Combinatorial Constructions for Effective Testing

Thesis approved by
the Department of Computer Science
Technische Universität Kaiserslautern
for the award of the Doctoral Degree
Doctor of Engineering (Dr.-Ing.)

to

Filip Nikšić

Date of Defense:	May 3, 2019
Dean:	Stefan Deßloch
Reviewer:	Rupak Majumdar
Reviewer:	Madan Musuvathi
Reviewer:	Stefan Kiefer

D 386

Summary

Large-scale distributed systems consist of a number of components, take a number of parameter values as input, and behave differently based on a number of non-deterministic events. All these features—components, parameter values, and events—interact in complicated ways, and unanticipated interactions may lead to bugs. Empirically, many bugs in these systems are caused by interactions of only a *small* number of features. In certain cases, it may be possible to test all interactions of k features for a small constant k by executing a family of tests that is exponentially or even doubly-exponentially smaller than the family of all tests. Thus, in such cases we can effectively uncover all bugs that require up to k-wise interactions of features.

In this thesis we study two occurrences of this phenomenon. First, many bugs in distributed systems are caused by *network partition faults*. In most cases these bugs occur due to two or three key nodes, such as leaders or replicas, not being able to communicate, or because the leading node finds itself in a block of the partition without quorum. Second, bugs may occur due to unexpected *schedules* (interleavings) of concurrent events—concurrent exchange of messages and concurrent access to shared resources. Again, many bugs depend only on the relative ordering of a small number of events. We call the smallest number of events whose ordering causes a bug the *depth* of the bug. We show that in both testing scenarios we can effectively uncover bugs involving small number of nodes or bugs of small depth by executing small families of tests.

We phrase both testing scenarios in terms of an abstract framework of tests, testing goals, and goal coverage. Sets of tests that cover all testing goals are called *covering families*. We give a general construction that shows that whenever a random test covers a fixed goal with sufficiently high probability, a small randomly chosen set of tests is a covering family with high probability. We then introduce concrete coverage notions relating to network partition faults and bugs of small depth. In case of network partition faults, we show that for the introduced coverage notions we can find a lower bound on the probability that a random test covers a given goal. Our general construction then yields a randomized testing procedure that achieves full coverage—and hence, find bugs—quickly.

In case of coverage notions related to bugs of small depth, if the events in the program form a non-trivial partial order, our general construction may give a suboptimal bound. Thus, we study other ways of constructing covering families. We show that if the events in a concurrent program are partially ordered as a tree, we can explicitly construct a covering family of small size: for balanced trees, our construction is polylogarithmic in the number of events. For the case when the partial order of events does not have a "nice" structure, and the events and their relation to previous events are revealed while the program is running, we give an *online* construction of covering families. Based on the construction, we develop a randomized scheduler called PCTCP that uniformly samples schedules from a covering family and has a rigorous guarantee of finding bugs of small depth. We experiment with an implementation of PCTCP on two real-world distributed systems—Zookeeper and Cassandra—and show that it can effectively find bugs.

Zusammenfassung

Große verteilte Systeme bestehen aus vielen Komponenten, hängen von vielen Parametern ab und leiten ihr Verhalten von vielen nicht-deterministischer Ereignissen ab. Alle diese Features—Komponenten, Parameter und Ereignisse—interagieren auf komplizierte Weise, und unerwartete Interaktionen können zu Fehlern führen. Erfahrungsgemäß werden viele Fehler in diesen Systemen durch eine Interaktion einer *kleinen* Anzahl von Features verursacht. In bestimmten Fällen ist es möglich, alle Interaktionen von k Features für eine kleine Konstante k zu testen, indem man eine Familie von Tests ausführt, die exponentiell oder gar doppelt-exponentiell kleiner als die Familie aller Tests ist. Man kann in diesen Fällen also alle Fehler, die durch die Interaktion von bis zu k Features auftreten, effizient aufdecken.

In dieser Dissertation untersuchen wir zwei Varianten dieses Phänomens. Zum einen werden viele Fehler in verteilten Systemen durch *network partition faults*[1] verursacht. In den meisten Fällen treten diese Fehler auf, wenn zwei oder drei wesentliche Knoten, beispielsweise Leader oder Replica, nicht kommunizieren können, oder wenn der Leader-Knoten sich in einem Block der Partition wiederfindet, der nicht mit einem Quorum der Knoten kommunizieren kann. Zum anderen können Fehler auftreten, wenn unerwartete *schedules* (interleavings) nebenläufiger Ereignisse auftreten—nebenläufiger Nachrichten-Austausch oder nebenläufiger Zugriff auf geteilte Ressourcen. Auch hier hängen viele Fehler von der relativen Ordnung einiger weniger Ereignisse ab. Wir nennen die kleinste Anzahl von Ereignissen, deren (Um)ordnung einen Fehler verursacht, die *Tiefe* des Fehlers. Wir zeigen, dass wir in beiden Test-Szenarien effektiv Fehler, die eine kleine Zahl von Knoten betreffen oder geringe Tiefe haben, aufdecken können, indem wir eine kleine Familie von Tests ausführen.

Wir beschreiben beide Szenarien mit Hilfe eines abstrakten Test-Frameworks bestehend aus Tests, Testzielen und Ziel-Überdeckung. Eine Menge von Tests, die alle Testziele überdeckt, nennen wir *covering family* (überdeckende Familie). Wir zeigen eine allgemeine Konstruktion, die beweist, dass es genügt, dass ein zufälliger Test ein festes Ziel mit hinreichend hoher Wahrscheinlichkeit überdeckt, dass mit hoher Wahrscheinlichkeit eine kleine, zufällig gewählten Menge von Tests eine covering family ist. Wir führen dann konkrete Überdeckungskriterien für network partition faults und Fehler kleiner Tiefe ein. Im Falle der network partition faults geben wir eine untere Schranke für die Wahrscheinlichkeit, dass ein zufälliger Test ein bestimmtes Ziel überdeckt, an. Unsere allgemeine Konstruktion ergibt dann ein randomisiertes Testverfahren, dass volle Überdeckung—und somit Fehlerfindung—schnell erreicht.

Im Falle der Überdeckung von Fehler kleiner Tiefe, wenn die Ereignisse des Programms eine nicht-triviale Halbordnung bilden, ergibt unsere Konstruktion eine suboptimale Schranke. Aus diesem Grund betrachten wir andere Wege, um covering families zu konstruieren. Wir zeigen, wenn die Ereignisse eines nebenläufigen Programms in baumartigen geordnet werden können, können wir explizit eine kleine covering family konstruieren: für balancierte Bäume ist unsere Konstruktion polylogarithmisch in der Ereignis-Zahl. Wenn die partielle Ordnung der Ereignisse keine "schöne" Struktur hat, und die Ereignisse sowie ihre Ordnung erst während der Ausführung bekannt werden, geben wir ein *online*-Konstruktion von covering families an. Aufbauend auf dieser Konstruktion entwickeln wir einen randomisierten Scheduler namens PCTCP, der gleichverteilte Stichproben von Schedules aus einer covering family wählt und eine starke Garantie, Fehler kleiner Tiefe zu finden, bietet. Wir experimentieren mit einer Implementierung von PCTCP auf zwei verbreiteten verteilten Systemen—Zookeeper und Cassandra—und zeigen, dass wir effektiv Fehler finden können.

[1] Netzwerkausfälle, bei denen ein Teil des Systems nicht mehr mit dem Rest kommunizieren kann

Acknowledgments

I would like to thank my advisor Rupak Majumdar for guidance and support during my PhD, for all the wise advice and the mathematical puzzles we would occasionally obsess over for days.

I would also like to thank Ruzica Piskac, my initial advisor before she left to Yale. Were it not for her, I would not have come to MPI-SWS in the first place.

I am forever grateful to Burcu Kulahcioglu Ozkan and Murat Ozkan, Ivan Gavran and Mia Majtan, Simin Oraee, Marko Doko, Kaushik Mallik, Soham and Nivedita Chakraborty, and Manuel Gomez Rodriguez for helping me through difficult times.

I wish to thank Beta Ziliani, Susanne van den Elsen, Natacha Crooks, and Johannes Kloos for being great office mates, and many individuals, too numerous to list, who made MPI-SWS such a great place. Thanks for all the scientific discussions, all the board games, quizzes, puzzles, movie nights, and all the fun we had together.

Finally, I am grateful to my family for providing constant support during my PhD.

Contents

Chapter 1

Introduction

Large-scale distributed systems are difficult to build and test. On the one hand, these systems feature nondeterminism in the interleaving of concurrent events—concurrent exchange of messages and concurrent access to shared resources. The number of possible interleavings of events is generally exponential in the number of events and thus difficult to reason about, and an unexpected interleaving may easily cause the system to fail. On the other hand, these systems must also account for partial failures, where components or communication can fail along the way and produce incomplete results. Fault-tolerant components are difficult to design and reason about, and usually require intricate protocols to ensure correct behavior for the global system. Even if individual components are correct, their composition may require further protocols to operate correctly under failure conditions. Thus, distributed systems are some of the most complex pieces of software. Given the critical role these systems play today, gaining high assurance of their behavior under failure conditions is a critical challenge (Lopes 2016; McCaffrey 2015).

The approaches to assuring correct behavior of distributed systems usually go in one of the two extreme directions. One direction is characterized by strong guarantees of finding bugs or showing their absence: one can try to build a fully verified system "from scratch" (Hawblitzel et al. 2015; Lamport 1994; Wilcox et al. 2015), or systematically explore the space of behaviors of the system (Baier and Katoen 2008; Fisman, Kupferman, and Lustig 2008; Konnov, Veith, and Widder 2017; Leesatapornwongsa et al. 2014a; Yang et al. 2009) under systematically injected faults (Alvaro, Rosen, and Hellerstein 2015; Gunawi et al. 2011). The former approach ensures the absence of bugs by construction, while the latter approach guarantees all bugs to be found by exhausting the search space. Unfortunately, the strong guarantees come with high price: for the fully verified systems it is difficult to match the functionality and performance of existing deployments, and the systematic exploration, despite techniques like partial order reduction (Godefroid 1996; Flanagan and

Godefroid 2005; Abdulla et al. 2014), still faces exponentially many behaviors to explore; hence it scales poorly to real-world deployments.

The other direction is stress testing with random schedulers and randomly inserted faults (Apache Hadoop 2016; Claessen et al. 2009; Izrailevsky and Tseitlin 2011; Kingsbury 2013–2018). This approach is usually straightforward to implement and it scales well, even to large geo-replicated systems in production (Izrailevsky and Tseitlin 2011; Chaos Engineering 2018). However, the guarantees of finding bugs using this approach are usually weak or non-existent.

Interestingly, despite the weak guarantees, the random testing approaches work remarkably well. For example, Jepsen (Kingsbury 2013–2018) is a framework for black-box testing of distributed systems under *partition faults*—faults that prevent portions of a system to communicate with other portions. Jepsen provides an infrastructure to set up a number of processes and exercise a system with random operations as well as randomly introduced partition faults. Using Jepsen, Kingsbury (2013) found a remarkably large number of rather subtle problems in many production distributed systems.

The success of random testing in Jepsen and similar tools presents a conundrum. On the one hand, academic wisdom asserts that random testing will be completely ineffective in finding bugs in faulty systems other than by "extremely unlikely accident": the probability that a random execution stumbles across the "right" combination of circumstances—failures, recoveries, reads, and writes—is intuitively so small that any soundness guarantee for a given coverage goal, even probabilistic, would be minuscule (many systematic testing papers start by asserting that random testing does not or should not work). On the other hand, in practice, random testing finds bugs within a small number of tests.

As one of the contributions of this thesis, we provide a theoretical understanding for the empirical success of Jepsen in exposing subtle bugs. As evidenced by a series of online articles by Kyle Kingsbury about testing distributed systems (Kingsbury 2013–2018), most of the bugs occur due to two or three key nodes, such as leaders or replicas, not being able to communicate, or because the leading node finds itself in a block of the partition without quorum. We show that a bug of this sort involving a small, fixed number of nodes k, can be detected by simulating a random network partition, and the detection probability is not astronomically small, as might be expected, but it is bounded from below by a sufficiently "high" constant that depends only on k. Using this bound we show that the bug can be detected with overwhelming probability by randomly generating a set of network partitions whose size is logarithmic in the size of the system. In fact, this is what Jepsen implicitly does, and hence it finds the bug quickly. Thus, we have an a posteriori justification of Jepsen's effectiveness.

The empirical observation that many bugs found by Jepsen involve a small number of nodes can be seen as an instance of the "small-world" phenomenon originally observed in

the context of social networks (Milgram 1967). The phenomenon also manifests itself in bugs arising from unexpected interleavings of concurrent events. Each interleaving is called a *schedule*. Empirically, many bugs in concurrent systems are exposed by considering the sequencing of a small number of events—the "bug depth"—independent of the ordering of the rest of the events (Burckhardt et al. 2010; Lu et al. 2008; Qadeer and Rehof 2005; Leesatapornwongsa et al. 2016a). Thus, instead of exploring all possible schedules, one can fix a bug depth d and only explore a sufficient family of schedules that guarantees that for every choice of d events, every ordering of these events is covered. The families of schedules providing this guarantee are called *d-hitting families of schedules*; we study them extensively in the thesis. In particular, we show that in many cases there are explicit constructions of d-hitting families of size logarithmic, polylogarithmic, or polynomial in the number of events. Executing schedules from these hitting families leads to an effective way to uncover bugs of depth up to d.

1.1 Abstract Testing

Both bug-finding scenarios can be phrased as testing problems in an abstract testing framework which we define in Chapter 3. The framework consists of tests, testing goals, and goal coverage; in a concrete situation, these are chosen so that covering all testing goals guarantees discovery of any bugs that may exist in the system. A set of tests that covers all testing goals is called a *covering family*. A covering family that is exponentially smaller than the set of tests yields, at least in principle, an effective testing procedure, since it usually means we can effectively execute sufficiently many tests to cover all testing goals, and hence discover any bugs in the system.

Even without knowing the exact nature of tests, testing goals, and the notion of goal coverage, there is a way to derive a useful bound on the size of optimal covering families using a combinatorial technique known as the *probabilistic method* (Alon and Spencer 2004). In this technique, in order to show that a combinatorial object with certain properties exists, one argues that a randomly chosen object from a suitable probability space has the property with positive probability. Since the probability is positive, there must be at least one object—we do not know which one—with that property.

To see the connection to testing, assume we have a set of tests, a set of testing goals, and some notion of goal coverage. The combinatorial object whose existence we want to show using the probabilistic method is a covering family of initially unknown size N; we will choose a suitable N at the end. The outline of the argument is as follows. Fix a testing goal and suppose that a randomly chosen test covers the goal with probability at least $p > 0$. Then, a set of N independently chosen random tests does not cover this goal with probability at most $(1 - p)^N$, and it is not a covering family with probability at most

$m \cdot (1-p)^N$ (by the union bound), where m is the number of testing goals. By picking N to be $\Omega(p^{-1} \log m)$, this probability can be made less than 1, showing that a covering family of this size exists. Moreover, for a given $\epsilon > 0$, by picking N to be $\Omega(p^{-1}(\log m + \log \epsilon^{-1}))$, the probability can be made less than ϵ, showing that a family of N randomly chosen tests is a covering family with probability at least $1 - \epsilon$. Thus, by running all tests from the constructed family, we are guaranteed with high probability that we have run at least one test for each coverage goal.

1.2 Testing with Random Partitions

Most bugs in distributed systems found by Jepsen can be manifested by tests of the following nature (see Chapter 2 for examples). First, a small sequence of operations "sets up" the system in a special state. Then, a carefully chosen network partition separates the system into two or more blocks which cannot communicate among each other. Then, a further sequence of operations are performed, often in each of the different blocks of the partition. Finally, the partition is healed and a final set of operations are performed. This sequence (perform operations, introduce failures, perform further operations) may need to be repeated a few times at most. Depending on the nature of the network partition introduced, we introduce and study the following coverage notions.

k-Splitting. Consider a distributed application consisting of processes $a_1, \ldots, a_n$. We fix $k > 0$, and consider partitions of the network into k disjoint blocks. Intuitively, we want to test the application for all possible ways of splitting the processes into k different groups. That is, for every k processes, we would like to test what happens if these k processes cannot communicate with each other. This is formalized using the notion of *splitting coverage*. A k-partition *splits* processes $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ if these processes all end up in different blocks of the partition. The splitting coverage asks that for every choice of k processes there is a test that splits them. For example, our application could involve two different types of quorums, and for each quorum there could be a leader (cf. Chronos in Section 2.3). A bug may occur if the two leaders are unable to communicate. Since we a priori do not know which two processes are leaders, we want tests such that for every pair (a, b) of processes, there is a test where a network failure partitions the system into two blocks such that the processes a and b cannot communicate with each other. This is an example of splitting coverage for $k = 2$. For the Jepsen bugs, values $k = 2$ and $k = 3$ sufficed for all bugs that fall under the scope of splitting coverage.

(k, l)-Separation. Processes in distributed applications often have a role. For example, processes may be clients, or replicas of shared state, or replicas managing consensus protocol (e.g., Zookeeper instances). Thus, a natural requirement is to ensure certain subsets of roles stay together in a split and are separated from some other subset. This is formalized using the notion of *separating coverage*: for fixed $k, l > 0$ and every pair of collections of k and l roles, separating coverage requires that the collection of k roles is separated by a partition from the collection of l roles. For example, suppose our application is a sharded system, where each shard is replicated with a replication factor $f = 3$. A bug may occur if the leading replica is separated from the two following replicas. Again, since we a priori do not know which processes form the set of replicas, and which one among them is the leader, we want tests such that every choice of one and two processes are separated by a partition. This is an example of splitting coverage for $k = 1$ and $l = 2$. Again, for bugs reported by Jepsen, values of k and l up to 3 suffice for all bugs that fall under the scope of separating coverage.

Minority isolation. In addition to separating small subsets of processes, we often wish to impose cardinality constraints. For example, it is important to cover cases in which the current leader is in the block of a partition with fewer nodes in order to force a new leader election. To study this, we introduce the notion of *minority isolation*: for each process, this coverage notion requires that the process is in the smaller block of a bipartition.

Empirically, these coverage notions capture a large class of fault tolerance bugs found by Jepsen in distributed systems. In each case, for a fixed testing goal (a process, a set of k processes, or a pair of k and l processes) we show we can bound the probability that a random test (a bipartition or a k-partition) covers the goal (isolates, splits, or separates the processes). The bound depends on parameters k, l, but does not depend on n—the size of the system. This allows us to construct small families of tests that cover all goals with high probability by picking sufficiently many tests uniformly at random. Our results thus provide a theoretical justification for the effectiveness of random testing in this domain.

One caveat is the notion of a "bug." A famous result from distributed systems, called the CAP theorem (Brewer 2000; Gilbert and Lynch 2002), asserts that no distributed system can be simultaneously consistent (roughly, linearizable), available, and partition tolerant. The precise intension of CAP and guarantees provided by specific systems is a matter of considerable debate in the systems community (Brewer 2012). For example, under partitions, a distributed database may give up availability or consistency. However, in real systems, programmers navigate a rich space of tradeoffs with many relaxed notions of availability or consistency. What constitutes a correct test oracle is usually up to the programmers of the system to decide (see, e.g., Kingsbury (2013), and discussions on his

bug reports on Github). In the thesis, in general we do not specify the property used to determine whether an error has occurred: we only guarantee that all testing goals are met. It is up to the programmer to write appropriate test oracles to check for problems.

1.3 Testing with Hitting Families

The testing task involving schedules of concurrent events can also be phrased in general terms: given a fixed bug depth d, tests are schedules of events, testing goals are ordered sets of d events, and a schedule covers an ordered set S if it "hits" S, that is, the events appear in the schedule in the same relative order as in S. A covering family in this setting is called a *d-hitting family of schedules*.

If the program consists of n concurrent events (that is, the events form an antichain), a schedule is simply a permutation of these events. An ordered set of d events appears in a random permutation in the same relative order with probability $1/d!$, and there are $\binom{n}{d}d! \leq n^d$ ordered sets of d events. Hence our general construction shows existence of a d-hitting family of size $d \cdot d! \log n$. While being optimal for $d \geq 3$, this bound is suboptimal for $d = 2$: all bugs involving two events can be exposed with just two schedules! Likewise, if the events in the program form a non-trivial partial order, our general construction may give a suboptimal bound. Thus, we explore other ways of constructing hitting families.

We show that if the events in a concurrent program are partially ordered as a tree of height h, we can explicitly construct a d-hitting family of size $O(\exp(d) \cdot h^{d-1})$. In particular, for balanced trees, our construction is polylogarithmic in the number of events. In the special case of $d = 3$, we can further improve the bound: we give an explicit construction of a 3-hitting family of size $4h$, which is optimal up to a constant factor. The construction in fact extends to a larger class of partial orders known as series-parallel orders, albeit with a more complicated upper bound whose optimality is still open.

Our notion of d-hitting families is closely related to the notion of order dimension for a partial order, defined as the smallest number of total orders, the intersection of which gives rise to the partial order (Dushnik and Miller 1941; Trotter 2001). Specifically, the size of an optimal 2-hitting family is the order dimension of a partial order, and the size of an optimal d-hitting family is a natural generalization. To the best of our knowledge, general d-hitting families have not been studied before for general partial orders.

From the dimension theory for partial orders we know that even for $d = 2$ there are examples of partial orders with $O(n)$ elements whose smallest 2-hitting family has $\Omega(n)$ schedules (see Example 5.1). Therefore, there is no hope for logarithmic or polylogarithmic d-hitting families for general partial orders. Moreover, computing and even approximating the dimension for general partial orders are known to be hard problems (Yannakakis 1982;

Hegde and Jain 2007; Chalermsook, Laekhanukit, and Nanongkai 2013). Nevertheless, general partial orders arise as execution models in realistic distributed systems. To make things worse, in realistic systems the partial order of events may be unknown in advance, that is, events and their relation to previous events may be revealed as the system is running. Thus, a challenge is to come up with a small d-hitting family *online* (i.e., along with the execution).

An online construction for d-hitting families was demonstrated by Burckhardt et al. (2010) for multithreaded, shared-memory programs. Their algorithm, called *PCT* (Probabilistic Concurrency Testing), instruments a program with randomized schedule points such that the resulting program is guaranteed to uniformly sample a d-hitting family of schedules. In fact, PCT guarantees its schedules are sampled from a stronger variant of d-hitting family, which we call a *strong d-hitting family*.[1] The key idea underlying the PCT construction is to represent the underlying partial order of events as a decomposition of k chains, one per thread. The events are then cleverly scheduled from these chains so that each d-tuple of events is hit with probability at least $1/(kn^{d-1})$, where n is the total number of instructions. Unfortunately, it was not known how this construction could be generalized for concurrency models in which the decomposition cannot be computed based on syntactic structures like threads. For example, an efficient PCT procedure was not known for distributed programs communicating via asynchronous message passing, where a naive mapping of each asynchronous task to a thread would lead to a very pessimistic procedure.

In this thesis, we provide an *online* construction of d-hitting families for *arbitrary* partial orders. Our construction uses the combinatorial notion of *adaptive chain covering* (Felsner 1997); we connect this notion with strong hitting families. In adaptive chain covering, the partial order is provided one element at a time, in an "upgrowing" manner. That is, the new element is guaranteed to be maximal among the elements seen so far. The adaptive chain covering algorithm must incrementally maintain a set of chains that form a *chain covering*—a decomposition of the partial order into a (not necessarily disjoint) union of chains. A sequence of deep results show that the optimal number of chains in an adaptive chain covering algorithm is exactly the size of an optimal strong 1-hitting family (Felsner 1997; Kloch 2007). We generalize this result to show that the size of an optimal strong d-hitting family is bounded above by the optimal number of chains times n^{d-1}, where n is the number of elements in the partial order. In particular, we re-derive the PCT result in this very general setting, since the size of the chain covering is k for k threads. The best known adaptive chain covering algorithms are in fact *online chain partitioning* algorithms—they decompose the partial order into a *disjoint* union of chains. It is not known how to effectively exploit the fact that we do not need partitions, but merely

[1] The precise relationship is that every strong d-hitting family is a (weak) $(d+1)$-hitting family.

coverings (Bosek, Felsner, et al. 2012). Optimal online chain partitioning algorithms use at most w^2 chains, where w is the width of the partial order (Agarwal and Garg 2007). (By Dilworth's theorem for partial orders, w is a lower bound (Dilworth 1950).) Thus, we get online strong d-hitting families of size w^2n^{d-1} for partial orders of width w and n elements. Using a general instrumentation technique, we get a randomized testing algorithm, named *PCTCP* (Probabilistic Concurrency Testing with Chain Partitioning), with a $1/(w^2n^{d-1})$ probability of hitting each d-tuple for arbitrary partial orders, presented online, with (unknown) width w.

While the proof of correctness is involved, the final algorithm is surprisingly simple: it involves maintaining prioritized chains of events, where the priorities are assigned randomly, picking the highest priority events at all times, and reducing the priorities of chains at $d - 1$ randomly chosen points in the execution.

We have implemented this algorithm for distributed protocol implementations written in P# (Deligiannis, McCutchen, et al. 2016), as well as for distributed applications such as Zookeeper and Cassandra, on top of the SAMC model checker (Leesatapornwongsa et al. 2014a). We show empirically that PCTCP is effective in finding bugs in these applications and usually outperforms naive random exploration.

1.4 Contributions

In summary, in this thesis we study the occurrence of the small-world phenomenon in distributed systems; in particular, how to use the phenomenon as a basis for effective testing procedures. We use combinatorial techniques to develop testing procedures with rigorous guarantees of finding bugs. Our contributions can be summarized as follows.

- We introduce an abstract framework of tests, testing goals, and goal coverage. We provide a general combinatorial construction relating the probability that a random test covers a testing goal to the size of a random test set that is overwhelmingly likely to provide full test coverage. This result applies to any random testing methodology.

- We introduce and study notions of test coverage for distributed systems with network partition faults. We show that our coverage notions can explain many different bugs found in production systems.

- We introduce d-hitting families as a framework for finding bugs of depth d, which arise due to a specific ordering of d concurrent events. The size of optimal d-hitting families generalizes the order dimension for partial orders, and the families themselves are natural combinatorial objects of independent interest.

- We provide explicit constructions of d-hitting families for trees and series-parallel partial orders. The constructions for trees are close to optimal: up to a small constant factor for $d = 3$ and up to a polynomial for $d > 3$. Our families of schedules can be exponentially smaller than the size of the partial order.
- We develop an algorithm for the online construction of hitting families for arbitrary partial orders. The construction incorporates online partitioning of a partial order into a number of disjoint linearizations and enables generalizing the PCT algorithm and its probabilistic guarantees to work with arbitrary partial orders. The algorithm is the basis for a simple randomized testing procedure with guaranteed lower bounds on the probability of finding depth-d bugs.
- We implement PCT with chain partitioning (PCTCP) for programs written in the P# framework, as well as the real-word distributed systems Zookeeper and Cassandra. We provide our practical design choices such as modeling node crashes as events in the system or handling livelocks that are likely to occur in some distributed systems.

The material presented in the thesis has been published in parts in the following publications.

1. Dmitry Chistikov, Rupak Majumdar, and Filip Niksic (2016). "Hitting Families of Schedules for Asynchronous Programs". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. vol. 9780. Lecture Notes in Computer Science. Springer, pp. 157–176. DOI: 10.1007/978-3-319-41540-6_9
2. Rupak Majumdar and Filip Niksic (2018). "Why is random testing effective for partition tolerance bugs?" In: *PACMPL* 2.POPL, 46:1–46:24. DOI: 10.1145/3158134
3. Burcu Kulahcioglu Ozkan et al. (2018). "Randomized Testing of Distributed Systems with Probabilistic Guarantees". In: *PACMPL* 2.OOPSLA, 160:1–160:28. DOI: 10.1145/3276530

Chapter 2

Motivating Examples

The *CAP theorem* (Brewer 2000; Gilbert and Lynch 2002) states that a distributed system running on an unreliable network cannot simultaneously satisfy **c**onsistency, **a**vailability, and **p**artition tolerance. Since the network is unreliable and network partitions do happen, at first glance this means a system has to make a choice between consistency and availability. However, in reality the situation is far from binary—various levels of consistency and availability form a rich landscape of tradeoffs, which system developers have to navigate knowingly or unknowingly. While doing so, they inevitably encounter pitfalls.

While there are certainly intricate pitfalls that are exposed only in extremely rare combinations of events and failures, many problems in existing systems are discoverable by random testing. An evidence to this is a series of online articles by Kingsbury (2013–2018), describing in-depth analyses of distributed systems within a testing framework called Jepsen[1]. Using Jepsen, Kingsbury has analyzed and discovered issues in a whole range of distributed systems: etcd, Postgres, Redis, Riak, MongoDB, Cassandra, Kafka, RabbitMQ, Consul, Elasticsearch, Aerospike, Zookeeper, and Chronos, to name a few. In each case, the approach is similar: the system under scrutiny is subjected to random sequences of operations under failure modes such as random network partitions. The recorded behavior of the system is then analyzed against a model to establish its correctness. The Jepsen framework provides a scripting framework to define operations, failure modes, correctness conditions, and checkers specific to a particular system. While setting up all these things for a given system need not be simple and often requires a lot of intuition and understanding of the system, in most cases, the test, once set up, uncovers subtle issues within seconds, even with random sequences of operations and partitions.

In the rest of the chapter, we showcase three typical bugs found by Kingsbury and use them to motivate the notions of coverage related to network partitions, which we study

[1] `http://jepsen.io/`

in Chapter 4. We also showcase two bugs found by the author of the thesis that do not arise due to network partitions, but an unexpected interleaving of concurrent events. We use these bugs to motivate the notions of coverage related to concurrent events, which we study in Chapters 5 and 6.

2.1 Etcd

Etcd[2] is a distributed key-value store. It is intended to be used for storing small amounts of critical data, which a complex distributed application might need for service coordination, distributed locking, write barriers etc. Hence, etcd's foremost design goal is to provide strong consistency. To achieve this, operations are committed through consensus, for which etcd uses an implementation of the Raft algorithm (Ongaro and Ousterhout 2014).

Since consensus requires communication between nodes, it may be unachievable while the network is partitioned. Hence, strong consistency in this case comes at the expense of reduced availability. In order to improve availability, in etcd's API version 2, read operations by default do not go through consensus. Instead, a node responds to a read request by simply returning the local copy of the value. This design choice is an example of a tradeoff between consistency and availability.

But what exactly is the manifestation of this tradeoff on the consistency side? We can check this with a Jepsen test. We set up etcd on five nodes and start five clients to issue random read, write, and compare-and-swap operations on a single key. Additionally, we start a special process (called *nemesis* in Jepsen) to randomly partition the network into two blocks to simulate network failures. We record a history of execution, and analyze it for linearizability—each operation should give an appearance of being executed atomically between its invocation and completion.

Figure 2.1 (left) shows an inconsistent state found by Jepsen. The arrows "w i" and "r" refer to client requests to write the value i or to read the shared state, respectively (for simplicity, we omit the key), and "w ok" and "r i" denote the system responses confirming the write and returning the value read, respectively. The blue rectangle shows the Raft consensus. In the picture, time flows downward, and the values show each node's own view of the shared state. The red line marks a network partition that separates n_1 and n_4 from n_2, n_3, and n_5. A write of the value 1 after the partition triggers a new leader election in Raft, after which the new value is committed (by the right block). The inconsistent behavior is that after the partition, the node n_1 returns its local stale value 0 even though it is not part of the quorum.

The developers of etcd were aware of this behavior. To give users stronger consistency,

[2] `https://github.com/coreos/etcd`

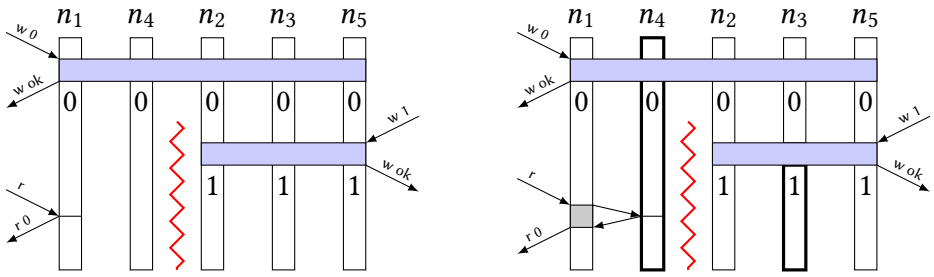

Figure 2.1: Non-linearizable execution histories in etcd. On the left, the read operation is invoked in the default mode—returning the local copy of the value—and on the right it is invoked in the consistent mode—redirecting the request to the Raft leader.

the read API provides an option called *consistent*, which causes nodes to redirect read requests to the leader elected as part of the Raft consensus algorithm. This option is another example of navigating the CAP landscape: we seemingly avoid both the overhead of full consensus and the inconsistent behavior. Unfortunately, the same Jepsen test setup quickly discovers another inconsistency, shown in Figure 2.1 (right). Assume that n_4 was the Raft leader to begin with. The write request w 0 is committed. At this point, the network partition separates n_1 and n_4 from the rest. Realizing that the leader is unavailable, nodes in the larger block elect n_3 as the new leader and successfully commit a new write of the value 1. Nodes n_1 and n_4 are unaware of the new leader election. Thus, n_1 forwards a read request to n_4, which returns the stale value 0. The problem is still that reads do not require consensus, so the smaller block does not realize the leader has changed.

Since the author of Jepsen reported this behavior[3], the developers of etcd have included another option for read called *quorum*. With this option, reads are committed through consensus the same way writes are. More recently, consensus for all operations has become the default behavior in the etcd API version 3.

2.2 Kafka

Kafka[4] is a distributed streaming system: it provides streams of records to which clients can publish or subscribe. To achieve scalability and fault-tolerance, records in Kafka's streams are partitioned into shards, and each shard is replicated by a set of in-sync replicas (ISR). Within an ISR, one node is designated as a leader: for leader election, Kafka uses

[3] `https://github.com/coreos/etcd/issues/741`
[4] `https://kafka.apache.org/`

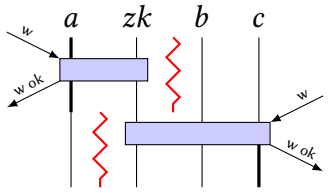

Figure 2.2: A diagram of an inconsistent behavior in Kafka.

Zookeeper[5], a distributed key-value store with strong consistency guarantees similar to etcd. The leader accepts write requests from clients and forwards them to all replicas in the ISR. When it receives acknowledgements from all replicas, it sends acknowledgement to the client. If a replica fails to acknowledge a request, the leader detects the request has timed out, and removes that node from the ISR. Remaining writes only have to be acknowledged by the healthy nodes still in the ISR. Should the leader ever fail, any replica in the ISR can take over, since all of them maintain the same history of records. In this way, Kafka can theoretically tolerate $f - 1$ failures with f replicas.

Note that with up to $f - 1$ failures Kafka provides both linearizability—all nodes in the ISR replicate records in the same order—and high availability—unresponsive nodes are automatically removed from the ISR. According to the CAP theorem, Kafka has to give up partition tolerance. The following Jepsen test, reported by Kingsbury (see Figure 2.2), shows not only that Kafka gives up partition tolerance, but that in presence of network partitions a *single* node failure can cause writes committed to the system to be lost. In a system consisting of a Zookeeper node and three in-sync-replicas a, b, c, with a being the leader, nodes b and c are separated from a and the Zookeeper node. Realizing that nodes b and c are unavailable, node a shrinks the ISR to just itself, and continues processing writes from the client. Node a then crashes (simulated by a partition that separates a from all other nodes), and the system enters a state called unclean leader election, in which any of the nodes b and c (whichever comes to life first) can be elected as a new leader. However, once the new leader starts processing writes, all intermediate writes processed by node a are lost.

2.3 Chronos

The third bug is from Chronos[6], a distributed and fault-tolerant job scheduler. Chronos is meant to schedule jobs with complicated dependency and periodicity specifications at the

[5] `http://zookeeper.apache.org/`
[6] `https://mesos.github.io/chronos/`

correct times. It is used in conjunction with Mesos[7]—a cluster management system that takes care of managing resources such as CPU and memory in a cluster. Both systems further depend on Zookeeper as a consistent data store.

There are three kinds of quorums involved in the system. First, Chronos has to maintain consensus over job scheduling, so Chronos nodes have a notion of leader and followers. Second, Mesos nodes are divided into "slaves," which offer resources and ultimately run jobs, and "masters," which take care of resource and job allocation. The latter also requires consensus, so Mesos masters have their own notion of leader and followers. And third, Zookeeper is a consistent data store and needs to maintain consensus over executed operations, so it also has a notion of leader and followers. All three kinds of leaders need to be able to communicate. Chronos and Mesos need Zookeeper for their internal coordination, but also for mutual discovery. It is thus not difficult to imagine partitions among these nodes may cause problems.

Kingsbury tested the system on five nodes $n_1, \ldots, n_5$. All five serve at the same time as Chronos and Zookeeper nodes. Additionally, nodes n_2, n_3, and n_4 are Mesos masters and the remaining two are Mesos slaves. During the test, simple jobs are generated and emitted, while the nemesis randomly partitions the network into two blocks and heals it back after some time. At the end, a checker examines the history to see whether all jobs were executed at correct time.

Not suprisingly, as soon as the Chronos leader is separated from the Zookeeper leader, problems start. In this case Chronos sometimes abruptly crashes, which turns out to be an undocumented but expected behavior[8]—the programmers considered this to be a conservative way of preventing inconsistencies while Zookeeper is unavailable. Suprisingly, however, in the same situation Chronos sometimes *does not* crash for unknown reasons.[9] This behavior is marked as a bug and was unresolved at the time this thesis was written.

The case when Chronos does not crash uncovered another bug[10] that has meanwhile been fixed. The observed execution is illustrated in Figure 2.3. Initially all three subsystems—Chronos, Mesos, and Zookeeper—go through leader election. Node n_1 becomes Chronos leader (blue timeline), node n_3 becomes Mesos leading master (orange timeline), and node n_5 becomes Zookeeper leader (green timeline). Chronos registers a new framework with Mesos—this is depicted by an exchange of messages "reg" and "reg ok" between n_1 and n_3—and starts scheduling jobs (omitted for simplicity). Next, a network partition separates n_1 and n_2 from n_3, n_4, and n_5. Chronos leader detects Zookeeper

[7] http://mesos.apache.org/

[8] https://github.com/mesos/chronos/issues/513

[9] https://github.com/mesos/chronos/issues/522

[10] https://github.com/mesos/chronos/issues/520

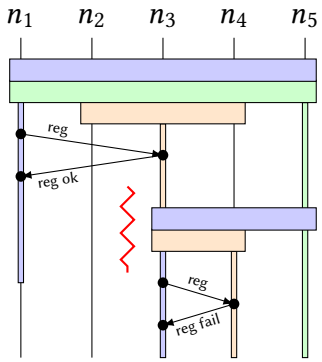

Figure 2.3: A diagram of a bug in Chronos that prevents jobs from being executed.

connection loss, but does *not* crash. After another round of elections, n_3 becomes Chronos leader, and n_4 becomes Mesos leading master. The partition is resolved, n_1 detects the Zookeeper leader and recognizes n_3 as the new Chronos leader.

At this point Chronos tries to register as a completely new framework with Mesos, instead of re-registring as the original framework. Since according to Mesos all resources in the cluster are owned by the original framework, there are no available resources for the new framework, and as a consequence no new jobs are ever started again.

2.4 CTStore

The next two bugs do not involve network partitions and were not discovered by Jepsen. Instead, they were discovered by a prototype testing framework built around the ideas of hitting families, which we present in Chapters 5 and 6. The bugs appear in a library called CTStore, developed by Lenin Ravindranath Sivalingam[11] as part of a research project at Microsoft. Unfortunately, at the time of writing, the library was not publicly available.

The goal of CTStore is to provide unified usage of a fast but ephemeral cache and a slow but persistent store, both of which are typically located in a cloud. CTStore provides an API to retrieve, insert, update, and delete entries. It retrieves an entry from the cache if it is available there. If not, it retrieves the entry from the persistent store and inserts it into the cache in order to make future retrievals faster. All of this is done behind the scene: the user perceives a fast unified store that does the right thing.

The main consistency property guaranteed by CTStore is that if an entry exists in the cache, it also exists in the persistent store. In order to provide this guarantee, the

[11] `https://www.microsoft.com/en-us/research/people/lenin/`

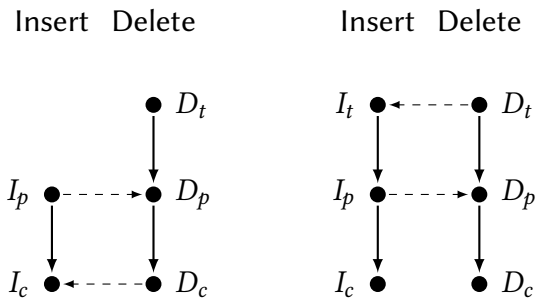

Figure 2.4: Two bugs in CTStore. The insert operation on the left does not tag the entity in the cache, which may lead to inconsistent executions. The insert operation on the right tags the entity, but the inconsistent executions are still possible.

operations that modify the store execute the following protocol: first they tag the entry in the cache with a unique tag (if the entry is not in the cache, they insert it there). Second, they modify the persistent store by inserting, replacing, or deleting the entry. Finally, they perform the corresponding modification in the cache, but only if the entry still holds the same tag from the first step.

We tested the library by concurrently executing multiple operations that modify the same entry, searching for executions that may lead to an inconsistent final state. In the initial version of the library, the insert operation did not follow the protocol described above: it would skip the first step and immediately insert the entry into the persistent store and the cache, regardless of the tag. Such a shortcut is not correct, as it was revealed in a test that concurrently inserts and deletes the same entry. The problem is conceptually depicted in Figure 2.4 (left). The nodes I_p, I_c, and D_t, D_p, D_c represent the steps of the insert and the delete operation, respectively. The subscripts t, p, and c denote tagging, persisting, and caching. The solid vertical arrows represent the order in which the steps are executed, and the dashed horizontal arrows represent the additional ordering constraints that *guarantee* the final state will be inconsistent. In particular, it is clear that if D_p is executed after I_p, that is, the entry is removed from the persistent store immediately after it is inserted, and I_c is executed after D_c, that is, the entry is inserted into the cache immediately after it is deleted, the execution will end in an inconsistent state. Note that it does not matter in which order I_p and D_t are executed.

After eliminating the shortcut in the insert operation, we repeated the same test, only to discover that the protocol still allows inconsistent executions. The problem is shown in Figure 2.4 (right). If I_t is executed after D_t, the entry is tagged by the insert's unique tag. The entry is then inserted into the persistent store by I_p only to be deleted by D_p. Finally,

regardless of when I_c is executed relative to D_p and D_c, the entry is inserted into the cache since it holds the insert's tag. Hence, the final state is inconsistent. Unfortunately, this time it is not obvious how to fix the protocol.

2.5 Summary and Coverage Notions

In each of the Jepsen examples, the test setup to manifest the misbehavior involved: setting up and running the system with a number of nodes and clients; running a random sequence of operations (reads, writes, etc.) to put the system in a specific state; introducing one or more carefully orchestrated partitions of the network; running a further sequence of operations, and optionally, further partitions of the network to demonstrate a violation. Jepsen provides an API to programmatically manage all these steps. In most Jepsen experiments, though, picking each of these steps *randomly* demonstrated the misbheaviors.

While the original Jepsen traces contain hundreds or thousands of events, the analysis of Kingsbury shows that for most bugs one requires a small number of "rounds," consisting of a small number of operations and one or two carefully chosen partitions in the system. In particular, the bugs in Chronos were exposed with a partition separating the Chronos leader from the Zookeeper leader. The inconsistency in Kafka was exposed with a partition separating node a and the Zookeeper node from nodes b and c. The inconsistencies in etcd were exposed with a partition isolating the Raft leader in a smaller block, while the system received a write request in the larger block, followed by a read request in the smaller block.

Accordingly, we study the following coverage notions.

k-splitting Given a set of k nodes, the goal is to *split* them with a network partition, that is, place each node in a separate block of the partition. The coverage problem is to split every set of k nodes with a family of partitions.

k, l-separation Given two sets of nodes of size k and l, the goal is to *separate* them with a network partition, that is, place the set of k nodes in one block and the set of l nodes in another block. The coverage problem is to separate every pair of sets of size k and l with a family of partitions.

minority isolation Given a node x in a system with n nodes, the goal is to *isolate* x in a *minority* block—a block containing less than $n/2$ nodes. The coverage problem is to isolate every element in a minority block with a family of partitions.

In addition to these coverage notions relating to network partitions, we require a set of *operations* to occur before or after the partition. Thus, we also study the following notion:

sequences of operations Given a sequence of k operations, the coverage problem is to observe this sequence as a contiguous subsequence of a larger sequence.

Our goal is to combine partition coverage with coverage w.r.t. short sequences of operations.

The test exposing the bugs in Chronos is an example of k-splitting with $k = 2$, but also of k, l-separation with $k = l = 1$, as the two notions overlap in this case. The test exposing the inconsistency in Kafka is an example of a sequenced k, l-separation: we need to observe a $2, 2$-separating partition followed by a $1, 3$-separating partition. Finally, the test exposing the inconsistencies in etcd is an example of minority isolation combined with a sequence of two operations. In each case, the separation is accompanied with short sequences of operations (e.g., reads or writes). Our results show that in each of these cases a small number of randomly generated partitions and operations achieves full coverage with overwhelming probability.

It is worth mentioning that our coverage notions are not specifically tied to the three systems we picked as examples. In fact, almost all anomalies discovered and described by Kingsbury fall under one or more of our coverage notions. There is, however, an exception: a split-brain issue in Elasticsearch[12] involving an *intersecting* partition—one in which blocks are not disjoint. In this particular issue, a node acts as a bridge between two otherwise disconnected blocks and facilitates a situation in which two nodes on the opposite sides of the partition become leaders and start processing write requests from clients. Note that an intersecting partition $\{X, Y\}$, where $Z = X \cap Y$ is nonempty, can be modeled by a partition $\{X \setminus Z, Z, Y \setminus Z\}$. Therefore, the Elasticsearch issue would fall under a straightforward generalization of k, l-separation with an additional block, and our techniques would apply. We omit this generalization for simplicity.

In the CTStore example, the test setup is different than the one in the Jepsen examples. Instead of searching for a combination of operation sequences and network partitions, we search for an interleaving of concurrent events, also called a *schedule*, that causes a bug. Even though the total number of events in the example is not large, we can see that both bugs depend on the relative ordering of exactly four events: the ordering $I_p \to D_p \to D_c \to I_c$ in the first case and the ordering $D_t \to I_t \to I_p \to D_p$ in the second case are guaranteed to lead to an inconsistent state. If the concurrent insertion and deletion of the same entry were part of a larger execution with a number of unrelated events, the same orderings of four events would still cause the bugs.

We study two coverage notions related to schedules of events:

d-hitting Given a d-tuple $(e_1, \ldots, e_d)$ of events from a partially ordered set of events, the goal is to *hit* the tuple with a schedule, that is, totally order all events in a way that

[12] `https://github.com/elastic/elasticsearch/issues/2488`

the events from the tuple appear in the relative ordering $e_1 \to \ldots \to e_d$. The hitting problem is to hit every d-tuple consistent with the partial order with a family of schedules.

strong d-hitting Given a d-tuple $(e_1, \ldots, e_d)$ of events from a partially ordered set of events, the goal is to *strongly hit* the tuple with a schedule, that is, totally order all events in a way that for every index i, $1 \leq i \leq d$, the event e_i appears at the last possible position before $e_{i+1}, \ldots, e_d$. In other words, if some event e appears after the event e_i in the schedule, it appears there because it is partially ordered after some e_j for $j \geq i$ to begin with. The strong hitting problem is to hit every d-tuple of events with a family of schedules.

The two CTStore bugs are examples of d-hitting for $d = 4$. As for the strong d-hitting coverage, at first it may seem unclear why one would study such a notion. The full reasons will become apparent in Chapter 6. For now let us note that the CTStore bugs are examples of strong d-hitting for $d = 2$. To see this, consider the first bug and note that every schedule that strongly hits (D_p, I_c) orders I_p before D_p and D_c before I_c, which guarantees an inconsistent state. Similarly, in the second case every schedule that strongly hits (I_t, D_p) orders D_t before I_t and I_p before D_p, again guaranteeing an inconsistent state. We can roughly think of the parameter d in strong d-hitting as a number of additional ordering constraints we need to add to the partial order in order to expose a bug. As we shall see in Chapter 6, the strong d-hitting coverage implies the $(d + 1)$-hitting coverage. In Chapters 5 and 6 we study constructions of families of schedules that achieve full hitting and strong hitting coverage.

Chapter 3

Abstract Testing

In this chapter, we introduce the notions of tests and testing goals, and we state and prove a general theorem on goal coverage. We leave the notions abstract—we will instantiate them in specific testing scenarios in the following chapters.

3.1 Covering Families

We begin with the following definition.

Definition 3.1. Let $\mathcal{T}$ be a nonempty set of *tests* and $\mathcal{G}$ a nonempty set of *testing goals*. A test $t \in \mathcal{T}$ may or may not *cover* a testing goal. A nonempty set $\mathcal{F}$ of tests is a *covering family* for $\mathcal{G}$ if for each $x \in \mathcal{G}$, there is a test $t \in \mathcal{F}$ such that t covers x.

Covering families are the central objects of study in this thesis. In particular, we are interested in their size. Constructing covering families of size exponentially or doubly-exponentially smaller than the set of all tests is the basic premise of effective testing, as this typically means we have a way to effectively cover all testing goals.

It may seem that in the abstract setting, without knowing exactly what the tests, testing goals, and the notion of goal coverage are, there is not much we can say about the size of covering families. However, there is a trivial but far-reaching observation we can make: a test can cover more than one testing goal, and conversely, a testing goal can be covered by more than one test. Consequently, covering families can contain fewer tests than $\mathcal{T}$, the set of all tests. We can quantify the size of covering families more precisely if we know a positive lower bound on the number of tests that cover a fixed goal. More generally, if we equip the set $\mathcal{T}$ with a probability distribution, and we know a positive lower bound on the probability that a random test covers a fixed goal, we obtain the following theorem.

Theorem 3.1. *Let $\mathcal{T}$ be a set of tests and $\mathcal{G}$ a set of m testing goals. Let $p > 0$ be a lower bound on the probability that a random test covers a fixed goal. Given $\epsilon > 0$, let $\mathcal{F}$ be a nonempty family of tests chosen independently and uniformly at random such that $|\mathcal{F}| \geq p^{-1}(\log m + \log \epsilon^{-1})$. Then $\mathcal{F}$ is a covering family with probability at least $1 - \epsilon$. Moreover, there exists a covering family of size $\left\lceil p^{-1} \log m \right\rceil$ (or 1 if $m = 1$).*

Proof. If $p = 1$, then every test with the positive probability of being chosen covers all goals. Therefore, any family containing at least one such test is a covering family, and the result trivially holds.

Thus, assume $p < 1$ and consider a fixed testing goal x. A random test does not cover x with probability at most $1 - p$. Since the tests in $\mathcal{F}$ are chosen independently, the probability that $\mathcal{F}$ does not cover x is at most $(1 - p)^{|\mathcal{F}|}$. By the union bound, the probability that there exists a testing goal not covered by $\mathcal{F}$ is at most $m(1 - p)^{|\mathcal{F}|}$.

If $m > \epsilon$, then $\log m + \log \epsilon^{-1} > 0$. Recall that $-\log(1 - p) = p + \frac{p^2}{2} + \ldots > p$, so $p^{-1} > -1/\log(1 - p)$. By combining the two inequalities, we get

$$|\mathcal{F}| \geq p^{-1}(\log m + \log \epsilon^{-1}) > \frac{-\log m - \log \epsilon^{-1}}{\log(1 - p)} = \log_{1-p}(m^{-1}\epsilon) \ .$$

Note that the final inequality trivially holds if $m \leq \epsilon$, since then $\log_{1-p}(m^{-1}\epsilon) \leq 0$. Therefore, in both cases $m(1 - p)^{|\mathcal{F}|} < \epsilon$, and hence the probability that $\mathcal{F}$ covers all testing goals is greater than $1 - \epsilon$. In particular, if we take $\epsilon = 1$, the probability that $\mathcal{F}$ covers all goals is positive. By the probabilistic method, there must exist a covering family of size $\lceil p^{-1} \log m \rceil$, or 1 if $m = 1$, since a covering family needs to be nonempty. □

There is always a trivial covering family of size $|\mathcal{G}|$ (assuming each testing goal can be covered by some test). The key observation in Theorem 3.1 is that there exist covering families of size proportional to $\log|\mathcal{G}|$; thus, if we can show the probability p is "high," we can get an exponentially smaller covering family of tests, and moreover, a randomly chosen test set can cover all goals with high probability. Of course, a caveat is that showing p is "high" usually means showing it is a constant. In general, this may not be the case, that is, p may depend on $|\mathcal{T}|$ and $|\mathcal{G}|$. For instance, suppose we only know that each testing goal is covered by some test, and the tests are sampled uniformly at random. Then we can take $p = 1/|\mathcal{T}|$ as the lower bound. The theorem then tells us there is a covering family of size $|\mathcal{T}| \log|\mathcal{G}|$, which is not very useful since we already know that the set $\mathcal{T}$ itself is a covering family of size $|\mathcal{T}|$.

As an example of non-trivial usage of Theorem 3.1, let us demonstrate how it can be used to analyze the coverage notion involving sequences of operations motivated in Section 2.5. Suppose we have $r \geq 1$ different operations, and we are generating a sequence

of operations s uniformly at random. Suppose we also have a set T of target sequences of length $k \geq 1$, and we want to observe any target sequence $t \in T$ as a contiguous subsequence of s.

Proposition 3.2. *Let $\epsilon > 0$, let T be a set of sequences of operations of length $k \geq 1$, and let s be a sequence of $n \geq 1$ operations chosen independently and uniformly at random such that $n \geq k + kr^k|T|^{-1} \log \epsilon^{-1}$. Then some target sequence $t \in T$ is a contiguous subsequence of s with probability at least $1 - \epsilon$.*

Proof. Split the sequence s into $\lfloor n/k \rfloor$ non-overlapping subsequences of length k. The probability that some $t \in T$ occurs among these subsequences is clearly lower than the probability that some $t \in T$ occurs in s. However, we can think of the non-overlapping sequences as $\lfloor n/k \rfloor$ sequences of length k generated independently and uniformly at random.

The probability that one of these sequences matches a sequence in T is $p = |T|/r^k$. The number of testing goals in this case is $m = 1$, namely any target sequence $t \in T$. Since $\lfloor n/k \rfloor > n/k - 1$, we have

$$\lfloor n/k \rfloor > r^k|T|^{-1} \log \epsilon^{-1} = p^{-1}(\log m + \log \epsilon^{-1}) \quad .$$

Hence the result follows from Theorem 3.1. □

Example 3.1. Consider the inconsistency in etcd in Figure 2.1 (left). In order to expose it, we need to observe a right combination of read and write operations during a network partition with two blocks. Note that it does not really matter how the network is partitioned, as long as we follow up with a write that changes the value in the larger block, and a read to any node in the smaller block. For simplicity, assume we only have three kinds of operations—read, write 0, and write 1—and each can be directed to any of the five nodes involved in the experiment. Thus, in total we have 15 operations, each occurring with equal probability.[1] We form an amalgamated operation by conjoining a partition with two read or write operations. Assuming we only partition into blocks of size 2 and 3, this gives us a total of $r = 10 \cdot 15 \cdot 15 = 2250$ amalgamated operations. Our target operations consist of a partition followed by one of two read operations and one of three write operations. Thus the set T of target amalgamated operations has size $10 \cdot 2 \cdot 3 = 60$. Applying Proposition 3.2 with $k = 1$, we see that in a sequence of 61 amalgamated operations, we would observe a target operation—and thus expose the inconsistency—with probability at least 80%. □

[1] In the real Jepsen experiment, clients were also issuing compare-and-swap operations with values ranging from 1 to 5, and the probability distribution over operations was not uniform.

As another example, we analyze the notion of d-hitting coverage from Section 2.5 for a special case of n concurrent events.

Example 3.2. Consider a set A of n concurrent events. Since the events are concurrent, every permutation of the set forms a valid schedule. Recall that for a given d-tuple of events $(e_1, \ldots, e_d) \in A^d$, the goal of the d-hitting coverage is to hit the tuple with a schedule, that is, totally order A so that $e_1, \ldots, e_d$ appear in the relative ordering $e_1 \to \ldots \to e_d$. Hence, the set of tests $\mathcal{T}$ in this case is the set of schedules of A, and the set of goals $\mathcal{G}$ is the set of d-tuples A^d.

If the schedules are sampled uniformly at random, what is the probability that a random schedule hits a fixed d-tuple? Since a schedule is just a permutation, it arranges the d events from the tuple in one of $d!$ ways without giving preference to any of them. Hence, the hitting probability is $p = 1/d!$. Since the number of d-tuples is $\binom{n}{d} d! \le n^d$, it follows from Theorem 3.1 that there exists a covering family of size $d! \cdot d \log n$. □

3.2 Independent Goals

In the proof of Theorem 3.1 we resorted to the union bound to bound the probability that some testing goal was not covered by the family $\mathcal{F}$. This step potentially introduced imprecision in the analysis, but without further assumptions on the independence of covering different goals, there does not seem to be a better way to complete the proof. Let us for a moment visit the other extreme, that is, let us assume that for any two goals $g, g' \in \mathcal{G}$, a random test covers g independently of covering g'. Since the family $\mathcal{F}$ covers a fixed goal with probability at least $1 - (1-p)^{|\mathcal{F}|}$, the probability that it covers all goals is at least $\left(1 - (1-p)^{|\mathcal{F}|}\right)^{|\mathcal{G}|}$, and this probability is positive if $(1-p)^{|\mathcal{F}|} < 1$, that is, if $|\mathcal{F}| \ge 1$. In other words, if the goals are mutually independent, we can cover all of them with a single test.

As it turns out, there is a middle ground between assuming nothing about the independence and assuming that any two goals are mutually independent. In particular, there is a theorem known as the Lovász local lemma (Erdős and Lovász 1975; Alon and Spencer 2004).

Theorem 3.3 (Lovász local lemma)**.** *Let $A_1, \ldots, A_n$ be events in an arbitrary probability space. Suppose that each event is independent of all the other events except for at most c of them, and that each event occurs with probability at most p. If $ep(c+1) \le 1$, where e is the base of the natural logarithm, then none of the events occurs with positive probability.* □

If we know that every testing goal is independent of all but at most c other goals, then

the Lovász local lemma can sometimes provide a better bound on the size of covering families than Theorem 3.1. We show an instance of this in the following example.

Example 3.3. As in Example 3.2, suppose we have a set A of n concurrent events and we want to hit every d-tuple from A^d. Let $g = (e_1, \ldots, e_d)$ and $g' = (e'_1, \ldots, e'_d)$ be two d-tuples that do not share any events, that is, $e_i \neq e'_j$ for all i, j such that $1 \leq i, j \leq d$. What is the probability that a random schedule hits g conditioned on hitting g'? Since g and g' do not overlap, we can repeat the same argument from Example 3.2: the schedule arranges the events $e_1, \ldots, e_d$ in one of $d!$ ways without giving preference to any of them. Hence, the conditional hitting probability is $1/d!$, the same as the unconditional one. Thus, the goals g and g' are mutually independent. On the other hand, if the goals overlap, in general they are not independent. To see this, take $A = \{1, 2, 3\}$ and let $g = (1, 2)$ and $g' = (1, 3)$. There are 3 schedules that hit g', namely 123, 132, and 213. Among these, 2 schedules hit g, so the conditional hitting probability is $2/3 \neq 1/2$.

From the above analysis, we conclude that every d-tuple is independent of $(n-d)^{\underline{d}}$ other d-tuples, where $(n-d)^{\underline{d}} = (n-d)(n-d-1)\cdots(n-2d+1)$ is the falling factorial power. Let $c = n^{\underline{d}} - (n-d)^{\underline{d}} - 1$ and let $\mathcal{F}$ be a family of schedules chosen independently and uniformly at random such that

$$|\mathcal{F}| \geq d! \left(1 + \log\left(n^{\underline{d}} - (n-d)^{\underline{d}}\right)\right) = d!\,(1 + \log(c+1)) \quad .$$

A calculation similar to the one in the proof of Theorem 3.1 shows

$$|\mathcal{F}| \geq \frac{-1 - \log(c+1)}{\log(1 - 1/d!)} = \log_{1-1/d!}\left(e^{-1}(c+1)^{-1}\right) \quad .$$

Since the probability that a fixed goal is not covered by $\mathcal{F}$ is $p = (1 - 1/d!)^{|\mathcal{F}|}$, using the above inequality we get $ep(c+1) \leq 1$; thus, by the Lovász local lemma, $\mathcal{F}$ is a covering family with positive probability.

To see that $d!(1+\log(c+1))$ is an asymptotic improvement over $d! \cdot d \log n$, we calculate the following limit using L'Hôpital's rule.

$$L := \lim_{n\to\infty} \frac{1 + \log(c+1)}{d \log n} = \lim_{n\to\infty} \frac{n(c+1)'}{d(c+1)}$$

Note that

$$\begin{aligned} c + 1 &= n^{\underline{d}} - (n-d)^{\underline{d}} \\ &= n^{\underline{d}} - (n-1)^{\underline{d}} + (n-1)^{\underline{d}} - \ldots + (n-d+1)^{\underline{d}} - (n-d)^{\underline{d}} \\ &= d(n-1)^{\underline{d-1}} + \ldots + d(n-d)^{\underline{d-1}} \\ &= dn^{d-1} + O(n^{d-2}) \quad . \end{aligned}$$

Therefore, $n(c+1)' = d(d-1)n^{d-1} + O(n^{d-2})$, and hence $L = (d-1)/d$. In other words, $d!(1+\log(c+1))$ is asymptotic to $d!(d-1)\log n$, which is a slight improvement compared to the bound obtained in Example 3.2. □

Chapter 4

Testing with Random Partitions

In this chapter, we study the coverage notions related to network partitions as instances of the general construction from Chapter 3. We start with combinatorial preliminaries about set partitions.

4.1 Combinatorial Preliminaries

Throughout this section, let $U = \{1, \ldots, n\}$ be a fixed set (a "universe") of n elements. A *partition* of U is a set of nonempty subsets of U that are pairwise disjoint and in the union give the whole U. We refer to the sets in a partition as *blocks*. If a partition has k blocks, we call it a k-partition. A *balanced partition* is a partition with blocks differing in size at most by 1.

Let us recall a few results about partitions. The number of k-partitions is given by a quantity called *Stirling number of the second kind*, denoted $\left\{ {n \atop k} \right\}$ and read "n subset k" (Graham, Knuth, and Patashnik 1994). It is not difficult to see that $\left\{ {n \atop 1} \right\} = \left\{ {n \atop n} \right\} = 1$ whenever $n \geq 1$. Moreover, $\left\{ {n \atop 2} \right\} = 2^{n-1} - 1$, as a 2-partition is uniquely determined by the block that does not contain the nth element, and this block needs to be nonempty. In general, Stirling numbers of the second kind satisfy the following recurrence:

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k \left\{ {n-1 \atop k} \right\} . \tag{4.1}$$

Combinatorially, we can partition n elements into k blocks by partitioning the first $n-1$ elements into $k-1$ blocks and adding a singleton block consisting of the nth element, or by partitioning the first $n-1$ elements into k blocks and placing the nth element into one of these blocks in k ways.

Lemma 4.1. *For every $n \geq 1$ and k such that $1 \leq k \leq n$, we have*

$$\left\{ {n \atop k} \right\} k! \leq k^n \quad .$$

Proof. The quantity on the right-hand side is the number of all functions from an n-element set to a k-element set, while the quantity on the left-hand side is the number of such functions that are *surjective*. To see this, note that a surjection induces a k-partition of the domain, and the induced blocks map to the codomain in one of $k!$ ways. □

For a fixed k, $\left\{ {n \atop k} \right\}$ asymptotically approaches $k^n/k!$. Intuitively, if we randomly assign n elements into k buckets and n is large, it is unlikely one of the buckets will be empty. Therefore, the difference between the left-hand side and right-hand side in Lemma 4.1 will be small.

4.2 Splitting Families

We formalize the notion of k-splitting from Section 2.5 using k-splitting families.

Definition 4.1. Given k, let P be a k-partition of U and let $S = \{x_1, \ldots, x_k\} \subseteq U$. We say P *splits* S if $P = \{B_1, \ldots, B_k\}$ and $x_1 \in B_1, \ldots, x_k \in B_k$. We say a family $\mathcal{F}$ of k-partitions is a *k-splitting family* if for every subset $S \subseteq U$ there is a partition in $\mathcal{F}$ that splits S.

Splitting families are an instance of the covering families from Section 3.1: a testing goal here is a subset of U of size k, a test is a partition of U with k blocks, and a covering family is a k-splitting family. Let the k-partitions be uniformly distributed. We shall invoke Theorem 3.1 to obtain a bound on the size of splitting families, but in order to do so, we need to derive a lower bound on the probability that a random k-partition splits a fixed set of k elements. As the following theorem shows, this probability can be bounded from below by a constant that depends only on k.

Theorem 4.2. *Let $S \subseteq U$ be a set of k elements, and let p be the probability that a random k-partition splits S. Then $p = k^{n-k} / \left\{ {n \atop k} \right\} \geq k^{-k} k!$.*

Proof. A k-partition that splits S is uniquely determined by a map $U \setminus S \to S$ that maps $x \in U \setminus S$ to $y \in S$ if x and y are in the same block of the partition. Hence, the probability that a random k-partition splits S is $p = k^{n-k} / \left\{ {n \atop k} \right\}$. From Lemma 4.1 it follows that $p \geq k^{-k} k!$. □

The bound on the size of splitting families is now a corollary of Theorem 3.1.

Corollary 4.3. *Let $\epsilon > 0$ and let $\mathcal{F}$ be a family of k-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \geq k^{k+1}(k!)^{-1}\log n + k^k(k!)^{-1}\log \epsilon^{-1}$. Then $\mathcal{F}$ is k-splitting with probability at least $1-\epsilon$. Moreover, there exists a k-splitting family of size $\left\lceil k^{k+1}(k!)^{-1}\log n \right\rceil$.*

Proof. By Theorem 4.2, a fixed subset of U of size k is split by a random k-partition with probability $p \geq k^{-k}k!$. Moreover, the number of subsets of U of size k is $m = \binom{n}{k} \leq n^k$. Therefore, $|\mathcal{F}| \geq p^{-1}(\log m + \log \epsilon^{-1})$, and the result follows from Theorem 3.1. □

In the experiments done by Jepsen, the most common case is to split the network into two blocks, that is, $k = 2$. In this case we can derive a more precise bound.

Corollary 4.4. *Let $\epsilon > 0$ and let $\mathcal{F}$ be a family of 2-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \geq 2\log_2 n + \log_2 \epsilon^{-1} - 1$. Then $\mathcal{F}$ is 2-splitting with probability at least $1-\epsilon$. Moreover, there exists a 2-splitting family of size $\lceil 2\log_2 n - 1\rceil$.*

Proof. We get this slightly more precise bound by performing a more precise version of the analysis from the proof of Theorem 3.1. Like there, we can bound the probability that $\mathcal{F}$ is not 2-splitting by $m(1-p)^{|\mathcal{F}|}$, with $m = \binom{n}{2}$ and $p = 2^{n-2}/\left\{{n \atop 2}\right\}$. However, since $\left\{{n \atop 2}\right\} = 2^{n-1} - 1$, we have $1 - p < 1/2$ whenever $n \geq 2$. Hence, $m(1-p)^{|\mathcal{F}|} < m2^{-|\mathcal{F}|}$. On the other hand, since $m = \binom{n}{2} \leq n^2/2$, we have $|\mathcal{F}| \geq 2\log_2 n - \log_2 \epsilon - 1 \geq \log_2(m/\epsilon)$. Hence, $m(1-p)^{-|\mathcal{F}|} < m2^{-|\mathcal{F}|} \leq \epsilon$. □

Remark 1. Note that there is a deterministic construction of a 2-splitting family of size $\lfloor \log_2 n \rfloor + 1$. To see this, take the binary representation of the elements in the universe. For each position $i \in \{0, \ldots, \lfloor \log_2 n \rfloor\}$, consider the partition obtained by separating all elements which have a 0 in the ith position from all elements which have a 1 in the ith position. Clearly, this set of $\lfloor \log_2 n \rfloor + 1$ partitions forms a 2-splitting family. Thus, the probabilistic construction in Corollary 4.4 is sub-optimal. □

Corollary 4.3 suggests that we can get k-splitting families with high probability by generating sufficiently many (logarithmically in n) k-partitions uniformly at random. But how do we generate a k-partition uniformly at random? We can do it recursively using the basic recurrence for Stirling numbers (4.1). The base cases are $k = 1$ (we generate a single block containing n elements) and $k = n$ (we generate n singleton blocks). Otherwise we choose between partitioning the first $n-1$ elements recursively into $k-1$ or k blocks with the following respective probabilities:

$$\left\{{n-1 \atop k-1}\right\} \Big/ \left\{{n \atop k}\right\} \quad \text{and} \quad k\left\{{n-1 \atop k}\right\} \Big/ \left\{{n \atop k}\right\} . \tag{4.2}$$

In the former case we add to the $k-1$ blocks a singleton block consisting of the nth element, and in the latter case we place the nth element into one of the k blocks uniformly at random.

It is not difficult to see that the described procedure indeed gives us k-partitions uniformly at random. However, the intermediate probabilities (4.2) involve computing Stirling numbers, which grow exponentially in n. Fortunately, there is a way around this obstacle. Note that we can split every k-element subset of U by at least one *balanced* k-partition. Recall that the blocks of a balanced partition differ in size at most by 1. Because of this, it is much easier to generate balanced partitions uniformly at random: we generate a random permutation of the set U and split it into k balanced blocks. Moreover, from the standpoint of lower bound on the splitting probability, we are no worse using balanced partitions instead of arbitrary partitions, as the following theorem shows.

Theorem 4.5. *Let $S \subseteq U$ be a set of k elements, let p be the probability that a random k-partition splits S, and let p_b be the probability that a random balanced k-partition splits S. Then $p_b \geq p$.*

In order to prove Theorem 4.5, we need an auxiliary combinatorial lemma of independent interest.[1]

Lemma 4.6. *Let $n, k \in \mathbb{N}$, and let $m = n \bmod k$. Then,*

$$k^{n-k}\binom{n}{k} \leq \left\lceil \frac{n}{k} \right\rceil^{m} \left\lfloor \frac{n}{k} \right\rfloor^{k-m} \left\{ {n \atop k} \right\} \ .$$

Proof. Let M be a binary matrix whose rows are indexed by k-partitions and columns by subsets of U of size k, and such that an entry corresponding to partition P and set S is 1 if and only if P splits S. We count the number of ones in the matrix in two different ways.

The number of ones in a column indexed by set S is the number of k-partitions that split S. As argued in the proof of Theorem 4.2, this number is k^{n-k}. Hence the total number of ones in M is $k^{n-k}\binom{n}{k}$. On the other hand, the number of ones in a row indexed by partition $P = \{B_1, \ldots, B_k\}$ is the number of sets split by P. It is not difficult to see this number is $|B_1| \cdots |B_k|$. Summing over all k-partitions, we get

$$k^{n-k}\binom{n}{k} = \sum_{P=\{B_1,\ldots,B_k\}} |B_1| \cdots |B_k| \tag{4.3}$$

[1] Independently of the author, a weaker variant of Lemma 4.6 was posted as Problem 11957 in the February 2017 issue of the American Mathematical Monthly (Edgar, Ullman, and D. B. West 2017). The proof given here solves the problem.

Note that the product $B = |B_1| \cdots |B_k|$ attains its maximal value for a balanced partition. For suppose the partition is not balanced; then there are blocks B_i and B_j such that $|B_i| - |B_j| \geq 2$. From these we obtain blocks B_i' and B_j' by moving an arbitrary element from B_i to B_j. Let $B' = B/(|B_i||B_j|)$; for the new product we have:

$$\begin{aligned} B'|B_i'||B_j'| &= B'(|B_i| - 1)(|B_j| + 1) \\ &= B'(|B_i||B_j| + |B_i| - |B_j| - 1) \\ &> B'|B_i||B_j| \\ &= B \end{aligned}$$

We have thus increased the value of the product.

It is not difficult to see that a balanced partition has m blocks of size $\lceil n/k \rceil$ and $k - m$ blocks of size $\lfloor n/k \rfloor$. Hence the maximal value of the product $|B_1| \cdots |B_k|$ is $\lceil n/k \rceil^m \lfloor n/k \rfloor^{k-m}$. With this we can bound the sum in (4.3) and complete the proof:

$$\sum_{P=\{B_1,\ldots,B_k\}} |B_1| \cdots |B_k| \leq \left\lceil \frac{n}{k} \right\rceil^m \left\lfloor \frac{n}{k} \right\rfloor^{k-m} \left\{ {n \atop k} \right\}$$

□

Proof of Theorem 4.5. We already know that $p = k^{n-k} / \left\{ {n \atop k} \right\}$. To calculate p_b, we need to calculate the number N_S of balanced k-partitions that split S, and the number N of all balanced k-partitions; then $p_b = N_S/N$.

Let $m = n \bmod k$. As noted earlier, a balanced k-partition has m blocks of size $\lceil n/k \rceil$ and $k - m$ blocks of size $\lfloor n/k \rfloor$. In order to uniquely determine a balanced k-partition that splits S, we first choose m elements of S that are placed in larger blocks, and then for each element of S we choose a completion of its block from $U \setminus S$. Thus,

$$N_S = \binom{k}{m} \binom{n-k}{\underbrace{\lceil n/k \rceil - 1, \ldots, \lceil n/k \rceil - 1}_{m \text{ times}}, \underbrace{\lfloor n/k \rfloor - 1, \ldots, \lfloor n/k \rfloor - 1}_{k-m \text{ times}}} .$$

Similarly, in order to uniquely determine a balanced k-partition, we choose m blocks of size $\lceil n/k \rceil$ and $k - m$ blocks of size $\lfloor n/k \rfloor$, and account for the fact that neither the order of larger nor the order of smaller blocks matters. Thus,

$$N = \frac{1}{m!(k-m)!} \binom{n}{\underbrace{\lceil n/k \rceil, \ldots, \lceil n/k \rceil}_{m \text{ times}}, \underbrace{\lfloor n/k \rfloor, \ldots, \lfloor n/k \rfloor}_{k-m \text{ times}}} .$$

After expanding the multinomial coefficients and rearranging the terms, we get

$$p_b = \left\lceil \frac{n}{k} \right\rceil^m \left\lfloor \frac{n}{k} \right\rfloor^{k-m} \Big/ \binom{n}{k} \ .$$

Hence, the inequality $p_b \geq p$ is precisely the inequality in Lemma 4.6. □

Example 4.1. Let us get back to the Chronos example from Chapter 2. In order to expose the bug, we had to split the Chronos leader from the Zookeeper leader in the set of five nodes. Corollary 4.4 tells us it is possible to do this with just 4 randomly generated partitions. Moreover, by generating just 2 additional partitions, we ensure the probability of the split is at least 80%. The optimal 2-splitting family in this case contains 3 partitions, and can be constructed explicitly (not randomly!) by the construction in Remark 1. □

Historical note. Splitting families appear in the context of perfect hashing (Yao 1981; Fredman, Komlós, and Szemerédi 1984; Czech, Havas, and Majewski 1997). Andrew Chi-Chih Yao calls them k-separating systems and gives an explicit construction of such systems of size $4^{k^2}(\log_2 n)^{k-1}$ (Yao 1981). He also references personal communication with Ronald Graham for a probabilistic construction of size $e^k\sqrt{k}\log n$. Another reference to personal communication with Ronald Graham appears in Fredman, Komlós, and Szemerédi (1984). It is very likely that Graham's construction is similar to the one given here.

4.3 Separating Families

We now turn to the notion of k, l-separation from Section 2.5, formalized using k, l-separating families. In this section we fix two positive integers k, l such that $k + l \leq n$.

Definition 4.2. Let $\mathcal{F}$ be a family of 2-partitions. We call $\mathcal{F}$ a *k, l-separating family* if for every pair of disjoint sets $S = \{x_1, \ldots, x_k\} \subseteq U$ and $T = \{y_1, \ldots, y_l\} \subseteq U$ there is a partition $P = \{X, Y\} \in \mathcal{F}$ such that $S \subseteq X$ and $T \subseteq Y$.

Like splitting families, separating families are also an instance of covering families from Section 3.1. A testing goal here is a pair (S, T) of disjoint sets $S, T \subseteq U$ such that $|S| = k$ and $|T| = l$. A test is a partition of U with two blocks, and a covering family is a k, l-separating family. Here we can also bound the probability that a random 2-partition separates two fixed sets of size k and l, and the bound depends only on k and l.

Theorem 4.7. *Let $S, T \subseteq U$ be sets of k and l elements, and let p be the probability that a random 2-partition separates S and T. Then $p = 2^{n-k-l} / \left\{ {n \atop 2} \right\} \geq 2^{1-k-l}$.*

Proof. A 2-partition that separates S and T is uniquely determined by a map $U \setminus (S \cup T) \to \{0, 1\}$ that maps $x \in U \setminus (S \cup T)$ to 1 if and only if x is in the block of the partition that contains S. Hence, the probability that a random 2-partition separates S and T is $p = 2^{n-k-l} / \left\{ {n \atop 2} \right\}$. From the fact that $\left\{ {n \atop 2} \right\} = 2^{n-1} - 1$ it follows that $p \geq 2^{1-k-l}$. □

With the obtained lower bound, we can invoke Theorem 3.1 to obtain a bound on the size of separating families.

Corollary 4.8. *Let $\epsilon > 0$ and let $\mathcal{F}$ be a family of 2-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \geq 2^{k+l-1}(k+l)\log n + 2^{k+l-1}\log \epsilon^{-1}$. Then $\mathcal{F}$ is k, l-separating with probability at least $1 - \epsilon$. Moreover, there exists a k, l-separating family of size $\left\lceil 2^{k+l-1}(k+l)\log n \right\rceil$.*

Proof. By Theorem 4.7, the probability that a random 2-partition separates two subsets $S, T \subseteq U$ of size k and l is $p \geq 2^{1-k-l}$. Moreover, the number of pairs of subsets (S, T) of size k and l is $m = \binom{n}{k}\binom{n-k}{l} \leq n^{k+l}$. Therefore, $|\mathcal{F}| \geq p^{-1}(\log m + \log \epsilon^{-1})$, and the result follows from Theorem 3.1. □

Example 4.2. Let us get back to the Kafka example from Chapter 2. By plugging $k = l = 2$ and $n = 4$ into Corollary 4.8, we see that we can separate node a and the Zookeeper node from nodes b and c with approximately 45 randomly generated partitions. We can get a more precise bound by using Theorem 3.1 directly with $p = 1/8$ and $m = \binom{4}{2,2} = 6$; this tells us separation is possible with 15 randomly generated partitions.

Of course, the separation of node a and the Zookeeper node from nodes b and c does not expose the inconsistency on its own—we need two consecutive partitions, one being 2, 2-separating, and another being 1, 3-separating. Suppose we alternate partitions with blocks of size 2 and 2 (first phase), and partitions with blocks of size 1 and 3 (second phase). Fix a pair of nodes (x, y)—x representing the Zookeeper node, and y representing node a. The probability of x and y ending up in the same block in the first phase is $1/3$, and the probability of y being isolated in the second phase is $1/4$, giving an overall probability of $1/12$. The number of pairs (x, y) in a set of four nodes is 12. Thus by invoking Theorem 3.1 directly, we get that we can expose the inconsistency with 30 alternations of the two phases, and we can expose it with probability at least 80% with 50 alternations. □

4.4 Minority Isolating Families

The next notion to analyze is the one of minority isolation. We formalize it using minority isolating families.

Definition 4.3. Let $\mathcal{F}$ be a family of 2-partitions. We call $\mathcal{F}$ a *minority isolating family* if for every $x \in U$ there is a partition $P = \{X, Y\} \in \mathcal{F}$ such that $x \in X$ and $|X| < |Y|$.

The analysis of minority isolating families depends on whether n is odd or even. The two cases differ slightly because if n is odd, there is a smaller block in every 2-partition, while if n is even, we can split the set into two blocks of equal size. We first analyze the case when n is odd.

Denote by p the probability that a random 2-partition isolates a fixed element x in the smaller (minority) block. By summing over the size of the block containing x, we get:

$$p = \sum_{0 \le j < \lfloor n/2 \rfloor} \binom{n-1}{j} \Big/ \left\{ {n \atop 2} \right\}$$

$$1 - p = \sum_{\lfloor n/2 \rfloor \le j < n-1} \binom{n-1}{j} \Big/ \left\{ {n \atop 2} \right\} = \sum_{0 < j \le \lfloor n/2 \rfloor} \binom{n-1}{j} \Big/ \left\{ {n \atop 2} \right\}$$

Subtracting the first equality from the second one gives us:

$$1 - 2p = \left(\binom{n-1}{\lfloor n/2 \rfloor} - 1 \right) \Big/ \left\{ {n \atop 2} \right\} \ .$$

By solving for p and using $\left\{ {n \atop 2} \right\} = 2^{n-1} - 1$, we get:

$$p = \left(2^{n-1} - \binom{n-1}{\lfloor n/2 \rfloor} \right) \Big/ (2^n - 2) \ .$$

Lemma 4.9. *If n is odd and $n \ge 3$, then $p \ge 1/3$.*

Proof. We show equivalently that $1 - 2p \le 1/3$, which is equivalent to showing $3\binom{n-1}{\lfloor n/2 \rfloor} \le 2^{n-1} + 2$. We show the latter inequality by induction on n.

If $n = 3$, it is easy to check that the inequality holds. Assume inductively that the inequality holds for an odd $n \ge 3$; then for $n + 2$ we have:

$$\begin{aligned} 3\binom{n+1}{\lfloor (n+2)/2 \rfloor} &= 3 \cdot \frac{4n}{n+1} \binom{n-1}{\lfloor n/2 \rfloor} \\ &\le \frac{4n}{n+1} (2^{n-1} + 2) \le 2^{n+1} + 2 \ . \end{aligned}$$

The last inequality boils down to $3n \le 2^n + 1$, which holds for every odd n. □

As before, we use the obtained lower bound on p to establish the bound on the size of minority isolating families as a corollary to Theorem 3.1.

Corollary 4.10. *Let n be odd and $n \geq 3$, let $\epsilon > 0$ and let $\mathcal{F}$ be a family of 2-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \geq 3(\log n + \log \epsilon^{-1})$. Then $\mathcal{F}$ is a minority isolating family with probability at least $1 - \epsilon$. Moreover, for an odd $n \geq 3$ there exists a minority isolating family of size $\lceil 3 \log n \rceil$.*

Proof. From Lemma 4.9, the probability p that a random 2-partition isolates a fixed element in the minority block satisfies $p \geq 1/3$. Moreover, in this case the testing goals are simply the elements of U, so $m = n$. The result then follows from Theorem 3.1. □

In the case of minority isolation with an odd n, balanced 2-partitions not only give us nicer random sampling, but improve the asymptotic bound on the size of minority isolating families.

Corollary 4.11. *Let n be odd and $n \geq 3$, let $\epsilon > 0$ and let $\mathcal{F}$ be a family of balanced 2-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \geq 2(1 + 1/(n-1))(\log n + \log \epsilon^{-1})$. Then $\mathcal{F}$ is a minority isolating family with probability at least $1 - \epsilon$. Moreover, for an odd $n \geq 3$ there exists a minority isolating family of size $\lceil 2(1 + 1/(n-1)) \log n \rceil$.*

Proof. The probability that a random balanced 2-partition isolates a fixed element in the minority block is

$$p_b = \binom{n-1}{\lceil n/2 \rceil} \Big/ \binom{n}{\lceil n/2 \rceil} = \frac{\lfloor n/2 \rfloor}{n} = \frac{n-1}{2n} \quad .$$

Again, in this case the testing goals are the elements of U, so $m = n$ and the result follows from Theorem 3.1. □

For completeness, let us now do the analysis with an even n. In this case, for the probability p that a random 2-partition isolates a fixed element in the smaller (minority) block we have:

$$p = \sum_{0 \leq j < n/2-1} \binom{n-1}{j} \Big/ \binom{n}{2}$$

$$1 - p = \sum_{n/2-1 \leq j < n-1} \binom{n-1}{j} \Big/ \binom{n}{2} = \sum_{0 < j \leq n/2} \binom{n-1}{j} \Big/ \binom{n}{2}$$

Subtracting the first equality from the second one gives us:

$$1 - 2p = \left(\binom{n-1}{n/2-1} + \binom{n-1}{n/2} - 1 \right) \Big/ \binom{n}{2} \quad .$$

By solving for p, using $\left\{ {n \atop 2} \right\} = 2^{n-1} - 1$, and noting that the two binomial coefficients in the expression are equal, the equality simplifies to:

$$p = \left(2^{n-2} - \binom{n-1}{n/2}\right) \Big/ (2^{n-1} - 1) \ .$$

Lemma 4.12. *If n is even and $n \geq 4$, then $p \geq 1/7$.*

Proof. We show equivalently that $1 - 2p \leq 5/7$, which is equivalent to $7\binom{n-1}{n/2} \leq 5 \cdot 2^{n-2} + 1$. We show the latter inequality by induction on n.

If $n = 4$, it is easy to check that the inequality holds. Assume inductively that the inequality holds for an even $n \geq 4$; then for $n + 2$ we have:

$$\begin{aligned} 7\binom{n+1}{(n+2)/2} &= 7 \cdot \frac{4(n+1)}{n+2}\binom{n-1}{n/2} \\ &\leq \frac{4(n+1)}{n+2}(5 \cdot 2^{n-2} + 1) \leq 5 \cdot 2^n + 1 \ . \end{aligned}$$

The last inequality boils down to $3n + 2 \leq 5 \cdot 2^n$, which holds for all n. □

Corollary 4.13. *Let n be even and $n \geq 4$, let $\epsilon > 0$ and let $\mathcal{F}$ be a family of 2-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \geq 7(\log n + \log \epsilon^{-1})$. Then $\mathcal{F}$ is a minority isolating family with probability at least $1 - \epsilon$. Moreover, for an even $n \geq 4$, there exists a minority isolating family of size $\lceil 7 \log n \rceil$.*

Proof. From Lemma 4.12, the probability p that a random 2-partition isolates a fixed element in the minority block satisfies $p \geq 1/7$. Moreover, in this case the testing goals are the elements of U, so $m = n$. The result then follows from Theorem 3.1. □

When n is even, balanced 2-partitions have two blocks of equal size, so we cannot use them for minority isolation. Instead, we use 2-partitions with the smaller block of size $n/2 - 1$ and the larger block of size $n/2 + 1$—we call these *semibalanced* partitions.

Corollary 4.14. *Let n be even and $n \geq 4$, let $\epsilon > 0$ and let $\mathcal{F}$ be a family of semibalanced 2-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \geq 2(1 + 2/(n - 2))(\log n + \log \epsilon^{-1})$. Then $\mathcal{F}$ is a minority isolating family with probability at least $1 - \epsilon$. Moreover, for an even $n \geq 4$, there exists a minority isolating family of size $\lceil 2(1 + 2/(n - 2)) \log n \rceil$.*

Proof. The probability that a random semibalanced 2-partition isolates a fixed element in the minority block is

$$p_b = \binom{n-1}{n/2+1} \Big/ \binom{n}{n/2+1} = \frac{n/2-1}{n} = \frac{n-2}{2n} \ .$$

Again, in this case the testing goals are the elements of U, so $m = n$ and the result follows from Theorem 3.1. □

Example 4.3. Minority isolation was motivated by the etcd example from Chapter 2. The number of nodes in the Jepsen test for etcd was 5; hence according to Corollary 4.11, with 9 randomly generated balanced partitions we will have isolated the leader in the minority block with probability at least 86%.

As in the Kafka example, the isolation here does not expose the inconsistent behavior on its own. We need to also consider read and write operations. As in Example 3.1, assume we only have three operations—read, write 0, and write 1—and each operation can be directed to any of the five nodes. Thus, in total we have 15 operations, each occurring with equal probability.

To expose the inconsistency in Fig. 2.1 (right), the partition first needs to isolate the leader in the minority block, which happens with probability 2/5. As in Example 3.1, this needs to be followed up with a write in the larger block, and read in the smaller block, which happens with probability $(2/15) \cdot (3/15) = 2/75$. Thus, the overall probability of covering a fixed goal is 4/375. Since there are five goals (any node could be the leader), full coverage is possible with approximately 150 tests, and approximately 302 tests will ensure full coverage with probability at least 80%. □

Chapter 5

Testing with Hitting Families

In this chapter, we study the notion of d-hitting coverage, introduced informally in Chapter 2. The corresponding covering families are called d-hitting families in this context.

5.1 Hitting Families of Schedules

We first recall the standard terminology of partial orders, and then proceed to define schedules (linearizations of these partial orders) and hitting families of schedules.

5.1.1 Preliminaries: Partial Orders

A *partial order* (also known as a partially ordered set, or a poset) is a pair $(\mathcal{P}, \leq)$ where $\mathcal{P}$ is a finite set[1] and $\leq$ is a binary relation on $\mathcal{P}$ that is:

1. reflexive: $x \leq x$ for all $x \in \mathcal{P}$,
2. antisymmetric: $x \leq y$ and $y \leq x$ imply $x = y$ for all $x, y \in \mathcal{P}$,
3. transitive: $x \leq y$ and $y \leq z$ imply $x \leq z$ for all $x, y, z \in \mathcal{P}$.

One typically uses $\mathcal{P}$ to refer to $(\mathcal{P}, \leq)$. The *size* of $\mathcal{P}$ is the number of elements in it, $|\mathcal{P}|$.

The relation $x \leq y$ is also written as $x \leq_{\mathcal{P}} y$ and as $y \geq x$. Elements x and y are *comparable* iff $x \leq y$ or $y \leq x$. Otherwise they are *incomparable*, which is written as $x \parallel y$. One writes $x < y$ iff $x \leq y$ and $x \neq y$; if $x < y$, we say the element x is a *predecessor* of y, and y is a *successor* of x. Furthermore, x is an *immediate predecessor* of y (and y is an *immediate successor* of x) if $x < y$ but there is no $z \in \mathcal{P}$ such that $x < z < y$. The *Hasse*

[1] In general, partial orders do not have to be finite, but in this thesis we only study finite partial orders.

diagram of a partial order $\mathcal{P}$ is a directed graph where the set of vertices is $\mathcal{P}$ and an edge (x, y) exists if and only if x is an immediate predecessor of y. Partial orders are sometimes identified with their Hasse diagrams.

For a partial order $(\mathcal{P}, \leq)$, an element $x \in \mathcal{P}$ is said to be *minimal* if no other element is smaller than x, that is, for any $y \in \mathcal{P}$, $y \leq x$ implies $y = x$. Analogously, x is *maximal* if no other element is greater than x, that is, for any $y \in \mathcal{P}$, $y \geq x$ implies $y = x$. We write $\min \mathcal{P}$ and $\max \mathcal{P}$ for the set of minimal and maximal elements of $\mathcal{P}$, respectively.

Partial orders $(\mathcal{P}_1, \leq_1)$ and $(\mathcal{P}_2, \leq_2)$ are *disjoint* if $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$; the *parallel composition* (or *disjoint union*) of such partial orders is the partial order $(\mathcal{P}, \leq)$ where $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$ and $x \leq y$ iff $x, y \in \mathcal{P}_k$ for some $k \in \{1, 2\}$ and $x \leq_k y$. In this partial order, which we denote by $\mathcal{P}_1 \parallel \mathcal{P}_2$, any two elements not coming from a single $\mathcal{P}_k$ are incomparable: $x_1 \in \mathcal{P}_1$ and $x_2 \in \mathcal{P}_2$ imply $x_1 \parallel x_2$.[2]

For a partial order $(\mathcal{P}, \leq)$ and a subset $\mathcal{Q} \subseteq \mathcal{P}$, the *restriction* of $(\mathcal{P}, \leq)$ to $\mathcal{Q}$ is the partial order $(\mathcal{Q}, \leq_{\mathcal{Q}})$ in which, for all $x, y \in \mathcal{Q}$, $x \leq_{\mathcal{Q}} y$ if and only if $x \leq y$. Instead of $\leq_{\mathcal{Q}}$ one usually writes $\leq$, thus denoting the restriction by $(\mathcal{Q}, \leq)$. We also say that the partial order $\mathcal{P}$ is an *extension* of the partial order $\mathcal{Q}$ and that $\mathcal{P}$ *contains* $\mathcal{Q}$. Note that $\mathcal{P} \setminus \mathcal{Q}$ is a restriction of $\mathcal{P}$. We say that $\mathcal{Q}$ is a *prefix* of $\mathcal{P}$ if $\mathcal{Q} \subseteq \mathcal{P}$ and $\mathcal{Q}$ is *downward closed*: for every $x \in \mathcal{Q}$ and $y \in \mathcal{P}$, if $y \leq x$, then $y \in \mathcal{Q}$. In general, partial orders $(\mathcal{P}_1, \leq_1)$ and $(\mathcal{P}_2, \leq_2)$ are *isomorphic* iff there exists an isomorphism $f : \mathcal{P}_1 \to \mathcal{P}_2$: a bijective mapping that respects the ordering, i.e., with $x \leq_1 y$ iff $f(x) \leq_2 f(y)$ for all $x, y \in \mathcal{P}_1$. Containment of partial orders is usually understood up to isomorphism.

5.1.2 Schedules and Their Families

A partial order is *linear* (or total) if all its elements are pairwise comparable. A linearization (linear extension) of the partial order $(\mathcal{P}, \leq)$ is a partial order of the form $(\mathcal{P}, \leq')$ that is linear and has $\leq'$ which is a superset of $\leq$. We call linearizations (linear extensions) of $\mathcal{P}$ *schedules*. In other words, a schedule α is a permutation of the elements of $\mathcal{P}$ that *respects* $\mathcal{P}$, i.e., *respects* all constraints of the form $x \leq y$ from $\mathcal{P}$: for all pairs $x, y \in \mathcal{P}$, whenever $x \leq_{\mathcal{P}} y$, it also holds that $x \leq_\alpha y$. We denote the set of all possible schedules by $S(\mathcal{P})$; a *family* of schedules for $\mathcal{P}$ is simply a subset of $S(\mathcal{P})$.

In what follows, we often treat schedules as words and families of schedules as languages. Indeed, let $\mathcal{P}$ have n elements $\{v_1, \ldots, v_n\}$, then any schedule α can be viewed as a word of length n over the alphabet $\{v_1, \ldots, v_n\}$ where each letter occurs exactly once. We say that α *schedules* elements in the order of occurrences of letters in the word that

[2]Note that we are using the same symbol to denote the parallel composition $\mathcal{P} \parallel \mathcal{Q}$ of partial orders $\mathcal{P}$ and $\mathcal{Q}$, and the statement $x \parallel y$ that the elements x and y are incomparable. The meaning will be clear from the context.

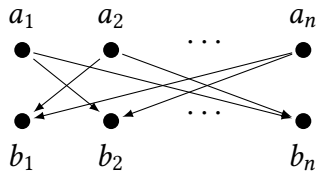

Figure 5.1: The standard example $\mathcal{S}_n$ with $2n$ elements

represents it.

Suppose α_1 and α_2 are schedules for disjoint partial orders $\mathcal{P}_1$ and $\mathcal{P}_2$; then $\alpha_1 \cdot \alpha_2$ is a schedule for the partial order $\mathcal{P}_1 \parallel \mathcal{P}_2$ that first schedules all elements from $\mathcal{P}_1$ according to α_1 and then all elements from $\mathcal{P}_2$ according to α_2. Note that we will use the $\cdot$ to concatenate schedules (as well as individual elements); since some of our partially ordered sets will contain strings, concatenation "inside" an element will be denoted simply by juxtaposition.

5.1.3 Admissible Tuples and Hitting Families

Fix a partial order $\mathcal{P}$ and let $\boldsymbol{a} = (a_1, \ldots, a_d)$ be a tuple of $d \geq 2$ distinct elements of $\mathcal{P}$; we call such tuples *d-tuples*. Suppose α is a schedule for $\mathcal{P}$; then the schedule α *hits* the tuple $\boldsymbol{a}$ if the restriction of α to the set $\{a_1, \ldots, a_d\}$ is the sequence $a_1 \cdot \ldots \cdot a_d$. Note that for a tuple $\boldsymbol{a}$ to have a schedule that hits $\boldsymbol{a}$, it is necessary and sufficient that $\boldsymbol{a}$ respects $\mathcal{P}$; this condition is equivalent to the condition that $a_i \leq a_j$ or $a_i \parallel a_j$ whenever $1 \leq i \leq j \leq d$. We call d-tuples satisfying this condition *admissible*.

Definition 5.1 (d-hitting family)**.** A family of schedules $\mathcal{F}$ for $\mathcal{P}$ is *d-hitting* if for every admissible d-tuple $\boldsymbol{a}$ there is a schedule $\alpha \in \mathcal{F}$ that hits $\boldsymbol{a}$.

Hitting families are yet another instance of covering families from Chapter 3: given a partial order $\mathcal{P}$, the tests here are all schedules for $\mathcal{P}$, the testing goals are all admissible d-tuples, and a schedule α covers a d-tuple $\boldsymbol{a}$ if α hits $\boldsymbol{a}$. However, unlike in Chapter 4 where most of the results were obtained by invoking the general construction of Theorem 3.1, here we will need more involved constructions to get small hitting families. Indeed, since there are at most $\binom{n}{d} d! \leq n^d$ admissible d-tuples, if there existed a lower bound $p > 0$ on the probability of hitting a fixed d-tuple depending only on d, Theorem 3.1 would give us an upper bound on the size of a d-hitting family of the form $O(f(d) \log n)$. Unfortunately, as the next example shows, there is a partial order with $2n$ elements whose smallest 2-hitting family contains n schedules.

Example 5.1 (standard example)**.** Consider a partial order $\mathcal{S}_n$, known as the *standard example* (Dushnik and Miller 1941; Trotter 2001), whose Hasse diagram is shown in Figure 5.1. The partial order consists of $2n$ elements $a_1, \ldots, a_n, b_1, \ldots, b_n$ and the ordering constraints $a_i < b_j$ if and only if $i \neq j$. Note that for every i, the elements a_i and b_i are incomparable; hence the pair (b_i, a_i) is admissible. Any schedule that hits (b_i, a_i) necessarily schedules a_j before b_i and a_i before b_j for all $j \neq i$. Thus, such a schedule does not hit any other pair (b_j, a_j) for $j \neq i$. In other words, we need at least n schedules to hit the pairs (b_i, a_i) for $1 \leq i \leq n$. It is not difficult to see that with n schedules we can hit all other admissible pairs; therefore, $\mathcal{S}_n$ has a 2-hitting family of size n, and this 2-hitting family is optimal. □

For $d = 2$, the size of the smallest 2-hitting family is known as the *dimension* of the partial order (Dushnik and Miller 1941; Trotter 2001). Example 5.1 shows a partial order with $O(n)$ elements of dimension n. Computing and even approximating the dimension for general partial orders are known to be hard problems (Yannakakis 1982; Hegde and Jain 2007; Chalermsook, Laekhanukit, and Nanongkai 2013), and the hardness results directly translate to computing and approximating the size of the optimal d-hitting families for $d \geq 3$.

As we have noticed earlier, a partial order with n elements has at most $\binom{n}{d} d! \leq n^d$ admissible d-tuples, which gives an upper bound of $O(n^d)$ on the size of the optimal d-hitting family. As it turns out, there are two simple constructions that slightly improve this bound: one for general partial orders, and one for partial orders of dimension 2.

Proposition 5.1. *For any $d \geq 2$, a partial order $\mathcal{P}$ with n elements has a d-hitting family of size $O(n^{d-1})$.*

Proof. Group all admissible d-tuples $\boldsymbol{a} = (a_1, \ldots, a_d)$ into sets that agree on $a_1, \ldots, a_{d-1}$. For each set, construct a schedule $\alpha = \alpha(a_1, \ldots, a_{d-1})$ as follows. First schedule $a_1, \ldots, a_{d-1}$: that is, start with an empty sequence of elements, iterate over $k = 1, \ldots, d-1$, and, for each k, append to the sequence all elements $x \in \mathcal{P}$ such that $x \leq a_k$. The order in which these $x$es are appended is not important as long as it respects the partial order $\mathcal{P}$. Elements that are predecessors of several a_k are only scheduled once, for the least k. Note that no a_k, $1 \leq k \leq d$, is a predecessor of any a_j for $j < k$, because otherwise the d-tuple $\boldsymbol{a} = (a_1, \ldots, a_d)$ would not be admissible. After the elements $a_1, \ldots, a_{d-1}$ and their predecessors have been scheduled, schedule the remaining elements in any order that respects $\mathcal{P}$. By construction, α hits all admissible d-tuples that agree on $a_1, \ldots, a_{d-1}$; collecting all such schedules for all possible $a_1, \ldots, a_{d-1}$ makes a d-hitting family for $\mathcal{P}$ of size at most n^{d-1}. □

If the partial order is additionally of dimension 2, we can do even better, as the next proposition shows.

Proposition 5.2. *For any $d \geq 2$, a partial order $\mathcal{P}$ of dimension 2 with n elements has a d-hitting family of size $O(n^{d-2})$.*

Proof. Let the two schedules that hit all admissible pairs be λ and ρ. Similarly as in the proof of Proposition 5.1, we group all admissible d-tuples $\boldsymbol{a} = (a_1, \ldots, a_d)$ into sets agreeing on $a_1, \ldots, a_{d-2}$. Instead of a single schedule per set, here we construct a pair of schedules $\lambda' = \lambda(a_1, \ldots, a_{d-2})$ and $\rho' = \rho(a_1, \ldots, a_{d-2})$. In both λ' and ρ', we first schedule $a_1, \ldots, a_{d-2}$ as before: start with an empty sequence of elements, iterate over $k = 1, \ldots, d-2$, and, for each k, append to the sequence all previously unscheduled elements $x \in \mathcal{P}$ such that $x \leq a_k$. After the elements $a_1, \ldots, a_{d-2}$ and their predecessors have been scheduled, the remaining elements are scheduled in λ' according to λ and in ρ' according to ρ. As a result, these two schedules hit all admissible d-tuples that agree on $a_1, \ldots, a_{d-2}$; collecting all such schedules for all possible $a_1, \ldots, a_{d-2}$ makes a d-hitting family for $\mathcal{P}$ of size at most $2n^{d-2}$. □

In the rest of the chapter we focus on specific partial orders for which we can show existence of small d-hitting families. We shall return to general partial orders in Chapter 6.

5.2 Specific Partial Orders

The specific partial orders we shall inspect are chains and antichains, series-parallel partial orders, and trees. We study chains and antichains primarily for theoretical reasons: they are contained in other partial orders, so any lower bounds we show for them automatically translate to other partial orders. On the other hand, while trees and series-parallel orders are also interesting from the theoretical standpoint, they arise in practice as execution models of concurrent programs with asynchronous task creation and fork-join constructs.

Of all the specific partial orders we have mentioned, series-parallel orders are the most general: they subsume chains, antichains, and trees. Their main property is that their dimension is two; we shall exploit this in Section 5.3 to construct near-optimal 3-hitting families for series-parallel orders. In Section 5.4 we give a general construction of d-hitting families for trees.

5.2.1 Chains and Antichains

A partial order is a *chain* if it is linear, that is, if any two elements are comparable. On the other hand, an *antichain* is a partial order where every two elements are incomparable. For a partial order $\mathcal{P}$, its *height* (sometimes called *length*) is the maximal cardinality of a chain it contains, and its *width* is the maximal cardinality of an antichain it contains.

Chain

Consider a chain of n elements: $C_n = \{1, \ldots, n\}$ with $1 < 2 < \ldots < n$. This partial order has a unique schedule: $\alpha = 1 \cdot 2 \cdot \ldots \cdot n$; a d-tuple $\boldsymbol{a} = (a_1, \ldots, a_d)$ is admissible iff $a_1 < \ldots < a_d$, and α hits all such d-tuples. Thus, for any d, the family $\mathcal{F} = \{\alpha\}$ is a d-hitting family for C_n.

Chain With Independent Element

Consider $C_n \parallel \{\dagger\}$, the parallel composition of a chain C_n and a singleton $\{\dagger\}$. There are $n + 1$ possible schedules, depending on how $\dagger$ is positioned with respect to the chain: $\alpha_0 = \dagger \cdot 1 \cdot 2 \cdot \ldots \cdot n$, $\alpha_1 = 1 \cdot \dagger \cdot 2 \cdot \ldots \cdot n$, …, $\alpha_n = 1 \cdot 2 \cdot \ldots \cdot n \cdot \dagger$. For $d = 2$, admissible pairs are of the form (i, j) with $i < j$, $(\dagger, i)$, and $(i, \dagger)$ for all $1 \leq i \leq n$; the family $\mathcal{F}_2 = \{\alpha_0, \alpha_n\}$ is the smallest 2-hitting family. Now consider $d = 3$. Note that all triples $(i, \dagger, i + 1)$ with $1 \leq i < n$, as well as $(\dagger, 1, 2)$ and $(n - 1, n, \dagger)$, are admissible, and each of them is hit by a unique schedule. Therefore, the smallest 3-hitting family of schedules consists of all $n + 1$ schedules: $\mathcal{F}_3 = \{\alpha_0, \ldots, \alpha_n\}$. For $d \geq 4$, it remains to observe that every d-hitting family is necessarily d'-hitting for $2 \leq d' \leq d$, hence $\mathcal{F}_3$ is optimal for all $d \geq 3$.

The following proposition is an important consequence of the analysis we have just made.

Proposition 5.3. *For any $d \geq 3$ and any partial order $\mathcal{P}$, every d-hitting family must contain at least $m + 1$ schedules, where m denotes the maximum number n such that $\mathcal{P}$ contains $C_n \parallel \{\dagger\}$. This m is upper-bounded (and this upper bound is tight) by the height of $\mathcal{P}$.* □

Antichain

Let $\mathcal{A}_n = \{e_1\} \parallel \{e_2\} \parallel \ldots \parallel \{e_n\}$ be an antichain with n elements. In Example 3.3, we have already derived an upper bound on the size of optimal d-hitting families for $\mathcal{A}_n$. In particular, there exists a d-hitting family of size $d!(1 + \log(n^{\underline{d}} - (n - d)^{\underline{d}}))$, which is asymptotic to $d!(d - 1) \log n$. Note that this bound is suboptimal for $d = 2$. Consider a family $\mathcal{F} = \{\lambda, \rho\}$, where $\lambda = e_1 \cdot e_2 \cdot \ldots \cdot e_n$ and $\rho = e_n \cdot e_{n-1} \cdot \ldots \cdot e_1$. It is not difficult to see that $\mathcal{F}$ is a 2-hitting family for $\mathcal{A}_n$ with only 2 schedules. This also shows that the dimension of $\mathcal{A}_n$ is two.

As for $d \geq 3$, we show that the upper bound from Example 3.2 is optimal up to a constant factor. More precisely, we have the following theorem.

Theorem 5.4. *For any $d \geq 3$, the smallest d-hitting family for $\mathcal{A}_n$ has size between $g(d) \log n - O(1)$ and $f(d) \log n + O(1)$, where $g(d) \geq d/(2 \log(2d + 2))$ and $f(d) \leq d!(d - 1)$.*

Proof. It remains to show the lower bound, that is, that any d-hitting family for $\mathcal{A}_n$ contains at least $g(d)\log n - O(1)$ schedules, where $g(d) \geq d/(2\log(2d+2))$. To see this, denote $r = \lfloor (d-1)/2 \rfloor \geq 1$ and observe that $2r+1 \leq d$. Take any d-hitting family $\mathcal{F} = \{\alpha_1, \ldots, \alpha_k\}$ and consider the following matrix $B = (b_{ij})$ of size $k \times (n-1)^{\underline{r}}$ (recall, $(n-1)^{\underline{r}} = (n-1)(n-2)\cdots(n-r)$ is the falling factorial power). The columns of B are indexed by all r-tuples of distinct elements from $\{e_1, \ldots, e_{n-1}\}$, of which there are exactly $(n-1)^{\underline{r}}$. Let $(a_1, \ldots, a_r)$ be the jth such tuple; then the entry b_{ij} is the number of elements from $\{a_1, \ldots, a_r\}$ that the schedule α_i places before e_n.

We claim that all columns of B are pairwise distinct. Indeed, if for some $j' \neq j''$ and all i it holds that $b_{ij'} = b_{ij''}$, then, for all $s \in \{0, \ldots, r\}$, no schedule from $\mathcal{F}$ can place exactly s elements from the j'th tuple before e_n without also placing exactly s elements from the j''th tuple before e_n, and vice versa. Since the j'th and j''th r-tuples—call them $\boldsymbol{a}'$ and $\boldsymbol{a}''$—are different, this implies that $\mathcal{F}$ cannot be d-hitting. Indeed, in the case where $\boldsymbol{a}'$ and $\boldsymbol{a}''$ have no elements in common, this is obvious: consider any d-tuple where all elements from $\boldsymbol{a}'$ come before e_n and all elements from $\boldsymbol{a}''$ after e_n. But if $\boldsymbol{a}'$ and $\boldsymbol{a}''$ have, say, $\ell > 0$ elements in common, then putting all the elements of $\boldsymbol{a}'$ before e_n and the remaining $r-\ell$ elements of $\boldsymbol{a}''$ after e_n produces a d-tuple that avoids getting hit by schedules from $\mathcal{F}$ (note that $r > \ell$ as $\boldsymbol{a}'$ and $\boldsymbol{a}''$ are different).

Now, since each b_{ij} can only assume values from the set $\{0, 1, \ldots, r\}$, it follows that B cannot have more than $(r+1)^k$ columns. Therefore, $(n-1)^{\underline{r}} \leq (r+1)^k$, and so $k \geq \log(n-1)^{\underline{r}} / \log(r+1)$. Recall that $x^{\underline{r}} = \binom{x}{r} \cdot r! \geq (x/r)^r \cdot r!$; we have

$$\begin{aligned} k &\geq \log\left(\left(\frac{n-1}{r}\right)^r \cdot r!\right) \Big/ \log(r+1) \\ &= \frac{r\log(n-1) - r\log r + \log r!}{\log(r+1)} \\ &= \frac{r}{\log(r+1)} \cdot \log(n-1) + w(r) \end{aligned}$$

where

$$w(r) = \frac{\log r! - r\log r}{\log(r+1)} \approx \frac{-r + (\log r + \log\pi + \log 2)/2}{\log(r+1)}.$$

Substituting $r = \lfloor (d-1)/2 \rfloor$ gives the desired result, because

$$\begin{aligned} \frac{r}{\log(r+1)} &= \frac{\lfloor \frac{d-1}{2} \rfloor}{\log\left(\lfloor \frac{d-1}{2} \rfloor + 1\right)} \geq \frac{\frac{d-2}{2}}{\log\left(\frac{d-1}{2} + 1\right)} \\ &= \frac{d-2}{2\log\left(\frac{d+1}{2}\right)} = \frac{d-2}{2\log(d+1) - 2\log 2} \geq \frac{d}{2\log(2d+2)} \end{aligned}$$

and $\log(n-1) = \log(n \cdot (1-1/n)) \geq \log n - 1$ for $n \geq d \geq 3$. □

5.2.2 Series-Parallel Orders

Series-parallel orders are defined inductively as follows.

singleton A singleton $\mathcal{S} = \{\bullet\}$ with $\bullet \leq \bullet$ is a series-parallel order. When it does not cause confusion, we also write $\bullet$ to refer to $\mathcal{S}$.

serial composition For two disjoint series-parallel orders $\mathcal{P}$ and $\mathcal{Q}$, the serial composition $\mathcal{P} \cdot \mathcal{Q}$ is a series-parallel order. The serial composition is defined as follows: $\mathcal{P} \cdot \mathcal{Q} = (\mathcal{P} \cup \mathcal{Q}, \leq)$, where $x \leq y$ if and only if $x \in \mathcal{P}$ and $y \in \mathcal{Q}$, or $x \leq y$ in $\mathcal{P}$, or $x \leq y$ in $\mathcal{Q}$.

parallel composition For two disjoint series-parallel orders $\mathcal{P}$ and $\mathcal{Q}$, the parallel composition $\mathcal{P} \parallel \mathcal{Q}$ is a series-parallel order. Recall that the parallel composition is defined as follows: $\mathcal{P} \parallel \mathcal{Q} = (\mathcal{P} \cup \mathcal{Q}, \leq)$, where $x \leq y$ if and only if $x \leq y$ in $\mathcal{P}$ or $x \leq y$ in $\mathcal{Q}$.

It is not difficult to see that the serial composition is associative, and the parallel composition is associative and commutative. Moreover, we can represent a series-parallel order of size n as an algebraic expression built using n disjoint copies of the singleton $\mathcal{S}$ and the operations $\cdot$ and $\parallel$. Such a representation is not necessarily unique; however, if we "flatten" the nested occurrences of both $\cdot$ and $\parallel$ by applying associativity, the representation becomes unique up to the order of the partial orders joined by the parallel composition. We refer to the flattened algebraic expression for $\mathcal{P}$ as the *canonical representation* of $\mathcal{P}$. In the canonical representation of $\mathcal{P}$, each maximal subexpression of the form $\mathcal{P}_1 \parallel \ldots \parallel \mathcal{P}_k$ for $k \geq 2$ is considered a single application of the parallel composition, and each maximal subexpression of the form $\mathcal{P}_1 \cdot \ldots \cdot \mathcal{P}_k$ for $k \geq 2$ is considered a single application of the serial composition.

Two quantities related to the canonical representation of $\mathcal{P}$ will be useful for describing properties of $\mathcal{P}$. The first one, denoted $\Gamma_{\mathcal{P}}$, is defined as the number of parallel compositions in the canonical representation of $\mathcal{P}$. The second quantity, denoted as $\Delta_{\mathcal{P}}$, is defined as the largest number k such that $\mathcal{P}_1 \parallel \ldots \parallel \mathcal{P}_k$ appears as a subexpression in the canonical representation of $\mathcal{P}$, or 1 if the canonical representation does not contain parallel composition. In other words, $\Delta_{\mathcal{P}}$ is the largest number of partial orders composed in a single application of the parallel composition while constructing $\mathcal{P}$.

Clearly, chains and antichains are series-parallel orders. The chain C_n of size n is a serial composition of n copies of a singleton, and the antichain $\mathcal{A}_n$ of size n is a parallel composition of n copies of a singleton. Thus, $\Gamma_{C_n} = 0$, $\Delta_{C_n} = 1$, $\Gamma_{\mathcal{A}_n} = 1$, and $\Delta_{\mathcal{A}_n} = n$.

For a partial order $\mathcal{P}$ (not necessarily series-parallel), we define the notion of *layers* of elements. The minimal elements of $\mathcal{P}$ form the 0th layer, and a non-minimal element

$x \in \mathcal{P}$ is on the $(k+1)$th layer if k is the largest number so that x has an immediate predecessor on the kth layer. We use $\ell_{\mathcal{P}}(x)$ to denote the layer of $x \in \mathcal{P}$. Clearly, a partial order of height h has exactly h layers. We usually write $\ell(x)$ instead of $\ell_{\mathcal{P}}(x)$ if $\mathcal{P}$ is understood from the context.

Lemma 5.5. *Let $\mathcal{P}$ be a partial order and $x, y \in \mathcal{P}$ two different elements. If $\ell(x) = \ell(y)$, then $x \parallel y$.*

Proof. First, note that for any $x', y' \in \mathcal{P}$, if $x' \leq y'$, then $\ell(x') \leq \ell(y')$. (This is easily shown by induction on $\ell(y')$.) We can now prove the claim by contraposition. Without loss of generality, let $x < y$. Then the set $\{y' \mid x \leq y' < y\}$ is nonempty, so it contains a maximal element, say y'. Clearly, y' immediately precedes y, so we have $\ell(x) \leq \ell(y') < \ell(y)$. □

We will need two important properties of series-parallel orders. The first one is that their dimension is at most 2.

Proposition 5.6. *The smallest 2-hitting family of schedules for a series-parallel order $\mathcal{P}$ has size at most 2, that is, the dimension of $\mathcal{P}$ is at most 2.*

Proof. Fix a non-flattened algebraic representation of $\mathcal{P}$. We use it to construct $\mathcal{F}_{\mathcal{P}} = \{\lambda_{\mathcal{P}}, \rho_{\mathcal{P}}\}$, where $\lambda_{\mathcal{P}}$ and $\rho_{\mathcal{P}}$ are two schedules defined recursively as follows. If $\mathcal{P}$ is a singleton $\{\bullet\}$, then $\lambda_\bullet = \rho_\bullet = \bullet$. If $\mathcal{P}$ is composed of $\mathcal{P}'$ and $\mathcal{Q}'$, let $\lambda_{\mathcal{P}'}, \rho_{\mathcal{P}'}$ and $\lambda_{\mathcal{Q}'}, \rho_{\mathcal{Q}'}$ be the schedules for $\mathcal{P}'$ and $\mathcal{Q}'$ defined using the same fixed algebraic representation. Then, if $\mathcal{P} = \mathcal{P}' \cdot \mathcal{Q}'$, we define $\lambda_{\mathcal{P}} = \lambda_{\mathcal{P}'} \cdot \lambda_{\mathcal{Q}'}$ and $\rho_{\mathcal{P}} = \rho_{\mathcal{P}'} \cdot \rho_{\mathcal{Q}'}$. Otherwise, if $\mathcal{P} = \mathcal{P}' \parallel \mathcal{Q}'$, we define $\lambda_{\mathcal{P}} = \lambda_{\mathcal{P}'} \cdot \lambda_{\mathcal{Q}'}$ and $\rho_{\mathcal{P}} = \rho_{\mathcal{Q}'} \cdot \rho_{\mathcal{P}'}$ (note the reversal of the schedules).

We show that $\mathcal{F}_{\mathcal{P}}$ is a 2-hitting family by induction. Clearly this is true for a singleton. Assume the claim holds for some $\mathcal{P}'$ and $\mathcal{Q}'$ such that the algebraic representations of $\mathcal{P}' \cdot \mathcal{Q}'$ or $\mathcal{P}' \parallel \mathcal{Q}'$ appear as subexpressions of the fixed algebraic representation of $\mathcal{P}$. Then $\mathcal{F}_{\mathcal{P}'} = \{\lambda_{\mathcal{P}'}, \rho_{\mathcal{P}'}\}$ and $\mathcal{F}_{\mathcal{Q}'} = \{\lambda_{\mathcal{Q}'}, \rho_{\mathcal{Q}'}\}$ are 2-hitting families for $\mathcal{P}'$ and $\mathcal{Q}'$, respectively. Now, if $\mathcal{P}$ is a serial or parallel composition of $\mathcal{P}'$ and $\mathcal{Q}'$ and (x, y) is an admissible pair in $\mathcal{P}$, then either both x and y are in $\mathcal{P}'$ or $\mathcal{Q}'$, in which case one of $\lambda_{\mathcal{P}}$ and $\rho_{\mathcal{P}}$ hits the pair by the inductive hypothesis, or one element is in $\mathcal{P}'$ and the other in $\mathcal{Q}'$, in which case one of $\lambda_{\mathcal{P}}$ and $\rho_{\mathcal{P}}$ hits the pair by construction. □

The second property we need is that a series-parallel order cannot contain the "zig-zag" order, whose Hesse diagram is shown in Figure 5.2. The zig-zag order on elements a, b, c, d is characterized by having only three ordering constraints: $a \leq b$, $c \leq b$, and $c \leq d$.

Lemma 5.7. *Let $\mathcal{P}$ be a series-parallel order. Then $\mathcal{P}$ cannot be an extension of the zig-zag order.*

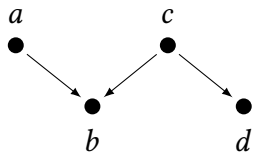

Figure 5.2: The "zig-zag" order with ordering constraints $a \leq b$, $c \leq b$, and $c \leq d$.

Proof. The following short argument is taken from Valdes, Tarjan, and Lawler (1982): suppose $\mathcal{P}$ contains the zig-zag order as a restriction. Since $\mathcal{P}$ is not a singleton, it is a composition of two components $\mathcal{P}'$ and $\mathcal{Q}'$. If the zig-zag is fully contained in either of the components, we can restrict ourselves to that component and repeat the argument. Therefore, we may assume that the components $\mathcal{P}'$ and $\mathcal{Q}'$ partition the zig-zag order into two disjoint blocks. However, notice that no matter how we partition the four elements, there are always two incomparable and two comparable elements in different blocks. Hence $\mathcal{P}$ can be neither the serial nor the parallel composition of $\mathcal{P}'$ and $\mathcal{Q}'$—a contradiction. □

In fact, the converse of Lemma 5.7 also holds. In other words, series-parallel orders are fully characterized by the absence of the zig-zag order, that is, a partial order is series-parallel if and only if it does not contain the zig-zag order (Valdes, Tarjan, and Lawler 1982).

5.2.3 Binary Semilattices

In this section we study a somewhat technical class of partial orders—*binary semilattices*—which are needed in the construction presented in Section 5.3.

Definition 5.2. A partial order $\mathcal{P}$ is a *semilattice* if every pair of elements has an *infimum*, that is, for every $x, y \in \mathcal{P}$ there exists an element $x \wedge y \in \mathcal{P}$ such that $x \wedge y \leq x$ and $x \wedge y \leq y$, and for any other element $z \in \mathcal{P}$ such that $z \leq x$ and $z \leq y$, we have $z \leq x \wedge y$. In a *binary semilattice* additionally every element has at most 2 immediate successors.

Note that each semilattice $\mathcal{P}$ has the *least element*, denoted $\bot_{\mathcal{P}}$ or $\bot$ for short, with the property that $\bot < x$ for every other $x \in \mathcal{P}$.[3] If a series-parallel order $\mathcal{P}$ is also a semilattice, then it cannot be a parallel composition. More precisely, in this case $\mathcal{P}$ is

[3]In order for the least element to exist, a sufficient condition is that the semilattice is finite; then the least element is simply the infimum of all the elements in the semilattice. Recall that all partial orders in this thesis are finite.

either a singleton, or it has the form $\{\bot\} \cdot \mathcal{P}'$ for a series-parallel order $\mathcal{P}'$ (which is not necessarily a semilattice).

The reason we study binary semilattices is that we will need to be able to extend series-parallel orders to series-parallel binary semilattices. Extending a series-parallel order $\mathcal{P}$ to a series-parallel binary semilattice $\mathcal{P}'$ may incur an increase in height. The next lemma gives an upper bound on the height of $\mathcal{P}'$ in terms of $\Gamma_{\mathcal{P}}$ and $\Delta_{\mathcal{P}}$.

Lemma 5.8. *Let $\mathcal{P}$ be a series-parallel order of height h. Then $\mathcal{P}$ can be extended to a series-parallel binary semilattice $\mathcal{P}'$ of height at most $h + \Gamma_{\mathcal{P}} \lceil \log_2 \Delta_{\mathcal{P}} \rceil$.*

Proof. We construct the extension $\mathcal{P}'$ by induction on the structure of $\mathcal{P}$. If $\mathcal{P}$ is a singleton, no extension is needed. Assume for some $k \geq 2$ and series-parallel orders $\mathcal{P}_0, \ldots, \mathcal{P}_{k-1}$ that each $\mathcal{P}_i$ for $0 \leq i < k$ can be extended to a series-parallel binary semilattice $\mathcal{P}'_i$ of height at most $h_i + \Gamma_i \lceil \log_2 \Delta_i \rceil$, where $\Gamma_i = \Gamma_{\mathcal{P}_i}$, $\Delta_i = \Delta_{\mathcal{P}_i}$, and $\mathcal{P}_i$ has height h_i. Let $\mathcal{P}$ be a composition of $\mathcal{P}_0, \ldots, \mathcal{P}_{k-1}$. If the composition is serial, that is, $\mathcal{P} = \mathcal{P}_0 \cdot \ldots \cdot \mathcal{P}_{k-1}$, then it is easy to verify that $\mathcal{P}' = \mathcal{P}'_0 \cdot \ldots \cdot \mathcal{P}'_{k-1}$ is a series-parallel binary semilattice. To establish the bound on the height h' of $\mathcal{P}'$, note that we have

$$h' \leq \sum_{0 \leq i < k} h_i + \Gamma_i \lceil \log_2 \Delta_i \rceil \leq h + \sum_{0 \leq i < k} \Gamma_i \lceil \log_2 \Delta_{\mathcal{P}} \rceil = h + \Gamma_{\mathcal{P}} \lceil \log_2 \Delta_{\mathcal{P}} \rceil \ .$$

The crux of the proof is the case when the composition is parallel, that is, $\mathcal{P} = \mathcal{P}_0 \parallel \ldots \parallel \mathcal{P}_{k-1}$. Assume without loss of generality that no $\mathcal{P}_i$ can be further decomposed as a parallel composition. We start by extending $\mathcal{P}$ to $\mathcal{P}' = \mathcal{P}'_0 \parallel \ldots \parallel \mathcal{P}'_{k-1}$. Note that if h' is the height of $\mathcal{P}'$, at this point we have the following upper bound:

$$h' \leq \max_{0 \leq i < k} \{h_i + \Gamma_i \lceil \log_2 \Delta_i \rceil\} \leq h + (\Gamma_{\mathcal{P}} - 1) \lceil \log_2 \Delta_i \rceil \ .$$

Next, we saturate $\mathcal{P}'$ with layers of new infima with two immediate successors by applying the following transformation. Introduce $\lfloor k/2 \rfloor$ new elements $x_0, \ldots, x_{\lfloor k/2 \rfloor - 1}$, and extend $\mathcal{P}'$ by replacing each parallel composition $\mathcal{P}'_{2i} \parallel \mathcal{P}'_{2i+1}$ in $\mathcal{P}'$ with $\mathcal{Q}_i = \{x_i\} \cdot (\mathcal{P}'_{2i} \parallel \mathcal{P}'_{2i+1})$ for $0 \leq i < \lfloor k/2 \rfloor$. If k is odd, additionally set $\mathcal{Q}_{\lfloor k/2 \rfloor} = \mathcal{P}'_{k-1}$. Regardless of the parity of k, the new $\mathcal{P}'$ is $\mathcal{P}' = \mathcal{Q}_0 \parallel \ldots \parallel \mathcal{Q}_{\lceil k/2 \rceil - 1}$, which is a parallel composition of $\lceil k/2 \rceil$ series-parallel orders, none of which is a parallel composition itself. We continue applying the transformation on $\mathcal{P}'$ until it is no longer a parallel composition.

Let us analyze the effect of a single iteration of the transformation. First note that each $\mathcal{Q}_i$ for $0 \leq i < \lfloor k/2 \rfloor$ is a binary semilattice: let $x, y \in \mathcal{Q}_i$ be two elements; if they are both in $\mathcal{P}'_{2i}$ or $\mathcal{P}'_{2i+1}$, the existence of their infimum follows from the inductive hypothesis. Otherwise they are in different components or one of them is x_i, so their infimum is x_i. Furthermore, by the inductive hypothesis, elements in $\mathcal{P}'_{2i}$ and $\mathcal{P}'_{2i+1}$ have at most 2

immediate successors. Note that x_i has exactly 2 immediate successors: the least elements $\bot_{\mathcal{P}'_{2i}}$ and $\bot_{\mathcal{P}'_{2i+1}}$ in $\mathcal{P}'_{2i}$ and $\mathcal{P}'_{2i+1}$, respectively. If k is odd, $\mathcal{Q}_{\lfloor k/2 \rfloor}$ is a binary semilattice by the inductive hypothesis. By induction on the number of iterations of the transformation, we conclude that once $\mathcal{P}'$ is no longer a parallel composition, $\mathcal{P}'$ is a binary semilattice, as required.

Now let us count the total number of iterations; let this number be l. Then l is the least number such that

$$\underbrace{\lceil\lceil\cdots\lceil\lceil k/2\rceil/2\rceil\cdots\rceil/2\rceil}_{l \text{ times}} \leq 1 \ .$$

In other words, l is the least number such that $k \leq 2^l$, that is, $l = \lceil \log_2 k \rceil$. Each iteration of the transformation adds a new layer of infima to $\mathcal{P}'$, which may increase the height by 1. Hence, the total contribution of the added infima to the height is at most $\lceil \log_2 k \rceil$, which is in turn at most $\lceil \log_2 \Delta_{\mathcal{P}} \rceil$. Therefore, the final height h' of $\mathcal{P}'$ is upper-bounded by $h + (\Gamma_{\mathcal{P}} - 1)\lceil \log_2 \Delta_{\mathcal{P}} \rceil + \lceil \log_2 \Delta_{\mathcal{P}} \rceil = h + \Gamma_{\mathcal{P}} \lceil \log_2 \Delta_{\mathcal{P}} \rceil$, as required. □

5.2.4 Trees

Trees are a special case of series-parallel orders constructed by applying only the restricted series-parallel composition of the form $C \cdot (\mathcal{T}_1 \parallel \ldots \parallel \mathcal{T}_k)$, where C is a chain and $\mathcal{T}_1, \ldots, \mathcal{T}_k$ for $k \geq 1$ are trees. It immediately follows that a tree is a semilattice. We will identify trees with their Hasse diagrams, and we will use without specific introduction the standard terminology regarding trees: nodes, the root node, leaves, the parent of a node, the children of a node, etc.

Consider a complete binary tree $\mathcal{T}$ of height h. The size of $\mathcal{T}$ is $n = 2^h - 1$, and it has $\lceil n/2 \rceil = 2^{h-1}$ leaves and $\lfloor n/2 \rfloor = 2^{h-1} - 1$ inner nodes. It is not difficult to see that $\Gamma_{\mathcal{T}}$ is precisely the number of inner nodes, i.e., $\Gamma_{\mathcal{T}} = \lfloor n/2 \rfloor$. In other words, for a general tree $\mathcal{T}$, the best asymptotic bound on $\Gamma_{\mathcal{T}}$ we can assume is $\Gamma_{\mathcal{T}} = O(|\mathcal{T}|)$. Now suppose we have a tree $\mathcal{T}$ of height h which is not binary, and we want to extend it to a binary semilattice. Then Lemma 5.8 bounds the height of the extension with $h + O(|\mathcal{T}|) \log_2 \Delta_{\mathcal{T}}$, which is not very useful.

Luckily, it turns out we can altogether ignore the quantity $\Gamma_{\mathcal{T}}$ when it comes to trees. Indeed, we can show a much tighter bound on the height of a binary extension of a tree.

Lemma 5.9. *Let $\mathcal{T}$ be a tree of height h. Then $\mathcal{T}$ can be extended to a binary tree $\mathcal{T}'$ of height at most $h \cdot \lceil \log_2 \Delta_{\mathcal{T}} \rceil$.*

Proof. We construct the extension $\mathcal{T}'$ by induction on the structure of $\mathcal{T}$. If $\mathcal{T}$ is a singleton, no extension is needed. Assume for some $k \geq 2$ and trees $\mathcal{T}_0, \ldots, \mathcal{T}_{k-1}$ that each

$\mathcal{T}_i$ for $0 \leq i < k$ can be extended to a binary tree $\mathcal{T}_i'$ of height at most $h_i \cdot \lceil \log_2 \Delta_i \rceil$, where $\Delta_i = \Delta_{\mathcal{T}_i}$, and $\mathcal{T}_i$ has height h_i. Let $\mathcal{T} = C \cdot (\mathcal{T}_0 \parallel \ldots \parallel \mathcal{T}_{k-1})$ for some chain C.

We proceed similarly as in the proof of Lemma 5.8. First we extend $\mathcal{T}$ to $\mathcal{T}' = C \cdot (\mathcal{T}_0' \parallel \ldots \parallel \mathcal{T}_{k-1}')$. Note that if h' is the height of $\mathcal{P}'$, at this point we have the following upper bound:

$$h' \leq |C| + \max_{0 \leq i < k} \{h_i \cdot \lceil \log_2 \Delta_i \rceil\} \leq |C| + (h - |C|) \lceil \log_2 \Delta_{\mathcal{T}} \rceil \ ,$$

since $h_i \leq h - |C|$ for all i, and $\Delta_i \leq \Delta_{\mathcal{T}}$.

Next, we saturate $\mathcal{T}'$ with layers of new infima with two immediate successors by applying the following transformation. Introduce $\lfloor k/2 \rfloor$ new elements $x_0, \ldots, x_{\lfloor k/2 \rfloor - 1}$, and extend $\mathcal{T}'$ by replacing each parallel composition $\mathcal{T}_{2i}' \parallel \mathcal{T}_{2i+1}'$ in $\mathcal{T}'$ with $\mathcal{Q}_i = \{x_i\} \cdot (\mathcal{T}_{2i}' \parallel \mathcal{T}_{2i+1}')$ for $0 \leq i < \lfloor k/2 \rfloor$. If k is odd, additionally set $\mathcal{Q}_{\lfloor k/2 \rfloor} = \mathcal{T}_{k-1}'$. Regardless of the parity of k, the new $\mathcal{T}'$ is $\mathcal{T}' = C \cdot (\mathcal{Q}_0 \parallel \ldots \parallel \mathcal{Q}_{\lceil k/2 \rceil - 1})$, which is a tree with $\lceil k/2 \rceil$ subtrees. Continue applying the transformation on $\mathcal{T}'$ as long as there are more than two subtrees.

Like before, we first note that each $\mathcal{Q}_i$ is a binary tree, and proceed with counting the number of iterations of the transformation. Let this number be l. Then l is the least number such that

$$\underbrace{\lceil \lceil \cdots \lceil \lceil k/2 \rceil / 2 \rceil \cdots \rceil / 2 \rceil}_{l \text{ times}} \leq 2 \ .$$

In other words, l is the least number such that $k \leq 2^{l+1}$, that is, $l = \lceil \log_2 k \rceil - 1$. Each iteration of the transformation adds a new layer of infima to $\mathcal{T}'$, which may increase the height by 1. Hence, the total contribution to the height is at most $\lceil \log_2 k \rceil - 1$, which is in turn at most $\lceil \log_2 \Delta_{\mathcal{T}} \rceil - 1$. Therefore, the final height h' of $\mathcal{T}'$ is upper-bounded by

$$\begin{aligned} h' &\leq |C| + (h - |C|) \lceil \log_2 \Delta_{\mathcal{T}} \rceil + \lceil \log_2 \Delta_{\mathcal{T}} \rceil - 1 \\ &= h \cdot \lceil \log_2 \Delta_{\mathcal{T}} \rceil - (|C| - 1)(\lceil \log_2 \Delta_{\mathcal{T}} \rceil - 1) \\ &\leq h \cdot \lceil \log_2 \Delta_{\mathcal{T}} \rceil \ , \end{aligned}$$

as required. □

5.3 3-Hitting Families for Series-Parallel Orders

In this section we present the first of the two main results of this chapter: a construction of 3-hitting families for series-parallel orders. For a series-parallel binary semilattice $\mathcal{P}$ of height h, the construction produces a 3-hitting family of size at most $4h$. For a general series-parallel order $\mathcal{P}$, the bound on the size is $4(h + \Gamma_{\mathcal{P}} \cdot \lceil \log_2 \Delta_{\mathcal{P}} \rceil)$. This has the following consequences for partial orders subsumed by series-parallel orders:

1. For an antichain of n elements, the bound collapses to $4\lceil \log_2 n \rceil + 4$, which differs by a constant factor from the bounds obtained in Examples 3.2 and 3.3 and summarized in Theorem 5.4. Unlike those bounds, which rely on the nonconstructive probabilistic method, the new bound follows from an explicit construction.

2. For a binary tree of height h, the bound is $4h$.

3. For a general tree of height h with maximal outdegree Δ, the bound is $4h\lceil \log_2 \Delta \rceil$.

The construction for a series-parallel order $\mathcal{P}$ proceeds in two phases. In the first phase, we extend $\mathcal{P}$ to a series-parallel order $\mathcal{P}'$ which is additionally a binary semilattice. In the second phase, for a series-parallel binary semilattice of height h, we explicitly construct a 3-hitting family of size $4h$. For the first phase we use Lemmas 5.8 and 5.9. The second-phase construction is given by the following theorem.

Theorem 5.10. *The smallest 3-hitting family of schedules for a series-parallel binary semilattice $\mathcal{P}$ of height h has size at most $4h$.*

Proof. Let $\mathcal{F}_2 = \{\lambda, \rho\}$ be a 2-hitting family for $\mathcal{P}$, whose existence is ensured by Proposition 5.6. We use λ as a reference total order on $\mathcal{P}$; in particular, given $x \in \mathcal{P}$, we denote by $s_1(x)$ and $s_2(x)$ the first and the second immediate successor of x in λ, if they exist.

For i such that $0 \le i < h$, we define four schedules: $\lambda_1^{(i)}$, $\lambda_2^{(i)}$, $\rho_1^{(i)}$, and $\rho_2^{(i)}$. For $\alpha \in \mathcal{F}_2$ and $j \in \{1, 2\}$, the schedule $\alpha_j^{(i)}$ is defined as follows: in the first phase, for any $x \in \mathcal{P}$ such that $\ell(x) = i$ and $s_j(x)$ is defined, the schedule omits all elements $y \ge s_j(x)$; it schedules the elements not omitted in this way according to α. In the second phase, the schedule schedules the elements omitted in the first phase according to α.

Collect $\lambda_1^{(i)}$, $\lambda_2^{(i)}$, $\rho_1^{(i)}$, and $\rho_2^{(i)}$ for all i, $0 \le i < h$, into a family $\mathcal{F}_3$; the family contains at most $4h$ schedules. We claim that $\mathcal{F}_3$ is a 3-hitting family for $\mathcal{P}$. To see this, let $\boldsymbol{a} = (x, y, z)$ be an admissible triplet. We first show that two schedules in $\mathcal{F}_3$ hit (x, z); then we show that one of them places y in between x and z.

Let $w = x \wedge z$. Since $\boldsymbol{a}$ is admissible, either $x \le z$, in which case $w = x$ and $w < z$, or $x \parallel z$, in which case $w < x$ and $w < z$. Consider the latter case first. Let $M_x = \min\{x' \mid w < x' \le x\}$ and $M_z = \min\{z' \mid w < z' \le z\}$. Clearly, both sets are nonempty. Moreover, they are disjoint, for suppose that $w' \in M_x \cap M_z$. Then by definition we have $w < w'$. However, since $w' \le x$ and $w' \le z$, we have $w' \le (x \wedge z) = w$, which is a contradiction. Finally, note that $M_x \cup M_z$ is contained in the set of immediate successors of w. For suppose that, say, $x' \in M_x$ is not an immediate successor of w; then there exists $x'' \in \mathcal{P}$ such that $w < x'' < x' \le x$, contradicting the minimality of x'. Since w has at most 2 immediate successors, we conclude that $M_x \cup M_z = \{s_1(w), s_2(w)\}$, that is, $M_x = \{s_{3-j}(w)\}$ and $M_z = \{s_j(w)\}$ for some $j \in \{1, 2\}$.

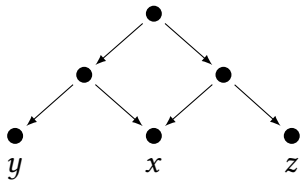

Figure 5.3: The obstacle for the layer-based construction of Theorem 5.10: unless the tuples (x, y, z) and (x, z, y) are already hit by the initial 2-hitting family, all schedules constructed by the layer-based construction will miss them.

Note that $x \not\geq s_j(w)$, since otherwise we would have $s_j(w) \leq (x \wedge z) = w$. Can it happen that there is another element $w' \in \mathcal{P}$ such that $\ell(w') = \ell(w)$ and $x \geq s_j(w')$? Suppose it can. Then by Lemma 5.5, the elements w and w' are incomparable. Also, w' and z are incomparable: it cannot be $z \leq w'$ because $w' \leq x$ and $x \parallel z$, and it cannot be $w' \leq z$ because then we would have $w' \leq (x \wedge z) = w$. Hence the elements w', x, w, z form a zig-zag, which cannot happen by Lemma 5.7. It follows that for $i = \ell(w)$ (note $i < h$), the schedules $\lambda_j^{(i)}$ and $\rho_j^{(i)}$ schedule x in the first phase and z in the second phase. Therefore, both schedules hit (x, z).

Now, there either exists or does not exist an element $w' \in \mathcal{P}$ such that $\ell(w') = i$ and $y \geq s_j(w')$; hence y is scheduled either in the first or in the second phase by both $\lambda_j^{(i)}$ and $\rho_j^{(i)}$. If it is scheduled in the first phase, then since the pair (x, y) is admissible, one of λ and ρ, and consequently one of $\lambda_j^{(i)}$ and $\rho_j^{(i)}$ hits the pair. Analogously, one of $\lambda_j^{(i)}$ and $\rho_j^{(i)}$ hits (y, z) if y is scheduled in the second phase. In both cases, one of $\lambda_j^{(i)}$ and $\rho_j^{(i)}$ hits (x, y, z).

It remains to consider the case when $x \leq z$, which implies that $x = x \wedge z$. By reasoning as before, we conclude that x has an immediate successor $s_j(x)$ for $j \in \{1, 2\}$ such that $z \geq s_j(x)$. Then, if $\ell(x) = i$, both $\lambda_j^{(i)}$ and $\rho_j^{(i)}$ hit (x, z), and one of them places y in between, thus hitting (x, y, z). □

By combining Lemma 5.8 and Theorem 5.10, we get the following corollary.

Corollary 5.11. *The smallest* 3*-hitting family of schedules for a series-parallel order* $\mathcal{P}$ *of height* h *has size at most* $4h + 4\Gamma_{\mathcal{P}} \lceil \log_2 \Delta_{\mathcal{P}} \rceil$. □

At the end of this section, let us mention that the layer-based construction of Theorem 5.10 does not work for a broader class of binary semilattices, one that would contain partial orders that are not series-parallel. This is because any such partial order necessarily contains the zig-zag order (Valdes, Tarjan, and Lawler 1982), and using any zig-zag one

can easily construct the obstacle that was ruled out in the proof of Theorem 5.10. For example, consider the binary semilattice in Figure 5.3. Assume that the initial 2-hitting family does not hit the tuple (x, y, z); say, the schedules λ and ρ order the elements as $y \cdot x \cdot z$ and $z \cdot x \cdot y$. As it turns out, at layers 0 and 1 there is always a node such that x is its left successor, and a node such that x is its right successor. Hence, all schedules constructed from λ and ρ schedule x in the second phase. This implies that in order for (x, y, z) to be hit, all of x, y, z need to be scheduled in the second phase. But then by construction the schedules must agree on x, y, z with either λ or ρ, both of which miss (x, y, z).

5.4 d-Hitting Families for Trees for $d \geq 3$

Consider a complete binary tree $\mathcal{T}^h$ of height $h \geq 1$. In this section we assume that elements of $\mathcal{T}^h$ are strings: $\mathcal{T}^h = \{0,1\}^{<h}$ with $x \leq y$ for $x, y \in \mathcal{T}^h$ iff x is a prefix of y. The kth layer of $\mathcal{T}^h$ is $\{0,1\}^k$, and nodes of the $(h-1)$th layer are *leaves*. Unless $x \in \mathcal{T}^h$ is a leaf, nodes $x\,0$ and $x\,1$ are the left and the right child of x, respectively. (Recall that the juxtaposition here denotes concatenation of strings, with the purpose of distinguishing individual strings and their sequences.) The tree $\mathcal{T}^h$ has $n = 2^h - 1$ nodes.

Fix some d and let $\mathcal{T}^h$ be a complete binary tree of height h. In this section we prove the second of the two main results of the chapter, namely the following theorem.

Theorem 5.12. *For any $d \geq 3$ the complete binary tree of height h has a d-hitting family of schedules of size $\exp(d) \cdot h^{d-1}$.*

Note that in terms of the number of nodes of $\mathcal{T}^h$, which is $n = 2^h - 1$, Theorem 5.12 gives a d-hitting family of size polylogarithmic in n. The proof of the theorem is constructive, and we divide it into three steps. The precise meaning to the steps relies on auxiliary notions of a *pattern* and of d-tuples *conforming to* a pattern; we give all necessary definitions below.

Lemma 5.13. *For each admissible d-tuple $\boldsymbol{a} = (a_1, \ldots, a_d)$ there exists a pattern p such that $\boldsymbol{a}$ conforms to p.*

Lemma 5.14. *For each pattern p there exists a schedule α_p that hits all d-tuples $\boldsymbol{a}$ that conform to p.*

Lemma 5.15. *The total number of patterns, up to isomorphism, does not exceed $\exp(d) \cdot h^{d-1}$.*

The statement of Theorem 5.12 follows easily from these lemmas. The key insight is the definition of the pattern and the construction of Lemma 5.14.

In the sequel, for partial orders that are trees directed from the root we will use the standard terminology for graphs and trees (relying on Hasse diagrams): node, outdegree, siblings, 0- and 1-principal subtree of a node, isomorphism. We denote the parent of a node u by $\operatorname{par} u$ and the *least common ancestor* of nodes u and v by $\operatorname{lca}(u, v)$.

If T is a tree and $X \subseteq T$ is a subset of its nodes, then by $[X]$ we denote the lca-closure of X: the smallest set $Y \subseteq T$ such that, first, $X \subseteq Y$ and, second, for any $y_1, y_2 \in Y$ it holds that $\operatorname{lca}(y_1, y_2) \in Y$. The following lemma is a variation of a folklore Lemma 1 in Fomin et al. (2012).

Lemma 5.16. $|[X]| \leq 2\,|X| - 1$.

Definition 5.3 (pattern)**.** A *pattern* is a quintuple $p = (D, \preccurlyeq, s, \ell, \pi)$ where:

- $d \leq |D| \leq 2d - 1$,
- $(D, \preccurlyeq)$ is a partial order which is, moreover, a tree directed from the root,
- the number of non-leaf nodes in $(D, \preccurlyeq)$ does not exceed $d - 1$,
- each node of $(D, \preccurlyeq)$ has outdegree at most 2,
- the partial function $s\colon D \rightharpoonup \{0, 1\}$ specifies, for each pair of siblings v_1, v_2 in $(D, \preccurlyeq)$, which is the left and which is the right child of its parent: $s(v_t) = 0$ and $s(v_{3-t}) = 1$ for some $t \in \{1, 2\}$; the value of s is undefined on all other nodes of D,
- the partial function $\ell\colon D \rightharpoonup \{0, 1, \ldots, h - 1\}$ associates a *layer* with each non-leaf node of $(D, \preccurlyeq)$, so that $u \prec v$ implies $\ell(u) < \ell(v)$; the value of ℓ is undefined on all leaves of D, and
- π is a schedule for $(D, \preccurlyeq)$.

We remind the reader that the symbol $\leq$ refers to the same partial order as $\mathcal{T}^h$.

Definition 5.4 (conformance)**.** Take any pattern $p = (D, \preccurlyeq, s, \ell, \pi)$ and any tuple $\boldsymbol{a} = (a_1, \ldots, a_d)$ of d distinct elements of the partial order $\mathcal{T}^h$. Consider the set $\{a_1, \ldots, a_d\}$: the restriction of $\leq$ to its lca-closure $A = [\{a_1, \ldots, a_d\}]$ is a binary tree, $(A, \leq)$. Suppose that the following conditions are satisfied:

a) the trees $(D, \preccurlyeq)$ and $(A, \leq)$ are isomorphic: there exists a bijective mapping $i\colon D \to A$ such that $v_1 \preccurlyeq v_2$ in D iff $i(v_1) \leq i(v_2)$ in $\mathcal{T}^h$;

b) the partial function s correctly indicates left- and right-subtree relations: for any $v \in D$, $s(v) = b \in \{0, 1\}$ if and only if $i(v)$ lies in the b-principal subtree of $i(\operatorname{par}(v))$;

c) the partial function ℓ correctly specifies the layer inside $\mathcal{T}^h$: for any non-leaf $v \in D$, $\ell(v) = |i(v)|$; recall that elements of $\mathcal{T}^h$ are binary strings from $\{0,1\}^{<h}$;

d) the schedule π for $(D, \preccurlyeq)$ hits the tuple $i^{-1}(\boldsymbol{a}) = (i^{-1}(a_1), \ldots, i^{-1}(a_d))$.

Then we shall say that the tuple $\boldsymbol{a}$ *conforms to* the pattern p.

We now sketch the proof of Lemma 5.14. Fix any pattern $p = (D, \preccurlyeq, s, \ell, \pi)$. Recall that we need to find a schedule α_p that hits all d-tuples $\boldsymbol{a} = (a_1, \ldots, a_d)$ conforming to p. We will pursue the following strategy. We will cut the tree $\mathcal{T}^h$ into multiple pieces; this cutting will be entirely determined by the pattern p, independent of any individual $\boldsymbol{a}$. Each piece in the cutting will be associated with some element $c \in D$, so that each element of D can have several pieces associated with it. In fact, every piece will form a subtree of $\mathcal{T}^h$ (although this will be of little importance). The key property is that, for every d-tuple $\boldsymbol{a} = (a_1, \ldots, a_d)$ conforming to p, if i is the isomorphism from Definition 5.4, then each element a_k, $1 \le k \le d$, will belong to a piece associated with $i^{-1}(a_k)$. As a result, the desired schedule α_p can be obtained in the following way: arrange the pieces according to how π schedules elements of D and pick any possible schedule inside each piece. This schedule will be guaranteed to meet the requirements of the lemma.

We now give detailed proofs of the Lemmas 5.16, 5.13, 5.14, and 5.15.

Proof of Lemma 5.16

Proof of Lemma 5.16. Consider the tree T as a partial order, $(T, \le)$, where the root is the smallest element. Let $X \subseteq T$. It is immediate that the restriction of $\le$ to $[X]$ is also a tree, $([X], \le)$. Suppose $[X] = L \cup B \cup U$ where L is the set of leaves of this new tree $([X], \le)$, B is the set of its non-leaf nodes with more than 1 child, and U is the set of its non-leaf nodes with exactly 1 child. Sets L, B, and U are disjoint.

We now trace the "provenance" of elements of these sets, i.e., look into why they are included in $[X]$. It is clear that $L \subseteq X$ and $U \subseteq X$, because only nodes with 2 or more children can belong to $[X] \setminus X$. Nodes of the set B are the only "branching points" of the tree $([X], \le)$, and thus their number cannot exceed $|L| - 1$. More formally, denote by n_i the number of nodes of $([X], \le)$ with exactly i children, $i \ge 0$. As each edge in the graph departs from some node and arrives at some node,

$$\sum_{v \in [X]} \mathrm{indeg}(v) = \sum_{v \in [X]} \mathrm{outdeg}(v).$$

The left-hand side of this equation is $n - 1$, where $n = |[X]|$, because each node except for

the root has a parent. Therefore,

$$n - 1 = \sum_{i \geq 0} n_i \cdot i,$$
$$n_0 + n_1 + n_2 + \ldots - 1 = n_1 + 2n_2 + 3n_3 + \ldots,$$
$$n_0 + n_1 + n_2 + \ldots - 1 \geq n_1 + 2n_2 + 2n_3 + \ldots$$

Denote $r = |B| = n_2 + n_3 + \ldots$, then $n_0 + n_1 + r - 1 \geq n_1 + 2r$, and so $r \leq n_0 - 1$, which is the same as $|B| \leq |L| - 1$.

To sum up, $|[X]| = |L \cup U| + |B| \leq |L \cup U| + |L| - 1$. Since $L \cup U \subseteq X$ as argued above, we conclude that $|[X]| \leq 2|X| - 1$. □

Proof of Lemmas 5.13 and 5.15

Proof of Lemma 5.13. Recall that we need to show that for each admissible d-tuple $\boldsymbol{a}$ there exists a pattern p such that $\boldsymbol{a}$ conforms to p. Take any such tuple $\boldsymbol{a} = (a_1, \ldots, a_d)$; since it is admissible, there exists a schedule α for $\mathcal{T}^h$ that hits $\boldsymbol{a}$. Consider the set $\{a_1, \ldots, a_d\}$ and take its lca-closure in $\mathcal{T}^h$: $D = [\{a_1, \ldots, a_d\}]$. Let $\preccurlyeq$ be the restriction of $\leq$ to D. Now for each non-leaf node $v \in D$ in the partial order $(D, \preccurlyeq)$ define $\ell(v) = |v|$; again, recall that elements of $\mathcal{T}^h$ are binary strings from $\{0,1\}^{<h}$. Furthermore, consider each node $v \in D$ in $(D, \preccurlyeq)$ with outdegree 2; if v' and v'' are the children of v in $(D, \preccurlyeq)$, then v' and v'' lie in different principal subtrees of v in $\mathcal{T}^h$ (because otherwise the equality $\mathrm{lca}(v', v'') = v$ cannot hold); that is, $v' = v\,0\,u'$ and $v'' = v\,1\,u''$ for some strings $u', u'' \in \{0,1\}^*$. Accordingly, define $s(v') = 0$ and $s(v'') = 1$. Finally, take the schedule α and restrict it to the set D; denote the obtained schedule by π.

It is not difficult to check that the tuple $\boldsymbol{a}$ conforms to the constructed pattern $p = (D, \preccurlyeq, s, \ell, \pi)$. Note that the upper bound on $|D|$ is by Lemma 5.16 and the upper bound on the number of non-leaf nodes in $(D, \preccurlyeq)$ holds by the following argument. Let $m \leq d$ be the number of leaves of $(D, \preccurlyeq)$ in the set $\{a_1, \ldots, a_d\}$; then $(D, \preccurlyeq)$ has exactly $m - 1$ binary nodes (none of them leaves). Furthermore, all non-leaf unary nodes in $(D, \preccurlyeq)$ cannot belong to the difference $D \setminus \{a_1, \ldots, a_d\}$ and thus all lie in the set $\{a_1, \ldots, a_d\}$; their number cannot exceed the number of all non-leaf nodes in $\{a_1, \ldots, a_d\}$, i.e., is at most $d - m$. Hence, the total number of non-leaf nodes in $(D, \preccurlyeq)$ does not exceed $(m - 1) + (d - m) = d - 1$. This concludes the proof. □

Proof of Lemma 5.15. We need to count the number of patterns, up to isomorphism. A pattern is fully specified by its components:

— the binary tree $(D, \preccurlyeq)$ with at most $2d-1$ nodes and the partial function s that specifies a planar embedding of this tree—the total number of such embeddings (for all trees) is at most $4^{2d-1}/3$;

— the partial function ℓ with domain of size at most the number of non-leaf nodes in D (i.e., at most $d-1$), and co-domain of size h—the number of suitable functions is at most h^{d-1};

— the schedule π for $(D, \preccurlyeq)$—of which there are at most $(2d-1)!$.

Thus the total number of patterns does not exceed

$$4^{2d-1}/3 \cdot h^{d-1} \cdot (2d-1)! = \exp(d) \cdot h^{d-1}.$$

This completes the proof. □

Proof of Lemma 5.14

Proof of Lemma 5.14. Fix any pattern $p = (D, \preccurlyeq, s, \ell, \pi)$. Recall that we need to find a schedule α_p that hits all d-tuples $\boldsymbol{a} = (a_1, \ldots, a_d)$ conforming to p. We will pursue the following strategy. We will cut the tree $\mathcal{T}^h$ into multiple pieces; this cutting will be entirely determined by the pattern p, independent of any individual $\boldsymbol{a}$. Each piece in the cutting will be associated with some element $c \in D$, so that each element of D can have several pieces associated with it. In fact, every piece will form a subtree of $\mathcal{T}^h$ (although this will be of little importance). The key property is that, for every d-tuple $\boldsymbol{a} = (a_1, \ldots, a_d)$ conforming to p, if i is the isomorphism from Definition 5.4, then each element a_k, $1 \le k \le d$, will belong to a piece associated with $i^{-1}(a_k)$. As a result, the desired schedule α_p can be obtained in the following way: arrange the pieces according to how π schedules elements of D and pick any possible schedule inside each piece. This schedule will be guaranteed to meet the requirements of the lemma.

We now show how to implement this strategy. We describe a procedure that, given p, constructs a suitable α_p. To simplify the presentation, we will describe cutting of $\mathcal{T}^h$ and constructing α_p simultaneously, although they can be performed separately. The cutting itself is defined by the following formalism. For each element $c \in D$, we define a set $E(c) \subseteq \mathcal{T}^h$, with the intention that elements from $E(c)$ point to the roots of all pieces associated with c. The pieces themselves stretch out down the tree up to (and including) layer $\ell(c)$; as the value $\ell(c)$ is undefined for leaves of $(D, \preccurlyeq)$, we will instead use the extension of ℓ that assigns $\ell(c) = h$ for all leaves c of $(D, \preccurlyeq)$, abusing the notation ℓ. As we go along, we add more and more elements to the schedule α_p, constructing it on the way; we will refer to this as *scheduling* these elements. The elements in $E(c)$

can be thought of as *enabled* after scheduling the elements from previously considered pieces: that is, all these elements have not been scheduled yet, but all their immediate predecessors (parents) in $(D, \preccurlyeq)$ have. This will allow us to schedule the pieces rooted at $E(c)$ at any suitable moment. We will not give any "prior" definition of $E(c)$: these sets will only be determined during the process.

Overall, the *invariant* of the procedure is that, when we define $E(c)$, the elements in $E(c)$ form an antichain, are enabled (not scheduled yet, but all predecessors already scheduled), and belong to layers $\leq \ell(c)$ of the partial order $\mathcal{T}^h$.

Let us now fill in the missing details of the process. At first, no elements are scheduled, and the set $E(c_*)$, where c_* is the root of $(D, \preccurlyeq)$, is defined as the singleton $\{\varepsilon\}$; recall that ε is the root of the tree $\mathcal{T}^h$. The procedure goes over the schedule π, which is part of the pattern p, and handles elements c scheduled by π one by one. The first element is, of course, the root of $(D, \preccurlyeq)$, which we called c_*. Note that at the beginning of the procedure, the invariant is satisfied.

To handle an element c scheduled by π, our procedures performs the following steps. It first schedules all elements in the set

$$U(c) = \{y \in \mathcal{T}^h \mid x \leq y \text{ for some } x \in E(c) \text{ and } |y| \leq \ell(c)\},$$

i.e., all elements $x \in E(c)$ and all elements that are successors of $x \in E(c)$ in layers up to and including $\ell(c)$. Note that this set $U(c)$ consists of a number of disjoint subtrees of the tree $\mathcal{T}^h$; these subtrees are the pieces that we previously discussed, and $U(c)$ is their union. Each piece is non-empty: for all $x \in E(c)$, the set of all y such that $x \leq y$ and $|y| \leq \ell(c)$ contains at least the element x itself, because, by our invariant, $\ell(x) \leq \ell(c)$; therefore, $E(c) \subseteq U(c)$. The pieces (subtrees) are disjoint because the elements in $E(c)$ form an antichain. Finally, scheduling these pieces is possible because, on one hand, no $x \in E(c)$ has been scheduled previously and, on the other hand, all predecessors of $x \in E(c)$ have already been scheduled. Note that we can schedule all elements from $U(c)$ in any order admitted by $\mathcal{T}^h$, for instance using lexicographic depth-first traversal.

After this, the procedure forms new sets E; the precise choice depends on the outdegree of c in $(D, \preccurlyeq)$. Recall that this outdegree does not exceed 2 by our definition of the pattern. Observe that after scheduling the pieces associated with c, as described in the previous paragraph, the following elements, for all $x \in E(c)$, are made enabled: $z \in \mathcal{T}^h \cap (\{0,1\}^{\ell(c)} \{0,1\})$ with $x \leq z$. In fact, this set is empty iff $\ell(c) = h$; by our choice of ℓ, this happens if and only if $d = 0$, i.e., when c is a leaf of $(D, \preccurlyeq)$. In such a case, no new set E is formed and the procedure proceeds to the next element of π. Otherwise $d \in \{1, 2\}$; we consider each case separately. If $d = 1$, then the element $c \in D$ has a single child in the tree $(D, \preccurlyeq)$. Denote this child by c' and define

$$E(c') = \{z \in \{0,1\}^{\ell(c)+1} \mid x \leq z \text{ for some } x \in E(c)\}.$$

If $d = 2$, then the element $c \in D$ has two children in the tree $(D, \preccurlyeq)$. Let these children be c_0 and c_1, such that $s(c_r) = r$ for both $r \in \{0, 1\}$. We now split the set of newly enabled elements as follows:

$$E(c_0) = \{\bar{z}\,0 \in \{0,1\}^{\ell(c)+1} \mid x \leq \bar{z}\,0 \text{ for some } x \in E(c)\},$$
$$E(c_1) = \{\bar{z}\,1 \in \{0,1\}^{\ell(c)+1} \mid x \leq \bar{z}\,1 \text{ for some } x \in E(c)\}.$$

Note that the elements z in $E(c')$ (or in $E(c_0)$ and $E(c_1)$, depending on d) form an antichain, are enabled, and, moreover, satisfy the inequality $\ell(z) \leq \ell(c')$, because $\ell(z) = \ell(c) + 1$ and $\ell(c) < \ell(c')$ by the choice of ℓ. This ensures that during the run of the procedure the invariant is maintained.

It is not difficult to see that the described procedure outputs some schedule α_p for $\mathcal{T}^h$. We now show why this α_p satisfies our requirements. Indeed, pick any admissible d-tuple $\boldsymbol{a} = (a_1, \ldots, a_d)$ conforming to the pattern p; we need to check that α_p hits $\boldsymbol{a}$. In fact, by the choice of our strategy, it is sufficient to check that each element a_k, $1 \leq k \leq d$, belongs to a piece associated with the element $i^{-1}(a_k)$ where i is the isomorphism from the definition of conformance. In other words, we need to ensure that each element a_k belongs to the set $U(c)$ for $c = i^{-1}(a_k)$; we will prove a stronger claim that $a_k \in U(c) \cap \{0,1\}^{\ell(c)}$ for this c. Note that the choice of $U(c)$ is such that $U(c) \subseteq \{0,1\}^{\leq \ell(c)}$.

The proof of this claim follows our construction of α_p. Indeed, consider the element a_1 first; we necessarily have $i^{-1}(a_1) = c_*$. By our definition of conformance, a_1 is on the $\ell(c_*)$th layer in the tree $\mathcal{T}^h$, that is, $|a_1| = \ell(c_*)$. By the description of our procedure, all elements from $\{0,1\}^{\ell(c_*)}$ are associated with c_*, i.e., belong to $U(c_*)$ and and are thus scheduled during the first step of the procedure. Note that since $\ell(c) > \ell(c_*)$ for all $c \neq c_*$ in D and ℓ correctly specifies the height in $\mathcal{T}^h$, none of the elements $a_2, \ldots, a_d$ can be scheduled before a_1. Also observe that the existence of the isomorphism i ensures that all the elements $a_2, \ldots, a_d$ are successors of a_1.

It now remains to follow the inductive step: suppose the claim holds for an element a_k with $i^{-1}(a_k) = c$ for some $c \in D$. As soon as our procedure schedules $U(c) \cap \{0,1\}^{\ell(c)}$, all its successors become enabled, because $U(c) \subseteq \{0,1\}^{\leq \ell(c)}$. We now need to consider three cases depending on the value of d. If $d = 0$, there is nothing to prove. If $d = 1$, both successors of a_k in $\mathcal{T}^h$ are included into $E(c') \subseteq U(c')$. where c' is the only child of c in $(D, \preccurlyeq)$. by our choice of ℓ it holds that $\ell(c') = |i(c')|$. Since the element $i(c')$ is a (not necessarily direct) successor of $i(c)$ and is different from $i(c)$, it follows that $x \leq i(c')$ for some element $x \in E(c)$. But then it follows that $i(c') \in U(c')$ by the choice of U. Similarly, consider $d = 2$. All 0-children and 1-children of a_k in $\mathcal{T}^h$ are included in $E(c_0)$ and $E(c_1)$, respectively, where by c_r, $r \in \{0,1\}$, we denote the (unique) child of c in $(D, \preccurlyeq)$ that has $s(c_r) = r$. Since s correctly specifies 0- and 1-principal subtree relations in $\mathcal{T}^h$, it follows that $i(c_r)$ belongs to the r-principal subtree of $i(c)$, for each $r \in \{0,1\}$. So our choice

of $E(c_0)$ and $E(c_1)$ ensures that, for each $r \in \{0,1\}$, there exists an $x \in E(c_r)$ such that $x \leq i(c_r)$. The conditions on the layer are checked in the same way as in the case $d = 1$; the upshot is that $i(c_r) \in U(c_r) \cap \{0,1\}^{\ell(c_r)}$ for both r. This completes the proof of the claim, from which the correctness of the procedure constructing α_p follows.

This concludes the proof of Lemma 5.14. □

Remark 2. The proof above cuts the tree right below the layers specified by the function ℓ; this choice is somewhat arbitrary and can be changed. Moreover, for presentation purposes we also decided to schedule all elements of sets $U(c)$ at once. This choice is essentially employing a *breadth-first* strategy: as soon as we get to process c, we necessarily schedule all possible candidates for its image $i(c)$. However, a *depth-first* strategy also works: in this strategy, elements $x \in E(c)$ are processed one-by-one. More precisely, the procedure can first schedule all elements of $U(c)$ that are successors of x, essentially going into the subtree of $\mathcal{T}^h$ rooted at x. After this, instead of switching to a different $x' \in E(c)$, the procedure could stay inside this subtree and follow, as usual, the guidance of π, *assuming* that the chosen subtree indeed contains $i(c)$. Only after scheduling *all* elements of the subtree (i.e., all $u \in \mathcal{T}^h$ such that $x \leq u$) does the procedure come back to its set $E(c)$ and proceed to the next candidate $x' \in E(c)$. In fact, during the run of this modified procedure *many* different sets $E(c)$ will be defined (as long as $c \neq c_*$); all these sets will be disjoint, and their union will be equal to the original set $E(c)$ as defined in the proof above.

5.5 From Hitting Families to Systematic Testing

Hitting families of schedules serve as a theoretical framework for systematically exposing all bugs of small depth. However, bridging the gap from theory to practice poses several challenges, which we describe in this section.

To make the discussion concrete, we focus on a specific scenario: testing the rendering of web pages in the browser. Web pages exhibit event-driven concurrency: as the browser parses the page, it concurrently executes JavaScript code registered to handle various automatic or user-triggered events. Many bugs occur as a consequence of JavaScript's ability to manipulate the structure of the page while the page is being parsed. Previous work shows such bugs are often of small depth (Jensen et al. 2015; Raychev, M. T. Vechev, and Sridharan 2013).

As an example, consider the web page in Fig. 5.4. In the example, the image (represented by the `<img>` tag) has an on-load event handler that calls the function `loaded()` once the image is loaded. The function, defined in a separate script block, changes the text of the paragraph `p` to *Loaded*. There are two potential bugs in this example. The first one is of depth $d = 2$, and it occurs if the image is loaded quickly (for example, from the cache),

```
<img src="..." onload="javascript:loaded()"/>
<script>
  function loaded() {
    document.getElementById('p').innerHTML = 'Loaded';
  }
</script>
<p id="p">Waiting...</p>
```

Figure 5.4: Example of bugs of depth $d = 2$ and $d = 3$ in a web page

```
<img src="..." onload="javascript:loaded()"/>
<script>
  function loaded() {
    var p = document.getElementById('p');
    if (p == null) {
      setTimeout(loaded, 10);
    } else {
      p.innerHTML = 'Loaded';
    }
  }
</script>
<p id="p">Waiting...</p>
```

Figure 5.5: Using a timer to fix the bug from Fig. 5.4 involving a non-existent element

before the browser parses the `<script>` tag. In this case, the on-load handler tries to call an undefined function. The second bug is of depth $d = 3$, and it occurs if the handler is executed after the `<script>` tag is parsed, but before the `<p>` tag is parsed. In this case, the function `loaded()` tries to access a non-existent HTML element.

Next, we identify and discuss three challenges.

Partial Orders Need Not Be Static

Our theoretical model assumes a static partially ordered set of elements, and allows arbitrary reordering of independent (incomparable) elements. For the web page in Fig. 5.4, there are three parsing events (corresponding to the three HTML tags) and an on-load event. The parsing events are chained in the order their tags appear in the code. The on-load event happens after the `<img>` tag is parsed, but independently of the other parsing events, giving a tree-shaped partial order.

In more complex web pages, the situation is not so simple. Events may be executions

of scripts with complex internal control-flow and data dependencies, as well as with effect on the global state. Once a schedule is reordered, new events might appear, and some events might never trigger. An example showing a more realistic situation is given in Fig. 5.5. In order to fix the bug involving a non-existent HTML element `p`, the programmer now explicitly checks the result of `getElementById()`. If `p` does not exist (`p == null`), the programmer sets a timer to invoke the function `loaded()` again after 10 milliseconds. As a consequence, depending on what happens first—the on-load event or the parsing of `<p>`—we may or may not observe one or more timeout events. Note that the chain of timeout events also depends on parsing the `<script>` tag. If the tag is not parsed, the `loaded()` function does not exist, so no timer is ever set. Moreover, the number of timeout events depends on when exactly the `<p>` tag is parsed.

The example shows that there is a mismatch between the assumption of static partially ordered events and the dynamic nature of events occurring in complex web pages. We study the mismatch in detail in Chapter 6, where we show that dynamic partial orders can be modeled using the notions of *upgrowing posets* and *scheduling posets*. We also give an *online* (on-the-fly) construction of hitting families for such posets. Here we note that even the "static" theory of hitting families can be applied as a testing heuristic. While we lose completeness (in the sense of hitting all depth-d bugs), we retain the variety of different event orderings. In the context of web pages, an initial execution of a page gives us an initial partially ordered set of events. We use it to construct a hitting family of schedules, which we optimistically try to execute. The approach is based on the notion of *approximate replay*, which is employed by R^4, a stateless model checker for web pages (Jensen et al. 2015). We come back to this approach later in the section.

Beyond Trees and Series-Parallel Orders

Our results on trees and series-parallel orders are motivated by the existing theoretical models of asynchronous programs (Jhala and Majumdar 2007; Ganty and Majumdar 2012; Emmi, Qadeer, and Rakamaric 2011a), where the partial orders induced by event handlers indeed form these specific partial orders. However, in the context of web pages, events need not necessarily be ordered as trees, not even as series-parallel orders. An example of a feature that introduces additional ordering constraints is deferred scripts. Scripts marked as deferred are executed after the page has been loaded, and they need to be executed in the order in which their corresponding `<script>` tags were parsed (Petrov et al. 2012). The tree approximation corresponds to testing the behavior of pages when the deferred scripts are treated as normal scripts and loaded right away.

In Chapter 6 we study a construction of hitting families that works for general partial orders. However, due to the existence of partial orders such as the standard example

Table 5.1: For each website, the table show the number of events in the initial execution, the height of the partial order (happens-before graph), the number of schedules generated for $d = 3$, and the number of schedules for $d = 3$ with pruning based on races.

Website	# Events	Height	$d = 3$	$d = 3$ (pruned)
abc.xyz	337	288	561	0
newscorp.com	1362	875	2689	100
thehartford.com	2018	1547	3913	138
www.allstate.com	4534	3822	9023	106
www.americanexpress.com	2971	2586	5897	340
www.bankofamerica.com	2305	2095	4561	150
www.bestbuy.com	301	248	576	10
www.comcast.com	188	118	337	16
www.conocophillips.com	4184	3478	8286	248
www.costco.com	7331	6390	14614	364
www.deere.com	2286	1902	4516	236
www.generaldynamics.com	2820	2010	5611	272
www.gm.com	2337	1473	4600	94
www.gofurther.com	1117	638	2154	568
www.homedepot.com	3780	2100	7515	1526
www.humana.com	5611	4325	11174	2058
www.johnsoncontrols.com	2953	2395	5881	450
www.jpmorganchase.com	4134	3519	8247	1316
www.libertymutual.com	3885	3560	7735	324
www.lowes.com	6938	4383	13778	3438
www.massmutual.com	3882	3313	7682	1852
www.morganstanley.com	2752	2301	5402	128
www.utc.com	4081	3266	8100	206
www.valero.com	2116	1849	4178	38

(see Example 5.1), no general construction can achieve polylogarithmic upper bounds. Therefore, the question of developing constructions for other special cases of partial orders that capture common programming idioms remains open.

Unbalanced Trees

For a tree of height h, constructions from Sections 5.3 and 5.4 give 3-hitting families of size $O(h)$ and $O(h^2)$, respectively. If the tree is balanced, the cardinality of these families are exponentially smaller than the number of events in the tree. However, in the web page setting, trees are not balanced.

In order to inspect the shape of partial orders occurring in web pages, we randomly

selected 24 websites of companies listed among the top 100 of Fortune 500 companies. For each website, we used R^4 (Jensen et al. 2015) to record an execution and construct the happens-before relation (the partial order). Table 5.1 shows the number of events and the height of the happens-before graph for the websites. The results indicate that a typical website has most of the events concentrated in a backbone of very large height, proportional to the total number of events.

The theory shows that going below $\Theta(h)$ is impossible in this case unless $d < 3$; and this can indeed lead to large hitting families: for example, our construction for $h = 1000$ and $d = 4$ corresponds to several million tests. However, not all schedules of the partial ordering induced by the event handlers may be relevant: if two events are independent (commute), one need not consider schedules which only differ in their ordering. Therefore, since hitting families are defined on an *arbitrary* partial order, not only on the happens-before order, we can use additional information, such as (non-)interference of handlers, to reduce the partial ordering first.

For web pages, we apply a simple partial order reduction to reduce the size of the input trees in the following way. We say a pair of events *race* if they both access some memory location or some DOM element, with at least one of them writing to this location or the DOM element. Events that do not participate in races commute with all other events, so they need not be reordered if our goal is to expose bugs.

R^4 internally uses a race detection tool called EventRacer (Raychev, M. T. Vechev, and Sridharan 2013) to over-approximate the set of racing events. In order to compute hitting families, we construct a pruned partial order from the original tree of events. As an example, for $d = 3$ and the simple $O(n^{d-2})$ construction from Proposition 5.2, instead of selecting a_1 arbitrarily, we select it from the events that participate in races. We then perform the left-to-right and right-to-left traversals as usual. In total, the number of generated schedules is $2r$, where r is the number of events participating in races. This number can be significantly smaller than $2n$, as can be seen in the fourth ($d = 3$) and fifth ($d = 3$ pruned) columns of Table 5.1.

Chapter 6

Online Construction of Hitting Families

As we have seen in the previous chapter, when the partial order is given explicitly and has a known structure, such as an antichain or a tree, one can provide explicit combinatorial constructions of d-hitting families of schedules; for antichains and trees, the size of a d-hitting family can be exponentially smaller than the number of events. Unfortunately, when testing a concurrent system implementation, it is unrealistic or impossible to know the partial ordering up front, e.g., if the events are exposed incrementally as the program executes, or to assume a specific "nice" structure. Thus, a challenge in systematic testing is to come up with a small d-hitting family *online* (i.e., along with the execution) and for an *arbitrary* partial ordering.

Recall that by Proposition 5.1 an arbitrary partial order with n elements has a d-hitting family of size $O(n^{d-1})$. Without going into details, we can say that the simple construction given there can be extended into an online construction with the same upper bound. In this chapter, we improve this bound to $O(w^2 n^{d-2})$, where w is the maximal width of the partial order during execution.[1] This may seem like a trivial improvement, but a lot of effort is required to obtain it. Unfortunately, we do not have a matching lower bound; however, we conjecture it is very close to $w^2 n^{d-2}$.

As we shall see in Section 6.3, the randomized scheduling algorithm PCTCP, which is based on our online construction, works reasonably well in practice regardless of the bound.

[1] The upper bound of $w^2 n^{d-1}$ stated in Corollary 6.8 is for *strong* d-hitting families, which will be defined shortly. As we shall see, every strong d-hitting family is a $(d+1)$-hitting family, hence the difference in the exponent.

6.1 Overview of the Approach

In this section, we informally introduce some of the notions used in the rest of the chapter, and we present the PCTCP algorithm and demonstrate it on a simple example.

Example 6.1. Consider a distributed system with three nodes[2]: a *Handler* which processes client requests, a *Logger* which logs transaction information, and a *Terminator* which terminates the system. When the *Handler* processes a *request* message from the client, it sends a *log* message to the *Logger* and a *terminate* message to the *Terminator*. When the *Terminator* receives a *terminate* message, it sends a *flush* message to the *Logger*. On receiving a *flush*, the *Logger* flushes the logs (i.e., writes the logs into the database and deallocates the file descriptors) and sends an acknowledgment *flushed* message back to the *Terminator*. The messages by the *Handler* are sent concurrently to *Logger* and *Terminator*. Hence, the *flush* message sent by *Terminator* and the *log* message sent by *Handler* arrive concurrently at the *Logger*. If the *log* message is processed before the *terminate* message, the system behaves as expected. However, if the *log* message is delayed and the *terminate* message is processed before the *log* message, the *Logger* accesses an invalid descriptor and crashes.

Partial Order of Events and Online Chain Partitioning

PCTCP abstracts messages in the system as partially ordered *events*. The partial order relation corresponds to the causality relation on the events in the execution. Note that the causality relation between the events of a system depends on the semantics and the guarantees of the system. In our example, the *log* and *terminate* messages depend on the *request* message, as they are created in response to *request*. They are concurrent to each other since they are sent to different receivers and they will be processed concurrently.

PCTCP intercepts all events in a running system and maintains the poset of events in an execution *online* as well as the current schedule. In each step, PCTCP selects an unexecuted event and schedules it. The execution of this event can cause further events in the system. These are intercepted and added to the partial order. The partial order of events is maintained as a *chain decomposition*—a partition of the order into a set of chains. Each chain is a linear ordering of events according to the partial order. When a new event is intercepted, it is added to one of these chains (or put in a new chain by itself) by an online chain partitioning algorithm.

The key to the theoretical properties of PCTCP is that the chain decomposition has a small number of chains, bounded by a function of the *width* of the partial order. PCTCP

[2]The example is adapted from Tasharofi et al. (2013), and it shows a simplified version of a bug found in a performance testing tool called *Gatling* (2011–2018).

Input: number of events n, depth bound d
Data: *chains* // chain partition of events, the first $d-1$ positions are initialized to null
Data: *eventsAdded* // number of events already added, initially 0
Data: *priorityChangePt* // vector of $d-1$ distinct integers, initialized randomly between 1 and n
Data: *schedule* // the current execution

Procedure `addNewEvent(`e`)`
1 insert e into the poset using online chain partitioning (Alg. 2)
2 **if** *a new chain is created* **then**
3 insert the new chain into a random position between d and $|chains|$ in *chains*
4 increment *eventsAdded*
5 **if** $\exists j : eventsAdded = priorityChangePt[j]$ **then**
6 // assign a label to the event
7 $e.label \leftarrow j$

Procedure `scheduleNextEvent()`
8 **while** $\exists j : chains[j] = \alpha \cdot e \cdot \alpha' \wedge e.isEnabled \wedge e.hasLabel \wedge e.label \neq j$ **do**
9 // we are at a priority change point
10 // note that $chains[e.label]$ = null due to the labels being distinct
11 swap $chains[j]$ and $chains[e.label]$
12 // select an enabled event from the chain with the highest priority
13 **if** $\exists j : chains[j] = \alpha \cdot e \cdot \alpha' \wedge e.isEnabled$ **then**
14 $e \leftarrow$ the event e corresponding to the highest index j s.t. $chains[j] = \alpha \cdot e \cdot \alpha' \wedge e.isEnabled$
15 $schedule.append(e)$
16 **return** e

Algorithm 1: PCTCP algorithm: adding new events and scheduling the next event from the poset

forms chains of events based on the causal dependency relation between them. It inserts the concurrently executable events into different chains, which bounds the number of chains to a function of the number of concurrently executable events. Therefore, the theoretical bug detection guarantee of PCTCP is not tied to the number of nodes in a system (some of which may be inactive in some parts of the execution) but to the *width* of the partial order, i.e., the maximum number of simultaneously executable events. (Note that since the partial order is revealed one element at a time, the chain partition is constructed *online*. Thus, while there always exists a chain partition whose size is the width of the ordering, we may not achieve this bound.) PCTCP uses an online chain partitioning algorithm (Agarwal and Garg 2007) that guarantees that we use at most $O(w^2)$ chains, where w is the (unknown) width of the partial order.

PCTCP Algorithm

PCTCP is a randomized scheduling algorithm for distributed programs. It takes as input the maximum number n of messages to be scheduled and a parameter d which determines the bug depth to be explored. It guarantees a lower bound on the probability of covering every execution of depth d based on n, d, and the width of the underlying partial order. PCTCP maintains a priority list of chains partitioning the partial order of events, where lower numbers indicate lower priorities. During execution, the scheduler schedules an event from a low priority chain only when all higher priority events are blocked (e.g., waiting on a synchronization action). In addition, the algorithm can change the priority of a chain during execution when the execution meets one of $d-1$ randomly chosen *priority change points* in the execution. When the execution reaches a change point, the scheduler changes the priority of the current chain to the priority value associated with the change point.

The algorithm is given in Algorithm 1. It maintains three main data structures. The first is a list of chains of events (called *chains*), which maintains a chain decomposition of events seen so far, where each chain in the list is assigned a priority. The chain decomposition data structure has two logical parts. The first $d-1$ indices in the list are reserved for chains with reduced priority and are all initialized to null. These positions are populated later during execution when a priority change point is encountered. The rest of the list maintains a prioritized list of chains, and higher indices in the list denote higher priority.

The second data structure, the priority change points *priorityChangePt*, is a list of $d-1$ distinct integers picked randomly from the range $[1, n]$ at the beginning of the algorithm and used to randomly change the priority of certain chains at run time. The third data structure, *schedule*, is a schedule of events executed so far.

The algorithm has two main procedures. Procedure addNewEvent inserts a new event into the chain decomposition by either inserting it at the end of an existing chain or creating a new chain, according to the online chain decomposition algorithm. If a new chain is created, the new chain is assigned a random priority by inserting it into the chain decomposition at a random position at or after the dth position. Additionally, this procedure uses the variable *eventsAdded* to keep track of the number of events added to the poset. Once *eventsAdded* becomes equal to *priorityChangePt*$[j]$ for some j, the procedure assigns a label j to the event that is being added to the poset. The label is used to adjust the priority of the chain containing the event once the event becomes ready to be scheduled.

Procedure scheduleNextEvent selects an enabled event and schedules it by appending it to *schedule*. We say an event e is enabled (denoted by $e.isEnabled$ in the pseudocode) if it is not yet scheduled, but all of its predecessors have been scheduled. To select an

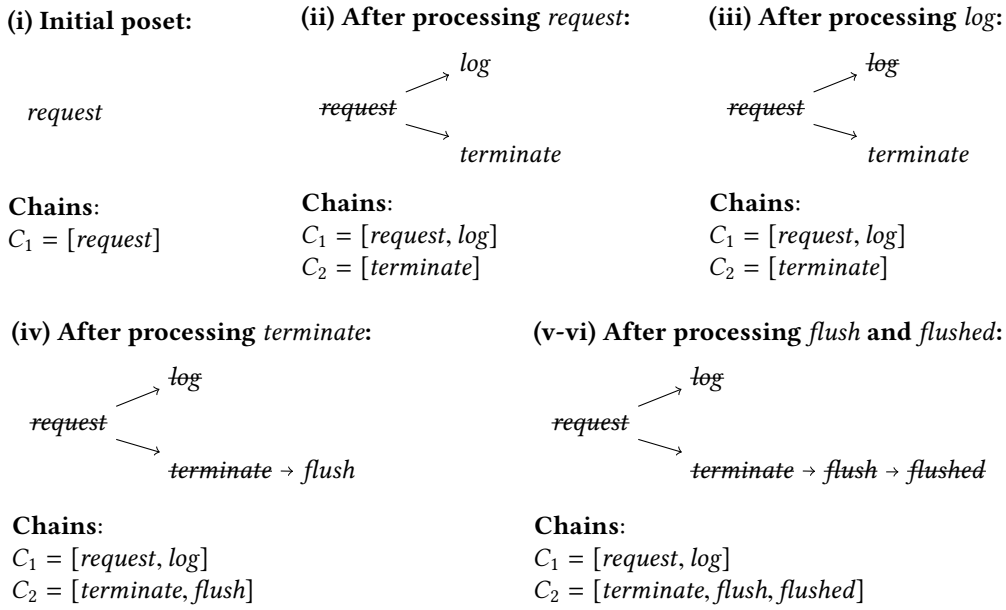

Figure 6.1: The poset of events in an execution and its decomposition into chains

enabled event, the procedure first adjusts the priorities of chains: if there is an enabled event e carrying a label i (the predicate $e.hasLabel$ is true in this case) that is placed in chain c currently in position $j \neq i$ in *chains*, the procedure moves c to position i in *chains*. Once the priorities are adjusted, the procedure picks the highest priority chain containing an enabled event, appends this event to *schedule*, and returns it to be executed. All new events resulting from the execution are added to the chain decomposition (using addNewEvent), and scheduleNextEvent is called again until n events are scheduled.

PCTCP on the Example

Figure 6.1 shows the online construction of the poset in our example for the bug depth parameter $d = 1$. In each step, the event that is executed is crossed out. (i) Initially, the poset contains only the *request* event in a single chain. The event is scheduled since it is the only event in the system. (ii) Executing *request* causes two new events: *log* and *terminate*. PCTCP extends the chain decomposition with these new events. Since the events are concurrent, the width of the partial order at this point is 2, and the chain partitioning algorithm needs to allocate a new chain. Say that in this example the chain partitioning algorithm inserts *log* into the same chain with *request* and *terminate* into a fresh chain. PCTCP now has two chains to select the next event from: $C_1 = [request, log]$ and $C_2 = [terminate]$, and randomly decides the priority between them. We follow the

algorithm first with the ordering that prioritizes C_1 over C_2. In this case, PCTCP schedules the *log* event. (iii) Processing *log* does not lead to more events, so we do not insert any events into the poset. Since all the events in the highest priority chain C_1 are executed, the PCTCP scheduler schedules the next event in C_2, i.e., the *terminate* event. (iv) Processing this event creates a *flush* even sent from the *Terminator* to the *Handler*. Since *flush* depends on *terminate*, PCTCP extends the chain C_2 with *flush*. (v) Since C_1 still does not have any events to schedule, PCTCP continues with C_2 and schedules *flush*. Similar to the previous step, the *flushed* message is inserted into the same chain. (vi) PCTCP schedules the *flushed* event and processes it. No more events are created and the execution ends.

Now assume C_2 was given a priority higher than C_1. In this case, PCTCP schedules the *terminate*, *flush*, and *flushed* events in this order, before the *log* event. This hits the buggy execution. Since each possible ordering between C_1 and C_2 is picked with probability $1/2$, we hit the bug with probability $1/2$.

On the other hand, a naive random strategy that uniformly picks one of the enabled events at each step would detect the same bug with probability $1/4$. The random scheduler would have to select *terminate* among the two concurrent events *log* and *terminate*, and then select *flush* among *log* and *flush* to be able to hit the bug. As the length of the chain in which *flush* is inserted increases, the probability of naive random testing to hit the bug decreases exponentially. On the other hand, the probabilities of detecting a bug with PCTCP and naive random testing intuitively get closer to each other as the width of the poset approaches the number of events in the system, i.e., when most of the newly added events are concurrent to each other. In our experimental evaluation in Section 6.3, we compare the performance of PCTCP and naive random testing on real-world benchmarks.

Priority Change Points

So far, we have ignored the priority change points, because exposing the bug in this example requires a single ordering constraint between two events. Hence, this bug can be detected without changing the initially assigned priorities of the chains. In a more complex setting, the priorities of chains may need to change in order to hit a bug, and this is handled by the priority change points.

Consider a modified version of our example, where the bug is exposed not just with the relative ordering of *flush* $\rightarrow$ *log* events, but also the ordering *flush* $\rightarrow$ *log* $\rightarrow$ *flushed*. Since two additional constraints trigger the bug, the PCTCP scheduler needs to be called with the bug depth parameter $d = 2$, causing it to change chain priorities at one randomly chosen priority change point. If initially C_2 has a higher priority than C_1 and the priority change point is picked to be 5, then the fifth event added to the poset, i.e. *flushed*, is assigned a label, and after *terminate* and *flush* events are executed, the priority of the

chain C_2 is reduced. At this point, the *log* event from the currently higher priority chain C_1 is scheduled. There are no more events in C_1 and PCTCP continues with scheduling *flushed* from C_2, hitting the buggy ordering of events. The probability of hitting the bug in this case is 1/10: the probability that C_2 initially has higher priority than C_1 is 1/2, and the desired priority change point is picked with probability 1/5.

Guarantees

Having generated a d-tuple of event labels $(x_0, \ldots, x_{d-1})$, the PCTCP algorithm produces a schedule which "strongly hits" this d-tuple. That is, PCTCP schedules an event labeled x_i at the last possible point in the execution before the events labeled $x_{i+1}, \ldots x_{d-1}$, thus ensuring the events are ordered in the way they appear in the tuple. In addition, PCTCP ensures a number of other ordering constraints: whenever an event x is concurrent with x_i and it is not forced to appear after x_i due to a constraint $x \geq x_j$ for some j, $i \leq j < d$, PCTCP schedules x before x_i. To guarantee these ordering constraints, PCTCP maintains a list of reduced-priority chains which are ordered based on the order of event labels in the d-tuple, e.g., the chain which has x_0 as the first unexecuted event is inserted as the first chain in the list of reduced-priority chains. When all the chains with initial priorities either finished or were reduced to a lower priority, the reduced-priority chains are executed in an order which preserves the relative order of event labels in the tuple. The crucial theoretical property we can ensure is that every possible d-tuple of events is hit with probability at least $1/(w^2 n^{d-1})$. The proof of this result appears in the next section.

6.2 Online Strong Hitting Schedulers

6.2.1 Scheduling Games

To formalize our scheduling task, we treat it as a scheduling game played by two players: Program, who reveals a poset of elements in the upgrowing fashion—each element being maximal when it appears—and Scheduler, who schedules the elements while adhering to the partial order.

We describe and analyze two versions of the scheduling game. In the first version, called *online hitting for upgrowing posets*, Scheduler maintains a family of schedules. In each step Program introduces a single new element, maximal among the old elements, and Scheduler responds by inserting the element into existing schedules without changing the order of the old elements. Scheduler is allowed to duplicate schedules before inserting the element. In this version of the game, Program has full freedom to select the relation

between the new and old elements, as long as the new element is maximal at the moment it is introduced.

In the second version of the game we will introduce a structure called *scheduling poset*. Thus, we call the game *online hitting for scheduling posets*. In this version, Scheduler maintains a single partial schedule, which it extends by appending elements at its end. Each time Scheduler schedules an additional element x, Program may extend the poset with one or more new elements, again in the upgrowing fashion, but with an additional restriction that each new element must be greater than x. This is to prevent Program in adding an element that could have been scheduled earlier in the partial schedule.

In both versions of the game, Scheduler's objective is to construct a *strong d-hitting family of schedules* for a fixed parameter $d \geq 1$, containing as few schedules as possible. The strong d-hitting property roughly says that for every d-tuple of elements $(x_0, \ldots, x_{d-1})$ in the poset there is a schedule constructed by Scheduler in which x_i appears at the last possible moment before $x_{i+1}, \ldots, x_{d-1}$, that is, if an element y is scheduled after some x_i, then it is scheduled there only because $y \geq x_j$ for some $j \geq i$. As we shall see, defining the property rigorously for scheduling posets is rather tricky.

Online hitting for scheduling posets closely corresponds to the execution model of distributed message passing programs. Scheduling an element corresponds to executing a receive event, that is, choosing a message that can be received and executing its receive handler. As a response, the handler may send new messages, inducing new receive events that can only be executed later, and never before the current receive event. This version of the game also straightforwardly generalizes the execution model of multithreaded programs and the PCT scheduler from Burckhardt et al. (2010). In this setting, scheduling an element corresponds to executing an instruction, to which the program responds by "making available" the next instruction from the same thread.

Online hitting for upgrowing posets extends the results about (weak) hitting families from Chapter 5, as well as the results about online dimension of upgrowing posets (Felsner 1997; Kloch 2007; Bosek, Felsner, et al. 2012). Recall, the (weak) d-hitting property requires that for every d-tuple $(x_0, \ldots, x_{d-1})$, if there exists a schedule α that schedules the elements in the order $x_0 <_\alpha \ldots <_\alpha x_{d-1}$ (i.e., the tuple is admissible), then such a schedule also exists in a d-hitting family of schedules. As we shall see, every strong d-hitting family is a $(d+1)$-hitting family. In the context of online dimension of upgrowing posets, online dimension can be defined as the smallest size of a 2-hitting family achievable by Scheduler.

6.2.2 Online Hitting for Upgrowing Posets

In the first version of the scheduling game, Program is arbitrarily extending a poset with new elements, and Scheduler is maintaining a strong d-hitting family of schedules for the

poset in each step, while trying to keep the number of schedules as small as possible. We start by precisely defining the objects constructed by each player.

Definition 6.1 (upgrowing poset)**.** An *upgrowing poset* of size n is a sequence of posets $\mathcal{P} = (\mathcal{P}_k)_{0\le k\le n}$ that satisfies the following conditions: (1) $\mathcal{P}_0 = \emptyset$, (2) $\mathcal{P}_{k+1} = \mathcal{P}_k \cup \{x\}$ for $k < n$, where x is a new elements such that $x \notin \mathcal{P}_k$, and (3) x is maximal in $\mathcal{P}_{k+1}$, that is, for every $y \in \mathcal{P}_{k+1}$, $y \not> x$.

Definition 6.2 (strong hitting family)**.** Let $d \ge 1$ be a fixed integer.

- Given a poset $\mathcal{P}$, we say a schedule α for $\mathcal{P}$ *strongly hits* a d-tuple of elements $(x_0, \dots, x_{d-1})$ if for every $y \in \mathcal{P}$, $y \ge_\alpha x_i$ in α for some $i \in \{0, \dots, d-1\}$ implies $y \ge x_j$ in $\mathcal{P}$ for some $j \ge i$.
- We call a set of schedules $\mathcal{F}$ a *strong d-hitting family* for $\mathcal{P}$ if for every d-tuple of elements in $\mathcal{P}$ there is a schedule in $\mathcal{F}$ that strongly hits it.
- Given an upgrowing poset $\mathcal{P} = (\mathcal{P}_k)_{0\le k\le n}$ of size n, we call a sequence of sets of schedules $\mathcal{F} = (\mathcal{F}_k)_{0\le k\le n}$ an *online strong d-hitting family* for $\mathcal{P}$ if each $\mathcal{F}_k$ is a strong d-hitting family for $\mathcal{P}_k$, and each schedule in $\mathcal{F}_{k+1}$ is an extension of a schedule in $\mathcal{F}_k$.

Remark 3. Strong hitting families are a stronger version of hitting families defined in Chapter 5, hence the name. Given a poset $\mathcal{P}$ and $d \ge 1$, a *d-hitting family* $\mathcal{F}$ is a set of schedules such that every *admissible* tuple $(x_0, \dots, x_{d-1})$ in $\mathcal{P}$ is *hit* by a schedule $\alpha \in \mathcal{F}$, that is, ordered by α as $x_o <_\alpha \dots <_\alpha x_{d-1}$. Recall, a tuple is admissible if it is hit by at least one schedule (not necessarily from $\mathcal{F}$).

Every strong d-hitting family is a $(d+1)$-hitting family. To show this, let $\mathcal{F}$ be a strong d-hitting family, and let $(x_0, \dots, x_d)$ be an admissible $(d+1)$-tuple. There is a schedule $\alpha \in \mathcal{F}$ that strongly hits $(x_1, \dots, x_d)$. We show that α hits $(x_0, \dots, x_d)$. Suppose it does not, and let i, j be indices such that $0 \le i < j \le d$ and $x_i \ge_\alpha x_j$. Since α strongly hits $(x_1, \dots, x_d)$ and $j \ge 1$, there exists $j' \ge j$ such that $x_i \ge x_{j'}$. But then, since $i < j'$, the tuple cannot be hit by any schedule, contradicting the admissibility.

The results of Felsner (1997) and Kloch (2007) (see also the survey by Bosek, Felsner, et al. (2012)) show that there is a close connection between constructing a strong 1-hitting family and an *adaptive chain covering* of an upgrowing poset. In the adaptive chain covering game, Scheduler constructs a decomposition of the poset into a (not necessarily disjoint) union of chains. That is, whenever Program adds a new element, Scheduler places it into several chains. Later on, the element may be removed from some chains to better accommodate new elements, but it must always remain in at least one chain. We formalize these requirements in the following definition.

Definition 6.3 (adaptive chain covering)**.** Let $\mathcal{P} = (\mathcal{P}_k)_{0\le k\le n}$ be an upgrowing poset of size n and Λ a set of *chain colors*. A sequence of functions $C = (C_k)_{0\le k\le n}$, where $C_k\colon \mathcal{P}_k \to 2^\Lambda$, is called an *adaptive chain covering* for $\mathcal{P}$ if the following conditions hold for all $0 \le k \le n$ and $x \in \mathcal{P}_k$: (1) $C_{k+1}(x) \subseteq C_k(x)$ for $k < n$, (2) $C_k(x) \neq \emptyset$, and (3) the set $\{x \in \mathcal{P}_k \mid \lambda \in C_k(x)\}$ is a chain for every $\lambda \in \Lambda$.

The result of Felsner and Kloch can be stated as follows. Let $\mathsf{hit}(w)$ be the least integer m such that Scheduler has a strategy for strong 1-hitting that uses at most m schedules, and let $\mathsf{adapt}(w)$ be the least integer m such that Scheduler has a strategy for adaptive chain covering that uses at most m chain colors, both on upgrowing posets of width at most w.

Theorem 6.1 (Felsner, Kloch)**.** $\mathsf{hit}(w) = \mathsf{adapt}(w)$.

Theorem 6.1 was never explicitly stated by Felsner and Kloch. In fact, they prove a stronger result that $\mathsf{dim}(w) = \mathsf{adapt}(w)$, where $\mathsf{dim}(w)$ is the maximal online dimension of upgrowing posets of width at most w. Felsner's proof of the stronger claim (Felsner 1997) originally had a flaw that was later corrected by Kloch (2007). In his correction, Kloch isolates strong 1-hitting under the name "property ($\star$)" as the key property, and essentially shows $\mathsf{dim}(w) = \mathsf{hit}(w)$ and $\mathsf{hit}(w) = \mathsf{adapt}(w)$. We emphasize the latter in Theorem 6.1 because Felsner and Kloch prove this claim by showing that a strategy for adaptive chain covering can be straightforwardly converted into a strategy for strong 1-hitting and vice versa. Thus, strong 1-hitting and adaptive chain covering are essentially the same problems.

In this chapter, we want to bound the number of schedules Scheduler needs to use to achieve strong d-hitting for arbitrary $d \ge 1$. Let $\mathsf{hit}_d(w, n)$ be the least integer m such that Scheduler has a strategy for strong d-hitting that uses at most m schedules on upgrowing posets of width at most w and size at most n. Our main result on online hitting for upgrowing posets is the following theorem.

Theorem 6.2. $\mathsf{hit}_d(w, n) \le \mathsf{adapt}(w) \cdot \binom{n}{d-1}(d-1)!\,.$

Proof. Let $\mathcal{P} = (\mathcal{P}_k)_{0\le k\le n}$ be an upgrowing poset of size n and width at most w, and let $C = (C_k)_{0\le k\le n}$ be an adaptive chain covering for $\mathcal{P}$ with m chain colors. We show how to transform C step by step into an online strong d-hitting family $\mathcal{F} = (\mathcal{F}_k)_{0\le k\le n}$ with at most $m \cdot \binom{n}{d-1}(d-1)!$ schedules.

The schedules in families $\mathcal{F}_k$ will be indexed by d-tuples $(\lambda, n_1, \ldots, n_{d-1})$, where $\lambda \in \Lambda$ is a chain color, and $n_i \in \{1, \ldots, n\}$ for $1 \le i < d$ are distinct numbers. We do not require the indexing scheme to be injective, that is, multiple indices may denote a single schedule. Intuitively, the numbers $n_1, \ldots, n_{d-1}$ serve to single out elements $x_1, \ldots, x_{d-1}$ added to the poset in steps $n_1, \ldots, n_{d-1}$, respectively. The schedule $\alpha = \alpha_{\lambda,n_1,\ldots,n_{d-1}}$ will be

constructed so that it strongly hits $(x_0, x_1, \ldots, x_{d-1})$ for every element x_0 with $\lambda \in C_k(x_0)$. More precisely, the construction will satisfy the following invariant: For $0 \leq k \leq n$, let $x_{i_1}, \ldots, x_{i_l}$ with $i_1 < \ldots < i_l$ be all the singled-out elements in step k, and let x_0 be an element such that $\lambda \in C_k(x_0)$. The schedule α strongly hits $(x_0, x_{i_1}, \ldots, x_{i_l})$.

Clearly, a family $\mathcal{F}_0$ consisting of a single empty schedule satisfies the invariant. Let x be the element added to $\mathcal{P}_k$ in step $k > 0$, and let $\mathcal{F}_{k-1}$ be a family of schedules that satisfies the invariant in step $k-1$. We show how to extend the schedule $\alpha = \alpha_{\lambda,n_1,\ldots,n_{d-1}} \in \mathcal{F}_{k-1}$ with x to obtain a schedule $\alpha' = \alpha'_{\lambda,n_1,\ldots,n_{d-1}} \in \mathcal{F}_k$ so that $\mathcal{F}_k$ satisfies the invariant in step k. We distinguish several cases:

1. If $k = n_i$ for some $i \in \{1, \ldots, d-1\}$, then x needs to be singled out: we define $x_i := x$. Let $x_{i_1}, \ldots, x_{i_l}$ with $i_1 < \ldots < i_l$ be all previously singled-out elements. If $l = 0$ or $i > i_l$ or $x \geq x_{i_l}$, we schedule x as the last element in α'. (This is possible because x is maximal in $\mathcal{P}_k$). Otherwise, let i_j be the least index such that $i < i_j$ and x is incomparable with all of $x_{i_j}, x_{i_{j+1}}, \ldots, x_{i_l}$. We schedule x right before x_{i_j}. (This is possible because of the invariant in step $k-1$: if $x \geq y$ for some $y \geq_\alpha x_{i_j}$, then $x \geq y \geq x_{i_{j'}}$ for some $j \leq j' \leq l$.)

   To see that the invariant holds for α', let x_0 be an element such that $\lambda \in C_k(x_0)$ and set $i_0 := 0$. If $x \geq_{\alpha'} x_{i_j}$ for some $j \in \{0, \ldots, l\}$, then either $i < i_j$ and $x \geq x_{i_{j'}}$ for some $j' \geq j$ by construction, or $i > i_j$ and $x \geq x_i$ trivially. Since x is singled out, we also need to inspect the case when $y \geq_{\alpha'} x$ for some $y \in \mathcal{P}_{k-1}$. By construction, x immediately precedes some previously singled-out element x_{i_j} such that $i < i_j$. Therefore, $y \geq_{\alpha'} x_{i_j}$, and by the invariant in step $k-1$, $y \geq x_{i_{j'}}$ for some $j' \geq j$.

2. If x is not to be singled out and $\lambda \in C_k(x)$, let again $x_{i_1}, \ldots, x_{i_l}$ with $i_1 < \ldots < i_l$ be all previously singled-out elements. If $l = 0$ or $x \geq x_{i_l}$, we schedule x as the last element in α'. Otherwise, let i_j be the least index such that x is incomparable with all of $x_{i_j}, x_{i_{j+1}}, \ldots, x_{i_l}$. We schedule x right before x_{i_j}.

   As before, to see that the invariant holds for α', let x_0 be an element such that $\lambda \in C_k(x_0)$ and set $i_0 := 0$. Assume $x \geq_{\alpha'} x_{i_j}$ for some $j \in \{0, \ldots, l\}$. If $j = 0$, then $x \geq x_{i_0}$ because both elements are in chain λ. If $j > 0$, by construction there exist $j' \geq j$ such that $x \geq x_{i_{j'}}$. Since $\lambda \in C_k(x)$, we also need to inspect the case when $y \geq_{\alpha'} x$ for some $y \in \mathcal{P}_{k-1}$. By construction, x immediately precedes some previously singled-out element x_{i_j}. Therefore, $y \geq_{\alpha'} x_{i_j}$, and by the invariant in step $k-1$, $y \geq x_{i_{j'}}$ for some $j' \geq j$.

3. Finally, if x is not to be singled out and $\lambda \notin C_k(x)$, we schedule x right after the last y such that $y \leq x$. To show the invariant, let $x_{i_1}, \ldots, x_{i_l}$ with $i_1 < \ldots < i_l$ be all previously singled-out elements, let x_0 be an element such that $\lambda \in C_k(x_0)$, and set $i_0 := 0$. If $x \geq_{\alpha'} x_{i_j}$ for some $j \in \{0, \ldots, l\}$, then we also have $y \geq_\alpha x_{i_j}$. Hence

$x \geq y \geq x_{i_{j'}}$ for some $j' \geq j$.

Now let $(x_0, \ldots, x_{d-1})$ be a d-tuple in $\mathcal{P}_k$. Since C is an adaptive chain covering, there exists $\lambda \in C_k(x_0)$. Moreover, there exist steps $n_1, \ldots, n_{d-1}$ in which the elements $x_1, \ldots, x_{d-1}$ were added to $\mathcal{P}_k$. Because of the invariant, the schedule $\alpha_{\lambda,n_1,\ldots,n_{d-1}} \in \mathcal{F}_k$ strongly hits the tuple. □

6.2.3 Online Hitting for Scheduling Posets

In the second version of the scheduling game, Scheduler maintains a single partial schedule of the upgrowing poset presented by Program. Scheduler takes a turn by scheduling an element x that is minimal among the non-scheduled elements. Program responds by introducing zero or more elements y such that $x < y$. New elements are introduced in the upgrowing fashion, that is, each element y is maximal in the step it is introduced.

There are two key complications in this version of the game. First, there is a mutual dependency of the upgrowing poset constructed by Program and the schedule constructed by Scheduler. And second, since only one schedule is constructed, it is unclear how to define strong hitting families. We deal with the first complication first: we upgrade the upgrowing poset to a new structure called *scheduling poset* that encodes all possible ways Scheduler can extend the schedule and Program can extend the poset.

Definition 6.4 (scheduling poset)**.** A *scheduling poset* is a pair $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$, where $\mathcal{S}$ is a set of schedules, each schedule $\alpha \in \mathcal{S}$ has an associated number $n_\alpha \geq 0$, and $\mathcal{P} = \{\mathcal{P}_{\alpha,k} \mid \alpha \in \mathcal{S}, 0 \leq k \leq n_\alpha\}$ is a set of posets satisfying the following conditions.

Initial conditions:

(1) $\epsilon \in \mathcal{S}$
(2) $\mathcal{P}_{\epsilon,0} = \emptyset$

Extending the schedule:

(3) $\alpha \cdot x \in \mathcal{S}$ if and only if $\alpha \in \mathcal{S}$ and $x \in \min(\mathcal{P}_{\alpha,n_\alpha} \setminus \alpha)$
(4) $\mathcal{P}_{\alpha\cdot x,0} = \mathcal{P}_{\alpha,n_\alpha}$

Extending the poset:

(5) $\mathcal{P}_{\alpha,k+1} = \mathcal{P}_{\alpha,k} \cup \{x\}$, where $k < n_\alpha$ and $x \notin \mathcal{P}_{\alpha,k}$
(6) x is maximal in $\mathcal{P}_{\alpha,k+1}$, that is, for every $y \in \mathcal{P}_{\alpha,k+1}$, $y \not> x$
(7) x is greater than the last scheduled element, that is, if $\alpha = \alpha' \cdot y$, then $y < x$

The numbers n_α in Definition 6.4 represent the number of new elements Program adds into the poset after the Scheduler extends the schedule to α. The poset $\mathcal{P}_{\alpha,k}$ for $0 \leq k \leq n_\alpha$ is the poset in the k-th step after scheduling α. We will also be referring to the *cumulative*

step for α and k: Let l be the length of α, and let α_i for $0 \leq i \leq l$ denote the prefix of α of length i. The cumulative step for α and k is the number $t = n_{\alpha_0} + \ldots + n_{\alpha_{l-1}} + k$. It is not difficult to see that a scheduling poset in cumulative step t has precisely t elements.

Note that Definition 6.4 of scheduling posets never explicitly requires a schedule α to extend a prefix of $\mathcal{P}_{\alpha,k}$. However, this fact can easily be derived from the definition.

Lemma 6.3. *Let $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$ be a scheduling poset. For every $\alpha \in \mathcal{S}$ and k such that $0 \leq k \leq n_\alpha$, α is a schedule of a prefix of $\mathcal{P}_{\alpha,k}$.*

Proof. The claim clearly holds for $\alpha = \epsilon$ and all k such that $0 \leq k \leq n_\epsilon$. Assume the claim holds for some $\alpha \in \mathcal{S}$ and all k such that $0 \leq k \leq n_\alpha$ and let $x \in \min(\mathcal{P}_{\alpha,n_\alpha} \setminus \alpha)$. Since $\mathcal{P}_{\alpha\cdot x,0} = \mathcal{P}_{\alpha,n_\alpha}$, and since x is minimal in $\mathcal{P}_{\alpha,n_\alpha} \setminus \alpha$, there exists no $y \in \mathcal{P}_{\alpha\cdot x,0} \setminus \alpha \cdot x$ such that $y \leq x$. Therefore, $\alpha \cdot x$ is a schedule of a prefix of $\mathcal{P}_{\alpha\cdot x,0}$. Now assume $\alpha \cdot x$ is a schedule of a prefix of $\mathcal{P}_{\alpha\cdot x,k}$ for some k such that $0 \leq k < n_{\alpha\cdot x}$. Let $\mathcal{P}_{\alpha\cdot x,k+1} = \mathcal{P}_{\alpha\cdot x,k} \cup \{y\}$. Since y is maximal in $\mathcal{P}_{\alpha\cdot x,k+1}$, we cannot have $y \leq x$. (This also follows from $x < y$.) Therefore, $\alpha \cdot x$ is a schedule of a prefix of $\mathcal{P}_{\alpha\cdot x,k+1}$. The claim now follows by sub-induction on k and induction on α. □

As with online hitting for upgrowing posets, our result for scheduling posets will be to show how to convert a strategy for adaptive chain covering to a strategy for online hitting for scheduling posets. Therefore, we need to extend the definition of adaptive chain covering to scheduling posets.

Definition 6.5 (adaptive chain covering for scheduling posets)**.** Let Λ be a set of *chain colors*, and $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$ a scheduling poset. A set of functions $C = \{C_{\alpha,k} \colon \mathcal{P}_{\alpha,k} \to 2^\Lambda \mid \alpha \in \mathcal{S}, 0 \leq k \leq n_\alpha\}$ is called an *adaptive chain covering for $\mathcal{SP}$* if the following conditions hold for all $\alpha \in \mathcal{S}$, $0 \leq k \leq n_\alpha$, and $x \in \mathcal{P}_{\alpha,k}$: (1) $C_{\alpha,k+1}(x) \subseteq C_{\alpha,k}(x)$ if $k < n_\alpha$, (2) $C_{\alpha\cdot y,0}(x) = C_{\alpha,n_\alpha}(x)$, (3) $C_{\alpha,k}(x) \neq \emptyset$, and (4) the set $\{x \in \mathcal{P}_{\alpha,k} \mid \lambda \in C_{\alpha,k}(x)\}$ is a chain for every $\lambda \in \Lambda$.

By defining scheduling posets, we have solved the first of the two complications mentioned earlier. We have also solved part of the second complication: given a scheduling poset $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$, a strong d-hitting family will be some subset $\mathcal{F} \subseteq \mathcal{S}$. But how do we define the strong d-hitting property? Note that we cannot quantify over d-tuples $(x_0, \ldots, x_{d-1})$, because as soon as we fix a domain for some d-tuple, say $\mathcal{P}_{\alpha,k}$, we have fixed the schedule α, and this schedule does not necessarily hit the tuple. We deal with this complication by employing a trick from Burckhardt et al. (2010): instead of tuples, we quantify over auxiliary functions called *labelings* that indirectly select the tuples for us.

Definition 6.6 (labeling)**.** Let $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$ be a scheduling poset, $d \geq 1$ a fixed integer, and $L = \{x_0, \ldots, x_{d-1}\}$ an ordered set of labels. A *d-labeling* for $\mathcal{SP}$ is a set of partial

functions $\mathcal{L} = \{\mathcal{L}_{\alpha,k} : L \nrightarrow \mathcal{P}_{\alpha,k} \mid \alpha \in \mathcal{S}, 0 \leq k \leq n_\alpha\}$ satisfying the following conditions for every $\alpha \in \mathcal{S}$ and $0 \leq k \leq n_\alpha$:

(1) $\mathcal{L}_{\alpha,k}$ is injective.

(2) $\mathcal{L}_{\alpha\cdot x,0} = \mathcal{L}_{\alpha,n_\alpha}$ for $x \in \min(\mathcal{P}_{\alpha,n_\alpha} \setminus \alpha)$, and if $k < n_\alpha$, then $\mathrm{dom}(\mathcal{L}_{\alpha,k}) \subseteq \mathrm{dom}(\mathcal{L}_{\alpha,k+1})$ and $\mathcal{L}_{\alpha,k+1}(x_i) = \mathcal{L}_{\alpha,k}(x_i)$ for every $x_i \in \mathrm{dom}(\mathcal{L}_{\alpha,k})$.

(3) If $k < n_\alpha$ and $\mathcal{P}_{\alpha,k+1} = \mathcal{P}_{\alpha,k} \cup \{x\}$, then $\mathrm{dom}(\mathcal{L}_{\alpha,k+1}) \setminus \mathrm{dom}(\mathcal{L}_{\alpha,k})$ contains at most one label x_i, for which $\mathcal{L}_{\alpha,k+1}(x_i) = x$.

(4) For every adaptive chain covering C for $\mathcal{SP}$, there exists a chain color λ such that $\lambda \in C_{\alpha,k}(\mathcal{L}_{\alpha,k}(x_0))$ for every schedule α and step $0 \leq k \leq n_\alpha$ in which $x_0 \in \mathrm{dom}(\mathcal{L}_{\alpha,k})$.

When α and k are clear from the context, we usually write x_i instead of $\mathcal{L}_{\alpha,k}(x_i)$.

Intuitively, the conditions in Definition 6.6 require the labels to be assigned to distinct elements; they require them to be stable, and only assigned to newly added elements. Condition 4 requires that for every adaptive chain covering there is a chain that contains x_0 irrespective of the way we schedule the elements.

Definition 6.7 (strong hitting family for scheduling posets)**.** Let $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$ be a scheduling poset, $d \geq 1$ a fixed integer, $\mathcal{L}$ a d-labeling for $\mathcal{SP}$, and $\alpha \in \mathcal{S}$ a schedule.

- We say α *partially hits* $\mathcal{L}_{\alpha,k}$ for $0 \leq k \leq n_\alpha$ if for every $x_i \in \mathrm{dom}(\mathcal{L}_{\alpha,k})$ scheduled by α and every $x \in \mathcal{P}_{\alpha,k}$ such that either $x \geq_\alpha x_i$ or x is not scheduled, there exists $x_j \in \mathrm{dom}(\mathcal{L}_{\alpha,k})$ with $j \geq i$ such that $x \geq x_j$ in $\mathcal{P}_{\alpha,k}$. We say α *partially hits* $\mathcal{L}$ if it partially hits $\mathcal{L}_{\alpha,k}$ for every $0 \leq k \leq n_\alpha$.

- If α is complete, that is, it schedules the whole $\mathcal{P}_{\alpha,n_\alpha}$, and it partially hits $\mathcal{L}$, we say it *strongly hits* $\mathcal{L}$.

- We say $\mathcal{L}$ is *complete* if for each of its strongly hitting schedules α all labels are assigned in $\mathcal{P}_{\alpha,n_\alpha}$, that is, $\mathcal{L}_{\alpha,n_\alpha}$ is a total function.

- A set of complete schedules $\mathcal{F} \subseteq \mathcal{S}$ is a *strong d-hitting family* for $\mathcal{SP}$ if for every complete d-labeling $\mathcal{L}$ for $\mathcal{SP}$ there is a schedule $\alpha \in \mathcal{F}$ that strongly hits $\mathcal{L}$.

The following lemma shows that in order to maintain partial hitting, it suffices for Scheduler to preserve the property on their move. In other words, Program cannot break the property by cleverly introducing a new element.

Lemma 6.4. *Let $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$ be a scheduling poset, $d \geq 1$ a fixed integer, $\mathcal{L}$ a d-labeling for $\mathcal{SP}$, and $\alpha \in \mathcal{S}$ a schedule. The following statements are equivalent:*

(1) α partially hits $\mathcal{L}$,

(2) α partially hits $\mathcal{L}_{\alpha,k}$ for some $0 \leq k \leq n_\alpha$,

(3) α partially hits $\mathcal{L}_{\alpha,0}$.

Proof. Clearly (1) implies (2). In order to show that (2) implies (3), assume α partially hits $\mathcal{L}_{\alpha,k}$ for some $k > 0$. We show α partially hits $\mathcal{L}_{\alpha,k-1}$ and conclude by downward induction on k. Let $\mathcal{P}_{\alpha,k} = \mathcal{P}_{\alpha,k-1} \cup \{x\}$, let $x_i \in \mathrm{dom}(\mathcal{L}_{\alpha,k-1})$ be an element scheduled by α, and let $y \in \mathcal{P}_{\alpha,k-1}$ be an element such that either $y \geq_\alpha x_i$ or y is not scheduled. Since α partially hits $\mathcal{L}_{\alpha,k}$, there exists $x_j \in \mathrm{dom}(\mathcal{L}_{\alpha,k})$ with $j \geq i$ such that $y \geq x_j$. If $x_j \in \mathcal{P}_{\alpha,k-1}$, we are done. Suppose $x_j \notin \mathcal{P}_{\alpha,k-1}$; then $x_j = x$. But then $x < y$ in $\mathcal{P}_{\alpha,k}$, contradicting the maximality of x.

We show that (3) implies (1) by (upward) induction on k. The statement (3) is the base case. Assume α partially hits $\mathcal{L}_{\alpha,k}$ for some $k < n_\alpha$, let $\mathcal{P}_{\alpha,k+1} = \mathcal{P}_{\alpha,k} \cup \{x\}$, let $x_i \in \mathrm{dom}(\mathcal{L}_{\alpha,k+1})$ be an element scheduled by α, and let $y \in \mathcal{P}_{\alpha,k+1}$ be an element such that either $y \geq_\alpha x_i$ or y is not scheduled. Note that $x_i \in \mathcal{P}_{\alpha,k}$. If $y \in \mathcal{P}_{\alpha,k}$, we are done; otherwise $y = x$. Since α schedules x_i, we know that $\alpha = \alpha' \cdot z$ for some $z \in \mathcal{P}_{\alpha,k}$, and moreover $z \geq_\alpha x_i$. By the induction hypothesis, there exists $x_j \in \mathrm{dom}(\mathcal{L}_{\alpha,k})$ with $j \geq i$ such that $z \geq x_j$. Since $y = x > z$, by transitivity we have $y \geq x_j$. □

In contrast to Theorem 6.2, which states the result for online hitting for upgrowing posets using quantities $\mathrm{hit}_d(w, n)$ and $\mathrm{adapt}(w)$, we state the result in this subsection in a more operational way. To that end, we define an auxiliary notion of *schedule indices*: Given a scheduling poset $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$ of size at most n, a fixed integer $d \geq 1$, and an adaptive chain covering C for $\mathcal{SP}$ with the set of chain colors Λ, we say a *schedule index* is a d-tuple of the form $(\lambda, n_1, \ldots, n_{d-1})$, where $\lambda \in \Lambda$ is a chain color, and $n_i \in \{1, \ldots, n\}$ for $1 \leq i \leq n$ are distinct numbers. Intuitively, given a labeling $\mathcal{L}$ and a schedule α, the numbers n_i represent the cumulative steps in which $\mathcal{L}$ assigns x_i to new elements in the poset, and λ represents the color of a chain containing x_0. If $\mathcal{L}$ assigns labels in this way by following α, we say $\mathcal{L}$ *conforms to* $(\lambda, n_1, \ldots, n_{d-1})$ *on* α.

Lemma 6.5. *Let $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$ be a scheduling poset of size at most n, $d \geq 1$ a fixed integer, and C an adaptive chain covering for $\mathcal{SP}$. For every schedule index $(\lambda, n_1, \ldots, n_{d-1})$ there is a schedule $\alpha = \alpha_{\lambda,n_1,\ldots,n_{d-1}}$ such that α strongly hits every complete d-labeling that conforms to $(\lambda, n_1, \ldots, n_{d-1})$ on α.*

Proof. Let $(\lambda, n_1, \ldots, n_{d-1})$ be a schedule index. We construct the schedule $\alpha = \alpha_{\lambda,n_1,\ldots,n_{d-1}}$ inductively. The invariant maintained during the construction is that α partially hits every labeling that conforms to $(\lambda, n_1, \ldots, n_{d-1})$ on α.

Base case: $\alpha = \epsilon$. Since $\mathcal{P}_{\epsilon,0} = \emptyset$, ϵ trivially hits $\mathcal{L}_{\epsilon,0}$ for any labeling $\mathcal{L}$. By Lemma 6.4, ϵ partially hits every labeling $\mathcal{L}$.

Induction step. Assume we have constructed some $\alpha = \alpha_{\lambda,n_1,\ldots,n_{d-1}}$ that satisfies the invariant. If all elements have been scheduled, we are done. Otherwise, we show how to select $x \in \min(\mathcal{P}_{\alpha,n_\alpha} \setminus \alpha)$ to extend α into $\alpha' = \alpha \cdot x$ without breaking the invariant. There are three cases:

1. There exists $x \in \min(\mathcal{P}_{\alpha,n_\alpha} \setminus \alpha)$ such that $\lambda \notin C_{\alpha,n_\alpha}(x)$ and x was not added in cumulative step n_i for any $1 \le i < d$. We extend α with any such x.

2. Otherwise, there exists $x \in \min(\mathcal{P}_{\alpha,n_\alpha} \setminus \alpha)$ such that $\lambda \in C_{\alpha,n_\alpha}(x)$ and x was not added in cumulative step n_i for any $1 \le i < d$. We extend α with any such x.

3. Otherwise, every $x \in \min(\mathcal{P}_{\alpha,n_\alpha} \setminus \alpha)$ was added in cumulative step n_i for some $1 \le i < d$. We extend α with x added in step n_i for the least index i.

Let $\mathcal{L}$ be a labeling conforming to $(\lambda, n_1, \ldots, n_{d-1})$ on α'. Since it also conforms to $(\lambda, n_1, \ldots, n_{d-1})$ on α, it is partially hit by α. We may have broken the partial hitting property if we have extended the schedule with x_0 in the second case, or with x_i for $1 \le i < d$ in the third case.

In the second case, let y be some element that is not yet scheduled, and let $y' \in \min(\mathcal{P}_{\alpha,n_\alpha} \setminus \alpha)$ be an element such that $y' \le y$ (such y' always exists). Since we are in the second case, either $\lambda \in C_{\alpha,n_\alpha}(y')$, implying $y \ge y' \ge x_0$, or $y' = x_j$ for some $1 \le j < d$. In either case, $y \ge x_j$ for some $0 \le j < d$.

In the third case, let y be some element that is not yet scheduled, and again, let $y' \in \min(\mathcal{P}_{\alpha,n_\alpha} \setminus \alpha)$ be an element such that $y' \le y$. Since we are in the third case, $y' = x_j$ for some $1 \le j < d$. Since we have extended α with x_i having the least index i, we have $j \ge i$.

This shows that α' partially hits $\mathcal{L}_{\alpha',0}$. By Lemma 6.4, α' partially hits $\mathcal{L}$. □

Lemma 6.6. *Let $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$ be a scheduling poset of size at most n, $d \ge 1$ a fixed integer, and C an adaptive chain covering for $\mathcal{SP}$. For every complete d-labeling $\mathcal{L}$ there is a schedule index $(\lambda, n_1, \ldots, n_{d-1})$ such that $\mathcal{L}$ conforms to $(\lambda, n_1, \ldots, n_{d-1})$ on $\alpha_{\lambda,n_1,\ldots,n_{d-1}}$.*

Proof. Let $\mathcal{L}$ be a complete d-labeling, and let λ be a chain color such that $\lambda \in C_{\alpha,k}(x_0)$ for every schedule α and step $0 \le k \le n_\alpha$ in which x_0 is defined. Note that we can repeat the construction from the proof of Lemma 6.5 with the knowledge of λ and the

actual elements $x_1, \ldots, x_{d-1}$ selected by $\mathcal{L}$ instead of the knowledge of the schedule index. During the construction, we take note of cumulative steps n_i in which $\mathcal{L}$ assigns labels x_i. By the invariant, the schedule α obtained at the end strongly hits $\mathcal{L}$. Since $\mathcal{L}$ is complete, all labels have been assigned at the end, hence $C_{\alpha,n_\alpha}(x_0)$ is a well-defined set of chain colors containing λ, and n_i are well-defined numbers for all $1 \leq i < d$. By construction, $\alpha = \alpha_{\lambda,n_1,\ldots,n_{d-1}}$ and $\mathcal{L}$ conforms to $(\lambda, n_1, \ldots, n_{d-1})$ on α. □

Theorem 6.7. *Let $\mathcal{SP} = (\mathcal{S}, \mathcal{P})$ be a scheduling poset of size at most n, $d \geq 1$ a fixed integer, and C an adaptive chain covering for $\mathcal{SP}$. The set*

$$\mathcal{F} = \{\alpha_{\lambda,n_1,\ldots,n_{d-1}} \mid (\lambda, n_1, \ldots, n_{d-1}) \text{ is a schedule index for } \mathcal{SP}\}$$

is a strong d-hitting family for $\mathcal{SP}$. If C uses m chain colors, then $\mathcal{F}$ has size at most $m\binom{n}{d-1}(d-1)!$.

Proof. Let $\mathcal{L}$ be a complete d-labeling for $\mathcal{SP}$. By Lemma 6.6, there exists a schedule index $(\lambda, n_1, \ldots, n_{d-1})$ such that $\mathcal{L}$ conforms to $(\lambda, n_1, \ldots, n_{d-1})$ on $\alpha = \alpha_{\lambda,n_1,\ldots,n_{d-1}}$. By Lemma 6.5, α strongly hits $\mathcal{L}$, and by definition, $\alpha \in \mathcal{F}$. Finally, the size of $\mathcal{F}$ is bounded by the total number of schedule indices. □

6.2.4 Online Chain Partitioning

Our two main results, Theorem 6.2 and Theorem 6.7, show that Scheduler can construct strong hitting families of bounded size provided they have a strategy for adaptive chain covering. Adaptive chain covering is essentially an online decomposition of an upgrowing poset into a (not necessarily disjoint) union of chains. In particular, any strategy for online chain *partitioning*, which decomposes the poset into a disjoint union of chains, is a strategy for adaptive chain covering.

By Dilworth's theorem (Dilworth 1950), the optimal chain partition of a poset of width w uses w chains. In the online setting, the optimal partition may not be achievable. As shown by Felsner (1997), for upgrowing posets of width at most w, Scheduler always has a strategy for chain partitioning that uses at most $\binom{w+1}{2}$ chains, and Program can force Scheduler to use $\binom{w+1}{2}$ chains. This bound translates into an upper bound for $\mathrm{adapt}(w)$, the minimal number of chains needed for adaptive chain covering over all upgrowing posets of width at most w. By plugging the bound into our main theorems, we can bound the size of strong hitting families.

Corollary 6.8. *Given $d \geq 1$, for any upgrowing or scheduling poset of width at most w and size at most n, there is a strong d-hitting family of schedules of size at most $\binom{w+1}{2}\binom{n}{d-1}(d-1)! \leq w^2 n^{d-1}$.*

Input: A new element y
Data: Sets of chains $B_1, \ldots, B_w$
Invariant: $\forall i : 1 \leq i \leq w \implies |B_i| \leq i$,
Invariant: $\forall i : 1 \leq i \leq w \implies \text{Last}(B_i) := \{x \mid \alpha \cdot x \in B_i\}$ is an antichain

Procedure `addNewElement(`y`)`

```
1 | for i = 1 to w do
2 | | if (∃α' · x ∈ B_i : x < y) or |B_i| < i then
3 | | | α ← α' · x if it exists, or a new empty chain otherwise
4 | | | α ← α · y
5 | | | if i > 1 then
6 | | | | (B_{i−1}, B_i) ← (B_i \ {α}, B_{i−1} ∪ {α})
7 | | | return
```

Algorithm 2: Chain partitioning algorithm: adding a new element into a chain

There is a surprisingly elegant algorithm for online chain partitioning given by Agarwal and Garg (2007), listed as Algorithm 2. The algorithm is optimal in the sense that it uses at most $\binom{w+1}{2}$ chains for upgrowing posets of width at most w. The algorithm maintains w sets of chains $B_1, \ldots, B_w$ such that each B_i contains at most i chains. Moreover, define $\text{Last}(B_i) := \{x \mid \alpha \cdot x \in B_i\}$. The algorithm maintains the invariant that $\text{Last}(B_i)$ is an antichain for every $1 \leq i \leq w$. Let y be the new maximal element added to the poset. The algorithm finds the least index i such that y is comparable with some $x \in \text{Last}(B_i)$ or B_i has less than i chains. (Such i exists, otherwise $\text{Last}(B_w) \cup \{y\}$ is an antichain of size $w + 1$.) Let $\alpha = \alpha' \cdot x$ if y is comparable with x for $\alpha' \cdot x \in B_i$, otherwise let $\alpha = \epsilon$. The algorithm extends α to $\alpha \cdot y$ in B_i. If $i > 1$, the algorithm swaps the chains in B_{i-1} and B_i so that in the next step $B'_{i-1} = B_i \setminus \{\alpha \cdot y\}$ and $B'_i = B_{i-1} \cup \{\alpha \cdot y\}$. It is not difficult to see that the invariant of the algorithm is preserved (Agarwal and Garg 2007). Notice that the width of the poset w need not be known upfront. As the algorithm inserts elements to the poset, it creates new chains as needed. By means of the invariant enforcing each $\text{Last}(B_i)$ to be an antichain, the largest set of chains B_w is at most of size as the width of the poset, w.

Note that since adaptive chain covering allows decompositions of the poset into non-disjoint chains, it is possible that there exist strategies which use fewer than $\binom{w+1}{2}$ chains. Unfortunately, no better strategies than online chain partitioning are currently known (Bosek, Felsner, et al. 2012). However, in case of future progress on adaptive chain covering, any new bounds on $\text{adapt}(w)$ will automatically translate into new bounds on the size of online strong hitting families.

6.2.5 PCTCP—PCT with Chain Partitioning

We now relate our algorithm *PCTCP (Probabilistic Concurrency Testing with Chain Partitioning)*, introduced as Algorithm 1 and described informally in Sec. 6.1, to the results discussed in Sec. 6.2.3 and Sec 6.2.4. PCTCP incorporates Agarwal and Garg's online chain partitioning algorithm into the construction of strong hitting families for scheduling posets. However, instead of constructing the whole strong hitting family, it selects a scheduling index uniformly at random and constructs only the corresponding schedule. Therefore, it provides a bound on the probability of hitting a bug of depth d:

Corollary 6.9. *Given a scheduling poset $\mathcal{SP}$ of size at most n and width at most w, a schedule constructed by PCTCP strongly hits a d-complete labeling for $\mathcal{SP}$ with probability at least $1/(w^2 n^{d-1})$.*

To pick a scheduling index uniformly at random, PCTCP uses a priority-based randomized scheduler similar to PCT—the randomized scheduler for multithreaded programs by Burckhardt et al. (2010). The PCTCP scheduler assigns a priority uniformly at random to each chain as it is constructed and added to the partition on-the-fly. It then, at each step of computation, schedules an *enabled* event from a chain with the highest priority. An event is enabled if it is not scheduled and all of its predecessors have been scheduled. The priority of a chain may change during the execution when it passes a *priority change point*. These points are steps in an execution with associated priorities which are lower than the priorities assigned to chains initially. When the execution reaches a priority change point, the scheduler adjusts the priority of the corresponding chain to the priority associated with the change point. More specifically, given inputs d and n, PCTCP assigns priority values $d, d+1, \ldots, d+w^2-1$ to chains which are constructed dynamically (we can have up to w^2 chains). It also picks initially $d-1$ random priority change points $n_1, \ldots, n_{d-1}$ in the range $[1, n]$, where each n_i has an associated priority value of i. Combining the initial priority assignments and the priority change points, PCTCP generates a schedule index $(\lambda, n_1, \ldots, n_{d-1})$, where λ is the chain with the lowest initial priority.

We argue that PCTCP is a natural generalization of PCT for multithreaded programs (Burckhardt et al. 2010). The main difference is that PCTCP uses a chain partition constructed on-the-fly, and PCT uses a chain partition provided by threads. Hence the difference in the probabilistic guarantee is $1/(w^2 n^{d-1})$ for PCTCP versus $1/(kn^{d-1})$ for PCT on a program with k threads.

Another difference between the two is in the phrasing of the objective of the generated schedule. PCTCP's objective is to "strongly hit a labeling," whereas PCT's objective is to "satisfy a directive that guarantees a bug." PCT's directive for a given $d \geq 1$ is a tuple $D = (\mathcal{L}, A_0, x_0, \ldots, A_{d-1}, x_{d-1})$, where $\mathcal{L}$ is a labeling whose set of labels L can contain labels other than $x_0, \ldots, x_{d-1}$, and each $A_i \subseteq L$ is a set of labels. A schedule α

satisfies a directive if it schedules all $a \in A_i$ before x_i, for every $0 \leq i < d$, and schedules $x_0, \ldots, x_{d-1}$ in the order $x_0 <_\alpha \ldots <_\alpha x_{d-1}$. Thus, a directive represents a set of additional ordering constraints a schedule should satisfy in order to expose a bug. The constraints are implicitly assumed to be consistent with the program's partial order.

It is not difficult to see that the strong hitting property subsumes PCT's directives. Let $D = (\mathcal{L}, A_0, x_0, \ldots, A_{d-1}, x_{d-1})$ be a directive, and assume a schedule α strongly hits $(x_0, \ldots, x_{d-1})$. We first show that the elements $x_0, \ldots, x_{d-1}$ are ordered as $x_0 <_\alpha \ldots <_\alpha x_{d-1}$. Suppose they are not, that is, suppose $x_j \geq_\alpha x_i$ for some $j < i$. By the strong hitting property, there exists $j' \geq i$ such that $x_j \geq x_{j'}$. But then, since $j < j'$, the directive is inconsistent with the partial order. We conclude that α correctly schedules $x_0 <_\alpha \ldots <_\alpha x_{d-1}$. Suppose now that $a \geq_\alpha x_i$ for some $0 \leq i < d$ and $a \in A_i$. By the strong hitting property, there exists $j \geq i$ such that $a \geq x_j$. Again, since $i \leq j$, according to the directive the element a should be scheduled before x_j, which is inconsistent with the partial order. We conclude all $a \in A_i$ are scheduled before x_i for every $0 \leq i < d$. Thus, α satisfies the directive D.

6.3 Experimental Evaluation

6.3.1 P# Benchmarks

We implemented PCTCP[3] to randomly test distributed applications written in Microsoft's P# framework[4] for building asynchronous message passing systems (Deligiannis, Donaldson, et al. 2015; Deligiannis, McCutchen, et al. 2016; Mudduluru et al. 2017). A P# program consists of a number of state machines that communicate by sending and receiving *messages*. Each P# machine executes a message handling loop and runs in parallel with other machines. Handling of a message can result in a state transition, creating new machines, sending messages to other machines, or updating local fields. The systematic testing engine of P# instruments a program at synchronization points, which are *send*, *create-machine*, and *receive* events. Upon execution of one of these events, the P# runtime calls the scheduler, which blocks the current machine and releases a possibly different machine for execution. Therefore, a machine may be interrupted in the middle of handling an incoming message in case the handling causes sending a new message or creating a new machine.

The original P# runtime does not keep track of causal dependencies between events, and thus does not have an explicit notion of chains. At synchronization points, the scheduler only knows the set of currently executing machines, and chooses one of them as the

[3]The source code is available at `https://gitlab.mpi-sws.org/fniksic/PSharp/tree/PCTCP`.

[4]`https://github.com/p-org/PSharp`

Table 6.1: Characteristics of benchmarks (LOC includes comments and blank lines, "Type of bug" refers to known bugs)

Benchmark	LOC	Machine types	Message types	Type of bug
BoundedAsync	288	2	7	safety
ChainReplication	1,562	5	46	safety
Raft	1,302	5	29	safety
Chord	917	3	22	liveness
ReplicatingStorage	978	7	37	liveness
FailureDetector	674	4	19	both
TwoPhaseCommit	725	5	29	-
MultiPaxos	1,095	5	26	-
CacheCoherence	420	3	17	-

next one to schedule. The choice is determined by the scheduling strategy; among others, the implemented strategies include the "random walk," which selects the next machine uniformly at random, and "prioritized strategy," which randomly selects d scheduling points during execution order and prioritizes them to make sure they are ordered in a particular way. The latter strategy is similar to PCT, and it is called PCT in the P# source code, but without the notion of chains it does not provide the same probabilistic guarantee. Therefore we call it "prioritized strategy" to avoid confusion.

In order to keep track of causal dependencies, we implemented our own version of the P# runtime called "PCTCP runtime". Additionally, we simplified the scheduler to only schedule the receive events. In fact, the underlying concurrency model of PCTCP runtime is coarser as it introduces fewer synchronization points. Therefore, it may miss behaviors arising from interleavings of different message handlers. However, it considers all possible reorderings between concurrent events which may lead to a concurrency bug. On top of this simplified concurrency model, we implemented the PCTCP and the random walk scheduling strategies. The random walk strategy selects the next event uniformly at random among the enabled chains.

We evaluate our method on 9 sample implementations of distributed algorithms in the P# framework, which were also used in previous work (Mudduluru et al. 2017; Deligiannis, Donaldson, et al. 2015). Table 6.1 shows the characteristics of the P# benchmarks including lines of code (LOC), number of machines and message types, and the type (safety or liveness) of the underlying (known) bug(s).

Table 6.2 shows the result of applying PCTCP on P# benchmarks. For each bench-

Table 6.2: Results of applying PCTCP to P# benchmarks including number of buggy schedules, average number of computed chains, maximum number of produced messages and the running time.

Benchmark	Event labels (d)	Runs	Buggy (%)	Avg chains	Max msgs (n)	Time (s)
BoundedAsync	1	10,000	98.97	12	128	71.48
ChainReplication	5	1,000	12.60	18	362	635.18
Raft	1	1,000	0.20	37	590	210.53
Chord	1	1,000	6.10	5	62	6.62
ReplicatingStorage	1	100	11.00	24	899	504.73
FailureDetector	1	5,000	0.36	27	172	360.87
TwoPhaseCommit	1	10,000	0.00	9	42	50.63
MultiPaxos	1	10,000	0.00	32	754	552.82
CacheCoherence	1	10,000	0.00	6	465	988.96

mark we ran PCTCP for a number of times (**Runs**) with a given value of parameter d (**Event labels**), and measured the average number of computed chains (**Avg chains**), the maximum number of messages in the partial order (**Max msgs**), the percentage of buggy schedules (**Buggy (%)**), and the execution time in seconds (**Time (s)**).

To choose the value of parameter d, we start with the minimum value of 1 and increment it only if it is not sufficient to catch at least one bug in 10,000 runs. As we can see in Table 6.2, for *ChainReplication* we increased the value of d up to 5 to catch the two underlying safety bugs. For the last three benchmarks in this table, we only experimented with $d = 1$. These examples do not have bugs discoverable by our reference methods from the P# framework as we will see in the following. Note that the bugs found with some specific value of d do not necessarily have the bug depth of d.

From Table 6.2, it can be inferred that the measured probability of catching a bug of depth d is higher than the guaranteed probability of PCTCP. This is mainly due to the fact that some benchmarks had more than one bug (assertion violation). For example, we observed two different assertion violations in *ChainReplication* and *FailureDetector*. Moreover, due to symmetry in these protocols various d-tuples can result in the same assertion violation.

Table 6.3 reports the result of comparing the effectiveness of PCTCP in detecting bugs (column 2) with other three reference methods: the random walk strategy in the PCTCP runtime (column 3), the prioritized strategy (column 4) and the random walk (column 5) in

Table 6.3: Comparison of effectiveness of PCTCP in bug detection with three other methods.

Benchmark	PCTCP runtime				Original P# runtime			
	PCTCP		Random walk		Prioritized strategy		Random walk	
	Buggy (%)	Time (s)	Buggy (%)	Time (s)	Buggy (%)	Time (s)	Buggy (%)	Time(s)
BoundedAsync	98.97	71.48	99.04	71.35	0.00	88.60	0.00	76.98
ChainReplication	12.60	635.18	17.50	325.09	0.00	34.98	0.00	25.42
Raft	0.20	210.53	1.10	124.17	0.00	29.11	1.30	27.82
Chord	6.10	6.62	5.90	4.35	5.50	8.09	4.80	7.03
ReplicatingStorage	11.00	504.73	23.00	112.47	0.00	9.28	24.00	24.21
FailureDetector	0.36	360.87	0.08	146.32	0.00	2,267.71	0.00	3,012.57
TwoPhaseCommit	0.00	50.63	0.00	35.97	0.00	27.21	0.00	26.57
MultiPaxos	0.00	552.82	0.00	398.86	0.00	129.59	0.00	106.19
CacheCoherence	0.00	988.96	0.00	758.06	0.00	197.52	0.00	197.34

the original P# runtime. Recall that the two runtimes differ in the underlying concurrency model—the PCTCP runtime only schedules message receive events. The number of runs and the parameter d for each benchmark are the same as in Table 6.2.

Table 6.3 shows that both PCTCP and its random walk version are more effective in detecting bugs than the original random walk and prioritized strategies of P#. The assertion violation in the *Process* machine of *BoundedAsync* was caught by the PCTCP runtime in nearly all runs (using either strategy). However, both the prioritized and the random walk strategies under the original P# runtime failed to reveal this bug by exploring up to 10,000 schedules. PCTCP found two assertion violations in *ChainReplication*. The one in *ChainReplicationMaster* machine was detected with $d = 1$; however for detecting the violation in *InvariantMonitor* we had to increase the value of d to 5 (Table 6.2). Both the prioritized (with $d \leq 5$) and the random walk strategies of the P# framework did not find these assertion violations by exploring up to 10,000 schedules (the running times given in Table 6.3 for this benchmark are for 1,000 runs for the sake of comparison). Only for some specific random seed values, the prioritized scheduler of P# runtime could find the assertion violation in *InvariantMonitor* in 1 out of 10,000 runs (0.01% buggy schedules). PCTCP also performed more effectively than any strategy under the original P# runtime by detecting one liveness and one safety bug in *FailureDetector*. Both strategies of the P# runtime failed to detect these bugs in 5,000 runs. The prioritized strategy could find this bug in 70 out of 5,000 when applying $d = 2$ and the specific random seed value given in the test suite of P# for this benchmark.

Note that, for *ReplicatingStorage*, we also compared the P# prioritized scheduler and the PCTCP scheduler based on similar time budgets. However, no bug was caught by the P# prioritized scheduler after exploring 100,000 schedules in 698.65 sec. We did the same

comparison for *Raft*.

As Table 6.3 shows, PCTCP is effective in detecting bugs in practice. It may not necessarily always outperform random walk, but in contrast it provides theoretical guarantees on finding bugs.

Livelocks

Because the PCTCP scheduler always schedules events from the chain with highest priority, it lacks fairness. This can result in livelocks in message-passing systems. For example, the *Raft* benchmark may livelock under the PCTCP scheduler due to new messages always being placed in the chain with the highest priority.

To avoid livelocks while searching for safety bugs, PCTCP identifies the chain which causes a livelock by detecting a cycle and using heuristics such as comparing the number of times a chain is consecutively scheduled with a given threshold. It then temporarily disables the identified chain and enables it again as soon as a new message is added to it.

The livelock problem also affects the original PCT algorithm and is discussed in Burckhardt et al. (2010). A more formal treatment of the issue is left for future work.

6.3.2 Case Study: Cassandra

In this and the next subsection we evaluate the effectiveness of PCTCP algorithm on two complex real-world systems. The bugs in real-world distributed systems are known to be hard to detect since in addition to message reorderings they often involve other kinds of faults, like node crashes and reboots (Leesatapornwongsa et al. 2016b). We demonstrate that PCTCP can effectively find bugs even in such realistic scenarios.

We start with Cassandra (Lakshman and Malik 2010)—a distributed NoSQL database management system, which provides lightweight transactions based on the Paxos consensus protocol.[5] Cassandra's Paxos protocol implementation in version 2.0.0 (Apache 2012) has a bug CASSANDRA-6023[6] which exposes in some subtle reorderings on the synchronization messages exchanged between the nodes. The bug is detected in a scenario where the nodes process different client transactions concurrently. In an execution where some commit messages of some transactions arrive at some nodes after the synchronization messages of other transactions, it is possible to commit a transaction more than once. This results in corrupting data and propagating it to the other nodes. The bug is deep and hard to detect as it requires several message reorderings in several transactions and nodes.

[5] `http://cassandra.apache.org`

[6] `https://issues.apache.org/jira/browse/CASSANDRA-6023`

Table 6.4: Test parameters and results for the Cassandra system.

	Max events	Event labels	Avg max chains	Runs	Buggy	Total time (min)
Random walk	54	-	6.97	1,000	0	481.95
PCTCP	54	4	5.65	1,000	0	505.73
PCTCP	54	5	5.73	1,000	1	503.81
PCTCP	54	6	5.80	1,000	1	512.00

We tested Cassandra on our PCTCP implementation[7] which we build on top of the SAMC/DMCK (Leesatapornwongsa et al. 2014a) model checker. Our PCTCP scheduler collects the distributed system events intercepted by SAMC and partitions them into chains. It selects the next event to be scheduled based on the chain priorities and sends this selected event to SAMC to enforce its execution in the distributed system.

We tested different schedules of a use case scenario with three concurrent client transactions using both PCTCP and a naive random scheduler which randomly selects one of the enabled events in the system. Table 6.4 shows the parameters and the results of our tests. The first row shows the results for the random walk and each of the next rows shows the PCTCP results for different values of d, the number of event labels. The columns list the maximum number of events produced by the benchmark (**Max events**), the size of the tuple of event labels (**Event labels**), the number of runs (**Runs**), the number of buggy schedules (**Buggy**) and the total running time of the tests in minutes (**Time (min)**). The column **Avg max chains** shows the average of the maximum number of concurrently enabled chains for PCTCP and the average of the maximum number of concurrently enabled events in the naive random tests.

The PCTCP algorithm hits the bug in one of the schedules determined by 5 and 6-tuples of events over 54 events. PCTCP detects the bug with a higher probability (0.1% in this evaluation) than its theoretical guarantee ($1/(w^2n^{d-1})$). This can be explained by several facts: (i) Considering the poset width parameter w, we can say that the number of concurrently enabled events (around 7 on average in our benchmark) is lower than the width of the poset (around 10 in our benchmark) in general. During the execution of PCTCP, it is typical to have several chains in which all the events are already executed and the algorithm selects from a smaller number of chains than the poset width. As an example consider a distributed system execution where the highest priority chain has some events pertaining to some protocol communication between a sender and receiver. The PCTCP scheduler moves to another chain (without a reduction in priorities) when all

[7]The source code is available at `https://gitlab.mpi-sws.org/burcu/pctcp-cass/tree/PCTCP`.

the events of this chain are executed. The first chain gets enabled only after some events in the other chains, e.g., when processing other receivers' responses in other chains cause the sender to send an event that is inserted into the first chain. (ii) Now let us consider the generation of a $d-1$ tuple of events from n events to characterize a bug. In practice, a bug is not only hit by a specific tuple of events but several tuples lead to the buggy schedule. First, more than one ordering of events in the tuple can expose the same bug, since some events in the tuple might be commutative. For example, a bug hit by a tuple of three events (a, b, c) might require only the relative ordering of $a - b$ and $a - c$, hence hit also by the tuple (a, c, b). In our experiment, the detected 4-tuple of events which leads to a buggy execution of Cassandra has two commutative events. Second, the problematic relative ordering of the events can be generated by tuples of different events. A distributed system bug which is exposed by scheduling an event at a later point of execution can be exposed by reducing the priority of a chain at this event as well as at an earlier event that creates that event. As an example, reducing the priority of a chain at an event pertaining to a certain part of the protocol communication between two nodes delays the execution of the next events of the communication protocol as well.

The naive random approach cannot detect the bug in a total of 1000 schedules which takes around eight hours to run. As shown in Table 6.4, the maximum number of concurrently enabled events to select from randomly in random testing is higher than the maximum number of concurrently enabled chains in PCTCP on average. Therefore, random testing has a lower probability of hitting the bug by naively selecting the next event among the enabled events.

6.3.3 Case Study: Zookeeper

Our next case study involves a system called Zookeeper[8]. Zookeeper is a distributed key-value store used by large distributed systems for maintaining configuration and naming information, providing distributed synchronization, and for other purposes that arise while coordinating nodes in a distributed setting. Its intended usage requires Zookeeper to provide strong consistency guarantees, which it does by running a distributed consensus algorithm called ZAB—Zookeeper Atomic Broadcast (Junqueira, Reed, and Serafini 2011). In our case study we only focus on the first phase of the algorithm—the leader election—and show that in the presence of node crashes and reboots, PCTCP can effectively find executions resulting in multiple node leaders.

In the experiments we are using Zookeeper v3.4.3, even though the most recent stable version was 3.4.11 at the time of writing this text. The reason is that the older version has some known bugs which can be detected by tools like SAMC. In fact, the authors of SAMC

[8] `https://zookeeper.apache.org/`

were kind enough to provide us with a version of their tool specifically tailored to catching bug ZK-1419[9] in Zookeeper v3.4.3. The bug in question is a liveness bug—the nodes fail to ever elect a leader. In a typical leader election involving 3 nodes, the nodes elect a leader by exchanging 15 to 18 messages. Based on this observation, the authors of SAMC set a bound of 50 events (including messages, but also node crashes and reboots)—any execution that goes beyond this bound is marked as faulty. Following the instructions for reproducing the bug, we ran SAMC in its semantic-aware exploration mode on 3 nodes, allowing 1 node crash and 1 reboot. Out of 174 executions explored in 2,798 seconds, 7 of them were marked as faulty: 4 of them because they reached the bound, but interestingly, 3 of them failed because they resulted in multiple leaders. Note that by itself the situation when multiple nodes believe they are leaders does not constitute a bug in the leader election protocol: the true leader has to be supported by a quorum of followers (cf. bug report ZK-1912[10]). However, such situations may still indicate underlying issues, e.g. the bug ZK-975[11] involves a node that unnecessarily goes into the leading state only to restart the leader election after a delay, thus affecting availability of the system. We set to reproduce the executions leading to multiple leaders, as well as followers following nodes that are not leaders, using PCTCP. We refer to such situations as *inconsistencies*.

We built our own scheduler called *HitMC*[12]. Like SAMC, HitMC can orchestrate Zookeeper nodes by imposing the order of messages during leader election, and it can crash and reboot nodes. Additionally, HitMC tracks causal dependencies among messages, allowing us to form more general chain decompositions than simple per-node chaining of messages. Unlike SAMC, which knows about commutativity among messages and can thus employ partial-order reduction techniques, HitMC does not have any semantic awareness—it treats messages opaquely and schedules them according to PCTCP.

We model the execution of the system with three kinds of events: node start, node crash, and message. Fig. 6.2 shows an example of a scheduling poset for Zookeeper with 3 nodes. In the figure, $\mathsf{Start}(n)$, $\mathsf{Crash}(n)$, and $\mathsf{Msg}(n, n')$ designate the start event for node n, the crash event for node n, and an event for a message from n to n', respectively. For each run of the system we specify a crash budget and a reboot budget. Each node's initial start event gets a corresponding crash event as a successor. We only effectively allow the execution of these events if they are within the crash budget. Later, each executed crash event gets a node start event as a successor and vice versa, as long as the corresponding budget is still positive. Nodes send messages either automatically after they are started, or in response to received messages. In the first case, a message is ordered after the last

[9] `https://issues.apache.org/jira/browse/ZOOKEEPER-1419`
[10] `https://issues.apache.org/jira/browse/ZOOKEEPER-1912`
[11] `https://issues.apache.org/jira/browse/ZOOKEEPER-975`
[12] The source code is available at `https://gitlab.mpi-sws.org/rupak/hitmc`.

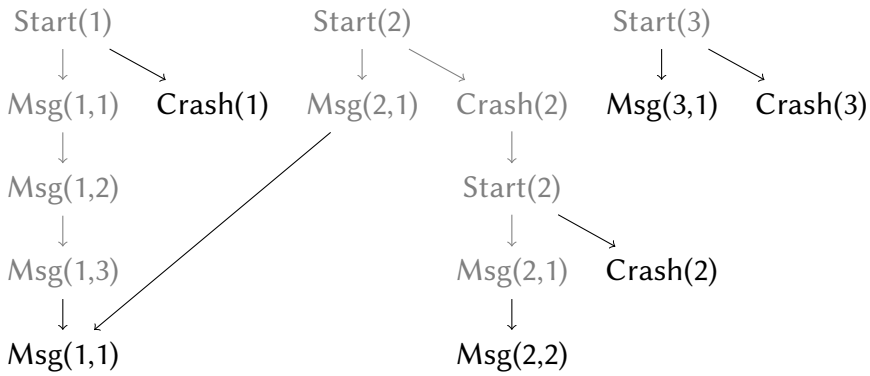

Figure 6.2: Scheduling poset for Zookeeper with 3 nodes. Gray events are executed and black events are enabled.

node start event or the last message sent from the same node, and in the second case a message is additionally ordered after the message it is responding to.

By default, HitMC schedules events according to PCTCP. The user specifies parameters d, the number of event labels, and n, the total number of events. If the number of events in an actual run exceeds n, HitMC makes sure to first schedule the n events that were added to the scheduling poset first. It then continues executing in the order specified by the chain priorities. No priority changes happen after the n-th event, and the probabilistic guarantee of PCTCP only applies to the first n events. In addition to PCTCP, HitMC provides the "random walk" scheduling strategy, which selects the next event in each step uniformly at random among the enabled events.

Like with SAMC, our experiments consist of executing Zookeeper's leader election on 3 nodes, with the crash and reboot budget of 1 each. We experiment with the random walk strategy, as well as PCTCP with the parameter d ranging from 1 to 5. In each case, we execute a total of 1,000 random runs. In addition, we execute 1,000 runs of PCTCP with $d = 2$ and no crashes and reboots, in order to see whether we can find inconsistent election results even without node crashes. Following the SAMC's bound for the number of events within which the leader should be elected, we set the parameter n for the maximal number of events to 50. Unlike SAMC, we do not stop the execution when the maximal number of events is reached. Instead, we let the system run for up to 1,000 events.

The results of the experiments are summarized in Table 6.5. With the random walk strategy we observe inconsistent election results in 12 out of 1,000 runs. With PCTCP we observe inconsistencies in at least 26 runs for $d = 4$, and up to 42 runs for $d = 5$. Note that the number of inconsistent runs for $d = 5$ amounts to 4.2% of all the runs. Even in

Table 6.5: Zookeeper results for 1,000 executions of each strategy. The columns are: average max enabled events (AMEE), max enabled events (MEE), average events (AE), max events (ME), average max chains (AMC), max chains (MC), number of faulty executions (F), and total time in seconds (T). The experiment marked by ⋆ had no crashes or reboots; in all other cases there was 1 node crash and 1 reboot per run. In the case of PCTCP with $d = 4$, we observed a non-terminating run, most likely exhibiting bug ZK-1419. The run was terminated after reaching 1,000 events. Numbers marked by † are given with the non-terminating run excluded.

Strategy	AMEE	MEE	AE	ME	AMC	MC	F	T
Random walk	6	6	20.9	36	-	-	12	7,040
PCTCP ($d = 1, n = 50$)	6	6	20.6	31	6.4	10	39	2,059
PCTCP ($d = 2, n = 50$)*	3	5	18.2	52	4.5	11	34	5,107
PCTCP ($d = 2, n = 50$)	6	6	20.7	34	7.5	11	32	2,060
PCTCP ($d = 3, n = 50$)	6	6	21.0	34	8.6	12	26	2,103
PCTCP ($d = 4, n = 50$)	6	7	†23.4	†63	10.1	18	†40+1	6,835
PCTCP ($d = 5, n = 50$)	6	6	23.7	70	11.2	19	42	7,111

the experiment without crashes and reboots, we detect inconsistencies in 34 runs. Except for $d \geq 4$, almost all runs finish within the predicted limit of 50 events. For $d \geq 4$, there are 10 runs that exceed this limit and go up to 63 and 70 events. Additionally, for $d = 4$ we observe a run that failed to terminate even after 1,000 events, possibly exhibiting the non-termination issue ZK-1419. However, it is also possible that the non-termination occurs due to the non-fair nature of our scheduler. Finally, we note that a run takes between 2 and 7 seconds on average, which allows us to execute a set of 1,000 runs in 30 to 120 minutes.

Chapter 7

Related Work

7.1 Combinatorial Testing

Our results can broadly be seen as an instance of *combinatorial testing* (Kuhn, Kacker, and Lei 2010; Colbourn 2004), the sub-field of testing that designs near-optimal test suites to cover all k-wise interactions among a large number of features. Like in our case, the key insight in combinatorial testing is that many bugs depend only on the interaction of a small number of features: in many cases, combinations of up to 6 features suffice for detecting most bugs in industrial applications (Kuhn, Kacker, and Lei 2010).

The main objects studied in combinatorial testing are *covering arrays*. An $N \times n$ array over values from the set $\{0, \ldots, v-1\}$ is said to be *k-covering* if every $N \times k$ sub-array contains all v^k possible rows (with multiple occurrences allowed). For given parameters n, k, and v, the goal is to find the smallest number of rows $N(n, k, v)$ such that there exists a k-covering array of size $N(n, k, v) \times n$. Many practical approaches for constructing covering arrays of small size have been studied, including greedy algorithms, hill-climbing algorithms, simulated annealing, and genetic algorithms (Colbourn 2004). Non-constructive bounds on $N(n, k, v)$ are usually shown using the probabilistic method. For example, Godbole, Skipper, and Sunley (1996) show the following bound using the Lovász local lemma (Alon and Spencer 2004):

$$N(n, k, v) \leq \frac{(k-1)\log n}{\log\left(v^k/(v^k-1)\right)}(1 + o(1)) \quad .$$

For concrete values of k and v, more precise bounds are known. For example, $N(n, 3, 2) \leq 7.56444 \log n(1 + o(1))$. This bound, attributed to Roux (Godbole, Skipper, and Sunley 1996), is proved by switching from arbitrary binary arrays to arrays with equal number

of zeros and ones in each column—a technique similar to our switching from arbitrary partitions to balanced and semibalanced partitions in Sections 4.2 and 4.4.

Note that covering arrays are an instance of our covering families. A test is a row of an array—a vector of size n with components taking values in the set $\{0, \ldots, v-1\}$—and a testing goal is a set of k positions $1 \le i_1 < i_2 < \ldots < i_k \le n$ and a vector $v \in \{0, \ldots, v-1\}^k$. A test $t \in \{0, \ldots, v-1\}^n$ covers this goal if the k components at positions $(i_1, i_2, \ldots, i_k)$ of t make up the vector v. A family $\mathcal{F}$ of vectors of size n is a covering family if it covers every testing goal, thus it is nothing but a covering array. The probability that a random vector $t \in \{0, \ldots, v-1\}^n$ covers a testing goal is v^{-k}. There are $\binom{n}{k} v^k$ testing goals in all. Using Theorem 3.1, we obtain that there is a covering family of size $O(kv^k \log n)$. By taking into account the independence of two testing goals with non-overlapping positions and using the Lovász local lemma, we can obtain a slightly better bound of $O((k-1)v^k \log n)$, which is asymptotically the same as the bound obtained by Godbole, Skipper, and Sunley (1996).

As a concrete example of combinatorial testing using covering arrays, consider the testing of combinational VLSI circuits (Seroussi and Bshouty 1988). We are given a circuit with n inputs, which consists of a large number of smaller components, and each component depends on at most k of the n inputs. In VLSI testing applications, n may be much larger than k. A test consists of a Boolean vector of n bits. Informally, our testing goal is to test every component with every possible input of k bits. Thus, our testing task precisely corresponds to finding small k-covering arrays for $v = 2$. From the discussion above, we obtain that there is a covering array of size $O(k2^k \log n)$. Moreover, by Theorem 3.1, every array of $2^k(k \log n + \log \epsilon^{-1})$ randomly generated rows is a covering array with probability at least $1 - \epsilon$. This was in fact the main result of Seroussi and Bshouty (1988). Note that the naive bound is $O(n^k 2^k)$. In fact, Seroussi and Bshouty (1988) show a lower bound of $\Omega(2^k \log n)$, so the probabilistic method is almost optimal in this case.

7.2 Randomized Approaches

Random simulation is the primary mode of testing systems with large and complex state spaces across many different domains: from sequential circuits to network protocol implementations and to large-scale distributed systems. Practitioners tend to use random schedulers and random fault-injection (Apache Hadoop 2016; Kingsbury 2013–2018; Izrailevsky and Tseitlin 2011; Claessen et al. 2009) to test their systems, sometimes even advocating doing this in production. The latter approach, dubbed Chaos Engineering and codified in the Principles of Chaos Engineering (2018), is the underlying philosophy of Netflix Simian Army (Izrailevsky and Tseitlin 2011), a collection of tools called monkeys

that randomly tamper with the system in production, with engineers monitoring the effects and addressing problems as they arise.

Our results from Chapters 3 and 4 are a step towards a theoretical understanding of random testing: we show that the effectiveness of testing can be explained in certain scenarios by providing lower bounds on the probability that a single random test covers a fixed coverage goal. For network partition tests, we introduce a set of coverage goals inspired by actual bugs in distributed systems and show lower bounds on the probability of a random test covering a goal.

7.2.1 Deterministic Families of Tests

Probabilistic constructions demonstrate existence of covering families, and we can amplify the probability of finding a test suite. However, the soundness guarantee is "with high probability." Unfortunately, even when a small covering family can be shown to exist, an *explicit, deterministic* construction of a covering family can be a significantly harder problem. This is a recurring theme in combinatorics (Alon 2010): for many problems, explicit deterministic constructions come much later than an existence argument using the probabilistic method. In fact, there are many combinatorial objects proved to exist using the probabilistic method for which we do not know optimal deterministic constructions (Alon and Spencer 2004)! For example, our notion of splitting families is related to perfect hash functions (Yao 1981; Fredman, Komlós, and Szemerédi 1984; Czech, Havas, and Majewski 1997). In this context, Yao (1981) gives a highly non-trivial deterministic construction of k-splitting families of size $4^{k^2}(\log_2 n)^{k-1}$, which is worse than the bound $k^{k+1}(k!)^{-1}\log n$ obtained through Corollary 4.3. Furthermore, decision problems related to minimal constructions or enumerations are usually computationally intractable (NP-hard (Seroussi and Bshouty 1988; Yannakakis 1982)); thus, it is unlikely that a simple deterministic approach can supplant the simplicity of random testing.

Even when deterministic construction of a covering family is known, it may be infeasible to execute all tests from the family. Consider Yao's construction of k-splitting families. Already for $n = 5$ and $k = 2$—the values used in Jepsen tests—the size of the family is approximately 595. Increase n to 10 and k to 3, and the size grows to 9,609,717. Compared to this, randomly constructed families have two additional benefits: they do not require a sophisticated test generation algorithm and we can always stop the construction and apply Theorem 3.1 or its instances in reverse to obtain the probability that the constructed family is covering. The probability can serve as a qualitative measure of coverage.

7.2.2 Random Walks over Graphs

We focus on "static" notions of coverage, where showing lower bounds on probabilities are relatively easier. In random simulation of reactive systems, such as network protocols, the testing process defines a random walk over the state space of the system. If the random walk *rapidly mixes*, that is, converges to a stationary distribution in a small number of steps, random simulation fairly simulates the reachable state space in a stationary distribution. Rigorous analysis of random walks using Markov chain mixing techniques was pioneered by C. H. West (1989) for a simple class of decoupled network protocols (technically, the state space was a hypercube). Mihail and Papadimitriou (1994) used coupling techniques to prove rapid mixing for the class of symmetric dyadic flip-flops (SDFF): a concurrent system of automata, each with two states, communicating pairwise by rendezvous and where each action has a reverse action. Unfortunately, it is very hard to prove rapid mixing for most Markov chains, indeed, there are counterexamples to rapid mixing when the rather severe restrictions of SSDF are relaxed; for example, reversibility is a strong requirement (Levin, Peres, and Wilmer 2009).

It is tempting to provide a random testing version of systematic testing procedures such as context-bounded reachability. For example, could one show that a random walk on the state space defined by a multithreaded program quickly visits any k-context-bounded reachable state? Unfortunately, this involves bounding the *hitting time* for a random walk on a directed graph, i.e., the expected time for a random walk to visit a node, and we do not have a sufficiently strong lower bound on the hitting time that gives better than exponential bounds.

Thus, we believe extensions of our techniques to "dynamic" coverage may require sophisticated methods.

7.3 Systematic Approaches

As we have already noted in the introduction, one direction of research in assuring correct behavior of distributed systems is to build fully verified systems "from scratch." Despite heroic efforts in this direction (Lamport 1994; Wilcox et al. 2015; Hawblitzel et al. 2015), we are quite far from replacing existing infrastructure with fully verified deployments of comparable functionality and performance.

The other direction of research are systematic approaches like model checking (Baier and Katoen 2008; Yang et al. 2009; Leesatapornwongsa et al. 2014b; Fisman, Kupferman, and Lustig 2008; Konnov, Veith, and Widder 2017) and systematic fault-injection (Alvaro, Rosen, and Hellerstein 2015; Gunawi et al. 2011), which design algorithms and heuristics that perform systematic search over behaviors which are sufficient to find all bugs. A

technique that is commonly employed by the model checking approaches is partial order reduction (Godefroid 1996; Flanagan and Godefroid 2005; Abdulla et al. 2014): instead of exploring all behaviors of the system, if suffices to only one representative behavior among equivalent behaviors, i.e., those differing only in the ordering of independent events. In their recent work, Yuan, Yang, and Gu (2018) introduce a randomized scheduling algorithm that takes partial order reduction into account. These techniques do not restrict the search space to depth-d bugs, as we do. We do not know how partial order reduction can be modified to efficiently explore depth-d bugs.

Different notions of bug depth are defined in the literature. These notions aim to parametrize the search space by a depth d, so that a d-bounded exploration provides a high coverage of the executions that are likely to be buggy. Context bounding (Qadeer and Rehof 2005) characterizes the depth as the number of context switches between threads required to hit a bug. Preemption bounding (Musuvathi and Qadeer 2007) bounds only the preemptive switches between the tasks. Although this notion yields a smaller depth for the cases where tasks run to completion, it is still not efficient for exploring schedules where many tasks need to be preempted to hit a schedule. Delay bounding (Emmi, Qadeer, and Rakamaric 2011b), which defines the depth as the number of deviations from a given deterministic scheduler, and phase bounding (Bouajjani and Emmi 2012), which bounds the number of process communication cycles, are applicable to distributed system setting, since the bug depth parameter does not limit the number of tasks/nodes involved in the execution. The work by Desai, Qadeer, and Seshia (2015) presents a randomized algorithm for asynchronous systems based on delay-bounded exploration. Their algorithm is parametrized by a delaying scheduler where the depth parameter does not characterize the bug but the search space. The bug depth we use in our work (which is also used in Burckhardt et al. (2010)) is defined only by the ordering constraints between the events in the execution, and it is independent of the exploration strategy.

7.4 Theory of Partial Orders

As we have already noted in Chapter 5, our notion of d-hitting families is closely related to the notion of *order dimension* for a partial order (Dushnik and Miller 1941; Trotter 2001). Specifically, the size of an optimal 2-hitting family is the order dimension of a partial order, and the size of an optimal d-hitting family is a natural generalization. To the best of our knowledge, general d-hitting families have not been studied before for general partial orders. A version of the dimension ($d = 2$) called fractional dimension is known to be of use for approximation of some problems in scheduling theory (Ambühl et al. 2008). Other generalizations of the dimension are also known (see, e.g., Trotter (1976)), but, to the best of our knowledge, none of them is equivalent to ours.

Online chain partitioning and its connection to online dimension for upgrowing posets was studied by Felsner (1997) and Kloch (2007). Their work is part of a larger context of studying unrestricted posets. In the unrestricted setting, bounds on the optimal number of chains are much worse than for upgrowing posets. While an upgrowing poset can be partitioned online into at most $\binom{w+1}{2}$ chains, for a long time the best known upper bound for an unrestricted poset was $(5^w - 1)/4$ (Henry A. Kierstead 1981). Bosek and Krawczyk (2010) found a subexponential upper bound of $w^{16 \log_2 w}$, which was recently improved to $w^{6.5 \log_2 w + 7}$ (Bosek, Hal A. Kierstead, et al. 2018), and even more recently to $w^{O(\log \log w)}$ (Bosek and Krawczyk 2018). A nice, albeit outdated survey of the results in this setting was done by Bosek, Felsner, et al. (2012).

The online chain partitioning algorithm we use in Chapter 6 is by Agarwal and Garg, and it appears in their work on chain clocks (Agarwal and Garg 2007): they compare vector clocks, where each component in the clock corresponds to a thread in the program, and chain clocks, where each component in the clock corresponds to a chain in a chain partition obtained by online chain partitioning. They show that in many cases chain clocks are considerably more efficient than vector clocks. Similarly to this, it would make sense to try running PCTCP side-by-side with PCT on multithreaded programs, and see whether there are scenarios in which PCTCP would find a better chain partition than the one induced by threads. An experiment along these lines is left for future work.

7.5 Practical Tools

On the application side, our work is related to a number of tools for finding bugs in concurrent and distributed systems. We start by mentioning Bita, a testing tool for actor programs implemented within Akka framework in Scala (Tasharofi et al. 2013). By using an arbitrary execution of the program as the initial schedule, Bita systematically reverses the order of pairs of concurrent messages in a given schedule to produce new schedules, and then executes these schedules. The exploration is guided by coverage goals similar to our 2-hitting goal. Unlike Bita, our algorithm PCTCP randomly samples schedules from a strong d-hitting family for arbitrary d, and does not rely on any initial execution of the program.

Another related tool is EventRacer, a race detector for client-side web applications (Raychev, M. Vechev, and Sridharan 2013). Similarly to Bita, EventRacer explores schedules by pairwise reversal of concurrent events, but with the goal of detecting races—concurrent conflicting accesses to the same memory location. One of the key contributors to EventRacer's efficiency is the use of chain clocks instead of vector clocks. EventRacer's chain clocks are based on a greedy chain decomposition. The authors report a 33-fold reduction

in the average length of chain clocks compared to standard vector clocks. Even with a highly optimized bit-vector representation of vector clocks, chain clocks are reported to consume on average 6.6 times less memory, which significantly improves overall performance. In a different paper on race detection, Dimitrov, M. T. Vechev, and Sarkar (2015) also use insights from theoretical work on partial orders to show that lattices of dimension two admit efficient race detection.

Bibliography

Abdulla, Parosh et al. (2014). "Optimal Dynamic Partial Order Reduction". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: ACM, pp. 373–384. DOI: 10.1145/2535838.2535845.

Agarwal, Anurag and Vijay K. Garg (2007). "Efficient dependency tracking for relevant events in concurrent systems". In: *Distributed Computing* 19.3, pp. 163–183. DOI: 10.1007/s00446-006-0004-y.

Alon, Noga (2010). "Algebraic and Probabilistic Methods in Discrete Mathematics". In: *Visions in Mathematics: GAFA 2000 Special volume, Part II*. Basel: Birkhäuser Basel, pp. 455–470. DOI: 10.1007/978-3-0346-0425-3_1.

Alon, Noga and Joel H. Spencer (2004). *The Probabilistic Method*. Wiley-Interscience series in discrete mathematics and optimization. Wiley.

Alvaro, Peter, Joshua Rosen, and Joseph M. Hellerstein (2015). "Lineage-driven Fault Injection". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, pp. 331–346. DOI: 10.1145/2723372.2723711.

Ambühl, Christoph et al. (2008). "Precedence Constraint Scheduling and Connections to Dimension Theory of Partial Orders". In: *Bulletin of the EATCS* 95, pp. 37–58.

Apache (2012). *Cassandra-2.0.0*. URL: http://archive.apache.org/dist/cassandra/2.0.0/ (visited on 04/13/2018).

Apache Hadoop (2016). *Fault Injection Framework and Development Guide*. URL: http://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-hdfs/FaultInjectFramework.html (visited on 07/07/2017).

Baier, Christel and Joost-Pieter Katoen (2008). *Principles of Model Checking*. MIT Press.

Bosek, Bartłomiej, Stefan Felsner, et al. (2012). "On-Line Chain Partitions of Orders: A Survey". In: *Order* 29.1, pp. 49–73. DOI: 10.1007/s11083-011-9197-1.

Bosek, Bartłomiej, Hal A. Kierstead, et al. (2018). "An Easy Subexponential Bound for Online Chain Partitioning". In: *Electr. J. Comb.* 25.2, P2.28.

Bosek, Bartłomiej and Tomasz Krawczyk (2010). "The Sub-exponential Upper Bound for On-Line Chain Partitioning". In: *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*. IEEE Computer Society, pp. 347–354. DOI: 10.1109/FOCS.2010.40.

— (2018). "On-line Partitioning of Width w Posets into $w^{O}(\log \log w)$ Chains". In: *CoRR* abs/1810.00270. arXiv: 1810.00270.

Bouajjani, Ahmed and Michael Emmi (2012). "Bounded Phase Analysis of Message-Passing Programs". In: *TACAS '12: Proc. 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. LNCS. Springer.

Brewer, Eric A. (2000). "Towards robust distributed systems (abstract)". In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*. ACM, p. 7. DOI: 10.1145/343477.343502.

— (2012). *CAP Twelve Years Later: How the "Rules" Have Changed*. URL: https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed (visited on 07/07/2017).

Burckhardt, Sebastian et al. (2010). "A randomized scheduler with probabilistic guarantees of finding bugs". In: *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*. ACM, pp. 167–178. DOI: 10.1145/1736020.1736040.

Chalermsook, Parinya, Bundit Laekhanukit, and Danupon Nanongkai (2013). "Graph Products Revisited: Tight Approximation Hardness of Induced Matching, Poset Dimension and More". In: *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*. SIAM, pp. 1557–1576. DOI: 10.1137/1.9781611973105.112.

Chaos Engineering (2018). *Principles of Chaos Engineering*. URL: http://principlesofchaos.org/ (visited on 09/18/2018).

Chistikov, Dmitry, Rupak Majumdar, and Filip Niksic (2016). "Hitting Families of Schedules for Asynchronous Programs". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Vol. 9780. Lecture Notes in Computer Science. Springer, pp. 157–176. DOI: 10.1007/978-3-319-41540-6_9.

Claessen, Koen et al. (2009). "Finding race conditions in Erlang with QuickCheck and PULSE". In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009*. ACM, pp. 149–160.

Colbourn, Charles J. (2004). "Combinatorial aspects of covering arrays". In: *Le Matematiche* 59.1,2, pp. 125–172.

Czech, Zbigniew J., George Havas, and Bohdan S. Majewski (1997). "Perfect Hashing". In: *Theor. Comput. Sci.* 182.1-2, pp. 1–143. DOI: 10.1016/S0304-3975(96)00146-6.

Deligiannis, Pantazis, Alastair F. Donaldson, et al. (2015). "Asynchronous Programming, Analysis and Testing with State Machines". In: *SIGPLAN Not.* 50.6, pp. 154–164. DOI: 10.1145/2813885.2737996.

Deligiannis, Pantazis, Matt McCutchen, et al. (2016). "Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!)" In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, pp. 249–262.

Desai, Ankush, Shaz Qadeer, and Sanjit A. Seshia (2015). "Systematic Testing of Asynchronous Reactive Systems". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, pp. 73–83. DOI: 10.1145/2786805.2786861.

Dilworth, Robert P. (1950). "A Decomposition Theorem for Partially Ordered Sets". In: *Annals of Mathematics* 51.1, pp. 161–166.

Dimitrov, Dimitar, Martin T. Vechev, and Vivek Sarkar (2015). "Race Detection in Two Dimensions". In: *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*. ACM, pp. 101–110. DOI: 10.1145/2755573.2755601.

Dushnik, Ben and E. W. Miller (1941). "Partially Ordered Sets". In: *American Journal of Mathematics* 63.3, pp. 600–610.

Edgar, Gerald A., Daniel H. Ullman, and Douglas B. West (2017). "Problems and Solutions". In: *The American Mathematical Monthly* 124.2, pp. 179–187.

Emmi, Michael, Shaz Qadeer, and Zvonimir Rakamaric (2011a). "Delay-bounded scheduling". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, pp. 411–422. DOI: 10.1145/1926385.1926432.

— (2011b). "Delay-bounded scheduling". In: *POPL '11: Proc. 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, pp. 411–422.

Erdős, Paul and László Lovász (1975). "Problems and Results on 3-Chromatic Hypergraphs and Some Related Questions". In: *Infinite and Finite Sets (to Paul Erdős on his 60th birthday)*. Ed. by A. Hajnal, R. Rado, and V. T. Sós. Vol. 2. North-Holland, pp. 609–627.

Felsner, Stefan (1997). "On-Line Chain Partitions of Orders". In: *Theor. Comput. Sci.* 175.2, pp. 283–292. DOI: 10.1016/S0304-3975(96)00204-6.

Fisman, Dana, Orna Kupferman, and Yoad Lustig (2008). "On Verifying Fault Tolerance of Distributed Protocols". 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. Vol. 4963. Lecture Notes in Computer Science. Springer, pp. 315–331. DOI: 10.1007/978-3-540-78800-3_22.

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: ACM, pp. 110–121. DOI: 10.1145/1040305.1040315.

Fomin, Fedor V. et al. (2012). "Planar $\mathcal{F}$-Deletion: Approximation, Kernelization and Optimal FPT Algorithms". In: *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pp. 470–479. DOI: 10.1109/FOCS.2012.62.

Fredman, Michael L., János Komlós, and Endre Szemerédi (1984). "Storing a Sparse Table with $O(1)$ Worst Case Access Time". In: *J. ACM* 31.3, pp. 538–544. DOI: 10.1145/828.1884.

Ganty, Pierre and Rupak Majumdar (2012). "Algorithmic verification of asynchronous programs". In: *ACM Trans. Program. Lang. Syst.* 34.1, 6:1–6:48. DOI: 10.1145/2160910.2160915.

Gatling Corp (2011–2018). *Gatling*. URL: https://gatling.io/ (visited on 09/07/2018).

Gilbert, Seth and Nancy Lynch (2002). "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2, pp. 51–59. DOI: 10.1145/564585.564601.

Godbole, Anant P., Daphne E. Skipper, and Rachel A. Sunley (1996). "t-Covering arrays: Upper bounds and poisson approximations". In: *Combinatorics Probability and Computing* 5.2, pp. 105–117.

Godefroid, Patrice (1996). *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Ed. by J. van Leeuwen, J. Hartmanis, and G. Goos. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Graham, Ronald L., Donald Ervin Knuth, and Oren Patashnik (1994). *Concrete Mathematics: A Foundation for Computer Science*. A foundation for computer science. Addison-Wesley.

Gunawi, Haryadi S. et al. (2011). "FATE and DESTINI: A Framework for Cloud Recovery Testing". In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association.

Hawblitzel, Chris et al. (2015). "IronFleet: proving practical distributed systems correct". In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. ACM, pp. 1–17. DOI: 10.1145/2815400.2815428.

Hegde, Rajneesh and Kamal Jain (2007). "The Hardness of Approximating Poset Dimension". In: *Electronic Notes in Discrete Mathematics* 29, pp. 435–443. DOI: 10.1016/j.endm.2007.07.084.

Izrailevsky, Yury and Ariel Tseitlin (2011). *The Netflix Simian Army*. URL: https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116 (visited on 07/07/2017).

Jensen, Casper Svenning et al. (2015). "Stateless model checking of event-driven applications". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pp. 57–73. DOI: 10.1145/2814270.2814282.

Jhala, Ranjit and Rupak Majumdar (2007). "Interprocedural analysis of asynchronous programs". In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. ACM, pp. 339–350. DOI: 10.1145/1190216.1190266.

Junqueira, F.P., B.C. Reed, and M. Serafini (2011). "Zab: High-performance Broadcast for Primary-backup Systems". In: *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*. DSN '11. IEEE Computer Society, pp. 245–256.

Kierstead, Henry A. (1981). "An Effective Version of Dilworth's Theorem". In: *Transactions of the American Mathematical Society* 268.1, pp. 63–77.

Kingsbury, Kyle (2013–2018). *Jepsen*. URL: http://jepsen.io/ (visited on 09/18/2018).

— (2013). *Partitions for Everyone!* URL: https://www.infoq.com/presentations/partitioning-comparison (visited on 07/07/2017).

Kloch, Kamil (2007). "Online dimension of partially ordered sets". In: *Reports on Mathematical Logic* 42, pp. 101–116.

Konnov, Igor, Helmut Veith, and Josef Widder (2017). "On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability". In: *Inf. Comput.* 252, pp. 95–109. DOI: 10.1016/j.ic.2016.03.006.

Kuhn, D. Richard, Raghu N. Kacker, and Yu Lei (2010). "Combinatorial Testing". In: *Encyclopedia of Software Engineering*. Ed. by Phillip A. Laplante. CRC Press, pp. 1–12.

Lakshman, Avinash and Prashant Malik (2010). "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2, pp. 35–40.

Lamport, Leslie (1994). "The Temporal Logic of Actions". In: *ACM Trans. Program. Lang. Syst.* 16.3, pp. 872–923. DOI: 10.1145/177492.177726.

Leesatapornwongsa, Tanakorn et al. (2014a). "SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems". In: *11th USENIX Symposium on*

Operating Systems Design and Implementation (OSDI 14). Broomfield, CO: USENIX Association, pp. 399–414.

Leesatapornwongsa, Tanakorn et al. (2014b). "SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems". In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. USENIX Association, pp. 399–414.

Leesatapornwongsa, Tanakorn et al. (2016a). "TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*. ACM, pp. 517–530. DOI: 10.1145/2872362.2872374.

— (2016b). "TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, pp. 517–530. DOI: 10.1145/2872362.2872374.

Levin, David Asher, Yuval Peres, and Elizabeth Lee Wilmer (2009). *Markov Chains and Mixing Times*. American Mathematical Society.

Lopes, Cristina Videira (2016). *Distributed Systems Testing: The Lost World*. URL: http://tagide.com/blog/research/distributed-systems-testing-the-lost-world/ (visited on 07/07/2017).

Lu, Shan et al. (2008). "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*. ACM, pp. 329–339. DOI: 10.1145/1346281.1346323.

Majumdar, Rupak and Filip Niksic (2018). "Why is random testing effective for partition tolerance bugs?" In: *PACMPL* 2.POPL, 46:1–46:24. DOI: 10.1145/3158134.

McCaffrey, Caitie (2015). "The Verification of a Distributed System". In: *ACM Queue* 13.9, p. 60. DOI: 10.1145/2857274.2889274.

Mihail, Milena and Christos H. Papadimitriou (1994). "On the Random Walk Method for Protocol Testing". In: *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings*. Vol. 818. Lecture Notes in Computer Science. Springer, pp. 132–141. DOI: 10.1007/3-540-58179-0_49.

Milgram, Stanley (1967). "The Small World Problem". In: *Psychology Today*.

Mudduluru, Rashmi et al. (2017). "Lasso detection using partial-state caching". In: *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. IEEE, pp. 84–91. DOI: 10.23919/FMCAD.2017.8102245.

Musuvathi, Madanlal and Shaz Qadeer (2007). "Iterative Context Bounding for Systematic Testing of Multithreaded Programs". In: *Proceedings of the 2007 ACM SIGPLAN Con-*

ference on Programming Language Design and Implementation. PLDI '07. San Diego, California, USA: ACM, pp. 446–455. DOI: 10.1145/1250734.1250785.

Ongaro, Diego and John Ousterhout (2014). "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. USENIX Association, pp. 305–319.

Ozkan, Burcu Kulahcioglu et al. (2018). "Randomized Testing of Distributed Systems with Probabilistic Guarantees". In: *PACMPL* 2.OOPSLA, 160:1–160:28. DOI: 10.1145/3276530.

Petrov, Boris et al. (2012). "Race detection for web applications". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pp. 251–262. DOI: 10.1145/2254064.2254095.

Qadeer, Shaz and Jakob Rehof (2005). "Context-Bounded Model Checking of Concurrent Software". In: *TACAS '05: Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3440. LNCS. Springer, pp. 93–107.

Raychev, Veselin, Martin T. Vechev, and Manu Sridharan (2013). "Effective race detection for event-driven programs". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. ACM, pp. 151–166. DOI: 10.1145/2509136.2509538.

Raychev, Veselin, Martin Vechev, and Manu Sridharan (2013). "Effective Race Detection for Event-driven Programs". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. Indianapolis, Indiana, USA: ACM, pp. 151–166. DOI: 10.1145/2509136.2509538.

Seroussi, Gadiel and Nader H. Bshouty (1988). "Vector sets for exhaustive testing of logic circuits". In: *IEEE Trans. Information Theory* 34.3, pp. 513–522. DOI: 10.1109/18.6031.

Tasharofi, Samira et al. (2013). "Bita: Coverage-guided, automatic testing of actor programs". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. IEEE, pp. 114–124. DOI: 10.1109/ASE.2013.6693072.

Trotter, William T. (1976). "A Generalization of Hiraguchi's: Inequality for Posets". In: *J. Comb. Theory, Ser. A* 20.1, pp. 114–123. DOI: 10.1016/0097-3165(76)90081-9.

— (2001). *Combinatorics and Partially Ordered Sets: Dimension Theory*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press.

Valdes, Jacobo, Robert Endre Tarjan, and Eugene L. Lawler (1982). "The Recognition of Series Parallel Digraphs". In: *SIAM J. Comput.* 11.2, pp. 298–313. DOI: 10.1137/0211023.

West, Colin H. (1989). "Protocol Validation in Complex Systems". In: *SIGCOMM '89, Proceedings of the ACM Symposium on Communications Architectures & Protocols, Austin, TX, USA, September 19-22, 1989*. ACM, pp. 303–312. DOI: 10.1145/75246.75276.

Wilcox, James R. et al. (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, Portland, OR, USA, June 15-17, 2015*. ACM, pp. 357–368. DOI: 10.1145/2737924.2737958.

Yang, Junfeng et al. (2009). "MODIST: Transparent Model Checking of Unmodified Distributed Systems". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*. USENIX Association, pp. 213–228.

Yannakakis, Mihalis (1982). "The Complexity of the Partial Order Dimension Problem". In: *SIAM Journal on Algebraic Discrete Methods* 3.3, pp. 351–358. DOI: 10.1137/0603036. eprint: https://doi.org/10.1137/0603036.

Yao, Andrew Chi-Chih (1981). "Should Tables Be Sorted?" In: *J. ACM* 28.3, pp. 615–628. DOI: 10.1145/322261.322274.

Yuan, Xinhao, Junfeng Yang, and Ronghui Gu (2018). "Partial Order Aware Concurrency Sampling". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. Vol. 10982. Lecture Notes in Computer Science. Springer, pp. 317–335. DOI: 10.1007/978-3-319-96142-2_20.

Appendix A

Curriculum Vitae

Research Interests

Analysis, verification, and testing of concurrent and distributed systems. In particular, I am interested in applying combinatorial techniques to systematic and random testing.

Education

09/2012—10/2018 Max Planck Institute for Software Systems, Kaiserslautern

Doctoral researcher in computer science, advised by Rupak Majumdar

09/2009—10/2011 Department of Mathematics, University of Zagreb

Enrolled in a doctoral program in mathematics

07/2004—10/2009 Department of Mathematics, University of Zagreb

Dipl. Ing. (4-year degree) in Mathematics (profile: Computer Science)
GPA: 4.7 / 5.0

Employment

10/2018— University of Pennsylvania, Philadelphia

Postdoctoral researcher in computer science

09/2012—10/2018 Max Planck Institute for Software Systems, Kaiserslautern

Doctoral researcher in computer science

05/2016—08/2016 Microsoft Corp., Redmond

Research intern working on a testing and fault-injection framework for concurrent software. Technologies: C#, .NET Compiler Platform ("Roslyn")

04/2010—09/2012 IN2 d.o.o., Zagreb

Software engineer developing financial software. Technologies: Oracle DB (SQL, PL/SQL), Java (Spring Framework), and Adobe Flex

Teaching Experience

10/2016—02/2017 Technische Universität Kaiserslautern

Teaching assistant: Program Analysis (Winter 2016/2017)

04/2014—07/2014 Technische Universität Kaiserslautern

Teaching assistant: Verification of Reactive Systems (Summer 2014)

03/2008—09/2009 Department of Mathematics, University of Zagreb

Student assistant: Set theory (Summer 2008), Introduction to parallel computing (Winter 2008), Application of parallel computers (Summer 2009).

09/2002—06/2005 Informatics Club NET, Ivanić-Grad

Tutoring high school students for programming competitions

Professional Service

- Artifact evaluation committee: ISSTA 2015, ECOOP 2018, CAV 2019
- Conference reviews: CAV 2013, CSL 2013, FMCAD 2013, EMSOFT 2014, FMCAD 2014, LICS 2014, CADE 2015, VMCAI 2015, POPL 2016, TACAS 2016, VMCAI 2017, ICALP 2018
- Journal reviews: ACM Transactions on Computational Logic, Acta Informatica

Technical Skills

Operating systems: GNU/Linux, Mac OS X, Windows

Programming languages: C/C++, C#, Java, Python, PL/SQL, ActionScript (Flex)

Databases: Oracle DB

Language Skills

Croatian (native), English (fluent), German (basic)

Publications

1. Burcu Kulahcioglu Ozkan, Rupak Majumdar, F. N. *Checking Linearizability Using Hitting Families.* PPoPP 2019
2. Burcu Kulahcioglu Ozkan, Rupak Majumdar, F. N., Mitra Tabaei Befrouei, Georg Weissenbacher. *Randomized Testing of Distributed Systems with Probabilistic Guarantees.* PACMPL 2 (OOPSLA) 2018
3. Rupak Majumdar, F. N. *Why is Random Testing Effective for Partition Tolerance Bugs?* PACMPL 2 (POPL) 2018
4. Dmitry Chistikov, Rupak Majumdar, F. N. *Hitting Families of Schedules for Asynchronous Programs.* CAV 2016
5. Ivan Gavran, F. N., Aditya Kanade, Rupak Majumdar, Viktor Vafeiadis. *Rely/Guarantee Reasoning for Asynchronous Programs.* CONCUR 2015
6. Sumit Gulwani, Mikaël Mayer, F. N., Ruzica Piskac. *StriSynth: Synthesis for Live Programming.* ICSE 2015
7. Javier Esparza, Ruslán Ledesma-Garza, Rupak Majumdar, Philipp Meyer, F. N. *An SMT-Based Approach to Coverability Analysis.* CAV 2014
8. Johannes Kloos, Rupak Majumdar, F. N., Ruzica Piskac. *Incremental, Inductive Coverability.* CAV 2013