other behaviors, e.g. the aforementioned message consolidation).

Trigger splicing. Trigger splicing identifies the places where triggers get activated (and operations invoked). They can be at the originating nodes, in case of external triggers, or inside existing behaviors that enable conditional triggers during execution (e.g. for `onChange(score)`, it's on the `server`, where `play` modifies `score`). EdgeC analyzes the current set of behavior graphs to discover possible places where these might get enabled and emits necessary run time checks.

Consistency analysis. EdgeC, when parametrized with strong consistency, checks for conflicts by analyzing each pair of behavior graphs. It employs an analysis based on interference conflict analysis, which allows detecting possible conflicting transactions statically [Shasha, Llirbat, et al., 1995]. We modify the original algorithm to handle distributed executions, accounting for the network model (and single-threaded execution per each node, which our prototype currently employs).

Scalability of analyses. All the analyses run in polynomial time with respect to the program size, except the splitting phase which run in exponential time. (A faster,

Algorithm 5 Execution model as an interpreter

```
Input: start state $s_0$, condition triggers $T_c$
 1: $s \leftarrow s_0, T \leftarrow \emptyset, E \leftarrow \emptyset$          ▷ state, triggers, execution graph
 2: loop
 3:   $T \leftarrow T$++ newEvents()                                   ▷ include new external events
 4:   $E \leftarrow E$++ chooseAndInstantiate(T, s)                    ▷ start new behaviors
 5:   $el \leftarrow$ removeTop(E)                                     ▷ next element in topological order
 6:   if $el = node(e, t)$ then                                        ▷ node case
 7:     $res(el) \leftarrow$ eval(s, $env(el)$); $s \leftarrow s$++$res(el)$   ▷ evaluate chunk
 8:     if $el$ is last chunk in operation then
 9:       $E \leftarrow E$++ instantiateAll($T_c, s$)                  ▷ check condition triggers
10:     else if $el$ is last chunk in condition trigger $t$ then
11:       $T \leftarrow T$++ checkEnabled($t$)                         ▷ new trigger enabled
12:   else $el = edge(n_1, n_2)$                                       ▷ edge case
13:     $env(n_2) \leftarrow env(n_2)$++$res(n_1)$                     ▷ communicate dependencies
```

heuristic-based, allocation method is left for future work.)

4.3 Language Semantics

We formalize the semantics of EdgeC programs with an event-driven model of execution, enriched with specifications of distribution, reactive events, and consistency concerns. It defines the requirements of the execution model, while allowing different concrete scheduling, consistency models, and optimizations.

Event-model semantics. We define dynamic semantics by interpretation [Reynolds, 1972]. The interpreter is given in Algorithm Algorithm 5. Inputs to the interpreter include the starting state and a set of all condition triggers (such as `onChange`, which need to be checked during execution). The system maintains the current state (across all nodes), the set of active triggers T and an execution graph E. The execution graph represents all active behaviors, waiting to get executed. Whenever an operation is invoked, its behavior is instantiated and added to the main graph E. These behaviors might be either operation invocations or condition checking (which need to evaluate an expression over current state, e.g. `onTrue(score>100)`).

The interpreter initializes variables and starts looping. It first collects new external triggers to the set of active triggers T. Then, it chooses a subset of T to instantiate behaviors (`chooseAndInstantiate`), adding them to the main execution graph E. The instantiated set depends on the assumed consistency model: for strong consistency it only instantiates behaviors which do not conflict with any of the active behaviors; for weak, it instantiates all behaviors of enabled triggers. Next, the algorithm picks a node (to execute) or an edge (to perform a communication step) from E: it picks either a non-visited node from E which has all incoming edges visited, or a non-visited edge which has the source node visited. (This is akin to topological order, but generalized to traverse edges as well.) If it visits a node, it executes its behavior chunk, with its environment ($env(el)$), for a result ($rel(el)$), and updates the state (s). If the node is the last non-visited node belonging to: 1) an operation invocation, it instantiates all condition triggers (to be checked, since some of them might trigger as a result of the current execution); 2) condition behavior, the evaluation result represents a Boolean which determines if the given trigger should fire, enabling new behaviors. If it visits an edge (from n_1 to n_2) it processes the communication step by updating environment for the chunk n_2.

4.4 Evaluation

This section evaluates the EdgeC prototype showing potential in performance gains due to implementation tweaking allowed by the expressiveness of the language. We evaluated the JVM implementation of EdgeC on an OpenStack Compute Cluster using 8 machines (3GHz clock speed and 2GB of RAM).The benchmarks include: 1) the standard Retwis benchmark ([I. Zhang et al., 2016]); 2) "reactive Retwis", with added reactive behaviors; 3) `play` from the running example, over a non-uniform network. The results are shown in Figure 4.5. EdgeC finds all optimizations of behaviors for the given benchmarks; we thus believe it performs similarly to manual implementations that rely on the same optimizations.

Redis over uniform network. The first row of graphs shows the Retwis benchmark, specifically: EdgeC implementations with strong and no consistency (*strong* and *weak*); Redis-based (no consistency) implementation with and without concurrent handler (*redis* and *redis_ 1thread*). EdgeC's performance is comparable to that of standard Redis in the weak consistency model. A performance penalty in EdgeC occurs for strong consistency, as expected, as around 3/4 of operations require coordination to maintain consistency.

Reactive behaviors. We added a new operation to the Retwis benchmark for notifications of new post or likes. In Redis, clients poll after each operation to check for new notifications. The results are shown in the second row of Figure 4.5. The experiments confirm the expected performance penalty due to polling in Redis; EdgeC exhibits better latency and throughput. The reason is direct splicing of triggers that avoids polling.

Non-uniform networks behaviors. We evaluated benefits of leveraging the network model in EdgeC, under a non-uniform network. We ran the `play` operation from the running example, with the network model from Section 4.1; we simulated this by adding delay of

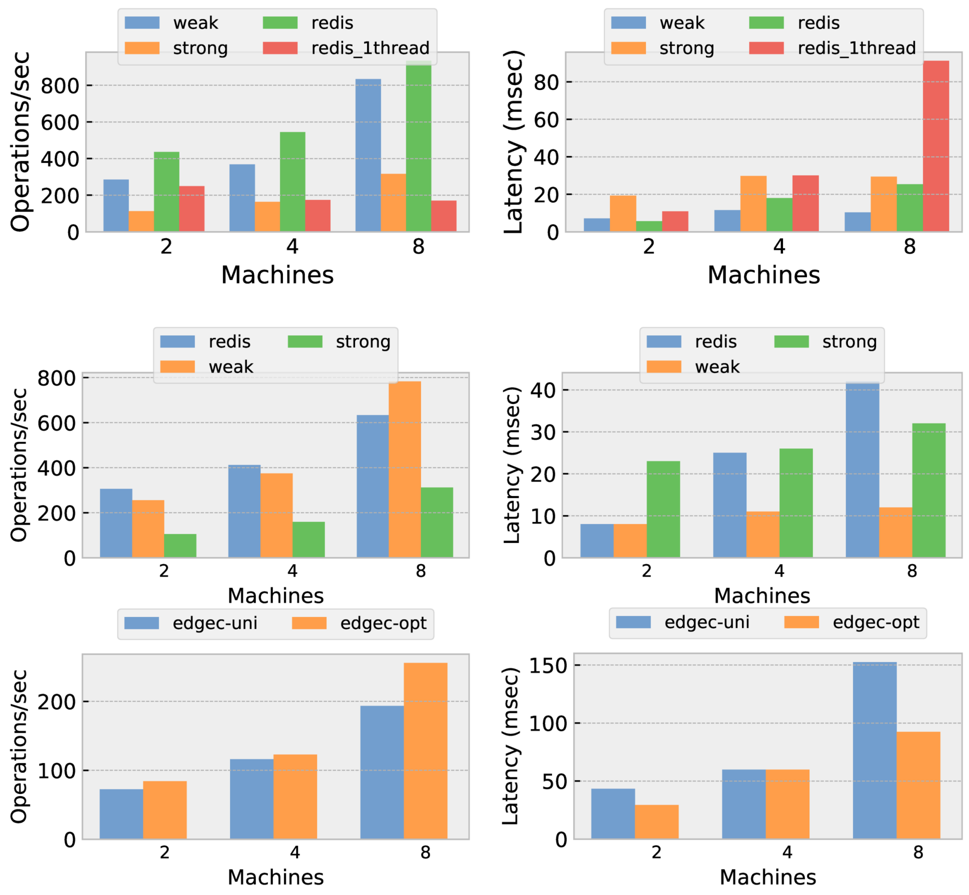


Figure 4.5: Performance evaluation

15ms to both computation and message receives on "slow" nodes. The results are shown in the third row of Figure 4.5. EdgeC outperforms the uniformly communicating system, in both the throughput and latency, since the adjustments in computation based on the given network avoid the unnecessary latency penalties. This becomes more significant with 8 nodes (with slowdown of around 25%).

Overall, the results demonstrate EdgeC achieves expected performance while providing standard level of consistency, on uniform networks, while the expressiveness of controlling distributed aspects can significantly improve performance in specific scenarios.

4.5 Extensions

We demonstrate the extensibility of EdgeC that could aid practical deployment of the generated distributed programs.

Fault tolerance. We implemented the support for the detection and restarting of nodes in situations of individual node failures in the system. Fault tolerance support in our prototype implements snapshot recovery thought the Akka persistence layer [*Akka persistence layer* 2023]. (Note that unlike other systems, this form of fault tolerance does not perform replication, and thus it does not offer availability under ongoing failures [Gray and Lamport, 2006; Lamport, 2001].) The bulk of the implementation is in the runtime: EdgeC synthesizes a global supervisor node, which, whenever a node fails (detected by the provided Akka failure detector) locks all the actors, resets them to the last snapshot, and replays events (including external stimuli).

While our implementation demonstrates the viability of EdgeC for further extensions, for handling additional practical concerns, it is prohibitively inefficient for many real-world applications during a failure, due to the "lock-the-world" behavior. In our benchmarks, it incurs up to 15% performance overhead in normal mode (where writes to disk are done asynchronously).

Eclipse Hono extension of EdgeC. To show flexibility of EdgeC, and extensibility for different implementation backends, including already existing and mature frameworks, in addition to the Akka backed, we implemented a backend that uses the Eclipse foundation, open source Internet-of-Things orchestration framework Hono [*Eclipse Hono IoT framework* 2023]. Eclipse Hono specifically targets range of devices indirectly, such as sensors and actuators, and low-energy communication networks directly, by relying on low-performance networking protocols. (It includes the support for Zigbee, MQTT, MQTT-IoT, and others.) Notably, since many of the mentioned protocols support much weaker guarantees for the network delivery (unreliable delivery is in the assumptions of the framework), EdgeC application-level mechanisms become required and provide an important part in achieving consistent delivery whenever required.

The support for Eclipse Hono as a backend was a matter of adding different backend, which besides few utility transformation data structures in the compilation process, required a new code generation component. We also added support for programmers to mark the incoming and outgoing traffic of EdgeC nodes as "event" and "telemetry" (specific types of packets in Hono, where "telemetry" provides weaker guarantees, while allowing less powerful underlying network protocols), so that the annotated nodes might get deployed as sensors or actuators through the framework in the system. We tested our benchmarks in the framework-provided Docker-based simulation environment for the given example applications, where we found the performance trends were comparable to those presented in Section 4.4.

4.6 Discussion

Overall, our results suggest that EdgeC can indeed simplify the development of distributed reactive applications and provide ways of controlling distributed aspects of reactivity, allocation and, uniformity of architectures, allowing programmers to pick the desired point in the trade-off space with: 1) *strong guarantees and a comparably small performance overhead*, 2) *weak guarantees and favorable performance* over uniform architectures, and 3) *strong guarantees and favorable performance, in the case of specialized, non-uniform architectures*, when compared to a state-of-the-art industry-level data management system [*Redis - in-memory data structure store* n.d.]. We believe the integration of consistency optimization approaches described in Chapter 2 and Chapter 3 can bring additional benefits and make the trade-off points more favorable for larger class of programs.

Semantic projection. While EdgeC allows programmers to describe application logic in sequential semantics simply by declaring abstract state and programs that modify it, EdgeC dictates the final shape of how the final behaviors in the implementation will look like. The extent of the control over the resulting distributed programs is limited by specifications that dictate such shapes: how the state is distributed across nodes, how the computation gets allocated at run-time, in response to which (external or internal) events in the system behaviors are invoked. This separation of concerns aims to allow programmers to simplify the reasoning about the relationship of the core logic of their applications and distributed aspects. To emit the final implementations, our synthesizer chooses and emits implementation as the "semantic projection" of the behavior given the allocation and timing specifications, following the semantic projection rules that hide the particular computation allocations, messaging patterns, and internal events that occur in the system to invoke behaviors. While we have shown the specifications can bring the needed flexibility for a class of programs, programmers that need more control over the projection of their behaviors, how the data and computation is allocated and invoked, might need to manually inspect and modify the emitted implementations.

Chapter 5

Related Work

Prior work explored different distributed programming frameworks and models that offer high-level abstractions useful in developing distributed applications. Many of the existing approaches, however, are too rigid. When developing programs that fall outside the supported classes of these approaches, programmers need to fall back and manually implement and reason about the low-level implementation and its distributed aspects, mainly consistency and the consistency model choice to adopt [G. A. Agha and Kim, 1999; Shapiro et al., 2011a; I. Zhang et al., 2016; Viennot et al., 2015; Christensen, Møller, and Schwartzbach, 2003; Andrew D. Birrell and Nelson, 1984b; Haridi et al., 1998]. Our approach aims to extend the class of supported programs, introducing the ability to handle consistency through high-level integrity specifications, while keeping high degree of automation.

Approaches that attempt to hide certain aspects related to consistency, such as conflict-free replicated data types (CRDTs), which offer certain operations with consistent updates without requiring coordination [Shapiro et al., 2011a], can be used as building blocks for constructing distributed programs with replicated data, but programmers still need to manually reason whether the desired high-level integrity is preserved. Studies of CRDTs and their properties, have identified properties that can be checked in order to satisfy certain classes of high-level integrity properties [Shapiro et al., 2011b; Burckhardt et al., 2014; Shapiro et al., 2011a; Zakhour, Weisenburger, and Salvaneschi, 2023; Bailis, Fekete, et al., 2014; Alvaro, Conway, Joseph M Hellerstein, et al., 2017; Y. Liu et al., 2020]. This led to work that motivated consistency optimization [Terry, Demers, et al., 1994] and led to creating automated approaches based on static analysis of program semantics [Sivaramakrishnan, Kaki, and Jagannathan, 2015; I. Zhang et al., 2016; C. Li, Leitão, et al., 2014; Balegas, Duarte, Ferreira, R. Rodrigues, Preguiça, Najafzadeh, et al., 2015a; Kaki, Earanky, et al., 2018; Houshmand and Lesani, 2019]. Our approach aims to explore a new point in the design space of consistency optimization systems and provide an end-to-end approach that:

1) achieves high levels of flexibility and granularity of the supported consistency through high-level invariants; 2) supports specifications of multiple practically important distributed aspects; 3) supports expressive operation executions in the distributed system; 4) maintains a high degree of automation.

The main inspiration for the work presented in this thesis came from analyzing key challenges in modern data management systems [Stonebraker and Çetintemel, 2018]. The idea of an end-to-end programming approach from high-level specification was motivated by modern multi-tier programming systems [Chlipala, 2015] and program synthesizers [Kneuss et al., 2013]. Our inference is motivated and takes inspiration from the fundamental concepts introduced in prior work that introduced the seminal ideas of static analysis of programs for concurrency [Shasha and Snir, 1988; Shasha, Llirbat, et al., 1995], introduced the problem of consistency inference [Sivaramakrishnan, Kaki, and Jagannathan, 2015], and formalized dependencies between consistency models [Terry, Demers, et al., 1994]. Next, we discuss the related work and group different approaches based on their relations to our approach.

5.1 Consistency Optimization

Prior work explored different points in the design space of consistency optimization of distributed programs and data-centric applications, focusing on the case where data is fully replicated across all nodes. Approaches for building and optimizing replicated data stores generally make tradeoffs in the supported consistency guarantees, and expressiveness for different behaviors and underlying configurations. In the expressive programming models, to satisfy the needed integrity constraints of their applications, programmers often choose to use off-the-shelf data stores, but may also need to implement additional safety mechanisms due to the limited support for consistency [Sivasubramanian, 2012; *Cassandra replication factor* 2020; Sivaramakrishnan, Kaki, and Jagannathan, 2015], or default to strong consistency and accept performance penalties [Deng et al., 2017; *Apache Ignite allocation modes* 2020; Bailis, Fekete, et al., 2014]. Our approach aims to bridge the gap between the two extremes and automate the process of fine-tuning consistency based on the programmers' needs.

Static Analysis for Consistency Optimization. The idea of using static analysis to optimize consistency was motivated by a line of work. Quelea [Sivaramakrishnan, Kaki, and Jagannathan, 2015] introduced static analysis for consistency optimization, using high-level contracts. Q9 [Kaki, Earanky, et al., 2018] added application-specific invariants and proposed the use of bounded symbolic reasoning. Hamsaz [Houshmand and Lesani, 2019] extended the idea with synthesis of replicated objects, and Hampa [X. Li, Houshmand, and Lesani, 2020] added reasoning about recency. ECROs [De Porre, Ferreira, et al., 2021] avoids coordination further by local reordering. Atropos [Rahmani et al., 2021] introduced consistency optimization by exploring adjustments in the program's data layout. PEEPCO combines static reasoning with runtime information to realize new optimization opportunities, while allowing specializing consistency model through integrity, and, in turn controlling and exposing state after convergence, periodically. PEEPCO's protocol, however, is blocking and requires agreement even for non-conflicting operations. In addition to replicated objects, PEEPCO extends the expressiveness to support partial replication.

Run-time Techniques. Sieve [C. Li, Leitão, et al., 2014; C. Li, Porto, et al., 2012] defines a consistency model called RedBlue which executes operations with two modes of consistency. Olisipo [C. Li, Preguiça, and R. Rodrigues, 2018], refines the approach and presents two coordination protocols, where one can be used for infrequent operations. PEEPCO supports static analysis in the extended consistency lattice, including weak consistency, automatically checks that integrity holds, and uses dynamic information to allow concurrent execution of operations. Homestasis [Roy et al., 2015] infers "treaties", run-time checks for node-local state to avoid coordination. By contrast, PEEPCO's run time checks optimize operations even when a treaty might be violated, using dynamic information.

Elaborate Specifications. Some approaches let developers aid the consistency optimization by providing, tags and conflicts between tags CISE [Gotsman et al., 2016; Najafzadeh et al., 2016], counterexample fixes for inductive generalizations [Padon et al., 2017], and conflict avoidance annotations in Indigo [Balegas, Duarte, Ferreira, R. Rodrigues, Preguiça, Najafzadeh, et al., 2015a; Balegas, Duarte, Ferreira, R. Rodrigues, Preguiça, Najafzadeh, et al., 2015b], conflict resolutions in IPA [Balegas, Duarte, Ferreira, R. Rodrigues, and Preguiça, 2018] and consistency guards in Carol [Lewchenko et al., 2019]. By contrast, PEEPCO provides automation while only requiring the specification of high-level invariants. While PEEPCO cannot provide the guarantees of unbounded full verification ([**lesani2016chapar**; Wilcox et al., 2015; Nagar and Jagannathan, 2019]), empirical evidence suggests it provides performance benefits by leveraging results from a relatively expressive SMT-based bounded-reasoning procedure, while preserving application-specific integrity at run time.

Type Systems. Disciplined inconsistency [Holt et al., 2016] presents a type system that disallows flow of weakly consistency values into strongly consistent operations, unless allowed by the programmer. MixT [Milano and Andrew C. Myers, 2018] introduced transactional support with types that allow accessing different stores with varying consistency guarantees and used information flow analysis to limit the influence of weakly-consistent data on transaction control flow. Some programming models allow mixing consistency levels to some extent [De Porre, Myter, et al., 2020; Köhler et al., 2020; Holt et al., 2016; Xie, Su, Kapritsos, et al., 2014]. PEEPCO splits methods into pieces under data partitioning, but it does not require consistency annotations to be associated with objects and locations, and it infers consistency of each piece automatically. PEEPCO, moreover, assigns consistency levels to particular operation splits, rather than fixing a particular object or memory location with a specific model. It considers control-flow dependent splits, as well as other operations that might occur at run-time, but it does not handle loops.

Replicated Objects. Replicated objects and data types have been proposed [Shapiro et al., 2011b; Burckhardt et al., 2014; Shapiro et al., 2011a; Zakhour, Weisenburger, and Salvaneschi, 2023], studied in terms of invariant preservation [Bailis, Fekete, et al., 2014; Alvaro, Conway, Joseph M Hellerstein, et al., 2017; Y. Liu et al., 2020], and used in systems [I. Zhang et al., 2016; Balakrishnan et al., 2013]. MRDTs [Kaki, Priya, et al., 2019] has shown that such objects allow deriving merge functions that can alleviate conflicts and provide safety. Quark [Kaki, Prahladan, and Lewchenko, 2022] then introduced convergence with run-time assistance, and Peepul [Laddad et al., 2022] verifies their efficient implementations. Katara [Laddad et al., 2022] used lightweight ordering constraints. The space of consistency models has been characterized in terms of a partially-ordered lattice [Viotti and Vukolić, 2016; Terry, Demers, et al., 1994]. PRACTI [Belaramani et al., 2006] motivated the partial replication model and defined flexible consistency as one of its key components. PEEPCO follows the approach of encoding different consistency levels as a lattice. PEEPCO, however, also supports partial replication by extending reasoning about the system to many different nodes and includes the flexibility of programmer-specialized consistency models, in addition to being "tunable" based on the operation semantics [Viotti and Vukolić, 2016].

Databases and Transactional Systems. Prior work has studied correctness conditions for finding the lowest isolation level for transactions [Lu, A. Bernstein, and P. Lewis, 2004; Fekete et al., 2005]. Alone-Together [Kaki, Nagar, et al., 2018] formulated a program logic for automated verification of weakly-isolated transactions. In addition, transaction decomposition was used to aid such reasoining, with transaction chopping [Shasha, Llirbat, et al., 1995; Cerone, Gotsman, and Yang, 2015], step-decomposition [A. J. Bernstein, Gerstl, and P. M. Lewis, 1999], transaction chains [Y. Zhang et al., 2013], locality-aware [Cheung et al., 2012] and security-aware decomposition [Zdancewic and Andrew C Myers, 2003]. Other work incorporates run-time heuristics into chopping approaches, to extract concurrency [Mu et al., 2014; Xie, Su, Littley, et al., 2015; Y. Zhang et al., 2013]. Similarly, PEEPCO splits methods, treating them as transactions, and employs bounded exploration for verification to achieve automation and avoid the pitfalls of limited information analysis of traditional chopping.

There is significant prior work on reducing coordination, by allowing (bounded) staleness in replicas [Yu and Vahdat, 2000; Bailis, Venkataraman, et al., 2012], by providing applications with a choice between fast inconsistent results and slower consistent results [Guerraoui, Pavlovic, and Seredinschi, 2016], or by dynamically tuning consistency in response to load [Kraska et al., 2009; Terry, Prabhakaran, et al., 2013]. In contrast to these approaches, PEEPCO reduces coordination only when it can still preserve application integrity, at the potential cost of some performance. PEEPCO, however, provides fast local reads, as well as congruent snapshots that do not slow the system down, regardless of conflicts in the update methods.

Concurrency synthesis. Similarly to PEEPCO, prior approaches leveraged bounded oracles and generalization within small scope of exploration [Wickerson et al., 2017; Solar-Lezama, Jones, and Bodík, 2008; Vechev and Yahav, 2008]. PSketch [Solar-Lezama, Jones, and Bodík, 2008] used CEGIS to synthesize concurrent data-structures from partial programs and test harnesses, using an explicit model checker. Paraglider [Vechev and Yahav, 2008] focused on partitioning atomic sections by introducing locks. The verified-synthesis loop inspired a line of work of inferring various concurrency mechanisms [Vechev, Yahav, and Yorsh, 2009; Kuperstein, Vechev, and Yahav, 2010]. Albeit expressive, practical scalability in the search space of distribution and partial replication requires mechanisms that go beyond low-level locks. Combinations of analysis and inference techniques were explored for memory models [Wickerson et al., 2017] and cache coherence protocols [Udupa et al., 2013]. PEEPCO explores a different point of specialization, within the expressive domain of distributed consistency, which requires specialized mechanisms for capturing and analyzing distributed executions across fine-grained replication. While avoiding the issues of generality, our framework specializes in distributed consistency and offers "small-scope" inference but also multiple directions for extendability.

5.2 Distributed Programming Languages

Prior work introduced approaches that allow programmers to specifying and control distributed aspects such as location and reactivity, either limited to certain classes of programs without expressiveness in consistency optimization, or requiring low-level manual implementation. We believe that our approach offers a new way of specifying reactive distributed programs by combining specifications of data distribution and triggers, together with high-level invariants on the system's state.

Programming models. Our approach shares some of the high-level goals with the following lines of research on programming distributed systems:

tierless programming models Similar in spirit of avoiding the complexity and complicating the underlying sequential programming model, tierless programming models focus on simplifying specification of aspects that cut across different tiers and unify them into a single model (and traditionally, focus on web development) [E. Cooper et al., 2007; Serrano and Berry, 2012; Chlipala, 2015]. Although these models simplify some of the aspects considered in this work, including communication, strong consistency transactional support, storage and interaction, their focus is to remove the complexity that arises due to handling different tiers of the system, rather than on preserving the semantics and structure of sequential computation within the same tier. Note that tierless models usually adopt existing mechanisms and constructs, such as client-server architecture and RPC for communication [Chlipala, 2015].

actor-based programming models Despite being flexible and providing clean abstractions for programming distributed event-driven systems that can easily be mapped to actual physical systems, actor models suffer from being close to the low-level implementations, where the structure of the system and behaviors need to match closely with the declared programs, making them complex and hard to reason about [G. Agha, 1986; Hewitt, Bishop, and Steiger, 1973; Haller and Odersky, 2009; Prokopec and Odersky, 2015]. Interestingly, actor-based programming frameworks represent a good fit for a low-level model that can be leveraged in the final emitted implementations [Kuraj and Jackson, 2016; Prokopec and Odersky, 2015].

partitioned global address space Partitioned global address space (PGAS) models aim to provide a simple programming model, and consequently allow better performance, for parallel programs by unifying the support for data and task parallelism, and abstracting the data model through a global address space [Charles et al., 2005; De Wael et al., 2015]. The concept of a “place” in these models allows allocating computations and data across the global address space, at a level that can be closer to the intended (sequential) behavior. Although places allow assigning a cost model to data accesses (based on the topology), automatic data distribution is usually restricted to partitioning of regular and dense data structures such as arrays; some PGAS languages require explicit distribution of data objects to remain expressive for irregular and sparse structures [De Wael et al., 2015]. Nodes in our model are similar to places in PGAS in that they contain running computations, which in turn might be spread across multiple different nodes. However, our model does not rely on specific patterns of data distribution and parallelism; it analyzes defined behaviors to emit event-driven implementations that need to satisfy consistency guarantees, and appropriately allocate both computation and data.

Reactive programming. Reactive programming itself has received significant attention from the research community, including in the context of distributed systems [Drechsler et al., 2014; Baldini et al., 2016; Dabek et al., 2002]. While allowing clean way of composing reactive values to achieve automatic updates of dependent computations, in distributed reactive programming, developers need to identify specific points in the graph, usually through a form of publish-subscribe mechanism, to forward the propagation through the network. In addition, since reactive values are often encapsulated as signals, adding new and merging existing functionalities often requires modifications to existing dependency graphs. Moreover, due to relying on automatic propagation algorithms they are often restricted to weaker consistency models like glitch-freedom and specialized to certain distributed architectures.

Location types. Location annotations as well as data and computation allocation were examined in the context of multi-tier programming models. Even though the formalization of the syntax and types shares similarities with our specifications, our approach differs in that it covers computations that span arbitrary numbers of nodes, as well as consistency criteria that naturally arise. In the previous approaches allocation is determined directly, while the consistency issue does not arise at all, due to the restriction to a two-party multi-tier model.

Multi-tier programming. Many approaches presented in prior work focus on using sequential computation to some extent while introducing additional abstractions, such as remote procedure calls, reactive values, and conflict-free replicated data type, for handling distributed aspects of the system [Chlipala, 2015; *JMacroRPC - reactive client/server web programming* n.d.; *Meteor - Pure JavaScript web framework* n.d.; Czaplicki and Chong, 2013; Meiklejohn and Van Roy, 2015]. A related line of research includes programming platforms based on writing sequential programs that aim at abstracting away infrastructure concerns to allow focusing on the application logic [Baldini et al., 2016; Kiciman et al., 2010]. An overview of different programming models and the influence of the sequential model on programming distributed systems is given in [Briot, Guerraoui, and Lohr, 1998; Bal, Steiner, and Andrew S. Tanenbaum, 1989]. In general, even though these models abstract away some of the complexity, due to the close match between the program and the final distributed implementation, expressing certain complex behaviors requires low-level reasoning and careful structuring of the program [Andrew Stuart Tanenbaum and Renesse, 1987; Prokopec and Odersky, 2015].

Sequential computation-based approaches. Our approach is aligned with the idea of using high-level specifications of distributed aspects and offloading the search for low-level implementations to the compiler. Some approaches lift the abstraction of specifying behaviors by using similar mechanisms to the ones employed by our approach, including logical formulas (used for triggering in our approach) in the form of event guards and await statements, and the concept of location, which allows automatic data distribution according to specifying computations [Y. A. Liu et al., 2012; Jayaram and Eugster, 2011; De Wael et al., 2015]. Prior work discusses the importance of preserving semantics of sequential computation and its effects on possible optimizations, as well as the potential role for programming distributed systems [Marino et al., 2015; Kuraj and Jackson, 2016]. In the similar spirit, this work tries to motivate lifting the level of abstraction by demonstrating potential gains in simplicity and performance. Moreover, it provides a different perspective on formalization of sequential computation and specifications to allow additional means for ensuring correctness and efficiency of the resulting implementation.

Development of distributed algorithms. While our approach focuses on implementing behaviors which can be conceptually expressed as sequential programs, it lacks expressiveness for programming distributed algorithms that inherently require dealing with aspects like processes and messages, and require control of low-level concerns [Y. A. Liu et al., 2012; Prokopec and Odersky, 2015]. While reimplementing such algorithms is rarely needed, they often cannot be used directly via an external library (e.g. if modifications to some of its internals are needed); our approach aims at utilizing different existing algorithms as means to an end whenever necessary, even in cases their code needs to be customized for specific needs of the intended distributed application.

Chapter 6

Conclusion

In this thesis, we have introduced a novel approach to the development of distributed programs, enabling the specification of behaviors in the sequential computation model. This approach empowers programmers to reason about and test their applications as sequential executions, while our tool, Peepco, automates the synthesis of low-level distributed code. By allowing programmers to specify distributed aspects using orthogonal specifications, our method facilitates a clear separation between concerns. This separation enables developers to modify functionality and shape distributed behaviors without the need for cross-cutting code, thereby streamlining the development process. Importantly, our approach implements the optimal consistency model based on the given input program's application-level integrity properties. Leveraging the exploration of bounded executions, we support an expressive programming model, presenting a unique set of challenges distinct from prior work on consistency optimization and static analysis for consistency inference. To tackle these challenges, we draw upon a diverse range of ideas and techniques, from static analysis of concurrency, model-checking, verification, synthesis, databases, and distributed algorithms.

Our emphasis on code generation establishes an important bar for the expressiveness of the programming model, the granularity of identified consistency requirements, and the precision of protocols used in resulting implementations. We believe that the integration of code synthesis, static analysis, and distributed protocol design can afford the requisite expressiveness for modern distributed application development. This integration enables our approach to address the inherent tension between the complexity of operation interleaving and the expressiveness of properties to be preserved in the resulting program, all while ensuring the desired distributed aspects. However, achieving the necessary level of automated reasoning for generating code in practical systems poses challenges. These challenges stem from the need for specifications expressive for controlling certain low-level aspects, scalability of static analysis, and efficiency of the run-time protocol that maintains strong guarantees.

For some programs, the proposed integrity properties might not be fully capable to specify the required behaviors of the intended implementations, and thus might require additional effort from the programmers. Our research demonstrates that our approach is capable of fully capturing various benchmarks with expressive requirements for the intended distributed programs, while our consistency inference analysis exhibits sufficient scalability to enable achieving significant speedups at run time. While our approach does not handle all aspects that might be needed for practical deployments, such as security and fault-tolerance, we believe it represents an important step towards the over-reaching goal of achieving a practical easy-to-use programming system for development of distributed programs.

Peepco's design offers a significant advantage by giving programmers the ability to enhance the potential of consistency optimization through extended static analysis, covering larger bounds as analysis runs longer. Furthermore, the flexibility of Peepco's consistency inference allows incorporating existing systems and their internally supported consistency levels into the execution trace checker. This integration can seamlessly expand the space of the covered resulting implementations, facilitating the generation of practical compositional implementations. This is supported by the compositional treatment of integrity and other distributed aspects specified in Peepco, enabling the combination of different implementations based on their properties. This compositional approach, coupled with efficient read-only methods, ensures that overall system requirements, including its integrity, are maintained as the sum of its individual parts. We believe the design and architecture of Peepco exemplifies a novel method for combining specifications, static analysis, and run-time protocols, that can open up new avenues for achieving expressiveness and scalability in modern software development.

References

Zhang, Irene, Niel Lebeck, Pedro Fonseca, Brandon Holt, Raymond Cheng, Ariadna Norberg, Arvind Krishnamurthy, and Henry M. Levy (2016). “Diamond: Automating Data Management and Storage for Wide-Area, Reactive Applications”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, pp. 723–738. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhang-irene.

Viennot, Nicolas, Mathias Lécuyer, Jonathan Bell, Roxana Geambasu, Jason Nieh, L Mathias, Jonathan Bell, Roxana Geambasu, and Jason Nieh (2015). “Synapse: A Microservices Architecture for Heterogeneous-Database Web Applications”. In: *EuroSys*.

Chlipala, Adam (2015). “Ur/Web: A simple model for programming the Web”. In: *POPL*.

Lesani, Mohsen, Christian J. Bell, and Adam Chlipala (2016). “Chapar: certified causally consistent distributed key-value stores”. In: *POPL*.

Sovran, Yair, Russell Power, Marcos K. Aguilera, and Jinyang Li (2011). “Transactional Storage for Geo-Replicated Systems”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP ’11. Cascais, Portugal: Association for Computing Machinery, pp. 385–400. ISBN: 9781450309776. DOI: 10.1145/2043556.2043592. URL: https://doi.org/10.1145/2043556.2043592.

Bailis, Peter, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica (2014). “Coordination avoidance in database systems (Extended version)”. In: *arXiv preprint arXiv:1402.2237*.

Papadimitriou, Christos H. (Oct. 1979). “The Serializability of Concurrent Database Updates”. In: *J. ACM* 26.4, pp. 631–653. ISSN: 0004-5411. DOI: 10.1145/322154.322158. URL: https://doi.org/10.1145/322154.322158.

Brewer, Eric (2012). “CAP twelve years later: How the "rules" have changed”. In: *Computer* 45.2, pp. 23–29.

DeCandia, Giuseppe, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels (2007). “Dynamo: Amazon’s Highly Available Key-value Store”. In:

Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles. SOSP '07. Stevenson, Washington, USA: ACM, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

Kleppmann, Martin, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan (2019). "Local-First Software: You Own Your Data, in Spite of the Cloud". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2019. Athens, Greece: Association for Computing Machinery, pp. 154–178. ISBN: 9781450369954. DOI: 10.1145/3359591.3359737. URL: https://doi.org/10.1145/3359591.3359737.

Sivaramakrishnan, KC, Gowtham Kaki, and Suresh Jagannathan (2015). "Declarative Programming over Eventually Consistent Data Stores". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: ACM, pp. 413–424. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737981. URL: http://doi.acm.org/10.1145/2737924.2737981.

Stonebraker, Michael, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland (2007). "The end of an Architectural Era: (It's Time for a Complete Rewrite)". In: *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. Vienna, Austria: VLDB Endowment, pp. 1150–1160. ISBN: 978-1-59593-649-3. URL: http://hstore.cs.brown.edu/papers/hstore-endofera.pdf.

Kaki, Gowtham, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan (Oct. 2019). "Mergeable Replicated Data Types". In: *Proc. ACM Program. Lang.* 3.OOPSLA. DOI: 10.1145/3360580. URL: https://doi.org/10.1145/3360580.

Kuraj, Ivan and Armando Solar-Lezama (2020). "Aspect-oriented language for reactive distributed applications at the edge". In: *Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking*, pp. 67–72.

Kuraj, Ivan, Armando Solar-Lezama, and Nadia Polikarpova (2022). "Optimizing consistency for partially replicated data stores". In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 457–458.

Kuraj, Ivan, John Feser, Nadia Polikarpova, and Armando Solar-Lezama (Nov. 2023). "PeepCo: Batch-Based Consistency Optimization". In: *PLDI '24, in submission.*

Alur, Rajeev, Emery Berger, Ann W Drobnis, Limor Fix, Kevin Fu, Gregory D Hager, Daniel Lopresti, Klara Nahrstedt, Elizabeth Mynatt, Shwetak Patel, et al. (2016). "Systems computing challenges in the Internet of Things". In: *arXiv preprint arXiv:1604.02980.*

Kaki, Gowtham, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan (Oct. 2018). "Safe Replication through Bounded Concurrency Verification". In: *Proc. ACM Program. Lang.* 2.OOPSLA. DOI: 10.1145/3276534. URL: https://doi.org/10.1145/3276534.

Houshmand, Farzin and Mohsen Lesani (2019). "Hamsaz: replication coordination analysis and synthesis". In: *PACMPL* 3.POPL, 74:1–74:32. DOI: 10.1145/3290387. URL: https://doi.org/10.1145/3290387.

Belaramani, Nalini Moti, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng (2006). "PRACTI Replication." In: *NSDI*. Vol. 6, pp. 5–5.

Wilcox, James R., Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson (2015). "Verdi: A Framework for Implementing and Formally Verifying Distributed Systems". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, pp. 357–368. ISBN: 9781450334686. DOI: 10.1145/2737924.2737958. URL: https://doi.org/10.1145/2737924.2737958.

Gotsman, Alexey, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, Marc Shapiro, Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro (2016). "'Cause i'm strong enough: reasoning about consistency choices in distributed systems". In: *POPL*.

Li, Cheng, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis (2014). "Automating the Choice of Consistency Levels in Replicated Systems". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, pp. 281–292. ISBN: 9781931971102.

Gray, Jim and Leslie Lamport (2006). "Consensus on transaction commit". In: *ACM Transactions on Database Systems (TODS)* 31.1, pp. 133–160.

Thomson, Alexander, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi (2012). "Calvin: fast distributed transactions for partitioned database systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 1–12.

Lamport, Leslie (1978a). "The implementation of reliable distributed multiprocess systems". In: *Computer Networks (1976)* 2.2, pp. 95–114. ISSN: 0376-5075. DOI: https://doi.org/10.1016/0376-5075(78)90045-4. URL: https://www.sciencedirect.com/science/article/pii/0376507578900454.

— (1977-03). "Proving the Correctness of Multiprocess Programs". In: *IEEE transactions on software engineering* SE-3.2, pp. 125–143. ISSN: 0098-5589.

Lamport, Leslie (1978b). “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM.*

Fidge, Colin J (1987). *Timestamps in message-passing systems that preserve the partial ordering.* Australian National University. Department of Computer Science.

Milano, Mae and Andrew C. Myers (2018). “MixT: a language for mixing consistency in geodistributed transactions”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2018. Philadelphia, PA, USA: Association for Computing Machinery, pp. 226–241. ISBN: 9781450356985. DOI: 10.1145/3192366.3192375. URL: https://doi.org/10.1145/3192366.3192375.

Pnueli, Amir (1981). “The temporal semantics of concurrent programs”. In: *Theoretical computer science* 13.1, pp. 45–60.

Saltzer, J. H., D. P. Reed, and D. D. Clark (Nov. 1984). “End-to-end Arguments in System Design”. In: *ACM Trans. Comput. Syst.* 2.4, pp. 277–288. ISSN: 0734-2071. DOI: 10.1145/357401.357402. URL: http://doi.acm.org/10.1145/357401.357402.

Baker, Jason, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh (2011). “Megastore: Providing Scalable, Highly Available Storage for Interactive Services”. In: *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pp. 223–234. URL: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf.

Akka – actor toolkit and runtime, http://akka.io/ (2023). URL: http://akka.io/.

Kuraj, Ivan and Armando Solar-Lezama (2017). “Leveraging sequential computation for programming efficient and reliable distributed systems”. In: *2nd Summit on Advances in Programming Languages (SNAPL 2017).* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Kuraj, Ivan and Daniel Jackson (2016). “Exploring the role of sequential computation in distributed systems: motivating a programming paradigm shift”. In: *Onward!*

Burckhardt, Sebastian, Alexey Gotsman, Hongseok Yang, and Marek Zawirski (2014). “Replicated Data Types: Specification, Verification, Optimality”. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’14. San Diego, California, USA: Association for Computing Machinery, pp. 271–284. ISBN: 9781450325448. DOI: 10.1145/2535838.2535848. URL: https://doi.org/10.1145/2535838.2535848.

Shapiro, Marc, Nuno Preguiça, Carlos Baquero, and Marek Zawirski (2011a). “Conflict-Free Replicated Data Types”. In: *Stabilization, Safety, and Security of Distributed Systems.* Ed. by Xavier Défago, Franck Petit, and Vincent Villain. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 386–400. ISBN: 978-3-642-24550-3.

Harding, Rachael, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker (2017). “An evaluation of distributed concurrency control”. In: *Proceedings of the VLDB Endowment* 10.5, pp. 553–564.

Abadi, Daniel (2012). “Consistency tradeoffs in modern distributed database system design: CAP is only part of the story”. In: *Computer* 45.2, pp. 37–42.

Herlihy, Maurice P. and Jeannette M. Wing (1990). “Linearizability: a correctness condition for concurrent objects”. In: *ACM Transactions on Programming Languages and Systems*. ISSN: 01640925.

Shapiro, Marc, Nuno Preguiça, Carlos Baquero, and Marek Zawirski (2011b). “A comprehensive study of convergent and commutative replicated data types”. In.

Fischer, Michael J, Nancy A Lynch, and Michael S Paterson (1985). “Impossibility of distributed consensus with one faulty process”. In: *Journal of the ACM (JACM)* 32.2, pp. 374–382.

Gilbert, Seth and Nancy Lynch (June 2002). “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”. In: *SIGACT News* 33.2, pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

— (2012). “Perspectives on the CAP Theorem”. In: *Computer*. ISSN: 0018-9162.

Viotti, Paolo and Marko Vukolić (June 2016). “Consistency in Non-Transactional Distributed Storage Systems”. In: *ACM Comput. Surv.* 49.1. ISSN: 0360-0300. DOI: 10.1145/2926965. URL: https://doi.org/10.1145/2926965.

Cachin, Christian, Rachid Guerraoui, and Luís Rodrigues (2011). *Introduction to reliable and secure distributed programming*. Springer Science & Business Media.

Flanagan, Cormac and Patrice Godefroid (2005). “Dynamic Partial-Order Reduction for Model Checking Software”. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’05. Long Beach, California, USA: Association for Computing Machinery, pp. 110–121. ISBN: 158113830X. DOI: 10.1145/1040305.1040315. URL: https://doi.org/10.1145/1040305.1040315.

Zhang, Yang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li (2013). “Transaction chains: achieving serializability with low latency in geo-distributed storage systems”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 276–291.

Breitbart, Yuri, Hector Garcia-Molina, and Avi Silberschatz (2010). “Overview of multidatabase transaction management”. In: *CASCON First Decade High Impact Papers*, pp. 93–126.

Berenson, Hal, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil (1995). "A critique of ANSI SQL isolation levels". In: *ACM SIGMOD Record* 24.2, pp. 1–10.

Kneuss, Etienne, Ivan Kuraj, Viktor Kuncak, and Philippe Suter (2013). "Synthesis modulo recursive functions". In: *OOPSLA*.

De Moura, Leonardo and Nikolaj Bjørner (2008). "Z3: An efficient SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, pp. 337–340.

Emerson, E Allen and A Prasad Sistla (1996). "Symmetry and model checking". In: *Formal methods in system design* 9.1-2, pp. 105–131.

TPC, Transaction Processing Performance Council (2010a). *Tpc benchmark™ E*.

Cooper, Brian F., Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears (2010). "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152. URL: https://doi.org/10.1145/1807128.1807152.

Shasha, Dennis, Francois Llirbat, Eric Simon, and Patrick Valduriez (Sept. 1995). "Transaction Chopping: Algorithms and Performance Studies". In: *ACM Trans. Database Syst.* 20.3, pp. 325–363. ISSN: 0362-5915. DOI: 10.1145/211414.211427. URL: http://doi.acm.org/10.1145/211414.211427.

GridGain data store benchmark (n.d.). URL: https://github.com/gridgain/yardstick-gridgain.

Shasha, Dennis and Marc Snir (Apr. 1988). "Efficient and Correct Execution of Parallel Programs That Share Memory". In: *ACM Trans. Program. Lang. Syst.* 10.2, pp. 282–312. ISSN: 0164-0925. DOI: 10.1145/42190.42277. URL: http://doi.acm.org/10.1145/42190.42277.

Kallman, Robert et al. (2008). "H-Store: a High-Performance, Distributed Main Memory Transaction Processing System". In: *Proc. VLDB Endow.* 1.2, pp. 1496–1499. ISSN: 2150-8097. DOI: http://doi.acm.org/10.1145/1454159.1454211. URL: http://hstore.cs.brown.edu/papers/hstore-demo.pdf.

Apache Ignite allocation modes (2020). URL: https://apacheignite.readme.io/docs/cache-modes.

Cassandra replication factor (2020). URL: https://docs.datastax.com/en/cql/3.3/cql/cql_using/useUpdateKeyspaceRF.html.

Alvaro, Peter, Neil Conway, and Joseph M. Hellerstein (2012). “Distributed Programming and Consistency: Principles and Practice”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC ’12. San Jose, California: Association for Computing Machinery. ISBN: 9781450317610. DOI: 10.1145/2391229.2391256. URL: https://doi.org/10.1145/2391229.2391256.

Alur, Rajeev, Emery D. Berger, et al. (2016). “Systems Computing Challenges in the Internet of Things”. In: *CoRR* abs/1604.02980. arXiv: 1604.02980. URL: http://arxiv.org/abs/1604.02980.

Solar-Lezama, Armando, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu (2005). “Programming by sketching for bit-streaming programs”. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pp. 281–294. DOI: 10.1145/1065010.1065045. URL: https://doi.org/10.1145/1065010.1065045.

Alford, Mack W, Jean-Pierre Ansart, Günter Hommel, Leslie Lamport, Barbara Liskov, Geoff P Mullery, and Fred B Schneider (1985). *Distributed systems: methods and tools for specification. An advanced course*. Springer-Verlag.

Alpern, Bowen and Fred B Schneider (1985). “Defining liveness”. In: *Information processing letters* 21.4, pp. 181–185.

Sarkar, Vivek (1998). “Analysis and optimization of explicity parallel programs using the parallel program graph representation”. In: *Languages and Compilers for Parallel Computing*. Ed. by Zhiyuan Li, Pen-Chung Yew, Siddharta Chatterjee, Chua-Huang Huang, P. Sadayappan, and David Sehr. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 94–113. ISBN: 978-3-540-69788-6.

Birrell, Andrew D and Bruce Jay Nelson (1984a). “Implementing remote procedure calls”. In: *ACM Transactions on Computer Systems (TOCS)* 2.1, pp. 39–59.

Epstein, Jeff, Andrew P Black, and Simon Peyton-Jones (2011). “Towards Haskell in the cloud”. In: *Proceedings of the 4th ACM symposium on Haskell*, pp. 118–129.

Apache Ignite allocation modes (n.d.). URL: https://github.com/gridgain/yardstick.

Slither.io game (2023). URL: http://slither.io/.

Phaser - A fast, fun and free open source HTML5 game framework (2023). URL: https://phaser.io/.

Scala.js (2023). URL: https://www.scala-js.org/.

I Shot You First: Networking the Gameplay of HALO: REACH (2023). URL: https://www.gdcvault.com/play/1014345/I-Shot-You-First-Networking.

Jackson, Daniel, Ian Schechter, and Hya Shlyahter (2000). "Alcoa: the Alloy constraint analyzer". In: *Proceedings of the 22nd international conference on Software engineering*, pp. 730–733.

TPC, Transaction Processing Performance Council (2010b). *TPC-C specification.*

Kuraj, Ivan, Viktor Kuncak, and Daniel Jackson (2015). "Programming with enumerable sets of structures". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 37–56.

Kuraj, Ivan and Viktor Kuncak (2014). "Scife: Scala framework for efficient enumeration of data structures with invariants". In: *Proceedings of the Fifth Annual Scala Workshop*, pp. 45–49.

Terry, Douglas B., Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch (1994). "Session Guarantees for Weakly Consistent Replicated Data". In: *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*. PDIS '94. USA: IEEE Computer Society, pp. 140–149. ISBN: 0818664002.

Reynders, Bob, Dominique Devriese, and F Piessens (2014). "Multi-tier Functional Reactive Programming for the Web". In: *Onward!*

Bykov, Sergey, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin (2011). "Orleans: Cloud Computing for Everyone". In: *SoCC*.

Reynolds, John C. (1972). "Definitional Interpreters for Higher-order Programming Languages". In: *Proceedings of the ACM Annual Conference - Volume 2*. ACM '72. Boston, Massachusetts, USA: ACM, pp. 717–740. DOI: 10.1145/800194.805852. URL: http://doi.acm.org/10.1145/800194.805852.

Akka persistence layer (2023). URL: https://doc.akka.io/docs/akka/2.3.6/scala/persistence.html.

Lamport, Leslie (2001). "Paxos made simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58.

Eclipse Hono IoT framework (2023). URL: https://www.eclipse.org/hono/.

Redis - in-memory data structure store (n.d.). URL: http://redis.io/.

Agha, Gul A and Wooyoung Kim (1999). "Actors: A unifying model for parallel and distributed computing". In: *Journal of systems architecture* 45.15, pp. 1263–1277.

Christensen, Aske Simon, Anders Møller, and Michael I. Schwartzbach (2003). "Extending Java for high-level Web service construction". In: *TOPLAS*. ISSN: 01640925.

Birrell, Andrew D. and Bruce Jay Nelson (1984b). "Implementing remote procedure calls". In: *ACM Transactions on Computer Systems*. ISSN: 07342071.

Haridi, Seif, Peter Van Roy, Per Brand, and Christian Schulte (1998). "Programming languages for distributed applications". In: *New Generation Computing*. ISSN: 1882-7055.

Zakhour, George, Pascal Weisenburger, and Guido Salvaneschi (June 2023). “Type-Checking CRDT Convergence”. In: *Proc. ACM Program. Lang.* 7.PLDI. DOI: 10.1145/3591276. URL: https://doi.org/10.1145/3591276.

Alvaro, Peter, Neil Conway, Joseph M Hellerstein, and David Maier (2017). “Blazes: Coordination Analysis and Placement for Distributed Programs”. In: *ACM Transactions on Database Systems (TODS)* 42.4, pp. 1–31.

Liu, Yiyun, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou (2020). “Verifying replicated data types with typeclass refinements in Liquid Haskell”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA, pp. 1–30.

Balegas, Valter, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro (2015a). “Putting Consistency Back into Eventual Consistency”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: Association for Computing Machinery. ISBN: 9781450332385. DOI: 10.1145/2741948.2741972. URL: https://doi.org/10.1145/2741948.2741972.

Stonebraker, Michael and Uĝur Çetintemel (2018). “"One size fits all" an idea whose time has come and gone”. In: *Making databases work: the pragmatic wisdom of Michael Stonebraker*, pp. 441–462.

Sivasubramanian, Swaminathan (2012). “Amazon DynamoDB: a seamlessly scalable non-relational database service”. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 729–730.

Deng, Dong, Albert Kim, Samuel Madden, and Michael Stonebraker (2017). “SilkMoth: An Efficient Method for Finding Related Sets with Maximum Matching Constraints”. In: *Proc. VLDB Endow.* 10.10, pp. 1082–1093. DOI: 10.14778/3115404.3115413. URL: http://www.vldb.org/pvldb/vol10/p1082-deng.pdf.

Li, Xiao, Farzin Houshmand, and Mohsen Lesani (2020). “Hampa: Solver-Aided Recency-Aware Replication”. In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, pp. 324–349. ISBN: 978-3-030-53288-8.

De Porre, Kevin, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix (Oct. 2021). “ECROs: Building Global Scale Systems from Sequential Code”. In: *Proc. ACM Program. Lang.* 5.OOPSLA. DOI: 10.1145/3485484. URL: https://doi.org/10.1145/3485484.

Rahmani, Kia, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan (2021). “Repairing serializability bugs in distributed database programs via automated schema refactoring”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 32–47.

Li, Cheng, Daniel Charles Ferreira Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Seromenho Miragaia Rodrigues (2012). "Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary". In: *USENIX Symposium on Operating Systems Design and Implementation.*

Li, Cheng, Nuno Preguiça, and Rodrigo Rodrigues (2018). "Fine-grained consistency for geo-replicated systems". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 359–372.

Roy, Sudip, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke (2015). "The homeostasis protocol: Avoiding transaction coordination through program analysis". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1311–1326.

Najafzadeh, Mahsa, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro (2016). "The CISE tool: proving weakly-consistent applications correct". In: *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, pp. 1–3.

Padon, Oded, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham (2017). "Reducing liveness to safety in first-order logic". In: *Proceedings of the ACM on Programming Languages* 2.POPL, pp. 1–33.

Balegas, Valter, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro (2015b). "Towards fast invariant preservation in geo-replicated systems". In: *ACM SIGOPS Operating Systems Review* 49.1, pp. 121–125.

Balegas, Valter, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça (Dec. 2018). "IPA: Invariant-Preserving Applications for Weakly Consistent Replicated Databases". In: *Proc. VLDB Endow.* 12.4, pp. 404–418. ISSN: 2150-8097. DOI: 10.14778/3297753.3297760. URL: https://doi.org/10.14778/3297753.3297760.

Lewchenko, Nicholas V, Arjun Radhakrishna, Akash Gaonkar, and Pavol Černý (2019). "Sequential programming for replicated data stores". In: *Proceedings of the ACM on Programming Languages* 3.ICFP, pp. 1–28.

Nagar, Kartik and Suresh Jagannathan (2019). "Automated parameterized verification of CRDTs". In: *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31*. Springer, pp. 459–477.

Holt, Brandon, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze (2016). "Disciplined inconsistency with consistency types". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pp. 279–293.

De Porre, Kevin, Florian Myter, Christophe Scholliers, and Elisa Gonzalez Boix (2020). "CScript: A distributed programming language for building mixed-consistency applications". In: *Journal of Parallel and Distributed Computing* 144, pp. 109–123.

Köhler, Mirko, Nafise Eskandani, Pascal Weisenburger, Alessandro Margara, and Guido Salvaneschi (2020). "Rethinking safe consistency in distributed object-oriented programming". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA, pp. 1–30.

Xie, Chao, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan (2014). "Salt: Combining {ACID} and {BASE} in a distributed database". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 495–509.

Balakrishnan, Mahesh, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck (2013). "Tango: Distributed data structures over a shared log". In: *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pp. 325–340.

Kaki, Gowtham, Prasanth Prahladan, and Nicholas V Lewchenko (2022). "RunTime-assisted convergence in replicated data types". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 364–378.

Laddad, Shadaj, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein (Oct. 2022). "Katara: Synthesizing CRDTs with Verified Lifting". In: *Proc. ACM Program. Lang.* 6.OOPSLA2. DOI: 10.1145/3563336. URL: https://doi.org/10.1145/3563336.

Lu, Shiyong, Arthur Bernstein, and Philip Lewis (2004). "Correct execution of transactions at different isolation levels". In: *IEEE Transactions on Knowledge and Data Engineering* 16.9, pp. 1070–1081.

Fekete, Alan, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha (2005). "Making snapshot isolation serializable". In: *ACM Transactions on Database Systems (TODS)* 30.2, pp. 492–528.

Kaki, Gowtham, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan (2018). "Alone together: compositional reasoning and inference for weak isolation". In: *Proc. ACM Program. Lang.* 2.POPL, 27:1–27:34. DOI: 10.1145/3158115. URL: https://doi.org/10.1145/3158115.

Cerone, Andrea, Alexey Gotsman, and Hongseok Yang (2015). "Transaction chopping for parallel snapshot isolation". In: *International Symposium on Distributed Computing*. Springer, pp. 388–404.

Bernstein, Arthur J, David S Gerstl, and Philip M Lewis (1999). “Concurrency control for step-decomposed transactions”. In: *Information Systems* 24.8, pp. 673–698.

Cheung, Alvin, Owen Arden, Samuel Madden, and Andrew C Myers (2012). “Automatic Partitioning of Database Applications”. In: *Proceedings of the VLDB Endowment* 5.11.

Zdancewic, Steve and Andrew C Myers (2003). “Observational determinism for concurrent program security”. In: *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.* IEEE, pp. 29–43.

Mu, Shuai, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li (2014). “Extracting more concurrency from distributed transactions”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 479–494.

Xie, Chao, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang (2015). “High-performance ACID via modular concurrency control”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 279–294.

Yu, Haifeng and Amin Vahdat (2000). “Design and Evaluation of a Continuous Consistency Model for Replicated Services”. In: *4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000*. Ed. by Michael B. Jones and M. Frans Kaashoek. USENIX Association, pp. 305–318. URL: http://dl.acm.org/citation.cfm?id=1251250.

Bailis, Peter, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica (2012). “Probabilistically Bounded Staleness for Practical Partial Quorums”. In: *Proc. VLDB Endow.* 5.8, pp. 776–787. DOI: 10.14778/2212351.2212359. URL: http://vldb.org/pvldb/vol5/p776%5C_peterbailis%5C_vldb2012.pdf.

Guerraoui, Rachid, Matej Pavlovic, and Dragos-Adrian Seredinschi (2016). “Incremental Consistency Guarantees for Replicated Objects”. In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, pp. 169–184. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/guerraoui.

Kraska, Tim, Martin Hentschel, Gustavo Alonso, and Donald Kossmann (2009). “Consistency Rationing in the Cloud: Pay only when it matters”. In: *Proc. VLDB Endow.* 2.1, pp. 253–264. DOI: 10.14778/1687627.1687657. URL: http://www.vldb.org/pvldb/vol2/vldb09-759.pdf.

Terry, Douglas B., Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh (2013). “Consistency-based service level agreements for cloud storage”. In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP ’13, Farmington, PA, USA, November 3-6, 2013*. Ed. by

Michael Kaminsky and Mike Dahlin. ACM, pp. 309–324. DOI: 10.1145/2517349.2522731. URL: https://doi.org/10.1145/2517349.2522731.

Wickerson, John, Mark Batty, Tyler Sorensen, and George A Constantinides (2017). “Automatically comparing memory consistency models”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 190–204.

Solar-Lezama, Armando, Christopher Grant Jones, and Rastislav Bodík (2008). “Sketching concurrent data structures”. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pp. 136–148. DOI: 10.1145/1375581.1375599. URL: https://doi.org/10.1145/1375581.1375599.

Vechev, Martin T. and Eran Yahav (2008). “Deriving linearizable fine-grained concurrent objects”. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pp. 125–135. DOI: 10.1145/1375581.1375598. URL: https://doi.org/10.1145/1375581.1375598.

Vechev, Martin T., Eran Yahav, and Greta Yorsh (2009). “Inferring Synchronization under Limited Observability”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pp. 139–154. DOI: 10.1007/978-3-642-00768-2_13. URL: https://doi.org/10.1007/978-3-642-00768-2%5C_13.

Kuperstein, Michael, Martin T. Vechev, and Eran Yahav (2010). “Automatic inference of memory fences”. In: *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pp. 111–119. URL: http://ieeexplore.ieee.org/document/5770939/.

Udupa, Abhishek, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur (2013). “TRANSIT: Specifying Protocols with Concolic Snippets”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’13. Seattle, Washington, USA: Association for Computing Machinery, pp. 287–296. ISBN: 9781450320146. DOI: 10.1145/2491956.2462174. URL: https://doi.org/10.1145/2491956.2462174.

Cooper, Ezra, Sam Lindley, Philip Wadler, and Jeremy Yallop (2007). “Links: Web Programming Without Tiers”. In: *FMCO*.

Serrano, Manuel and Gérard Berry (2012). “Multitier programming in Hop”. In: *Communications of the ACM*. ISSN: 00010782.

Agha, Gul (1986). “Actors: a Model of Concurrent Computation in Distributed Systems”. In: *MIT Press*.

Hewitt, Carl, Peter Bishop, and Richard Steiger (1973). "A universal modular ACTOR formalism for artificial intelligence". In: *IJCAI*.

Haller, Philipp and Martin Odersky (2009). "Scala Actors: Unifying thread-based and event-based programming". In: *Theoretical Computer Science*. ISSN: 03043975.

Prokopec, Aleksandar and Martin Odersky (2015). "Isolates, Channels, and Event Streams for Composable Distributed Programming". In: *Onward!*

Charles, Philippe et al. (2005). "X10: an object-oriented approach to non-uniform cluster computing". In: *OOPSLA*.

De Wael, Mattias, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter (2015). "Partitioned Global Address Space Languages". In: *ACM Computing Surveys*. ISSN: 03600300.

Drechsler, Joscha, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini (2014). "Distributed REScala: An Update Algorithm for Distributed Reactive Programming". In: *OOPSLA*.

Baldini, Ioana, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter (2016). "Cloud-native, event-based programming for mobile applications". In: *MOBILESoft*.

Dabek, Frank, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris (2002). "Event-driven programming for robust software". In: *ACM SIGOPS European workshop: beyond the PC*.

JMacroRPC - reactive client/server web programming (n.d.). URL: http://hackage.haskell.org/package/jmacro-rpc.

Meteor - Pure JavaScript web framework (n.d.). URL: http://meteor.com.

Czaplicki, Evan and Stephen Chong (2013). "Asynchronous Functional Reactive Programming for GUIs". In: *PLDI*.

Meiklejohn, Christopher and Peter Van Roy (2015). "Lasp: A Language for Distributed, Coordination-free Programming". In: *PPDP*.

Kiciman, Emre, Benjamin Livshits, Madanlal Musuvathi, and Kevin C. Webb (2010). "Fluxo: a system for internet service programming by non-expert developers". In: *SoCC*.

Briot, Jean-Pierre, Rachid Guerraoui, and Klaus-Peter Lohr (1998). "Concurrency and Distribution in Object-oriented Programming". In: *ACM Computing Surveys*. ISSN: 0360-0300.

Bal, Henri E., Jennifer G. Steiner, and Andrew S. Tanenbaum (1989). "Programming languages for distributed computing systems". In: *ACM Computing Surveys*. ISSN: 03600300.

Tanenbaum, Andrew Stuart and Robbert van Renesse (1987). *A critique of the remote procedure call paradigm*. Tech. rep. Vrije Universiteit.

Liu, Yanhong A., Scott D. Stoller, Bo Lin, Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski (2012). “From clarity to efficiency for distributed algorithms”. In: *OOPSLA*.

Jayaram, K. R. and Patrick Eugster (2011). “Program analysis for event-based distributed systems”. In: *DEBS*.

Marino, Daniel, Todd Millstein, Madanlal Musuvathi, Satish Narayanasamy, and Abhayendra Singh (2015). “The Silently Shifting Semicolon”. In: *SNAPL*. ISSN: 1868-8969.