Formal Verification of Fault-Tolerant Systems

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

OGNJEN MARIĆ

MSc in Computer Science, University of Belgrade
born 29.04.1981
citizen of Bosnia and Herzegovina

accepted on the recommendation of

Prof. Dr. David Basin, examiner
Dr. Stephan Merz, co-examiner
Dr. Christoph Sprenger, co-examiner

2017
Abstract

The goal of formal methods is to enhance our confidence in the correctness of computer systems, by proving that a model of a system obeys a given specification. For software systems, the models are typically built with the assumption that the underlying hardware is perfect. This assumption greatly simplifies both the model and the reasoning, but it is not valid for software designed with the explicit goal of fault-tolerance. In this thesis, we examine the verification problem for such software, in different fault scenarios.

In the first part of the thesis, we focus on crashes (for example, due to power failures) and persistent storage faults (for example, random bit flips) within a single computer system. We formally verify the correctness of the persistent memory manager in IBM’s 4765 secure coprocessor, which provides a transactional semantics of persistent memory updates. The inclusion of storage faults is novel in this area and incurs a significant jump in complexity of both the system and its verification. We tackle the resulting verification challenge by a combination of a monad-based model, an abstraction that reduces the system’s non-determinism, and stepwise refinement. We derive novel proof rules for handling repeated system restarts and for compositional reasoning that exploits the system’s layered structure. The entire development is formalized in the theorem prover Isabelle/HOL.

The second part of the thesis deals with faults in a distributed setting. We consider benign (non-Byzantine) faults: crashes and recoveries of entire nodes, as well as network faults such as message delay and loss. A standard approach to distributed fault-tolerance is node replication. The central issue with replication is ensuring state consistency across replicas. A standard approach exists again: make replicas deterministic, and have them always agree on the next operation to be performed. Achieving such agreement is known as the consensus problem. The problem is typically parameterized in the size of the system, that is, the number of replicas; the solution should work for all values of the parameter. Past research has yielded a large number of algorithms that solve this problem. However, precisely understanding how these algorithms work and proving their correctness is non-trivial due to the complex and failure-prone environment they operate in. In fact, due to their parameterized nature, no fully automated procedure for verifying their correctness exists in the literature. Moreover, many of the algorithms
evidently share similar basic constructs, but it is unclear whether and which underlying ideas are also shared.

Our first main contribution in this area is a development, based on stepwise refinement, which provides an abstract and unified view of a sizable family of consensus algorithms. The models we build provide insights into the main ideas and design choices underlying the different algorithms, and classify them based on those choices. The results guide us in constructing a new consensus algorithm that answers an open theoretical question. All our results are formalized and verified in Isabelle/HOL, yielding precision and strong correctness guarantees.

For our second main contribution in this setting, we again exploit the observation that many algorithms share the same basic constructs. We design a simple but expressive language based on these constructs, and prove a cutoff theorem for consensus algorithms written in the language. Given an algorithm \( \mathcal{A} \), the theorem gives a number \( B \) (the cutoff bound) and states that the correctness of \( \mathcal{A} \) when applied to a system with \( B \) processes implies the correctness of \( \mathcal{A} \) for any system with any process count. The proof relies on a zero-one principle for the language, an analogue of the same principle for sorting networks. We encode a number of real-world algorithms in the language; for these, the theorem yields small bounds, either 5 or 7. This formalizes the so-called small scope hypothesis for the language. The hypothesis is a well-known informal observation that most bugs in parameterized systems manifest themselves already for small parameter values. It implicitly underlies approaches such as testing and model checking. As a consequence of the main theorem, we show that model checking can be leveraged to provide the first complete and fully automated verification method applicable to consensus algorithms.
Zusammenfassung


Unserer erster Beitrag auf diesem Gebiet ist eine Entwicklung, die auf schrittweiser Verfeinerung basiert und einen abstrakten und einheitlichen Blick auf eine beträchtliche Anzahl von Konsensalgorithmen liefert. Die Modelle, die wir bauen, gewähren Einblick in die Grundideen und die Entwurfseinscheidungen der verschiedenen Algorithmen und klassifizieren sie entsprechend. Die Resultate leiten uns zur Konstruktion eines neuen Konsensalgorithmus, der eine offene theoretische Frage beantwortet. All unsere Resultate sind formalisiert und verifiziert in Isabelle/HOL, sodass Präzision und Korrektheit garantiert sind.

# Contents

1 Introduction .................................................. 1
   1.1 Problems ................................................ 3
       1.1.1 Faults within a node .................................. 3
       1.1.2 Node-level and communication faults .............. 4
   1.2 Contributions ........................................... 7
       1.2.1 Handling crashes and storage failures within a node 7
       1.2.2 Handling node-level and communication faults ...... 8
   1.3 Outline ................................................ 9

2 Preliminaries ................................................ 11
   2.1 Set-theoretic Preliminaries ................................ 11
   2.2 Unlabeled and Labeled Transition Systems ............... 12
       2.2.1 Transition systems ................................... 12
       2.2.2 Traces ............................................... 13
       2.2.3 Trace properties and inclusion ...................... 14
   2.3 Simulations Between Transition Systems .................. 15
       2.3.1 Forward simulations .................................. 16
       2.3.2 Backward-forward simulations ....................... 17
   2.4 Modeling Computational Effects with Monads ............ 19
       2.4.1 Basic monadic operations ............................ 19
       2.4.2 Hoare logic ......................................... 20
   2.5 Simulations Between Monadic Computations .............. 21

I Handling Faults Within a Node ................................ 23

3 Verification of a Persistent Memory Manager Under Crashes and Hardware Faults ........................................ 25
   3.1 The Setting and Our Approach ............................ 25
   3.2 System Overview .......................................... 27
       3.2.1 The environment ...................................... 28
       3.2.2 The requirements .................................... 28
       3.2.3 The implementation ................................. 29
3.3 The Specification and Verification Framework . . . . . . . . . 31
  3.3.1 Modeling hardware failures and restarts (inner layer) . 31
  3.3.2 Specifications and refinement (outer layer) . . . . . . . 33
  3.3.3 Properties and their preservation . . . . . . . . . . . . 33
  3.3.4 Compositional reasoning . . . . . . . . . . . . . . . . . 34
3.4 Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
  3.4.1 The abstract model . . . . . . . . . . . . . . . . . . . . 37
  3.4.2 The concrete model . . . . . . . . . . . . . . . . . . . . 38
  3.4.3 Abstractions . . . . . . . . . . . . . . . . . . . . . . . 38
3.5 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
  3.5.1 Establishing the requirements . . . . . . . . . . . . . . . 40
  3.5.2 Statistics and discussion . . . . . . . . . . . . . . . . . 41
3.6 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

II Handling Node-Level and Communication Faults 45

4 Consensus and Models of Distributed Systems 47
  4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
  4.2 Failure and Timing Models . . . . . . . . . . . . . . . . . . . 48
    4.2.1 Failure assumptions . . . . . . . . . . . . . . . . . . . 48
    4.2.2 Timing assumptions . . . . . . . . . . . . . . . . . . . 50
    4.2.3 Double trouble: asynchrony and failures . . . . . . . . 51
  4.3 Formal Models . . . . . . . . . . . . . . . . . . . . . . . . . . 51
    4.3.1 Failure detectors . . . . . . . . . . . . . . . . . . . . . 52
    4.3.2 Lockstep models . . . . . . . . . . . . . . . . . . . . . 53
    4.3.3 The Heard-Of model . . . . . . . . . . . . . . . . . . . 55
  4.4 Consistency in Distributed Systems . . . . . . . . . . . . . . . 60
    4.4.1 The consensus problem . . . . . . . . . . . . . . . . . 61

5 Explaining and Unifying Consensus Algorithms 63
  5.1 Our Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
  5.2 The Specification and Verification Framework . . . . . . . . . 65
  5.3 Voting, Quorums and Defection . . . . . . . . . . . . . . . . . 66
    5.3.1 Formalizing Voting . . . . . . . . . . . . . . . . . . . . 67
    5.3.2 Voting ensures agreement . . . . . . . . . . . . . . . . 69
    5.3.3 Towards an implementation . . . . . . . . . . . . . . . 69
  5.4 Fast Consensus: Enlarging Quorums . . . . . . . . . . . . . . . 70
    5.4.1 Optimizing Voting . . . . . . . . . . . . . . . . . . . . 71
    5.4.2 Implementations: fast consensus . . . . . . . . . . . . . 72
  5.5 Same Vote . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
    5.5.1 Formalizing Same Vote . . . . . . . . . . . . . . . . . . 74
    5.5.2 Towards an implementation . . . . . . . . . . . . . . . 75
  5.6 Observing Quorums . . . . . . . . . . . . . . . . . . . . . . . 76
C Proof of the Cutoff Theorem 155
   C.1 Calculating the Cutoff Bound .............................. 156
   C.2 Simulation Relations ...................................... 157
      C.2.1 Basic relations on multisets ......................... 157
      C.2.2 Basic state relations ................................. 158
      C.2.3 Simulation relations ................................. 159
   C.3 The Lemmas ................................................. 161
      C.3.1 Multiset relations .................................. 161
      C.3.2 Preservation of guards .............................. 161
      C.3.3 Preservation of updates .............................. 163
      C.3.4 Preservation of update distributions ............... 168
      C.3.5 Simulation: the inductive step .................... 179
      C.3.6 Simulation ........................................... 191
      C.3.7 Property preservation .............................. 192

D Bibliography 193
“Well Ognjen, you can always tell them that the dog ate your thesis”, said my supervisor as I asked him what to put down as the reason for my defense deadline extension request. He thought he was joking, but two days earlier the proverbial dog almost did add my thesis to its diet: my computer had suddenly died, leaving me without several weeks’ worth of thesis work that I failed to back up. Fortunately, after a day of tinkering, I managed to recover the data, and the crisis was avoided.

At the expense of some anxiety, the incident provided me with a personal example for the underlying problem that this work addresses: today’s computer systems are unreliable. The reader has likely had their own experiences paralleling mine. Of course, computer failures can have much worse consequences than a delayed PhD thesis. The list of well-publicized examples is long: from the destruction of NASA’s Mariner 1 space probe in 1962, caused by a software error, to the 2012 system interoperability failure that lost Knight Capital’s Group 440 million USD in 45 minutes. The total yearly cost of software bugs to the US industry was estimated at 60 billion USD in 2002 by NIST [Tas02], and to the industry worldwide at 312 billion USD in 2013 by a team from Cambridge University [BJC+13]. The cost is not just monetary. In the 1980s, a bug in the control software of the Therac-25 radiation therapy machine caused six massive radiation overdoses leading to the death of three patients [LT93]. The unintended acceleration problems in a range of Toyota vehicles in 2000s, linked to deaths of 89 people, is strongly suspected to have been caused by the failures of the electronic throttle control system [Koo14].

The problem of computer reliability was already recognized early on, and gave rise to the field of formal verification. Its roots can be traced back to as early as 1949, when Alan Turing published a paper [MJ84] addressing the reliability problem. The three main ingredients of any formal verification method (or formal method for short) are a mathematical model of the system, a mathematical specification of the system’s desired proper-
ties, and a mathematical proof that the model is correct, that is, that it obeys the specification. The research community has produced a plethora of formal methods, with a multitude of ways to model and specify systems (e.g., automata, logic, programming language semantics) and to perform the correctness proofs (e.g., interactive and automated theorem proving, model checking, static analysis).

The main obstacles to the wider adoption of formal methods are cost and scaling. At the very minimum, the methods require a system model and a specification, which can already be laborious to produce. The larger problem is the correctness proof, the construction of which is usually either undecidable, or of very high algorithmic complexity; in most cases, it requires some form of manual interaction. One well-known way of reducing the overall proof complexity is the stepwise refinement (or just refinement) method. It describes a system by a sequence of models, starting from a simple abstract model that captures the main ideas of the system, and then gradually adding details in the subsequent, more concrete models. A typical example is an abstract model that uses a simple but inefficient data structure that is replaced by a more complex but efficient implementation in the later models. A key feature of the refinement method is property preservation: we can prove a property on the simpler, abstract model, and have it automatically hold on the concrete model, provided we prove refinement. Still, even with techniques such as refinement, the complexity of the models, specifications and of the proof worsen as the system size increases. Formal verification has thus traditionally been limited to small and special-purpose systems or even simplified models thereof, such as systems used in the aerospace domain, with extremely high costs of failure that surpass the verification costs. Advances in formal verification have, however, gradually driven the verification cost down, and in the last ten years formal methods have been scaled to source and binary code level verification of realistic general-purpose systems, such as the CompCert C compiler [Ler06] or the seL4 microkernel [KEH+09].

An assumption that both [Ler06] and [KEH+09] make is that the hardware does not fail. More precisely, their system model does not include any form of hardware faults. This assumption is shared by most formal methods literature, and is certainly not unreasonable. It greatly simplifies the system model, and it is indeed often the software that is the weak link. Nevertheless, hardware also can and does fail. The anecdote in the beginning is in fact an example: it was the disk controller that was at fault, and not the software. The loss of throttle control in Toyota vehicles could be triggered in testing by a single bit flip [Koo14, Bar14]. Thus, many systems take hardware failures into account in their design. A prominent example is storage. Most modern hard disks come with a pool of spare disk sectors that are inaccessible to the user. When one of the other sectors fails, the hard disk controller transparently remaps it onto one of the spare sectors. The hard disk is thus an example of a fault-tolerant system. Another example is Internet traffic
routing, which was designed to work around the failures of individual links. Obviously, fault tolerance cannot be absolute. If the failures become too severe or are of an unexpected nature, the system can still fail. For example, once the spare sectors are used up, the sector faults become visible to the rest of the system. Misconfigured routers can cause Internet traffic outages, even when all links are still operational. Each fault-tolerant system thus assumes a particular fault model; the system is not expected to tolerate faults that fall outside of the model. Furthermore, even if the faults are within the model, they need not be fully masked and their effect on the system’s behavior can still be visible to the client. For example, an Internet packet might have to be retransmitted in the presence of link failures, which would not be necessary in a fault-free system.

The fact that a system is designed to be fault-tolerant is a good indicator that the system might be mission-critical, with a high cost of failure. Such systems are then obvious candidates for formal verification, but they also pose significant verification challenges compared to their non-fault-tolerant counterparts. First, the system model must be extended to include the fault model. Second, since the faults might have a visible influence on the system’s behavior, the specification may also be complicated by incorporating the new behaviors. Lastly, and most importantly, the system itself is usually significantly more complicated than its non-fault-tolerant counterparts. It thus tends to be significantly larger, more difficult to understand, and, consequently, more difficult to prove correct, even on an informal level.

This motivates the main broad research questions behind this thesis. What kind of fault-tolerant systems can we scale today’s formal methods to? How can we improve our understanding of these systems? Can we derive methods that facilitate (and thus cheapen) the verification?

1.1 Problems

The answers to the above questions depend on the fault model and the system at hand. The main fault types that we consider are crashes that completely halt a single computer system, such as power failures. We consider their effects first on a single computer system, which we call a node, and then in a distributed setting, where the nodes that crash are a part of a higher-level distributed system. Both of these settings have additional fault types that have a natural connection to crashes and their effects. We thus next look into each setting separately, and precisely identify the problems that we address in the thesis.

1.1.1 Faults within a node

When a node crashes, the only system part that reliably retains its contents is the system’s persistent storage. However, persistence alone is not enough.
1. Introduction

As the crashes can occur at any time, the programs that interact with (and in particular update) the persistent state can be interrupted before they finish executing, and some data loss is often unavoidable. However, this data loss can be disproportionately large (for example, the complete loss of a file system’s root directory [YTEM06]), since the interrupted program might leave the persistent storage in a logically corrupted state. To minimize the loss, after the system is restarted (rebooted), it must ensure that the storage contents are brought back to a state consistent with the system’s specification, regardless of when the crash occurred, and even if further crashes occur during the restart. A typical requirement is that the storage updates be transactional: in case of a crash, the update is either executed to the end, or not at all. For example, a possible requirement for a POSIX file system is that system calls (such as `unlink()` or `fwrite()`) be transactional.

In addition to crash faults, persistent storage is also often hit by spontaneous failures. The most severe problem is data degradation, where the storage contents spontaneously changes. This is a common problem both in solid state disks [CLGM15] and hard drives [Kle16]. For critical applications, the system should not only be able to reliably detect such failures, but it should also provide some form of resilience to failures and prevent data loss when the failures are not too severe.

Systems that provide transactional updates in the presence of crashes have already been analyzed before in the literature [SL00, HBdJL01, PHdJ02, DB09]. However, the fault models in these systems include neither writes corrupted by a crash (i.e., the hardware write operations are assumed to be atomic) nor data degradation, and none of the systems provide any resilience against it. Resilient systems are significantly more complicated, as they require redundancy, not only of data, but also of metadata. Scaling formal verification to a system resilient to these types of failures is thus an open question.

1.1.2 Node-level and communication faults

A distributed system consists of multiple communicating nodes that are working towards a common goal. As these systems are inherently concurrent, building them correctly is non-trivial even if no faults are present. However, distributed systems are rarely fault-free [RGO06]. Node failures are one major class of faults; they include not only node crashes, but also stalled nodes that are taking longer than anticipated to complete their tasks, clock drifts, and even Byzantine faults, where the nodes start behaving arbitrarily, and potentially maliciously. The other major class are communication faults: the messages might get dropped, delayed, and even duplicated. The messages might even become corrupted, but this can be efficiently mitigated using error correction. It is perhaps surprising that we classify communication and processing delays and clock drifts as faults; however, such timing
1.1. Problems

**assumptions** and the effects of the other fault types are closely related in distributed systems [She15]. In particular, in *synchronous* models, where no such delays or drifts are possible, it is possible to identify crashed nodes precisely. In contrast, in the *asynchronous* models where such delays are possible, the node failures, message loss and the delays themselves become indistinguishable [FLP85]. Obviously, this failure-prone environment significantly increases the difficulty of ensuring the correctness of distributed systems. With the strong shift towards cloud computing services (whose worldwide market was estimated at 110 billion USD in 2015 [Col16], with a 28% annual growth rate) and big data that requires distributed processing, these correctness issues now increasingly also concern the “common programmer” [BK14]. In response, the last few years have seen the rise of industrial testing frameworks that succeeded in finding bugs in a range of popular open-source distributed systems [Kin13], and model checkers being used in-house in software companies to verify models of distributed systems (e.g. Amazon [New14] is using TLA+ [Lam02]).

Regardless of the fault model, if the system’s nodes are to act coherently, they must coordinate their actions. Coordination in turn requires the nodes to maintain a consistent shared state of some sort. A well-known technique for this is state machine replication [Lam78], which reduces the coordination problem to solving the so-called *distributed consensus* problem. In fact, it can be shown [Her91] that consensus is not only sufficient, but also necessary for solving the consistency problem (in particular, for a notion of consistency called *linearizability*). Informally, the consensus problem requires the nodes in the system to each propose a value, and then have all nodes agree on exactly one of those values. The problem is deceitfully simple: while it is trivial to solve in a fault-free system, this changes sharply once faults are introduced.

**Understanding consensus algorithms**

In this thesis, we do not consider Byzantine faults. However, even without them, the interplay of concurrency and faults can still drive the algorithms’ executions in many different ways. This makes the consensus problem not only difficult, but even impossible to solve deterministically in asynchronous systems when even just a single node fails [FLP85]. Assumptions such as partial synchrony [DLS88] and bounds on the number of node failures impose just enough constraints on the faults to admit solutions, but still retain the main difficulties of the full fault model. Hence the number of possible executions of an algorithm in this setting is still immense, making the understanding of both the algorithms and their correctness non-trivial. For example, Lamport’s seminal Paxos algorithm [Lam98, Lam01a] is often deemed to be hard to understand. In response, [PLL00, BDFG03b, BDFG03a, Cac09, Lam01b] all aim to explain it in simpler, more abstract terms. In general,
1. Introduction

Abstractions that simplify the algorithms or the setting can be immensely helpful in understanding distributed algorithms, making the development of such abstractions an appealing research topic.

As examples of setting simplifications, Chandra and Toueg [CT96] introduce failure detectors, which effectively abstract timing issues away. In a similar vein, Gafni [Gaf98] and Charron-Bost and Schiper [CBS09] provide an abstraction of the asynchronous (or partially synchronous) setting for the class of algorithms operating in so-called communication-closed rounds. For this class, the asynchronous setting is replaced by what is essentially a synchronous model weakened by message loss (dual [RS13] to strengthening the asynchronous model by failure detectors). As the resulting models provide the illusion that all the nodes operate in lockstep, we refer to them as lockstep models.

More than 30 years of research on consensus has also yielded a large collection of consensus algorithms. Many of them appear to share similar underlying algorithmic ideas, although their presentation, structure, and details differ. A natural question is whether their similarities can be distilled and captured in a uniform and generic way, and this has led to another substantial body of work [MR99, HMR02, MRR02, GR04, GR07, SRSD08, RMS10]. In the same vein, one may ask whether the algorithms can be classified by some natural criteria.

Summing up, we see a clear need for (1) abstraction and simplification and (2) unification and classification of consensus algorithms, in order to understand their essence and relationships. Additionally, as the setting they operate in is complex, it is necessary that both (1) and (2) are addressed in a precise and correct manner. We thus add to our wish list (3) precision and correctness guarantees.

**Parametric verification of consensus algorithms**

The problem of understanding algorithms is naturally related to the problem of verifying their correctness. As already mentioned, a main criterion for the industrial applicability of a verification method is its cost. It is therefore unsurprising that the above industrial success stories in the area of fault-tolerant systems’ verification utilize push-button methods such as testing and model checking. These methods analyze individual system configurations with a small, fixed number of participating nodes. However, many real distributed systems are expected to work for any given number of nodes, i.e., they are parameterized in this number. The deployed instances are often larger than the analyzed ones, and the above analyses then offer no a priori guarantees for the deployed system. Still, an informal observation known as the small scope hypothesis [Jac12] states that analyzing small system instances of parameterized systems (distributed or otherwise) suffices in practice. Empirical studies [ADKM03, OPP+12, YLZ+14] support this
hypothesis in different settings. In the distributed setting, a recent analysis [YLZ+14] of 198 bug reports for several popular systems found that 98% of those bugs could be triggered by three or fewer nodes.

A crucial question is then: can we state and formally prove this hypothesis? That is, given a parameterized system and a property $\psi$, can we determine a cutoff bound: a number $B$ such that whenever all systems with parameter values of $B$ or less satisfy $\psi$ then systems with arbitrary parameter values also satisfy $\psi$? The answer is no in general as the parametric verification problem (verification a parameterized system for all parameter values) is undecidable even when we can decide the system’s correctness for each parameter value [AK86, Suz88]. The best we can hope for is to find cutoff bounds for interesting classes of systems and properties. While such results exist [EK00, EK03, EN03, GHJS08, KKW10, JM12], none of them apply to fault-tolerant distributed systems in general, and to algorithms for solving the consensus problem in particular.

In addition to the lack of cutoff results, no fully automated method for parametric verification of consensus algorithms exists. The invariant verification approach of [DHV+14] comes the closest, but it is not fully automated as the user must find inductive invariants that are automatically checked. Also, while the authors report good practical results, their main algorithm is only a semi-decision procedure. Other reported results have either performed bounded verification (e.g., [TS08, TS10, DTT14]) or used interactive verification methods (e.g., [JM05, KNR12, DM12, HHK+15, SRvR+14]).

1.2 Contributions

As we address problems in two different settings, we present our contributions for each setting separately.

1.2.1 Handling crashes and storage failures within a node

Our contributions in this area are twofold:

1. We propose novel modeling and reasoning techniques for systems tolerating crashes and storage faults. Unlike previous work, we model restart handling using special language-level constructs. This allows us to derive structured refinement proof rules for crashes and restarts (including repeated restarts) that enable compositional reasoning. The rules exploit the existing logical layering of the system, where the lower system layers deal with nested transactions that handle the metadata redundancy.

We also model restarts and failures as occurring only during low-level storage operations, unlike previous work where restarts were inserted in
between all system operations. This limits the asynchrony of these effects and hence simplifies the verification by reducing non-determinism.

2. The above combination of techniques enabled the success of our case study, the verification of the persistent memory manager in IBM’s 4765 cryptographic coprocessor. The case study is substantial, industrially relevant, and more complex than related ones published hitherto, due to the system’s resilience to hardware failures, and since the system is not based on the standard write-ahead log technique. We believe that our approach is applicable in related areas such as smart cards and file systems.

1.2.2 Handling node-level and communication faults

We identify two main contributions of our work in this area. Our first main contribution is a development which provides a simple and abstract explanation of many consensus algorithms found in the literature, as well as a classification of the algorithms. In more detail:

1. The development combines stepwise refinement, lockstep models, and formal modeling and reasoning. This combination of techniques was not present in the previous work, and enables us to address all three problems that we have identified: abstraction and simplification, unification and classification, and precision and correctness guarantees. The refinement approach provides a natural framework for the abstraction, unification, and classification of a variety of algorithms, the lockstep model additionally increases the abstraction level, and the formality of our approach provides the desired precision and correctness guarantees.

2. Our abstract models provide insight into different classes of consensus algorithms by laying bare the underlying mechanisms in a clear, simple, and concise way. These models consist of a single non-deterministic event modeling a round of the algorithm. The enabling predicates of this event precisely capture the safety conditions needed to preserve agreement.

3. The branching points in our refinement tree directly reflect the design choices behind the different types of algorithms. For the setting with benign failures, our classification encompasses more algorithms and algorithm types than any other one presented in the literature, testifying to the flexibility of our approach.

4. While we focus on existing algorithms, we also derive a new one. Our development provided insights that allowed us to affirmatively answer a question raised in [CBS09], asking whether there exists a leaderless
consensus algorithm with a wait-free messaging layer that tolerates up to \( \frac{N}{2} \) node failures.

Our second main contribution in this area is to prove the small scope hypothesis for an expressive class of consensus algorithms. In more detail:

1. We define a language \( \text{ConsL} \) that captures numerous consensus algorithms found in the literature, using just a few simple primitives, where we focus on algorithms for the asynchronous and partially synchronous setting with benign (i.e., non-Byzantine) failures.

2. For \( \text{ConsL} \) algorithms, we prove a zero-one principle for consensus: the algorithm’s correctness for binary inputs (from the set \{0, 1\}) entails the algorithm’s correctness for inputs from any ordered set, finite or infinite. This is an analogue of the same principle for sorting networks [Knu73].

3. We give cutoff bounds for algorithms run on binary inputs: given a \( \text{ConsL} \) algorithm, we determine a number \( B \) such that the algorithm solves consensus on binary inputs if it solves it for exactly \( B \) nodes. Together with Step 2, this proves the small scope hypothesis for \( \text{ConsL} \) algorithms.

4. The bounds we obtain for real-world algorithms are indeed small: 5 or 7 nodes for all algorithms considered in this paper. We can thus leverage model checking to provide the first fully automated decision procedure applicable to a range of consensus algorithms, and we provide a tool that generates Promela/Spin [Hol04] models from \( \text{ConsL} \) algorithms. The resulting verification times are competitive with the semi-automated method of [DHV+14].

1.3 Outline

This thesis consists of two parts, that correspond to the two settings outlined previously. The preliminaries that are shared between the two parts are presented in Chapter 2. The conclusions and the future work are also presented jointly, in Chapter 7.

The first part consists of Chapter 3, where we present our work on the verification of the transactional persistent memory manager used in the IBM 4765 cryptographic coprocessor, and discuss our verification framework, our compositional reasoning approach, and the related work. The chapter is based on the publication [MS14]. All of the results in this chapter are verified using the theorem prover Isabelle/HOL, and can be found online [Mar16].

The second part of the thesis starts with some further background material on distributed systems and consensus protocols, presented in Chapter 4.
Chapter 5 then presents our work on explaining and unifying consensus algorithms using stepwise refinement. We also present our new algorithm, and discuss the related work. The chapter is based on the publication [MSB15], and the results presented there are also formalized in Isabelle/HOL. The formalization is available [MS15] in the Isabelle Archive of Formal Proofs; however, the presentation in this thesis differs slightly from the one in the paper and the archive. An adjusted version of the proofs is also available [Mar16]. Finally, Chapter 6 presents our language ConsL for expressing consensus algorithms, sketches the proof of the small scope hypothesis for it, shows our experimental verification results, and discusses the related work. The proofs for the theorems of this chapter are presented in the Appendices A–C. The chapter is based on the publication [MSB17].
Chapter 2
Preliminaries

In this chapter, we review the basic notions that we will be using throughout the thesis. Only known results and variants thereof are presented here. The notions we rely on are standard, although we do use several flavors of them, as convenient for the particular setting.

2.1 Set-theoretic Preliminaries

Given a relation $R \subseteq A \times B$, we write $\text{dom}(R)$ for its domain, and $R[X]$ for the image of the set $X \subseteq A$ under $R$, that is, the set $\{y \mid \exists x \in X. (x, y) \in R\}$. The range of $R$, written $\text{ran}(R)$, is then $R[A]$. In general, we view functions as relations where convenient, and use the same notation $\text{dom}(f)$, $f[X]$ and $\text{ran}(f)$ for partial (respectively total) functions $f$ from $A$ to $B$, the set of which we denote by $A \rightarrow B$ (respectively $A \rightarrow B$). Often, when writing functions as relations, we use $\mapsto$ to denote the pairs. For example, $\{a \mapsto x, b \mapsto y\}$ denotes the function $f$ with $\text{dom}(f) = \{a, b\}$, $f(a) = x$ and $f(b) = y$. In the special case where the function $f$ is constant on its domain $D$ and equals some $c$, we write $f = [D \mapsto c]$. For two relations $R_1$ and $R_2$, we define their left relational composition $R_1 \circ R_2 \triangleq \{(x, y) \mid \exists z. (x, z) \in R_1 \land (z, y) \in R_2\}$. Next, we say that a relation $R \subseteq X \times Y$ is inverse total if for all $y \in Y$, the set $\{x \mid (x, y) \in R\}$ is not empty.

We liberally apply the currying transformation, identifying a function $g : A \times B \to C$ with the corresponding function $g' : A \to (B \to C)$ where $\to$ associates to the right, so the parenthesis around $B \to C$ can be dropped. That is, given a $g$ of either type, we use $g(x, y)$ and $g(x)(y)$ interchangeably. We use the lambda notation for anonymous functions, as standard. In a functional programming context (e.g., when talking about monads), we will also often use simple juxtaposition to denote function application, writing $f x$ instead of $f(x)$. When doing so, we assume function application to be left-associative, as standard.

Given a partial (or total) function $f$ and a partial (or total) function $g$,
we write $f \triangleleft g$ for the update of $f$ by $g$, that is, the function $h$ such that $h(x) = g(x)$ if $x \in \text{dom}(g)$ and $h(x) = f(x)$ otherwise. We also define point updates, where $f(x := y) \triangleq f \cup \{x \mapsto y\}$.

Given two sets $S_1$ and $S_2$, $S_1 + S_2$ denotes the right-associative sum of $S_1$ and $S_2$, with injections (or constructors) $\iota_1 : S_1 \to S_1 + S_2$ and $\iota_2 : S_2 \to S_1 + S_2$. The unit set $\textbf{1}$ consists of a single element $\ast$. Given a set $S$, we use $S_{\bot}$ to denote the set $S \cup \{\bot\}$, where $\bot$ is a value not present in $S$. We will sometimes represent partial functions $A \to B$ as total functions $A \to B_{\bot}$, as common, by setting $f(x) = \bot$ when $x \notin \text{dom}(f)$. However, we will explicitly mention when this is the case.

A multiset $M$ over a set $S$ is a function $S \to \mathbb{N}$, where $M(x)$ is the multiplicity of $x$ in $M$. We define $|M| \triangleq \sum_{s \in S} M(s)$ and, for a set $X$, $M \setminus X = M'$, where $M'(x) = 0$ if $x \in X$ and $M'(x) = M(x)$ otherwise. Next, for a partial (or total) function $f : A \to B$, we define the multiset image $\#f[X] : B \to \mathbb{N}$ of a finite set $X \subseteq A$ under $f$ by $\#f[X](b) \triangleq |f^{-1}(b) \cap X|$. We also define the multiset range of $f$ by $\#f[A]$. We introduce notation for specifying finite multisets. For example, we write $M = \{m_x \times x, m_y \times y\}$ to denote the multiset $M$ where $M(x) = m_x$, $M(y) = m_y$, and $M(z) = 0$ for $z \notin \{x, y\}$.

Finally, we define $[a,b]_{\mathbb{Q}} \triangleq \{c \in \mathbb{Q} \mid a \leq c < b\}$.

### 2.2 Unlabeled and Labeled Transition Systems

Our treatment of transition systems and simulations between them is largely inspired by [LV95] and [AL91]. As we use both labeled and unlabeled transition systems, virtually all definitions come in two flavors. We specialize and adjust the standard notions when appropriate, to simplify our proofs. The main such changes are:

1. we define traces of labeled transition system to include not only the labels but also the states;

2. as we do not compose systems, we do not use the notion of internal (also known as silent) transitions in labeled systems.

In the rest of the section, we define the used notions formally.

#### 2.2.1 Transition systems

An unlabeled transition system is a triple $(\Sigma, \Sigma^0, \to)$, where $\Sigma$ is the set of states, $\Sigma^0 \subseteq \Sigma$ is the set of initial states, and $\to \subseteq \Sigma \times \Sigma$ the transition relation. We will typically write the transition relation in an infix notation, hence $s \to s'$ stands for $(s, s') \in \to$.

A labeled transition system is a quadruple $(\Sigma, \Sigma^0, \mathcal{L}, \to)$, where $\Sigma$ is the set of states, $\Sigma^0 \subseteq \Sigma$ is the set of initial states, $\mathcal{L}$ is the set of labels, and
→ ⊆ Σ × L × Σ the transition relation. Similar to the unlabeled case, we write \( s \xrightarrow{\alpha} s' \) for \( (s,\alpha,s') \in \rightarrow \).

For convenience, in this thesis we specify the states of a system by a record containing the system’s variables. Moreover, for unlabeled systems we sometimes specify the system’s transitions by a set of parameterized events, in the style of [Abr10]. An event is specified by a guard and an action. The guard is a predicate defining when the event is enabled in a given state. The action describes a relation between the source and the target state, specified as a state update. Here is a prototypical unlabeled event with a vector of parameters \( \vec{p} \), a guard \( G \), and an action that simultaneously updates the state variables \( \vec{x} \) using the update functions \( \vec{a} \), one for each variable in \( \vec{x} \):

\[
\text{Event} \ \text{evt}(\vec{p}):
\]
\[
\text{Guard} \quad G(\vec{x},\vec{p})
\]
\[
\text{Action} \quad \vec{x} := \vec{a}(\vec{x},\vec{p})
\]

An event \( \text{evt}(\vec{p}) \) has a straightforward relational semantics, which we denote by \( \rightarrow_{\text{evt}(\vec{p})} \). The (unlabeled) system’s transition relation \( \rightarrow \) is then defined as the union over all relations induced by the events, for all parameter values \( \vec{p} \). Sometimes, we will assume that the system includes a special event called \( \text{skip} \), whose relational semantics is the identity relation. We say that such systems are closed under stuttering.

### 2.2.2 Traces

The semantics of a transition systems is given by its set of traces. An unlabeled trace over \( \Sigma \), where \( \Sigma \) is a set of states, is any (finite or infinite) sequence \( s_0, s_1, \ldots \) of states from \( \Sigma \). When \( \Sigma \) is clear from the context, we will just use the term unlabeled trace. The length of a trace \( \tau \) is the usual length of the sequence, denoted by \( \text{len}(\tau) \), where \( \text{len}(\tau) = \infty \) for infinite traces. We sometimes view unlabeled traces as partial functions \( \tau : \mathbb{N} \rightarrow S \), whose domain \( \text{dom}(\tau) \) is \( \{0, \ldots, \text{len}(\tau) - 1\} \). We write \( \tau|_l \) for the prefix of a trace \( \tau \) of length \( l + 1 \) (where \( \tau|_l = \tau \) when \( l \geq \text{len}(\tau) \)). Given an unlabeled transition system \( T \) of the above form, an execution fragment of \( T \) is any unlabeled trace \( \tau = s_0, s_1, \ldots \) such that for all \( i < \text{len}(\tau) - 1 \), \( s_i \rightarrow s_{i+1} \). A trace of \( T \) is any execution fragment with \( s_0 \in \Sigma^0 \). We denote the set of all traces of \( T \) by \( \text{tr}(T) \). We say that a state \( s \) is reachable in \( T \) if there exists a trace of \( T \) such that \( s \) appears in the trace. We denote the set of all reachable states of \( T \) by \( \text{reach}(T) \).

We extend the notions to labeled systems as follows. A labeled trace is any alternating sequence \( s_0, l_0, s_1, l_1, s_2, \ldots \) of states and labels that starts with a state, and if it is finite, also ends with a state \( s_l \). The length of a labeled trace \( \tau \), again denoted by \( \text{len}(\tau) \), is the number of states appearing
in it, with \( \text{len}(\tau) = \infty \) for infinite traces. As for unlabeled traces, \( \tau|_l \) denotes the prefix \( s_0, l_0 \ldots s_l \) of the trace \( \tau = s_0, l_0, s_1, \ldots \), where again, \( \tau|_l = \tau \) if \( l \geq \text{len}(\tau) \). Given a labeled system \( T \), an execution fragment of \( T \) is a labeled trace \( \tau = s_0, l_0, s_1, \ldots \) such that for all \( i < \text{len}(\tau) \), \( s_i \xrightarrow{l} s_{i+1} \). A trace of \( T \) is any execution fragment with \( s_0 \in \Sigma^0 \), and all traces of \( T \) are again denoted by \( tr(T) \). For a labeled trace \( \tau \), we denote by \( \text{unlabel}(\tau) \) the unlabeled trace obtained by removing the labels from \( \tau \), and by \( \text{labels}(\tau) \) the sequence obtained by removing the states from \( \tau \). Given a labeled system \( T \) and a sequence of labels \( \lambda \), a state \( s \) is reachable by \( \lambda \) in \( T \) if there exists a \( \tau \in tr(T) \) such that \( \text{labels}(\tau) = \lambda \) and \( s \) is the last state of \( \tau \).

Occasionally, it will be useful to distinguish between finite and infinite traces of a system, whether labeled or not. We denote the former by \( tr^*(T) \), and the latter by \( tr^\omega(T) \). In general, whether traces are labeled or not will be clear from the context, that is, the system being discussed. Furthermore, we will omit making explicit distinctions between the two kinds of systems where the definitions or results apply to both kinds.

### 2.2.3 Trace properties and inclusion

A **property** is a set of traces. A system \( T \) satisfies the property \( P \) if \( tr(T) \subseteq P \). Due to their trace semantics, we can also view systems as properties and relate two systems by relating their sets of traces. If \( tr(T_2) \subseteq tr(T_1) \), we write \( T_2 \sqsubseteq_T T_1 \), and say that \( T_2 \) refines \( T_1 \) or conversely, that \( T_1 \) abstracts \( T_2 \). We call \( T_2 \) the concrete system and \( T_1 \) the abstract system. We also write \( T_2 \sqsubseteq_{T^*} T_1 \) for \( tr^*(T_2) \subseteq tr^*(T_1) \), and \( T_2 \sqsubseteq_{T^\omega} T_1 \) for \( tr^\omega(T_2) \subseteq tr^\omega(T_1) \).

So far, we assumed that both the system and the property (or the abstract system) use the same set of states. We can relax this assumption by providing a relation between two different sets of states. For labeled systems, we will still assume that the set of labels remains the same in both the system and the property.

Given unlabeled traces \( \tau = s_0, s_1, \ldots \) over state set \( \Sigma \), and \( \tau' = s'_0, s'_1, \ldots \) over state set \( \Sigma' \), and given a relation \( R \subseteq \Sigma \times \Sigma \) we say that \( \tau \) is an *image of \( \tau' \) under \( R \), written \( \tau \in R[\tau'] \), if \( \text{len}(\tau) = \text{len}(\tau') \) and \( (s_i, s'_i) \in R \) for all \( i < \text{len}(\tau) \). Given an unlabeled system \( T \) with the state set \( \Sigma_T \), a property \( P \) over a state set \( \Sigma_P \), and a relation \( R \subseteq \Sigma_P \times \Sigma_T \), we say that \( T \) satisfies \( P \) under \( R \), denoted \( T \sqsubseteq^R_T P \) if for each trace \( \tau_T \in tr(T) \) there exists a trace \( \tau_P \in P \) such that \( \tau_T \in R[\tau_P] \). We also define the satisfaction of \( P \) by the finite traces of \( T \) under \( R \), denoted \( T \sqsubseteq^F_T P \), and the satisfaction of \( P \) by the infinite traces of \( T \) under \( R \), denoted \( T \sqsubseteq^R_T P \), in the obvious manner.

Given a labeled system \( T \) with the state set \( \Sigma_T \), a property \( P \) over a state set \( \Sigma_P \), and a relation \( R \subseteq \Sigma_P \times \Sigma_T \), we say that \( T \) satisfies \( P \) under

---

1 Many authors point the \( \sqsubseteq_T \) relation in the opposite direction, writing \( T_1 \sqsubseteq_T T_2 \) where we write \( T_2 \sqsubseteq_T T_1 \). This is often done when considering \( \sqsubseteq_T \) in the context of a lattice of programs. Since we do use an algebraic view, we choose \( \sqsubseteq_T \) by the analogy with \( \subseteq \).
2.3. Simulations Between Transition Systems

$R$, denoted $T \sqsubseteq^R P$ if for each $\tau_T \in \text{tr}(T)$ there exists a trace $\tau_P \in P$, such that $\text{unlabel}(\tau_T) \in R[\text{unlabel}(\tau_P)]$ and $\text{labels}(\tau_T) = \text{labels}(\tau_P)$. Again, we define $T \sqsubseteq^R_\# P$ and $T \sqsubseteq^R_\omega P$ in the obvious manner.

Having extended the notion of property satisfaction to different state spaces, we do the same for the notions of refinement and abstraction. We define $T_2 \sqsubseteq^R_1 T_1$, $T_2 \sqsubseteq^R_\# T_1$, and $T_2 \sqsubseteq^R_\omega T_1$ in the analogous manner. The meaning of property satisfaction (or refinement) now also depends on the relation $R$. It is easy to see that refinement is transitive in the following sense: if $T_2 \sqsubseteq^R_1 T_1$ and $T_3 \sqsubseteq^R_2 T_2$, then $T_3 \sqsubseteq^{R_1 \circ R_2}_1 T_1$ (where $\circ$ is the left relational composition).

Furthermore, if $T_1$ satisfies some property $P$, then so do $T_2$ and $T_3$ under the relations $R_1$ and $R_1 \circ R_2$, respectively. This allows us to carry out abstraction (or refinement) in a stepwise manner, producing a sequence of models. The more abstract models are typically simpler, making it easier to prove properties of interest on them. The concrete systems then immediately satisfy all the properties of the systems they refine, under suitable relations. Note, however, that this does not hold for conditional properties, that hold only for a particular subset of a system’s traces. A typical example are liveness properties, that are often conditioned on some fairness assumptions. To prove refinement for such properties, one would need to prove that every trace from the desired subset (e.g., subset of fair traces) of the concrete model is an image of some trace from the desired subset (e.g., the subset of fair traces) of the abstract model. While there do exist extensions to the above refinement framework (e.g., [AL91]) that can handle conditional properties, we do not use them in this thesis.

In general, instead of a sequence of models, we can also construct a tree, by refining the same abstract model in several different ways. Even more generally, one can also construct a directed graph of models. Such a graph need not even be acyclic, although all models on a graph cycle are identical, i.e., their sets of traces are the same.

2.3 Simulations Between Transition Systems

Simulation techniques allow us to prove refinement in a local fashion, looking only at individual states and transitions rather than entire traces. While a variety of such techniques exist [LV95], we will use only two of them in this thesis: forward simulation and backward-forward simulation.

In the following, we will assume that $T_1$ and $T_2$ are systems over state sets $\Sigma_1$ and $\Sigma_2$ respectively. Furthermore, when they denote labeled systems, we assume that their label sets are identical.
2. Preliminaries

2.3.1 Forward simulations

A relation $R$ on $\Sigma_1 \times \Sigma_2$ is a forward simulation from a system $T_2$ to a system $T_1$, written $T_2 \sqsubseteq^R T_1$, if:

(F1) For all $s_2 \in \Sigma_2^0$ there exists an $s_1 \in \Sigma_1^0$ such that $(s_1, s_2) \in R$.

(F2) Whenever $s_2 \rightarrow s'_2$ (respectively $s_2 \overset{l}{\rightarrow} s'_2$ for labeled transition systems) and $(s_1, s_2) \in R$, there exists an $s'_1$ such that $s_1 \rightarrow s'_1$ (respectively $s_1 \overset{l}{\rightarrow} s'_1$) for labeled transition systems) and $(s'_1, s'_2) \in R$.

For unlabeled transition systems, the above conditions can also be written in a more compact form. The initial condition (F1) can be written as $\Sigma_2^0 \subseteq R[\Sigma_1^0]$. The step condition (F2) can be written as $(R \circ \rightarrow_2) \subseteq (\rightarrow_1 \circ R)$. It is also often represented graphically by a “refinement square”:

![Refinement Square Diagram]

where the dashed circle indicates that we must find the $s'_1$ to complete the diagram.

Moreover, for unlabeled transition systems specified by events we follow [Abr10] and decompose the step condition (F2) as follows. We relate every concrete event $evt_2(\vec{p}_2)$ (with guard $G_2$ and action functions $\vec{a}_2$) to some abstract event $evt_1(\vec{p}_1)$, and decompose the proof obligation into:

1. guard strengthening, i.e., the concrete guard implies the abstract one under $R$, and
2. action refinement, i.e., the updated states are also related by $R$.

More formally, for any $\vec{x}_1$ and $\vec{x}_2$ such that $(\vec{x}_1, \vec{x}_2) \in R$, we must prove that $G_2(\vec{x}_2, \vec{p}_2)$ implies $G_1(\vec{x}_1, \vec{p}_1)$ (guard strengthening), and $(\vec{a}_1(\vec{x}_1, \vec{p}_1), \vec{a}_2(\vec{x}_2, \vec{p}_2)) \in R$ (action refinement).

**Theorem 1** (Soundness of forward simulation). $T_2 \sqsubseteq^R T_1 \implies T_2 \sqsubseteq^R T_1$.

**Proof.** Assume $T_1$ and $T_2$ are labeled systems, and assume $\tau_2 = s_0, t_0, s_1, \ldots$. By (F1), there exists an initial state $t_0$ of $T_1$ such that $(t_0, s_0) \in R$. Define a trace $\tau_1$ as follows: $\text{len}(\tau_1) = \text{len}(\tau_2)$, the first state of $\tau_1$ is $t_0$, the labels are $t_0, t_1, \ldots$, and, for $0 \leq i < \text{len}(\tau_2)$, the $(i+1)$-th state $t_{i+1}$ is some state (that is, defined by Hilbert’s choice operator) such that $t_i \overset{i}{\rightarrow} t_{i+1}$ (where $t_i$ is the $i$-th element of $\tau_1$) and $(t_{i+1}, s_{i+1}) \in R$. From (F1) and (F2), for every $i < \text{len}(\tau_2)$ we conclude by induction that the prefix of $\tau_1$ of length $i + 1$ is well defined, $\text{labels}(\tau_1[i]) = \text{labels}(\tau_2[i])$ and $(t_i, s_i) \in R$. Thus, $\tau_2 \in R[\tau_1]$, and by construction, $\tau_1$ is a trace of $T_1$.

The proof for unlabeled systems is analogous. \[\square\]
2.3. Simulations Between Transition Systems

However, forward simulations, while sound, are not complete, in the sense that $T_2 \sqsubseteq^R T_1 \not\Rightarrow T_2 \sqsubseteq^R T_1$, even when we replace $\sqsubseteq^R$ by $\sqsubseteq^R_*$. Figure 2.1 shows an example with two unlabeled systems, adapted from Lynch and Vaandrager [LV95]. $T_1$ and $T_2$ have a single initial state each (the leftmost state in the graph), and only two (finite) traces corresponding to the paths from the final states. Clearly, $T_2 \sqsubseteq^R T_1$, but $T_2 \not\sqsubseteq^R_* T_1$. The incompleteness arises because $T_1$ makes a choice before the concrete system $T_2$. In Section 6.4.2 we will see a practical example where this incompleteness arises.

Thus, forward simulation is sometimes too weak to prove refinement, and we will need a stronger notion. One such stronger (and in fact, complete for $\sqsubseteq^R_* [LV95]$) notion is backward-forward simulation, which we will use only for labeled transition systems. We describe it next.

### 2.3.2 Backward-forward simulations

An inverse total relation $R$ on $\mathcal{P}(\Sigma_1) \times \Sigma_2$ is a backward-forward simulation from a labeled system $T_2$ to a system $T_1$, written $T_2 \sqsubseteq^R_{BF} T_1$, if:

1. **(BF1)** for all $s_2 \in \Sigma_2^0$ and for all $S_1$ such that $(S_1, s_2) \in R$, $S_1 \cap \Sigma_1^0 \neq \emptyset$

2. **(BF2)** whenever $s'_2 \xrightarrow{l} s_2$ and $(S_1, s_2) \in R$, then there exists a set $S'_1$ such that $(S'_1, s'_2) \in R$ and for every $s'_1 \in S'_1$ there exists an $s_1 \in S_1$ such that $s'_1 \xrightarrow{l} s_1$.

Just as for forward simulation, we would like to prove the soundness of backward-forward simulation for refinement. But there are two complications. First, the relations in the definition of refinement and backward-forward simulation are defined over different ranges. To overcome that, given a relation $R \subseteq \mathcal{P}(X) \times Y$ define $\text{join}(R) \subseteq X \times Y$ to be the relation $\{(x, y) \mid \exists X. (X, y) \in R \land x \in X\}$. The second problem is that, in general, backward-forward simulation is not a sound method for proving refinement of infinite traces. The following definition and Lemma 1 help us state a sufficient condition for making it sound.

---

**Figure 2.1:** An example of forward simulation’s incompleteness. Stated related by $R$ are aligned vertically.
A labeled transition system $T$ is said to have finite non-determinism if its set of initial states is finite, and for all $s$ and $l$, the set of $s'$ such that $s \xrightarrow{l} s'$ is also finite.

**Lemma 1** (A variant of König’s Lemma). Let $G$ be an infinite directed graph that satisfies the following properties.

- $G$ has finitely many roots.
- Each node of $G$ has a finite outdegree.
- Each node of $G$ is reachable from some root of $G$.

Then there is an infinite path in $G$ starting from some root.

**Proof.** The proof appears in [LV95] as the proof of Lemma A.1.

**Theorem 2** (Soundness of backward-forward simulation). If $T_1$ and $T_2$ are labeled transition systems, then:

1. $T_2 \sqsubseteq_{BF} T_1 \implies T_2 \sqsubseteq_{T}^{\text{join}(R)} T_1$.

2. if $T_1$ has finite non-determinism:

   \[ T_2 \sqsubseteq_{BF} T_1 \implies T_2 \sqsubseteq_{T}^{\text{join}(R)} T_1. \]

**Proof.** The proof of item 1 is straightforward, combining the proof of item 1 of Theorem 3.17 in [LV95] with the proof of Theorem 1.

For item 2, we employ the same technique as in item 2 of Theorem 3.17 in [LV95]. Given an infinite trace $\tau_2 \in tr(T_2)$ of the form $s_0, l_0, s_1, \ldots$, define a directed graph $G$ with nodes from $\Sigma_1 \times \mathbb{N}$, such that:

- $(s', i)$ is a node of $G$ if:
  
  1. $(s', s_i) \in \text{join}(R)$
  2. $s'$ is reachable by $\lambda$ in $T_1$, where $\lambda$ is the sequence $l_0, l_1, \ldots l_{i-1}$.

- an edge between $(s', i)$ and $(s'', j)$ exists in $G$ if and only if $j = i + 1$, and $s' \xrightarrow{l_{i-1}} s''$.

We first note that, since the set of traces of a transition system (and in particular $T_2$) is prefix-closed, from item 1 we have that $G$ has at least one node for every $i$, and thus an infinite total number of nodes. Next, finite non-determinism of $T_1$ ensures that $G$ has only a finite number of source nodes, and that each node has a finite outdegree. Finally, the construction of $G$ also guarantees that each node of $G$ is reachable from some root of $G$. Thus, the conditions of Lemma 1 are met, and we conclude that there exists an infinite path through the graph $G$. From the construction of $G$, we conclude that the path corresponds to the required infinite trace in $tr(T_1)$.
2.4 Modeling Computational Effects with Monads

Part I of the thesis describes the verification of an imperative program in Isabelle/HOL. Since the latter is (or can be viewed) as a pure functional language, we need a way of modeling imperative features (termed effects) in a functional manner. We choose to do so using monads. The essence of this approach was already used by Landin as early as 1965 [Lan65], and was generalized by Moggi [Mog91]. We limit our description to a combination of three effects that we use: stateful computations, non-determinism, and exceptions. While the combined effect can be described in terms of its constituents in a principled manner, using monad transformers [LHJ95], for the sake of simplicity we give direct definitions.

To model stateful computations, that is, computations that use a mutable state (store), we use state monads. A state monad on a state of type $\sigma$, denoted by $s$-monad$(\sigma)$, is a function of type $\sigma \rightarrow (\alpha \times \sigma)$, for some type $\alpha$. Thus, starting from some state of the system, a state monad produces a value and a new state, which can be viewed as an update of the old state. We will write $s$-monad$(\sigma)(\alpha)$ for a state monad with a value type $\alpha$. For non-deterministic stateful computations, we use non-deterministic state monads instead. These produce a set of possible outcomes instead of a single output; their type is then $nds$-monad$(\sigma)(\alpha) = \sigma \rightarrow \mathcal{P}(\alpha \times \sigma)$. Finally, for non-deterministic stateful computations that may fail, we resort to non-deterministic state-exception monads. In case of a failure, these produce an exception value of some fixed exception type $\epsilon$ instead of values of type $\alpha$. The resulting type is then $ndse$-monad$(\sigma,\epsilon)(\alpha) = \sigma \rightarrow \mathcal{P}((\epsilon + \alpha) \times \sigma)$. Notice that in this definition states are still included in the result even in case of a failure.

2.4.1 Basic monadic operations

Crucially, for any monad, we need to define a way to compose two computations. A standard way of doing so is by defining a bind function for the monad. Given a monad $\mu$, bind has the type $\mu(\alpha) \rightarrow (\alpha \rightarrow \mu(\beta)) \rightarrow \mu(\beta)$. We write $m \gg f$ for bind $m f$. For $s$-monad, we define:

$m \gg_S f \triangleq \lambda s. \text{let} \ (v, s') = m \ s \ \text{in} \ f \ v \ s'$

where we use the standard functional programming notation, using currying and not using parenthesis around function arguments. For $nds$-monad, we define

$m \gg_N f \triangleq \lambda s. \text{let} \ S = m \ s \ \text{in} \ \bigcup \ f \ v \ s' \ \text{where} \ (v, s') \in S$

We write $m \gg f$ for bind $m f$. For $s$-monad, we define:

$m \gg_S f \triangleq \lambda s. \text{let} \ (v, s') = m \ s \ \text{in} \ f \ v \ s'$

where we use the standard functional programming notation, using currying and not using parenthesis around function arguments. For $nds$-monad, we define

$m \gg_N f \triangleq \lambda s. \text{let} \ S = m \ s \ \text{in} \ \bigcup \ f \ v \ s' \ \text{where} \ (v, s') \in S$
Finally, for ndse-monad, we additionally define a function \( \text{lift}_E \), whose type is \( (\alpha \to \beta) \to (\epsilon + \alpha \to \epsilon + \beta) \):

\[
\text{lift}_E f \ ev \triangleq \text{case } ev \text{ of } \iota_1 e \Rightarrow \iota_1 e \mid \iota_2 v \Rightarrow \iota_2 (f \ v)
\]

The choice of \( \text{bind} \) as the composition operation allows us to use Haskell-like syntax sugar for monads in the form of do-notation. We write:

\[
\text{do } \{ \ x \leftarrow m; \ f \ x \ \}
\]

for \( m \gg f \). Moreover, given two monadic values \( m_1 \in \mu(\alpha) \) and \( m_2 \in \mu(\beta) \), we write

\[
\text{do } \{ \ m_1; \ m_2 \ \}
\]

for \( m_1 \gg \lambda_. \ m_2 \). We also write just \( m_1; m_2 \), dropping the do part.

Furthermore, for any monad, we also have to define an injection function of type \( \alpha \to \mu(\alpha) \). For compatibility with the naming used in Haskell, we use the name \( \text{return} \) for this injection. For the s-monad, we define \( \text{return} x \triangleq \lambda s. (x, s) \). For the nds-monad, we define \( \text{return}_N x \triangleq \lambda s. \{(x, s)\} \). For ndse-monad, we define \( \text{return}_{NE} x \triangleq \lambda s. \{\iota_2 x, s\} \).

For any monad \( \mu \), the functions \( \gg \) and \( \text{return} \) must respect three laws:

\[
\begin{align*}
(m \gg \text{return}) &= m \\
(\text{return} \ x \gg f) &= f \ x \\
((m \gg f) \gg g) &= (m \gg (\lambda x. f \ x \gg g))
\end{align*}
\]

for any \( m \in \mu(\alpha) \), \( x \in \alpha \), \( f : \alpha \to \mu(\beta) \), and \( g : \beta \to \mu(\gamma) \), and for any types \( \alpha \), \( \beta \) and \( \gamma \). It is easy to check that these laws hold for all our monads.

For the non-deterministic state and state-exception monads, the non-deterministic choice operator \([+]\) combines the results of two monadic computations: \( m_1 [+] m_2 \triangleq \lambda s. (m_1 \ s) \cup (m_2 \ s) \). For the non-deterministic state-exception monad, we define the standard try/catch construct, with:

\[
\text{try } m \ \text{catch } h \triangleq m \gg_N (\lambda v. \text{case } v \text{ of } \iota_1 e \Rightarrow h \ e \mid \iota_2 v \Rightarrow \text{return}_{NE} v)
\]

### 2.4.2 Hoare logic

We express properties of stateful monadic computations using pre- and post-conditions, written as Hoare triples. While such triples can be defined on
any of the monads we have described, in this thesis we only use and define
them on the non-deterministic state monad:

\[ \{ P \} m \{ Q \} \triangleq \forall s \in P. \forall (v, s') \in m \quad s' \in Q \quad v \]

Note that, unlike the standard post-conditions for Hoare triples on im-
perative programs, the post-conditions also depend on the computed values.

### 2.5 Simulations Between Monadic Computations

Just like in the case of transition systems, we will often want to relate
two monadic computations. For this we resort to relational Hoare tuples
\[ \{ R \} m_a m_c \{ S \} \], defined as:

\[ \{ R \} m_a m_c \{ S \} \triangleq \forall s_a, s_c, v_c, s'_c. \quad (s_a, s_c) \in R \land (v_c, s'_c) \in m_c \quad s_c \Rightarrow \]

\[ (\exists v_a s'_a, (v_a, s'_a) \in m_a \quad s_a \land (s'_a, s'_c) \in S \quad v_a \quad v_c) \]

where \( m_a \in \text{nds-monad}(\sigma_a)(\alpha) \), \( m_c \in \text{nds-monad}(\sigma_c)(\beta) \), \( R \subseteq \sigma_a \times \sigma_c \), and
\( S : \alpha \rightarrow \beta \rightarrow \mathcal{P}(\sigma_a \times \sigma_c) \). That is, for any pair \((s_a, s_c)\) of states related by \( R \),
any result that \( m_c \) produces when run on \( s_c \) must be related by \( S \) to some
result that \( m_a \) produces when run on \( s_a \).

Note that by choosing \( S \) to be \( R \) (more precisely, \( \lambda \_ \_ . \ R \)), we nearly
recover the condition (F2) of forward simulation on transition systems (ig-
noring the return values). Indeed, such a relational Hoare tuple on two
computations can conceptually be seen as a forward simulation on the tran-
sition systems arising from the operational semantics of the computations.
However, in the monadic case, we are typically only interested in relating
the initial and final states, and relating the intermediate states is just an
auxiliary step. Hence, when \( m_a \) and \( m_c \) are intermediate computations, \( S \) is
in general independent of \( R \).

Similar definitions of Hoare tuples appear in [Ben04, SB07, CKS08]; we
briefly point out the differences. In [Ben04], the computations are determin-
istic, and written in an imperative language with no return values (they only
affect the state). In [SB07], the computations are deterministic. In [CKS08],
special provisions are made for a halt operation that stops the system – an
operation that does not exist in our models.

**Preservation of Hoare triples.** Just as the transition system simula-
tions, relational Hoare triples are used to prove properties on abstract mod-
els and transfer them to the related concrete models. The following proof
rule is the Hoare tuple analogue of the simulation soundness theorems:

\[
\frac{\{ R \} m_a m_c \{ S \} \quad \{ P \} m_a \{ Q \}}{\{ R[P] \} m_c \{ \lambda v_c. \bigcup_{v_a} (S \ v_a \ v_c)[Q \ v_a] \} \quad \text{RH-TRANSFER}}
\]
2. Preliminaries

The presence of return values in the computations’ results complicates the rules somewhat, compared to the transition systems case. Luckily, the post-relation $S$ is often in a form that simplifies this rule. Namely, we often require that the abstract and concrete operations compute the same value. Given a relation $R$, we then define $(s, t) \in eq(R)(u, v)$ iff $u = v$ and $(s, t) \in R$. When $S = eq(R')$, the above rule then simplifies to:

$$\frac{\{R\} \, m_a \, m_c \, \{eq(R')\} \quad \{P\} \, m_a \, \{Q\}}{\{R[P]\} \, m_c \, \{\lambda v_c. \, R'(Q \, v_c)\}} \text{ RH-TRANSFER=}$$

Compositional reasoning. In the previous section we listed several basic constructs for composing monadic computations. An important property of the relational Hoare logic is that the proof obligations decompose over these constructs. For example, we prove the rule:

$$\frac{\{R\} \, m_a \, m_c \, \{S'\} \quad \forall v_a, v_c. \, \{S' \, v_a \, v_c\} \, (f_a \, v_a) \, (f_c \, v_c) \, \{S\}}{\{R\} \, (m_a \, \gg_N \, f_a) \, (m_c \, \gg_N \, f_c) \, \{S\}} \quad \gg_N -\text{SPLIT}$$

which allows us to relate two compound computations, with the topmost connective $\gg_N$, by relating their components. Similar rules exist for the other basic constructs, such as return, try/catch, and $[+]$. 

Stepwise refinement. Just as for transition systems, we can also perform the refinement of monadic computations stepwise. This is captured by the following proof rule:

$$\frac{\{R\} \, m_a \, m_b \, \{S\} \quad \{R'\} \, m_b \, m_c \, \{S'\}}{\{R \, \gg_N \, R'\} \, m_a \, m_c \, \{\lambda v_a, v_c. \, \bigcup_{v_b} (S \, v_a \, v_b \, \gg_N \, S' \, v_b \, v_c)\}} \text{ RH-COMPOSE}$$

This rule too can be simplified for post-relations that require abstract and concrete computations to compute the same values:

$$\frac{\{R\} \, m_a \, m_b \, \{eq(S)\} \quad \{R'\} \, m_b \, m_c \, \{eq(S')\}}{\{R \, \gg_N \, R'\} \, m_a \, m_c \, \{eq(S \, \gg_N \, S')\}} \text{ RH-COMPOSE=}$$
Part I

Handling Faults Within a Node
Chapter 3

Verification of a Persistent Memory Manager Under Crashes and Hardware Faults

In the first part of the thesis, we focus on the persistent storage of a single computer system. More precisely, we investigate how to verifiably ensure the consistency of the storage state in the presence of crashes and hardware faults. To ensure practical applicability of our results, we use the persistent memory manager of IBM’s 4765 secure coprocessor as our driving case study.

We describe the setting and our approach in Section 3.1. Then, we give overviews of the system architecture and requirements (Section 3.2), our verification framework (Section 3.3), and our system models (Section 3.4). We discuss our results in Section 3.5. Finally, Section 3.6 provides an overview of the related work.

3.1 The Setting and Our Approach

The IBM 4765 [ABC+12] cryptographic coprocessor resembles a general-purpose computer, encased in a tamper-proof housing and packed onto a PCIe card. Its security policies require that most access to the persistent storage be brokered through the built-in bootloader, and in particular its subsystem called the Persistent Memory Manager (PMM).

The PMM’s API offers a rudimentary persistent storage service. It abstracts the persistent memory into an arbitrary, but fixed number of storage slots of different capacities. The slots are called regions, and they are addressed by their indices. The API provides just two operations: update and fetch. The main requirement for this API are transactional, or atomic updates: given new contents for a set of regions, an update operation updates either all or none of them.

The API does not support concurrency. Hence, designing and verifying
such a system appears to be easy at first. However, we will require atomicity to hold even in the presence of:

1. system crashes (e.g., due to power-downs), possibly resulting in garbled writes. At power-up a startup procedure is called (that might itself be subject to abrupt restarts).

2. failures of persistent storage, such as spontaneous corruption ("data rot") or permanent hardware failures.

Algorithms that provide atomic updates in the presence of restarts have already been analyzed in the literature [SL00, HBdJL01, PHdJ02], but this is the first work we are aware of that also addresses storage failures. Moreover, our target system does not just detect such failures, but also aims for resilience against them, restoring corrupted data from spare copies when possible. This necessitates full redundancy in both user and metadata (i.e., administrative data used by the algorithm) stored in the persistent memory. It also complicates the details of the algorithm, requiring nested transactions in the metadata and permeating the implementation with special cases, integrity checks, and potential recovery actions. An example of the resulting implementation complexity is the seemingly innocuous fetch procedure, which simply retrieves the contents of a single region. Figure 3.1 shows its call graph. This implementation complexity is then amplified by restarts. Longer implementations induce new restart points, which in turn increases the complexity of the startup procedure. This procedure must itself be tolerant to crashes and storage failures, further increasing its complexity and length; thus, restarts and storage failures form a positive feedback loop with respect to the implementation complexity, which then translates into the reasoning complexity.

To tame this complexity and enable verification, we proceed using abstraction (or dually, refinement), building a stack of progressively more abstract models. We prove a refinement rule that enables our abstractions to take advantage of the layered logical structure of the system, where subsystems ensure atomic access to the data and the metadata. We also prove a rule that leverages the idempotence of the startup procedure to reduce the reasoning about an arbitrary number of restarts to reasoning about a single restart. Our abstractions then use these rules to gradually remove redundancy, first in the metadata, then in the user data, and finally replace repeated restarts by a single one. Combining proofs of refinement between neighboring models with the general property preservation results of Section 2.3, we then transfer proofs of requirement compliance from the top of the model stack to the concrete model of the PMM on its bottom. We have formalized our entire development [Mar16] in the Isabelle/HOL theorem prover [NPW02].
3.2 System Overview

The task of the PMM is to provide a simple API for transactional access to persistent memory, effectively resulting in an abstraction of the memory as a function $index \rightarrow contents$. To achieve this task, the PMM in turn relies on an API called the lower-level interface (LLI) for accessing the persistent storage. The PMM (sub)system consists of three main procedures: update, fetch, and startup. The first two constitute the PMM API. The fetch procedure takes a single parameter, the index of the target region, and is supposed to return the corresponding contents. The update procedure also takes a single parameter, a map (partial function) $index \rightarrow contents$, and is supposed to override the memory abstraction with the given map, updating all the regions in the map’s domain with the given contents. However, the behavior of the API is also conditioned on possible abrupt crashes and subsequent restarts, and hardware failures.

The restarts cause the startup procedure to be run, which then performs cleanups and integrity checks. An API call to update or fetch may thus result in one or more (if startup is itself restarted) executions of startup. The same procedure is also executed in case a restart happens in between API calls.

The hardware failures come in several flavors, which we describe shortly. At the moment, we focus on their global effect on the card, reflected in the three PMM modes of operation: Normal, Degraded, and Fail, corresponding to normal operation, read-only mode, and complete failure. To achieve resilience, the system stores all its data in two copies. This includes the data of the user regions, exposed by the API, but also the metadata stored in the extra administrative regions. Each copy of a region is called a region instance. Figure 3.2 gives an overview of the system’s abstraction levels.

To signal irremediable hardware failures to the caller, the API calls use an exceptional mode of termination. To facilitate modeling, we will also use the exceptional mode to signal the completion of the startup procedure. This simplifies the typing of our operations, as restarts lose the information about the original call and its return value. If an API call terminates in the normal (non-exceptional) mode, we expect it to behave as described at the beginning of the section. In the exceptional mode, however, we will have to
3. Verification of a Persistent Memory Manager Under Crashes and Hardware Faults

loosen the requirements. We will make this more precise in Section 3.2.2.

3.2.1 The environment

Next, we present the environment that the PMM interacts with, and our assumptions about it. The PMM controls the persistent storage, which consists of battery-backed RAM and flash memory. The LLI abstracts the memory into logical memory blocks of varying sizes. It can read and write each block independently (regardless of the type of the underlying memory), by transferring data between the persistent memory and the DRAM (dynamic RAM). We assume both the DRAM and the CPU to be reliable.

The PMM maps each region instance to a unique memory block. The two blocks corresponding to the two instances of the same region have equal capacity. The LLI provides a convenient addressing scheme for mapping instances to blocks, but is otherwise oblivious of the connection between blocks and regions. Its task consists, first and foremost, of mapping the logical addresses onto the appropriate hardware ones, and performing blockwise read and write operations.

Additionally, the LLI tries to eliminate transient failures (e.g., bus interconnect problems) by repeating its reads, and checking the success of each write. It also detects and reports two kinds of permanent (irrecoverable) failures, namely:

- Read failures, where a block becomes completely unusable (e.g., due to a dead memory bank). We call such a block dead.

- Write failures, where a block can no longer be overwritten with new contents. We call such a block degraded.

A block without permanent failures is called ok. Other failures are undetectable by the LLI and the PMM must detect and try to correct them. An example is “data rot”, where some content can be retrieved from an instance, but it differs from the content that was last written to it. These failures are recoverable, as the block can still be overwritten with the correct contents, if they are available.

The environment can also trigger restarts, whereby control is transferred to the startup procedure. Restarts may interrupt write operations, causing another (recoverable) kind of write failure. We will show how we model hardware failures and restarts in Section 3.3.1.

3.2.2 The requirements

We specify the requirements on the PMM in terms of the abstract view on the memory it provides to API users. We express this view as elements of the type \((index \rightarrow contents) \perp\), where \(\perp\) corresponds to a failure, that is, an irrecoverably inconsistent state. The requirements concern entire API calls,
including the possible runs of the startup procedure. We call a user region instance active if it matches the view’s content.

(R1) Atomic updates. Given the current view $v$ and an update map $u$ of type $\text{index} \rightarrow \text{contents}$, an update results in either the view $v \triangleleft u$ (successful update, where $\triangleleft$ is as in Section 2.1), $v$ (rollback), or $\perp$ (failure, also if $v = \perp$). A rollback may only be performed in case of exceptional termination.

(R2) Correctness of fetch. Fetch returns the value of the view at the given index, or results in an exception in Fail mode or when interrupted by a restart.

(R3) Unchanged view during fetches, updates in non-Normal mode, and restarts in between API calls, except for when the mode is changed to Fail.

(R4) Matching modes of operation and termination. API calls can terminate exceptionally only in the case of restarts or non-Normal mode of operation.

(R5) Correctness of the mode of operation. In Normal mode, all region instances are active. In Degraded mode, each region has at least one active instance and there is at least one degraded block. In Fail mode, there exists a region with no reliable and up-to-date instances.

(R6) Maximum redundancy. In all but Fail mode, any 'ok' region instance is active (and hence all 'ok' instances of a region match).

3.2.3 The implementation

We now give a high-level overview of the PMM implementation. It relies on the LLI interface described in Section 3.2.1, and uses two administrative regions to store metadata. As in the case of user regions, the two administrative regions have two instances each. The first region is used for checking data integrity. It stores the checksums of all logical blocks, including its own instances. These checksums are realized with hash functions, and the region is thus named the hash region. The second region is the pending-transactions register (PTR), used by the startup procedure to “break ties” between instances of user regions, as will be explained shortly.

The centerpiece of the system is the update algorithm. Figure 3.3 sketches a sample execution, where we write new contents to regions 0 and 2. Each image shows the state of the memory at a different update step. We divide the process into three stages: pre-commit, commit, and roll-forward.

As mentioned earlier, each region has two instances, which we refer to as primary and secondary instances. In the pre-commit stage, the new content
3. Verification of a Persistent Memory Manager Under Crashes and Hardware Faults

![Diagram showing state sequence of PMM update operation]

Figure 3.3: The state sequence of a sample PMM update operation, which updates regions 0 and 2

is sequentially written to the secondary instances of each target region. Then, during the commit stage, we record the set of updated regions’ indices (the domain of our update map) in the PTR. This will give precedence to the corresponding secondary instances in the startup procedure, in case the system is abruptly restarted. The final stage is the roll-forward stage, where we iteratively synchronize the two instances of every freshly-written region, by successively overwriting the contents of the primary instance with the contents of the secondary instance. In each iteration, once the instances are synchronized, we remove the region index from the PTR.

Missing from the diagram are updates to the hash region. Every single step shown actually entails three block writes. First, we write the new content to the target instance, and then update the two corresponding hashes, first in the secondary hash region instance, and then in the primary one.

Also missing is the treatment of restarts and hardware failures. These complicate matters greatly, as a number of special cases arise, especially if the failures occur in instances of the administrative regions. As an example, we will describe the handling of restarts here, giving a short account of the startup procedure.

This procedure brings the system into a maximally redundant state. It first synchronizes the instances of the hash region. Both directions of synchronization are possible, depending on the exact scenario. They correspond to a “mini” roll-back or roll-forward, and result in either a failure or a single value in both instances. We can thus view writes to the hash region as implicit nested transactions within our system. Synchronization is then also performed in the PTR, forming another layer of transactions on top (since writes to the PTR also involve writes to the hash region). At this point we have unambiguous metadata. The procedure next iterates through all of the user regions, again performing checks and synchronizations as necessary. To determine the direction of the synchronization, it needs to figure out which of the two instances is current. The criterion is as follows: if the hashes of both instances match, both are current. Otherwise, we examine the PTR. If it contains the region’s index, only the secondary instance is current, otherwise only the primary one is current. If the synchronization completes successfully, the index is removed from the PTR.

The global effect of a restart on an update thus depends on the stage
where it occurs: during pre-commit, the state is rolled back; during roll-forward, the update is applied; and during commit, either is possible, as the nested transactions (to the PTR and the hash region) can still be rolled either back or forward.

3.3 The Specification and Verification Framework

We embedded a framework for modeling and reasoning about imperative programs with restarts and failures in the theorem prover Isabelle/HOL [NPW02]. Similar to Klein et al. [KEH+09], we build a series of models at different levels of abstraction, with each model having two layers: an outer layer based on transition systems and an inner layer that uses monads to provide a structured input language.

All of the work is done in the inner layer, where we model the API and startup procedures in an imperative fashion. The structured nature of the language makes this modeling straightforward. This layer also provides facilities for modeling hardware failures and restarts, including repeated restarts. The last point allows us to model entire API calls within the inner layer. We note that our treatment of both hardware failures and restarts is possibilistic, since our requirements do not include probabilistic properties. The outer layer is a simple shell around the inner one, with the purpose of providing an (unlabeled) trace semantics as a record of the interaction with the API's users. Its transitions are derived directly from the definitions in the inner layer, as the union over all API calls and over restarts in idle states. Given this trace semantics, we use the refinement infrastructure of Sections 2.3 and 2.5 to relate the different models. We transfer the refinement proof obligations from the outer to the inner layer, whose structured nature allows us to prove the obligations in a compositional manner. As described in Sections 2.2.3 and 2.5, the refinements guarantee that the concrete models inherit the properties expressing our requirements, which we prove on the simpler abstract models.

3.3.1 Modeling hardware failures and restarts (inner layer)

We start our description with the inner layer. The salient features of the system we wish to model are the imperative nature of the target algorithm and the non-determinism in the environment stemming from hardware failures and abrupt restarts. Our modeling of these features in HOL’s functional language is based on a non-deterministic state monad \( \text{nds-monad}(\alpha, \sigma) \) described in Section 2.4. We call the monad’s elements \( \text{computations} \). We use a function \( \text{to-rel} : \text{nds-monad}(\alpha, \sigma) \to \mathcal{P}(\sigma \times \sigma) \) to derive outer-layer transition relations from given computations by simply forgetting the return values.

Hardware failures can, in reality, happen asynchronously, at any time. However, the PMM can only observe them through the LLI. We thus model
them as happening synchronously (and non-deterministically), upon calls to
the LLI. Restarts are also asynchronous in reality. They transfer control to
the startup procedure. However, it is impossible to model the exact start and
des of this transfer as well as the precise system state handed to the
startup procedure, without getting into electrical properties of circuits. All
practical models of restarts are thus necessarily approximations – they must
choose a granularity and approximate the effect on the state. Existing struc-
tured models (such as [And06, PHdJ02]) choose the granularity of a language
statement, inserting non-deterministic restarts between statements. Fortu-
nately, one observation allows us to enlarge this granularity and simplify
our model: the persistent memory is accessed only during LLI calls. Hence,
restarts outside of LLI calls can only affect the volatile memory, and their
effects can be (over)approximated by inserting restarts only right before and
after LLI calls, and allowing them to arbitrarily modify the volatile memory.
The effects of restarts during LLI calls are call-specific (e.g., setting a block’s
contents to an arbitrary value during a write). We thus model all restarts
as synchronous, by putting them in and around LLI calls.

Since the transfer of control that a restart triggers can happen at ar-
bitrarily deep levels of the call stack, we chose to model them as excep-
tions. As we also use exceptions for error handling, we model restarts
with a distinguished exception, in order to avoid accidentally capturing
restarts by the error handlers in the code. We thus transform the non-
deterministic state monad into a PMM restart-exception monad, defined
as \( \text{pre-monad}(\alpha, \epsilon, \sigma_v, \sigma_p) \triangleq \text{nds-monad}(1 + \epsilon + \alpha, \sigma_v \times \sigma_p) \). Here, \( \alpha \) rep-
resents (normal) return values, \( \epsilon \) represents (regular) exceptions, and the
unit type \( 1 \) represents restart exception. Moreover, the state is partitioned
into the volatile \( (\sigma_v) \) and persistent \( (\sigma_p) \) components. It will often be use-
ful to distinguish between the restarting (R) and non-restarting (N) re-
results of the computation, so we define two projections of a computation
\( m : \text{pre-monad}(\alpha, \epsilon, \sigma_v, \sigma_p) \), namely \( m \downharpoonright_R \) of type \( \text{nds-monad}(1, \sigma_v \times \sigma_p) \) and
\( m \downharpoonright_N \) of type \( \text{nds-monad}(\epsilon + \alpha, \sigma_v \times \sigma_p) \). Formally, recalling that \( \iota_1 \) and \( \iota_2 \)
are the two injections into the sum type, we have:

\[
\begin{align*}
    m \downharpoonright_R & \triangleq \lambda s. \{(*, s') \mid (\iota_1, *, s') \in m \, s\} \\
    m \downharpoonright_N & \triangleq \lambda s. \{(ev, s') \mid (\iota_2, ev, s') \in m \, s\}
\end{align*}
\]

We lift \text{bind} and \text{return} as expected and also redefine the \text{try/catch} con-
struct to handle only regular exceptions.

We also define a \text{tryR/catchR} construct to handle the restart exception.
Here, the “handler” is normally the \text{startup} procedure. However, this con-
struct does not suffice to accurately model the possibility of \text{startup} being
itself interrupted by a restart. Hence, we need a construct for repeated
restarts. Given a handler \( h \), we denote this construct by \text{recTryR}(h) \) and
define it inductively, by preceding a single run of \( h \downharpoonright_R \) by zero or more runs
3.3. The Specification and Verification Framework

of \( h \parallel_R \) and lifting the resulting computation back to the \textbf{pre-monad}. We then define

\[
\text{tryR } m \ \text{catchR}^* \ h \equiv \text{tryR } m \ \text{catchR} \ \text{recTryR}(h)
\]

which accounts for an arbitrary number of restarts in both the computation \( m \) and the handler \( h \).

3.3.2 Specifications and refinement (outer layer)

The outer layer consists of an unlabeled transition system, whose state space is the same as in the inner layer. Its transitions correspond to API calls and restarts in an idle state, both of which include the possibility of (arbitrarily often repeated) restarts during the execution of the call and the \texttt{startup} procedure. The return values of the API calls are thrown away. For instance, let \texttt{fetchAPI ind} be \( \text{tryR fetch ind catchR}^* \texttt{startup} \). Then, the part of the transition relation that corresponds to the \texttt{fetch} API call consists of all state pairs \((s, s')\) such that, for some \( v \) and \( \text{ind} \):

\[
(v, s') \in \texttt{fetchAPI ind } s
\]

The outer layer enables us to specify the system’s intended invariants (such as the requirement (R5)) in a simple fashion. Due to the simplicity of the outer layer, all proof obligations for this layer can be straightforwardly reduced to proof obligations for the inner layer. We next discuss these proof obligations and the reduction in more detail.

3.3.3 Properties and their preservation

We formalize the system requirements either as outer-layer invariants or as Hoare triples of Section 2.4.2 in the inner layer. In both cases, we prove the requirements stepwise, building a sequence of models. For any two neighboring models in the sequence, we first relate their inner-layer API calls using the relational Hoare tuples of Section 2.5. The tuples always require that the computations’ results be equal. For instance, for the \( i \)-th and \((i + 1)\)-th models of the sequence we prove:

\[
\{ R_{i+1} \} \ (\texttt{fetchAPI}_{i+1} \ ind) \downarrow_N (\texttt{fetchAPI}_i \ ind) \downarrow_N \{ eq(R_{i+1}) \}^N.
\]

where \( R_{i+1} \) is the appropriate simulation relation, and we use subscripts to distinguish between the implementations of \texttt{fetchAPI} in the two models. Notice that we only consider the non-restarting parts of computation here, as the user can only interact with the systems once it stops restarting. After proving such tuples for all API calls, the proof obligation (F2) of forward simulation for the outer layers follows immediately. Hence, we can prove the requirements on the most abstract (and thus simplest) model possible.
3. Verification of a Persistent Memory Manager Under Crashes and Hardware Faults

and transfer them onto the concrete model using the preservation results of Sections 2.3 and 2.5. We will give an example in Section 3.5.1.

Two implicit system requirements are termination and deadlock freedom, where the latter means that no branch of a (non-deterministic) computation yields an empty set of results. We do not prove them formally, but we still informally argue that these properties hold for our system. The primitives used in our concrete model neither deadlock nor use non-terminating constructs, and their compositions are well-defined since HOL is a logic of total functions.

3.3.4 Compositional reasoning

As mentioned above, there are two kinds of properties we wish to prove of our monadic computations. First, we want to establish properties of individual computations. We express these using Hoare triples of Section 2.4.2. Given an \( m : \text{pre-monad}(\alpha, \epsilon, \sigma_v, \sigma_p) \), we sometimes only care about the non-restarting results of the computation; then, we use the following variant:

\[
\{ P \} m \{ Q \}_N = \{ P \} m \downarrow_N \{ Q \}.
\]

Second, we want to relate two implementations of the same API call at different levels of abstraction. For this, we resort to the relational Hoare tuples of Section 2.5. Same as for triples, we define

\[
\{ R \} m_a m_c \{ S \}_N = \{ R \} (m_a \downarrow_N) (m_c \downarrow_N) \{ S \}.
\]

In Section 2.5 we already showed rules that enable compositional reasoning about relational Hoare tuples. For instance, if both \( m_a \) and \( m_c \) are sequential compositions, the rule \( \gg=_{N} \text{-SPLIT} \) decomposes the Hoare tuple into two, one for each component computation. These decomposition rules are applicable if the two related implementations share the same structure. Usually, we apply them for as long as possible, until we are left with proving Hoare tuples between pairs of “small” monadic operations. At this point, the proof obligations usually become simple enough to discharge them by unfolding the relevant definitions and using Isabelle’s proof automation. This decomposition strategy might fail, however, either because the two implementations have different structures, or because the rules yield unprovable goals. For these cases we have to derive two important novel rules, which we present next.

The first one relates a restart handler \( m \) with its repeated version realized as \( \text{recTryR}(m) \). This rule is typically used with the \text{startup} procedure, which checks the system state and repairs inconsistencies; if the procedure is itself restarted, we would intuitively expect it to pick up where it left off (at least when viewed abstractly enough). That is, a restarting run of the procedure, followed by a non-restarting run does not yield more results than
just a single, non-restarting run. This property can be considered as a form of idempotence and is captured in the premise of the following inductively justified proof rule:

\[
\begin{align*}
\text{\{}\text{Id}\text{\}} (m \downarrow_N) (m \downarrow_R; m \downarrow_N) \text{\{}eq(\text{Id})\text{\}} N \\
\text{\{}\text{Id}\text{\}} m \ \text{recTryR}(m) \text{\{}eq(\text{Id})\text{\}} N \ \text{IDEM}
\end{align*}
\]

where the composition \(m_1; m_2\) and \(eq(\cdot)\) are as in Sections 2.4 and 2.5 respectively, and \(Id\) is the identity relation. The conclusion states that \(m\) itself retains all the possible non-restarting behaviors of \(\text{recTryR}(m)\). At a high enough abstraction level, \(\text{startup}\) becomes simple enough to prove the rule’s premise directly by unfolding the definitions.

The other important proof rule allows us to exploit the logical layering present in our target system. We use it to abstract the non-atomic persistent memory operations of a (logical) subsystem in the concrete model by atomic operations in the abstract model. For example, our implementation defines a \(\text{write_hashes}\) operation that writes the given new value to both hash region instances sequentially. The next abstraction in the abstraction stack collapses the two hash region instances into a single instance, and the \(\text{write_hashes}\) operation becomes atomic. Due to this difference in granularity, the abstract \(\text{write_hashes}\) has fewer restart points than the concrete one, which causes the standard decomposition rule for \(\text{tryR}/\text{catchR}\) to fail to prove the goal:

\[
\begin{align*}
\text{\{}R\text{\}} \\
\text{(tryR write_hashes}_a \ \text{catchR}^* \ \text{startup}_a) \ \\
\text{(tryR write_hashes}_c \ \text{catchR}^* \ \text{startup}_c) \ \\
\text{\{}eq(R)\text{\}} N
\end{align*}
\]

The cause of the failure is that the concrete computation can be interrupted in an intermediate state, where the write can still be either completed or rolled back. The abstract computation, however, must make this choice before the restart happens. This is the same problem that makes forward simulation incomplete in the transition system setting, as discussed in Section 2.3. We overcome this problem by leveraging the computations’ structured nature to formulate the following rule:

\[
\begin{align*}
\text{\{}R\text{\}} (m_a \downarrow_R; \text{recTryR}(h^1_a)) (m_c \downarrow_R; \text{recTryR}(h^1_c)) \text{\{}eq(T)\text{\}} N \\
\text{\{}R\text{\}} m_a \ m_c \ \text{\{}eq(S)\text{\}} N \\
\text{\{}T\text{\}} (\text{tryR } h^2_a \ \text{catchR}^* (h^1_a; h^2_a)) (\text{tryR } h^2_c \ \text{catchR}^* (h^1_c; h^2_c)) \text{\{}eq(S)\text{\}} N \ \text{GRAN} \\
\text{\{}R\text{\}} (\text{tryR } m_a \ \text{catchR}^* (h^1_a; h^2_a)) (\text{tryR } m_c \ \text{catchR}^* (h^1_c; h^2_c)) \text{\{}eq(S)\text{\}} N
\end{align*}
\]

The rule assumes that the abstract and concrete startup procedures are both sequential compositions, of the form \(h^1; h^2\). For example, the startup
procedure of the PMM implementation (Section 3.2.3) is decomposed such that $h_1$ is the `check_hashes` procedure that synchronizes the hash instances. The key assumption is that in case of a restart, $h_1^c$ completes the (non-atomic) operation of $m_c$ and matches the behavior of the abstract counterpart. Our first premise captures this requirement. The second premise requires a refinement between non-restarting computations of $m_a$ and $m_c$. The third premise is similar to the conclusion, but concerns $h_2^a$ and $h_2^c$. It connects to the first premise via an intermediate relation $T$. We can prove it either using the standard proof rule for tryR/catchR, or by reapplying the rule $\text{Gran}$ on this premise if $h_2^c$ uses the same non-atomic operation as $m_c$.

This rule works in synergy with the $\text{Idem}$ rule: if we prove the idempotence of $h_1^a$, we can use the compositionality of Hoare tuples (more precisely, the rule $\text{RH-compose}$ of Section 2.5) to simplify the first premise of $\text{Gran}$ by dropping the recTryR constructs. That is, $\text{Gran}$ then simplifies to:

$$
\begin{align*}
\{R\} (m_a \uparrow_R; h_1^a) (m_c \uparrow_R; h_1^c) \& \{eq(T)\} \land \{R\} m_a m_c \{eq(S)\} \land
\{T\} (\text{tryR } h_2^c \text{ catchR}^* (h_1^c; h_2^c)) (\text{tryR } h_2^a \text{ catchR}^* (h_1^a; h_2^a)) \& \{eq(S)\}
\end{align*}
$$

Notice that, unlike with $\text{Gran}$, the first premise of $\text{Gran}^*$ poses no requirements on the restarting behavior of $h_1^a$. This behavior is still constrained by the last premise, but this premise can be further decomposed by either reapplying $\text{Gran}^*$ or applying the standard rule for tryR/catchR. As the restarting behavior of $h_1^a$ is unconstrained, we can remove restarts from $h_1^a$ completely. In fact, since $m_a$ is atomic, $h_1^a$ almost becomes a no-op in our abstract models. In our `write_hashes` example, the abstract version of `check_hashes` is a non-deterministic choice between a skip operation and error operations that degrade or fail the hash region. This simplicity of $h_1^a$ not only simplifies the abstract model, but also facilitates the idempotence proof for the entire abstract restart handler, as no restarts occur in $h_1^a$.

Lastly, we implemented custom Isabelle tactics to automate the simulation proofs between the different models. All our models share the same structure, but with some simple concrete operations replaced by atomic abstract ones. For example, in the abstraction step of replacing the two hash region instances by a single one, we replace the operations `write_hashes` and `read_hashes` by their atomic versions, and the rest of the program stays the same. After we prove (wh-goal) and its `read_hashes` equivalent, our tactics then automatically derive the simulation proof for the entire program, which includes choosing the appropriate proof rules and intermediate simulation relations.
3.4 Models

This section gives an overview of our development. We first present the abstract model, followed by some notes on the concrete model. Then we describe the series of models and refinements connecting these two. Finally, we sketch our formalizations and proofs of the requirements from Section 3.2.2.

3.4.1 The abstract model

This model directly represents the abstract memory view exposed by the API, as introduced in Section 3.2. Recall that our state is split into the volatile and persistent components (Section 3.3.1). In the abstract model, the persistent component is of type \( \text{abs}_\text{mem} \), defined as follows:

\[
\text{record abs_mem =}
\begin{align*}
\text{memory} : \text{index} \to \text{contents} \\
\text{reg_health} : \text{index} \to \text{log-health} \\
\text{global_health} : \text{log-health}
\end{align*}
\]

The memory field models the abstract memory, where where \text{index} is an abstract data type and \text{content} is a type variable. For each region (i.e., index), the \text{reg_health} field tracks its health, which is either ‘ok’, ‘degraded’, or ‘dead’, depending on whether any permanent failures have happened to it. Similarly, \text{global_health} records failures that are not directly related to individual regions. It serves to capture those behaviors of concrete models where card failure or degradation occur during the handling of metadata.

The volatile state component consists of the card’s mode of operation. Based on the two state components, we formalize the abstract memory view as follows. We define the function \( \text{view} : \text{abs_mem} \to (\text{index} \to \text{contents}) \perp \) which maps the persistent state component \( sp \) to the abstract memory view. The \( \text{view} sp \) is \( \perp \) if \text{global_health} or some region are ‘dead’, in which case the card goes into \text{Fail} mode and the memory contents become inaccessible to the user. Otherwise, the \( \text{view} sp \) is defined as \text{memory} \( sp \).

Even at this level of abstraction, the procedures are not entirely trivial, since they need to capture the variety of possibilities present in the concrete models. To get a flavor of what they look like, recall the \text{do} notation for \text{bind} (Section 2.4) and consider the definition of \text{fetch}:

\[
\text{fetch}_6(\text{ind}) \equiv \text{do} \{
\begin{align*}
\text{fail_if_fail_mode;}
(\text{do} \{
\begin{align*}
\text{cnt} \leftarrow \text{read_success ind}; \\
\text{degrade [+]} \text{ skip;}
\text{throw_mode_error (EC cnt);} \\
\text{return cnt}
\end{align*}
\}) [+]
\text{fail [+]} \text{restart_mangle_sp}
\}
\]
3. Verification of a Persistent Memory Manager Under Crashes and Hardware Faults

Here, \texttt{fail_if_fail_mode} ensures that the card is not in \texttt{Fail} mode and throws an exception otherwise; \texttt{read_success} reads the contents of the selected region if possible; \texttt{degrade} and \texttt{fail} respectively degrade and fail one or more regions and set the card mode appropriately; \texttt{throw_mode_error} checks the card mode and potentially throws the appropriate exception, which is value-carrying (constructed using the \texttt{EC} constructor) in \texttt{Degraded} mode; and \texttt{restart_mangle_sp} restarts, non-deterministically lowering the health of zero or more regions.

Still, the definitions are simple enough to keep the proofs performed on this model reasonably easy. This includes the idempotence of the \texttt{startup} procedure, which, by the restart reduction rule \texttt{Idem}, allows us to remove the \texttt{recTryR} construct from the (outer-layer) transitions.

### 3.4.2 The concrete model

The concrete model contains our implementation of the PMM algorithms, which we briefly described in Section 3.2.3. It is based on informal descriptions provided by IBM researchers, and discussions with their PMM developers. Currently, it exists only in terms of Isabelle/HOL definitions. That is, it is neither extracted from a program executable on the coprocessor, nor do we synthesize code for it. We thus verify the PMM algorithms rather than a concrete system. However, our implementation is roughly at the same level of abstraction as what one might see in, e.g., C++ code, in that almost all the statements could be mapped 1-1 to C++ statements (save for unfolding monadic maps and folds and equality checks on lists). It consists of about 700 lines of (Isabelle) code. While we had some freedom in the implementation, the algorithm itself was fixed (and already deployed in the IBM 4765). Its verification is therefore essentially post-hoc, making our task more challenging.

### 3.4.3 Abstractions

We now sketch the refinement between the abstract and the concrete models, explaining the intermediate models and their relations. We build five such models, in a bottom-up fashion. At first, we tried the more conventional top-down approach; but this made finding the right abstractions of our fixed target hard, as the many different failure and restart behaviors would often only creep up low in the stack, breaking the models higher up. Going bottom-up exposed them more quickly, and allowed us to gradually build usable abstractions.

Figure 3.4 gives a schematic view of our abstractions in four main steps. We (A) extract the metadata from the memory in a preparatory step; (B) successively remove the redundancy it contains by merging the different instances, giving us unambiguous metadata; (C) interpret the metadata in a
more abstract way; (D) merge the pairs of user regions’ instances. To simplify the presentation we elide most details about the blocks’ health status from the figure. In the rest we give some additional details of this process.

abs0: Concrete model. As described in Sections 3.2.3 and 3.4.2.

abs1: Extract hash and PTR regions. An auxiliary step. All hash and PTR instances are pulled out of the memory, leaving only the user regions there. This improves the automation of the various Isabelle tools (e.g., the simplifier).

abs2/3: Eliminate metadata redundancy. We replace the logical subsystems which handle metadata transactions described in Section 3.2.3 by their atomic versions, using the proof rule Gran introduced in Section 3.3.4. We achieve this by successively collapsing the pairs of hash and PTR region instances into a single instance. This eliminates the complexity of keeping the metadata copies in sync and provides us with unambiguous metadata. The simulation relation states that an abstract administrative (hash or PTR) region coincides with a concrete instance whenever that instance is not ‘dead’ and its integrity is intact, i.e., its computed hash matches the one stored in the hash region.

abs4: Abstract the metadata and the physical status. We abstract the hash and PTR regions into a combination of per-instance reliability and writability flags, and a per-region arrow field. Moreover, we combine the hash and PTR regions’ health status into a single abstract administrative status.

A region instance is reliable if neither it nor the hash region is ‘dead’ and its integrity is intact (i.e., its computed hash equals its stored hash). It is writable if both it and the hash region are ‘ok’. The arrow indicates the possible directions of instance synchronization for each
region. A region instance is current (as defined in Section 3.4.2 in terms of the administrative regions) exactly if the arrow is bi-directional or points away from the instance. Figure 3.4 shows a possible abs4 state, but it omits the full description of the abs3 state it is extracted from, so we provide some more details here. The secondary instance of the second (user) region is writable, but not reliable. This implies that the physical block corresponding to the instance is ‘ok’, but that either its contents does not match the stored hash, or that the hash region itself is logically ‘dead’. The arrow points away from the secondary instance, implying that the stored hashes of the primary and secondary instances of the second region differ, and that the region’s index is currently stored in the PTR.

abs5/6: Eliminate user regions’ redundancy. Abstract model. The persistent state becomes the one described in Section 3.4.1. It is obtained by collapsing the two user region instances into one. Each abs5 region matches all of its reliable and current abs4 region instances. If no such instance exists, the region’s contents are arbitrary and its health status is ‘dead’. Otherwise, the status is either ‘ok’ (if both instances are writable), or ‘degraded’ (if at least one is unwritable). For example, the second (user) region is ‘dead’, since its only current instance (the secondary one, as determined by the arrow) is not reliable. The third region is ‘degraded’ since it contains a current and reliable instance, but both instances are degraded (that is, unwritable). In abs5, update operations are still performed sequentially and region-wise. We turn these into one-shot atomic updates in abs6 and replace repeated by single restarts as sketched in Section 3.4.1.

3.5 Results

Next, we give a brief overview of how we have formalized the requirements from Section 3.2.2 and verified in Isabelle/HOL that the concrete model satisfies them. We also provide some statistics on the development and discuss our experience.

3.5.1 Establishing the requirements

Requirements (R1-R4) describe properties of individual API calls, which we express and prove as Hoare triples on abs6. We state and prove requirements (R5) and (R6) as (outer-layer) invariants of abs4, since these refer to individual region instances, which disappear in abs5. Our refinement proofs and property preservation theorems then enable us to transfer these properties onto the concrete model. We will sketch this on the example of our main requirement, (R1).
On the abstract model, we can state this property using the following two sets of states, where the left and right constructors of the sum type, $\iota_1$ and $\iota_2$, correspond to exceptional and normal termination respectively.

$$\text{view}_\text{in}(S) = \{(sv, sp) \mid \text{view}(sp) \in S\}$$

$$\text{view}_\text{post}_\text{upd}(v, u, r) = \text{case } r \text{ of}$$

$$\iota_1 \_ \Rightarrow \text{view}_\text{in}(\{v \triangleleft u, v, \perp\})$$

$$\mid \iota_2 \_ \Rightarrow \text{view}_\text{in}(\{v \triangleleft u\})$$

The following Hoare triple then expresses (R1) on the abstract model:

$$\{\text{view}_\text{in}(\{v\})\}$$

$$\text{tryR update}_6 \ u \ \text{catchR startup}_6$$

$$\{\lambda r. \text{view}_\text{post}_\text{upd}(v, u, r)\}_N$$

We then combine the refinement results from (Section 3.4.3) and the general property preservation theorems (Section 2.5) to derive the triple:

$$\{R[\text{view}_\text{in}(\{v\})] \cap \text{reach}(T_c)\}$$

$$\text{tryR update}_0 \ u \ \text{catchR' startup}_0$$

$$\{\lambda r. R[\text{view}_\text{post}_\text{upd}(v, u, r)]\}_N$$

Here, $T_c$ is the transition system of the concrete model, and $R$ is the relation obtained by a (forward) composition of the simulation relations $R_6, \ldots, R_1$ that we sketched in Section 3.4.3, where $R_6$ relates $\text{abs}_i$ and $\text{abs}_{i-1}$. Notice that the property on the concrete model encompasses an arbitrary number of restarts and calls to $\text{startup}$.

3.5.2 Statistics and discussion

All our Isabelle/HOL theories amount to around 39,000 lines. These are composed of the modeling and reasoning infrastructure (~12,000 lines), the models (ranging from ~700 for the concrete to ~200 lines for the abstract model), the refinement proofs (~11,000 lines), and the invariant proofs (~9,000 lines). We approximate our development effort at somewhere between 1 and 1.5 person years. Our first attempt (not counted in the above development time) used an Event-B inspired modeling framework in Isabelle/HOL. However, the considerations of hardware failures render our system quite complex. Since restarts force a small event granularity, the models eventually became unmanageable, due to the large number of events and the fact that the framework did not provide any structured way of expressing the control flow between events.

Overall, the choice of Isabelle/HOL was a mixed bag. HOL’s expressiveness was crucial for representing our system’s unorthodox features. While
3. Verification of a Persistent Memory Manager Under Crashes and Hardware Faults

Isabelle’s connection to external provers and our own custom proof tactics helped a great deal, the degree of automation we were able to obtain was modest. It is clear that significant improvements in this area are needed if one is to scale our verification approach to larger systems.

We also note that our development lacks an executable implementation. However, we believe that deriving one from our concrete model would only require a modest effort, leveraging modern Isabelle tools for C code. Unfortunately, non-technical barriers would likely have prevented a deployment of an implementation on actual devices, thus disincentivizing us from pursuing this further.

3.6 Related Work

Two transaction mechanisms similar to the one described here have been studied before in the literature, both of them targeting smart cards. One is due to Sabatier and Lartigue [SL00], who use the B method for development and verification. As usual in B, the system is modeled as an (unstructured) transition system, which makes modeling restarts easy. Their main proof technique is refinement. From the final model, they derive a C implementation by hand, without a formal link to the B development. As discussed in Section 3.5.2, we were unable to scale this approach to our case study.

Another transaction mechanism was proposed by Hartel et al. [HBdJL01, PHdJ02]. They combine Z notation and SPIN [HBdJL01] (resp. JML in [PHdJ02]) to analyze a C implementation, but the unclear relationship between the different formalisms and the lack of machine-checked proofs obscure the resulting guarantees.

Andronick [And06] discusses a general verification methodology for reasoning about C programs under restarts, but aimed at transaction mechanisms. Her approach is the one most similar to ours, in that restarts are modeled as exceptions in a structured input language, while allowing for an arbitrary number of successive restarts to be analyzed. Verification is performed directly on C source code, by leveraging the Why/Caduceus tool. However, her model of restarts does not include any effects on the state and the paper describes only a toy case study. It also mentions a larger one, but without providing any details.

The PMM could also be viewed as a highly primitive file system. In response to Hoare’s Grand Verification Challenge, Joshi and Holzman [JH07] propose verifying a file system as a “mini challenge”, identifying restarts and hardware failures as major hurdles in overcoming it. While the PMM is a far cry from a full-blown POSIX file system, we may claim to have completed a micro challenge with its verification. Moreover, since the publication of this work, significant progress has been made in addressing the challenge.

Ernst et al. [EPSR16] present a framework for modular reasoning about
3.6. Related Work

crashes in programs formalized as abstract state machines (ASMs). They define black-box machine semantics, which corresponds to a big-step semantics without crashes, and a white-box semantics which corresponds to a big-step semantics with crashes (i.e., it includes intermediate states). The formalism supports submachines, and they extend it with crash recovery operations. The main result is a theorem that enables them to show the refinement of a white-box (outer) ASM with a black-box submachine to a black-box outer ASM, if the entire state of the outer ASM is volatile and if the submachine is “crash-neutral”, which roughly means that a crash before a submachine call can be simulated by first calling the submachine, and then performing a crash. They use this framework in [EPSR15] to verify a transactional journal (based on a write-ahead log) for a flash file system called Flashix, which appears to be single-threaded. The hardware in their model can fail to execute operations, but the system does not seem to provide resilience against such failures. It is unclear if (hardware) block writes are assumed to be atomic.

Chen et al. [CZC+15] develop a domain specific language, which includes an extension of Hoare logic with crash specifications, within the Coq prover. The crash specifications describe the post-conditions that must hold in case of a crash. In this sense, their specifications must talk about more states than ours, since our Gran rule enables us to avoid talking about crashed states explicitly. They use the language to model and prove the correctness of a simple file system called FSCQ, which is modeled after the xv6 file system used for teaching. The setting is incomparable to ours. Like our model, the file system is single-threaded. However, it assumes atomic block writes and no failures; but unlike our model, it supports asynchronous writes. The central role in crash handling is played by the transaction subsystem, which roughly corresponds in functionality to the PMM. Unlike the PMM, this system is based on a write-ahead log. All top-level file system operations are wrapped inside a single transaction, guaranteeing atomicity in case of a crash. They do not use refinement explicitly, but different predicates appearing in their Hoare pre- and post-conditions can be grouped into separate conceptual levels. Furthermore, a separation logic embedded in their Hoare logic helps them improve automation.

In [SBTW16], Sigurbjarnarson et al. perform an almost fully automatic verification of a file system similar to FSCQ. They develop a verification framework for relating two programs written in a subset of Python, using the notion of crash refinement. The definition of crash refinement is almost identical to the version of our Gran rule where the Idem rule is applied to remove the recTryR(·) construct. We believe that this can be taken as a further testament to the utility of the proof rules we have devised. Then, they build a system consisting of six layers, where the highest layer is the complete file system, and the lower layers correspond to subsystems (e.g. for managing inode and free block data). Each layer comes with a specification, and an automated refinement proof relates the two. The proof is performed
by an SMT solver, and the input to the solver are symbolic representations of the two programs, created by a symbolic execution engine. The symbolic representations of the higher layers use the specifications of the lower layers instead of their implementations.

Amani et al. [AHC+16] provide a domain-specific language for file systems, called Cogent; they also provide reasoning tools for the language, within Isabelle/HOL. Cogent is designed for performance, and it can be compiled straight to C, and thus also to native code. The authors use Cogent to implement a single-threaded flash memory file system called BilbyFS. The generated C code for BilbyFS is then verified to be correct with respect to a high-level specification, and comparable in performance to a hand-coded version of BilbyFS. However, the specification does not include crash behaviors.
Part II

Handling Node-Level and Communication Faults
Chapter 4

Consensus and Models of Distributed Systems

In the first part of the thesis, we investigated a key component in providing tolerance to system crashes: transactional access to the persistent storage. In the second part, we take a wider view, where the system is now not standalone anymore; instead, it is just a node of a distributed system. A key component for making the system as a whole tolerant to individual node crashes (whether with or without recovery) is solving the so-called distributed consensus problem. The second part of the thesis is devoted to this problem.

This chapter reviews the relevant background from distributed computing. We start with the various failure and timing models present in the literature and then review the basic formal models of distributed systems. We then move on to the problem of consistency in distributed systems, and show how this problem can be reduced to the problem of distributed consensus.

4.1 Introduction

Distributed systems [Lyn96, CGR11] are computer systems consisting of multiple separate processes, working towards a common goal. The processes receive inputs from the system’s clients and produce outputs for them. The processes run concurrently, on separate physical machines, and communicate by message passing. This is in contrast to parallel computing systems, where the separate processes operate on shared memory, with the main goal of increasing performance. However, the distinction between distributed and parallel systems is not entirely clear cut, especially as message passing can be used to simulate certain properties of shared memory.

As the processes communicate using message passing, we can view distributed systems as graphs, with the processes as nodes and the communication links as edges. We will thus often refer to the individual processes
as nodes. Different layouts of physical links yield different graph topologies. The topology is mostly relevant for lower-level (e.g. routing) protocols. Higher-level protocols, such as the ones considered in this thesis, can rely on the different transport and network layers (e.g., TCP/IP) to provide full connectivity over the underlying physical links. We thus assume throughout this thesis that the system’s graph is complete. Note that this completeness does not mean that all sent messages get through, but merely that the possibility of direct communication exists between all pairs of nodes.

The need for distributed systems arises for two reasons. First, the input data might be produced and used at different nodes of the system. In this case, the system is distributed by its very nature, and cannot be centralized. Second, a distributed system might not be strictly necessary to solve a problem, but it can improve the following properties of the solution:

Fault tolerance/availability If one of the nodes fails, the distributed system might still be able to keep on running, as the system’s clients can still use the other nodes.

Throughput The distributed system can combine the processing power of the individual nodes.

Latency A centralized system only has a single location, and the response latency increases with the distance of the client. In a distributed system, a client can get a response from the node closest to it.

We focus on the first one of those, fault tolerance. This obviously implies the existence of faults in the systems we consider. It turns out that discerning the effects of faults in distributed systems is closely related to timing assumptions. We thus next describe the standard failure and timing models considered in the literature.

4.2 Failure and Timing Models

Distributed computing algorithms are constructed under different assumptions about the hosts that they execute on, and the links used for communication. The two main classes of assumptions are failure and timing assumptions, giving rise to the different models the algorithms are targeted for, and analyzed in. By “model”, in this section we mean an approximation of reality, and not necessarily a formal mathematical model.

4.2.1 Failure assumptions

Most models distinguish between two kinds of failures: process and link failures. Process failures are most often modeled in three varieties:
• *Crash-stop*, also called *fail-stop* failures, where processes stop executing, and never restart.

• *Crash-recovery* failures, where the processes can restart after a failure. In this case, their state prior to the crash is usually assumed to be wholly or in part recovered at startup. While it is sometimes possible for the processes to restart from a “blank” state and relearn its old state from the other processes before rejoining the system, this is often difficult to do correctly. For example, Google’s Paxos-based implementation of the so-called total-order broadcast service [CGR07] was later found incorrect precisely for this reason [vRSS14]. Thus, the process state (or a part thereof) is usually assumed to be correctly persisted through crashes. We note that this requires a recovery mechanism, such as the one presented in the first part of the thesis.

• *Byzantine* failures, where the processes can fail in arbitrary ways. This can include malicious behavior of the processes. In contrast, the crash-stop and crash-recovery failures are referred to as *benign* failures.

Additionally, *omission failures* are sometimes also considered, where a process “forgets” to send or receive messages. Unsurprisingly, all of these failures hamper the ability of distributed systems to perform their tasks. One must thus often bound the number of processes that are allowed to fail. The bounds are usually represented as fractions of the total number of processes; for example, we can assume that less than $\frac{n}{2}$ of the processes are allowed to fail, where $n$ is the total number of processes in the system. This number, which we denote by $n_{\text{fail}}$, is often used as the main measure of an algorithm’s fault tolerance. Note that what constitutes a failed process depends on the failure model. For instance, in the crash-recovery model, a process is considered failed only if it crashes and never recovers, or if it crashes infinitely often [CGR11].

Link failures also come in different varieties. Some models assume links to be perfectly reliable, that is, they do not consider link failures at all. In practice, however, *network partitions* often occur [BK14]. They correspond to a partition in the system’s graph, where links between two groups of processes are severed; the connectivity is typically restored after a while, *healing* the partition. Such behavior is captured by assuming *fair-lossy links*. These can lose messages, but guarantee that sending a message infinitely often will cause it to be delivered infinitely many times. On top of these, one can again build reliable links, although delays might be introduced. On the extreme end, one could consider completely unreliable links, with no delivery guarantees; however, in a system using (only) such links the communication could be blocked completely, rendering the system useless.
4. Consensus and Models of Distributed Systems

4.2.2 Timing assumptions

Timing assumptions constrain the relative running speeds of processes or message transmission delays. If both are constrained to some known values, and all processes have access to synchronized clocks, every process can bound the time it has to wait to receive a response from another process. In such a model, the system typically operates in rounds, where a round is long enough to allow each pair of processes to exchange messages, and to allow each process to finish the necessary local computations. Rounds follow a fixed structure, where in each round every process:

1. starts by sending out the round’s messages;
2. waits for inbound messages to arrive; and
3. updates its local state based on the received messages.

Since the timing of the rounds is synchronized at all the processes, we obtain a synchronous system model. We note that the synchrony assumption is not completely independent of the failure assumptions. The assumption of known transmission delays usually also implicitly assumes that the links are reliable, and that no omission failures occur.

In contrast, dropping the timing assumptions leads us to asynchronous models. In most cases, dropping even a single assumption (constrained relative process speeds, bounded message delays and synchronized clocks) has the same visible effect on the resulting system as dropping all of them. Thus, the asynchronous models typically negate all three assumptions: processes do not have access to clocks, their relative speeds are unconstrained, and messages can be delayed without a bound.

A middle ground is to assume partial synchrony. Traditional partial synchrony assumptions, presented in the classic paper of Dwork et al. [DLS88], come in two main flavors. In the first one, the system is synchronous, but the timing bounds are unknown. In the other, the system behaves arbitrarily for some unknown global stabilization time. After this time, the system enters an infinitely long stable period, in which it behaves completely synchronously, obeying a priori known timing bounds, and no failures occur.

These traditional assumptions are not valid for many real distributed systems, since they require the stable period to be infinitely long. A more realistic set of assumptions is presented in the timed asynchronous model of Cristian and Fetzer [CF99]. There, the processes are assumed to be equipped with local clocks, which are loosely synchronized. However, the relative process speeds and transmission delays are unconstrained. The model is then also extended to include stable periods, which they call progress periods, in which the system behaves synchronously. Unlike the traditional stable period, the progress period need not last forever. In [CF99], the authors also
provide extensive measurements to validate the model in practice. Nevertheless, the traditional assumptions of Dwork et al. are more prevalent in the literature. They are easier to incorporate into the traditional models of asynchronous systems, and the stable periods of real distributed systems are long enough to be practically infinite compared to the timescales in which distributed systems function.

4.2.3 Double trouble: asynchrony and failures

The different failure and timing assumptions affect the problem-solving ability of systems. To describe this difference precisely, we first need a notion of a problem suitable for distributed systems. One such notion are decision tasks [RS13], where each of the \(n\) system processes is given an input, and each has to produce output. A decision task is defined by a relation on the \(n\)-element input and (allowed) output vectors (in the presence of failures, only subsets of processes might receive inputs and produce outputs). Unsurprisingly, with full synchrony assumptions, and with no failures, all tasks defined by computable relations are solvable. Perhaps surprisingly, it turns out that synchronous and asynchronous models are equivalent in their problem-solving ability [RS13]; but only as long as no failures occur. The difficulty appears once one allows the possibility of both asynchrony and failures.

With the possibility of failures, asynchronous models become very weak in their problem-solving ability, that is, very few useful problems can be deterministically solved in such a setting (probabilistic solutions often still exist). The essential difficulty is that the asynchronous setting makes it impossible to distinguish a slow process (or a process whose messages are delayed for a long time) from a crashed one. Indeed, this difficulty underlies many of the impossibility results in the area of distributed computing, such as the famous FLP [FLP85] and CAP [GL12] results.

4.3 Formal Models

In the last section we covered the main classes of assumptions about the environment that distributed systems operate in. To reason about the system precisely, we must capture both the environment and the system in a formal model. In this section, we give an overview of the main types of such models that exist in the literature. In Section 4.3.3 we then focus on a particular type of model, called the Heard-Of model, which we use for formalizing the distributed systems in this thesis.

Distributed computation assumes concurrency, with only partial temporal ordering between actions. Models which capture such partial orderings have been investigated (e.g., [GG01]), however, for simplicity, most works
adopt the interleaving-based model of transition systems (often with additional models or languages layered on top). The timing and failure assumptions outlined above then drive transition systems’ design in different ways. As synchronous systems are usually structured in rounds, they are typically modeled by global transitions, which change the state of all processes. Asynchronous systems are modeled by letting each process take its steps individually, and by explicitly modeling the network as a buffer into which messages are sent and from which they are delivered. The process and network steps can be interleaved arbitrarily.

Both synchronous and asynchronous systems can thus be modeled in a straightforward fashion. We will henceforth call the corresponding types of formal models synchronous and asynchronous models. However, as mentioned previously, most realistic systems lie somewhere in between the two types of systems. A direct approach of modeling them is to add clocks to the asynchronous model, or to use timed transition systems. However, proofs in systems with such explicit timing assumptions are complicated and unwieldy [Lyn96]. Certain other extensions to the ordinary asynchronous and synchronous models turned out to provide a simpler approach. We discuss those next.

4.3.1 Failure detectors

In their groundbreaking work, Chandra and Toueg [CT96] augment the asynchronous model with failure detectors, which are oracles that inform processes about crashes of other processes. These oracles remove the main weakness of the asynchronous models, the impossibility to distinguish failed from slow processes. Detectors are characterized by different levels of completeness (whether an actually crashed process is reported crashed) and accuracy (whether a process reported crashed is actually crashed). Different assumptions on accuracy and completeness of the available detectors capture the different timing assumptions, without having to explicitly reason about time. As an example, an eventually perfect failure detector (denoted $\Diamond P$ in [CT96]) is eventually complete and eventually accurate. Such a detector can be realized in a partially synchronous system with infinitely long stable periods using timeouts; yet neither timeouts nor time ever appear in the formal model. We now see why the assumption of an infinite stable period is useful: as the asynchronous model has no notion of time, it is impossible to define a failure detector that can be implemented if stable periods are shorter than any fixed finite length. Other examples of failure detectors are the perfect failure detector, which is eventually complete and always accurate, and can be easily implemented in synchronous systems; and the eventually strong detector that is eventually complete, and has eventual weak accuracy: some correct process is eventually not suspected by any other process.

A lot of work, starting with [CHT96], has been devoted to the question
4.3. Formal Models

of weakest failure detectors required to solve a problem. A detector $D$ is the weakest for a problem $P$ if any other failure detector which can be used to solve $P$ can also be used to implement $D$. Freiling et al. [FGK11] give an overview of recent work on failure detectors.

While the notion of failure detectors significantly improved the state-of-the-art of distributed computing, the detectors are not without their shortcomings. They primarily target the crash-stop failure model; they often have to be augmented for the crash-recovery model [ACT00]. Adapting them to the Byzantine setting is hard. Finally, while reasoning with failure detectors is significantly simpler than reasoning about partial synchrony models with explicit timing, it is still far from trivial. Steps of the individual processes can be interleaved in many different ways, and some processes can advance far ahead of the other ones. The design of such algorithms is difficult. To provide some additional structure, many algorithms in the asynchronous model (for example, for the consensus problem such as [CT96, Lam98, DLS88]), adopt the concept of a round from the synchronous model. For such algorithms, simpler formal models exists which eliminate the interleaving. We describe them next.

4.3.2 Lockstep models

Inspired by this ubiquity of round-based algorithms, a different line of research [Gaf98, CBS09, RS13] starts from synchronous, round-based and lockstep systems. Whereas the failure detectors strengthen the asynchronous model, the idea here is to weaken the synchronous model by allowing messages to be changed or dropped. The first such model was introduced by Gafni in [Gaf98], using round-by-round fault detectors (RRFD).

The model applies to round-based algorithms in which each process individually follows the three-step synchronous round structure described in Section 4.2.2: it (1) sends out the round messages, (2) collects the round messages, and (3) performs a local step based on the received messages. Clearly, due to asynchrony, messages can arrive out of order, so the sender must tag each message with the round number it belongs to. The receiver needs to decide what to do with messages from rounds different than the current round. If it ignores them (typically by discarding messages with older round numbers and by buffering messages with newer round numbers), we say that the rounds are communication-closed. This is a crucial requirement of Gafni’s model. Figure 4.1 shows an example algorithm, the OneThirdRule algorithm [CBS09], that complies with requirements. The algorithm is designed to solve the consensus problem, but we postpone the explanation of both the problem and how the algorithm solves it. At this point, we simply use it to illustrate the above structure. The algorithm’s presentation uses a style of pseudocode often found in the literature. The state of each process is a record consisting of the round number $r$, and two additional fields $\text{inp}$ and
4. Consensus and Models of Distributed Systems

1. Initially: $r_p = 0$
2. $\text{inp}_p$ is $p$’s input value
3. $\text{dec}_p = \perp$
4. repeat forever:
5. send $(r_p, \text{inp}_p)$ to all processes
6. $R_p := (\lambda \_ . 0)$ \hspace{1cm} // a multiset
7. collect messages $(r, v)$:
8. if $r = r_p$ then
9. $R_p(v) = R_p(v) + 1$
10. if $\exists v. R_p(v) = |R_p| > \frac{2n}{3}$ then
11. $\text{dec}_p := v$
12. if $|R_p| > \frac{2n}{3}$ then
13. $\text{inp}_p :=$ the $v$ such that for all $w$:
14. $R_p(v) \geq R_p(w) \land$
15. $(R_p(v) = R_p(w) \implies v \leq w)$
16. $r_p := r_p + 1$

Figure 4.1: Pseudocode for the OneThirdRule algorithm. It is executed by every process $p$.

dec. Line 5 corresponds to Step (1); lines 6-9 to Step (2), and lines 10-15 to Step (3).

A crucial question is when the message collection (Step (2)) is terminated. In Gafni’s model, in each round $r$, each process $p$ waits until, for each process $q$, it either receives the round-$r$ message from $q$, or the RRFD instructs it that $q$ is faulty in round $r$, written $q \in D(p, r)$. The detector is unreliable: $q \in D(p, r)$ does not mean that the process $q$ has actually crashed. It might simply be late, or be communicating over a slow or lossy link. Thus, crashed processes are indistinguishable from processes whose messages are dropped by the network. In the next section, we will exploit this observation and model crashed processes as processes whose messages are no longer received by the other processes. However, the failed processes can still continue receiving and processing messages.

The RRFD effectively weakens the synchronous system model. In contrast to the asynchronous setting, where the failure detectors are seen as helpful, the fault detectors can be viewed as reporting the actions of adversaries suppressing messages [Gaf98, RS13]. The power of the adversary over the execution of the system is then constrained by so-called communication predicates.

Similar to the properties of failure detectors in asynchronous systems, the communication predicates reflect the timing and failure assumptions about the environment. For instance, crash-stop failures in a synchronous system
correspond to the following predicate:

\[ \forall p \ q \ r. \ q \in D(p, r) \implies (\forall p'. \ \forall r' > r. \ q \notin D(p', r')) \]

Better said, a fault detector satisfying this predicate can be implemented in synchronous systems with crash-stop failures. There might also be other predicates (i.e., fault detectors satisfying them) that can be implemented by this type of systems. Gafni proposes predicates for several kinds of systems, which should both specify the systems precisely and “resemble” them, but neither of these definitions is formal. Similar to the failure detectors for asynchronous systems, one can also pose the question of weakest communication predicates for a given task.

As mentioned earlier, Gafni’s model provides a much simpler semantics for round-based algorithms than the usual asynchronous one. Like in the synchronous model, the processes in this semantics proceed in lockstep, which is why we refer to them as lockstep models. However, the semantics is left implicit in Gafni’s paper. In the next section, we will formally define the semantics of a variation of Gafni’s model, called the Heard-Of model. We remark that an added benefit of lockstep models is that the rounds provide the model with an implicit notion of time. This allows one to elegantly encode the partial synchrony assumptions of (finite) progress periods, while still avoiding the need to explicitly reason about time.

4.3.3 The Heard-Of model

A variation on Gafni’s model is the Heard-Of (HO) model of Charron-Bost and Schiper [CBS06, CBS09]. Instead of the round-by-round fault detectors, they define the notion of Heard-Of sets (HO sets). The two are essentially identical, with \( HO(p, r) \) being the complement of \( D(p, r) \). Additionally, the message collection step is removed from the HO model and is assumed to be handled by a lower-level messaging layer. As the HO model has a more fully developed formal specification, we choose it as the basis of our work.

Formally, the HO model is parameterized by:

- a set \( \Pi \) of processes;
- a set \( S_p \) for each \( p \in \Pi \), the local state space of each process;
- a set \( \mathcal{M} \) of messages.

We will also write \( \{HO^r_p\}_{p \in \Pi}^{r \in \mathbb{N}} \) to refer to a HO collection, that is, a function \( HO : \Pi \to \mathbb{N} \to \mathcal{P}(\Pi) \). Following [CBS09], we write \( HO_p^r \) for \( HO(p, r) \). We use the same notation for other functions, writing \( f_p^r \) for a function parameterized in processes \( p \in \Pi \) and rounds \( r \in \mathbb{N} \), and \( f_p \) for a function parameterized in just the processes \( p \in \Pi \).

Then, an algorithm in the model is defined by three components:
4. Consensus and Models of Distributed Systems

Initially: \( \text{inp}_p \) is \( p \)'s input value, 
\[ \text{dec}_p = \bot \]

\( \text{send}_p^r \):
send \( \text{inp}_p \) to all

\( \text{next}_p^r \):
if \(|HO_p^r| > \frac{2n}{3}\) and all received messages equal some \( v \) then
\[ \text{dec}_p := v \]
if \(|HO_p^r| > \frac{2n}{3}\) then
\[ \text{inp}_p := \text{smallest most often received value} \]

Figure 4.2: The HO model of the OneThirdRule algorithm

- a set \( I_p \subseteq S_p \) of initial states for each \( p \in P \);
- a send function \( \text{send}_p^r : S_p \times \Pi \rightarrow M \), where \( \text{send}_p^r(s_p, q) \) determines the message \( p \) sends to \( q \) in round \( r \), based on \( p \)'s current state \( s_p \). This function is total; not sending a message is modeled by a special dummy message, which we model by the element \( \star \in 1 \);
- an update function \( \text{next}_p^r : S_p \times (\Pi \rightharpoonup M) \rightarrow \mathcal{P}(S_p) \). Let \( \mu_p^r : \Pi \rightharpoonup M \) model the messages that the messaging layer of \( p \) collects in round \( r \), such that \( \mu_p^r(q) \) is undefined if \( q \notin HO(p, r) \), i.e., \( q \)'s message was not received by \( p \). Then, \( \text{next}_p^r(s_p, \mu_p^r) \) determines the set of possible successor states of \( p \)'s current state \( s_p \).

Figure 4.2 shows the pseudocode for the HO model of the OneThirdRule consensus algorithm previously shown in Figure 4.1. Note that the round variable \( r \) has now been dropped from the process state in the HO algorithm description. As all processes execute the same code in each round, \( \text{send}_p^r \) and \( \text{next}_p^r \) are the same for all processes \( p \) and rounds \( r \).

Semantics of the Heard-Of model

The HO model has a straightforward unlabeled transition system semantics. Each HO collection \( \{HO_p^r\}_{p \in \Pi} \) gives rise to a set of traces as follows. The global \( n \)-process state consists of the Cartesian \( n \)-product of local process states (that is, \( \prod_{p \in \Pi} S_p \)), together with the current round number, denoted by \( \text{rnd}(s) \) for state \( s \). The set of the initial states of the system is \( \prod_{p \in \Pi} I_p \) paired with the round number 0. A step \( s \rightarrow s' \) of the system, with \( \text{rnd}(s) = r \), corresponds to the simultaneous execution of the round \( r \) by all processes, based on the send and next functions and an HO collection. More precisely, given an HO collection \( \{HO_p^r\}_{p \in \Pi} \), let \( \mu_p^r(q) = \text{send}_q^r(s_q, p) \) if \( q \in HO_p^r \) and let it be undefined otherwise. Moreover, let \( s_p \) (respectively \( s'_p \) be
the projection of $s$ (respectively $s'$) onto $p$’s local state. Then, we have $\text{rnd}(s') = \text{rnd}(s) + 1$ and, for all $p$, $s'_p \in \text{next}_p^r(s_p, \mu_p^r)$. As this advances the state of every process, the HO model has no notion of failed processes, formalizing our earlier observation about the indistinguishability of failed and silent processes. In fact, all processes are assumed to run forever.

Since each transition includes an instantaneous exchange of messages, this semantics does not require an explicit representation of the network. Furthermore, all processes proceed in lockstep, and we do not have to consider all the different possible interleavings of process actions. This greatly simplifies reasoning about these models compared to an asynchronous model.

Reality, of course, does not proceed in lockstep. Hence, there is a second, asynchronous semantics of the HO model [CSCBM09]. As before, each HO collection gives rise to a set of traces. However, in this semantics, each process has its own view of the current round number. All messages carry the sender’s round number and are explicitly transmitted over a network. A process $p$ executing its round $r$ receives only the $r$-round messages from processes in the set $\text{HO}_r^p$. Once it has received all such messages, it can take a $\text{next}_p^r$ transition to move on to the next round. Each process does this independently. This asynchronous semantics thus closely corresponds to the real world, where the sequence of HO sets is, however, generated dynamically, depending on when the processes decide to move on to the next round.

The main theorem of [CSCBM09] tells us that a certain class of so-called local trace properties, when proved under the lockstep semantics (also called coarse-grained semantics) also hold in the asynchronous one (called the fine-grained semantics). Local trace properties are defined as follows. Let $s_p$, as before, be the process $p$’s component of a global state $s$. Given a trace $\tau$, the local trace for process $p$ is obtained by projecting the configurations onto the state of $p$ and removing the stuttering steps. A trace property is local if, for every trace $\tau$ it contains, it also contains all traces that have the exact same local trace for every process $p$. Intuitively, the property must be insensitive to the interleavings of steps of the individual processes. We exploit this result to simplify our correctness proofs, since consensus can be specified as a set of such local properties. Thus, throughout this thesis, we will use the lockstep semantics.

**Communication predicates**

As we have seen earlier, the solvability of the different distributed problems requires assumptions on the environment. For instance, if the network drops all messages, then most problems become unsolvable. As the effects of the environment are only visible in the HO collection, the environment assumptions in the HO model (like in the RRFD model) are encapsulated in communication predicates, which determine the set of allowed HO collections.

While arbitrary communications predicates are a priori possible, if they
are to be useful, they must reflect realistic assumptions on distributed systems. In this thesis, we focus on the asynchronous and partially synchronous setting with crash-stop and crash-recovery failures. In Section 4.2, we have already seen the two main classes of assumptions for this setting. The first one is the number of failed processes that can be tolerated by an algorithm. The second one are stable periods. These assumptions are reflected in two types of round formulas, $\phi_{th}$ and $\phi_{uf}$, of the forms:

$$\phi_{th}(c, r) \triangleq \forall p. |HO_p^r| > c \cdot |\Pi| \quad (\phi_{th})$$

$$\phi_{uf}(r) \triangleq \forall p, q. HO_p^r = HO_q^r \quad (\phi_{uf})$$

The threshold formula $\phi_{th}(c, r)$ requires that, in round $r$, all processes receive messages from at least the fraction $c \in [0, 1)$ of processes. This formula encapsulates the assumptions about the number of failures and timeouts within round $r$. The uniformity formula $\phi_{uf}(r)$ requires that all processes receive messages from the same set of processes. This reflects the partial synchrony assumption of a stable period that spans an entire round. For example, the communication predicate for the OneThirdRule algorithm is specified by:

$$\exists r_1, r_2 > r_1. \phi_{th}(\frac{2}{3}, r_1) \land \phi_{uf}(r_1) \land \phi_{th}(\frac{2}{3}, r_2).$$

There are two straightforward strategies to implement a messaging layer that ensures communication predicates that contain a $\phi_{th}(c, r)$ formula. We use the term “strategies” since we are not aware of any general proofs of their correctness; Hutle and Schiper [HS07] do give a rigorous correctness proof for the second strategy, but only for implementing a variant of the above predicate for OneThirdRule, and not in general. Both strategies assume $n_{fail} < (1 - c) \cdot n$.

The first strategy is to employ waiting: all processes simply never progress to the next round until they receive more than $c \cdot n$ messages. Then, clearly, $\phi_{th}(c, r)$ holds for all rounds $r$. The downside of this approach is twofold. First, if the network can drop messages then to guarantee progress each process $p$ must keep retransmitting each message $m$ that it ever sent, until it receives an acknowledgment of receipt for $m$ from all the processes. Second, the processes must keep waiting even though they know that a new algorithm round has been started (e.g., when they see a message tagged with a round number higher than their current one).

The second strategy is simpler, but it can only ensure that $\phi_{th}(c, r)$ holds for some round, and it additionally relies on the existence of progress periods. It is based on timeout, where for each round each process collects the round messages for at most the system’s timing bound: the time required for a round of message exchange and processing. Moreover, the messaging layer fast-forwards through the rounds as soon as a message is received with a round number higher than the current one.
4.3. Formal Models

The $r$ in $\phi_{of}(r)$ formula is typically existentially quantified. The above timeout-based strategy can also be used to implement such predicates, as is also shown in [HS07], again in the context of the OneThirdRule’s communication predicate.

Model extensions

There exist several extensions to the basic HO model. The first one was already described in the original HO paper [CBS09], targeting leader-based (also called coordinated) algorithms, which use a distinguished process called the leader (or the coordinator). Leaders are often used in agreement problems, such as consensus, since the leader can propose a unique value that the other processes can agree on.

The types of next and send are enriched with an additional parameter ldr of type $\Pi$. The global state $s$ additionally contains a field ldr of type $\Pi \rightarrow \Pi$; the additional parameter ldr of send$_{\text{round}}$ and next$_{\text{round}}$ is instantiated to ldr($s$)(p). Intuitively, ldr($s$)(p) = q denotes that p believes that q is the current leader. The algorithm specifies the leader policy, the set of allowed ldr functions. In this work, we will always assume that the leader of each round is unique and known in advance by all processes. Overloading the name ldr, this can be achieved by supplying all processes with a function ldr : $\mathbb{N}$ → $\Pi$, which determines the leader of a round $r$. Many practical leader-based algorithms rely on this policy, which is easily implementable, for example by a rotating coordinator paradigm, where the leader of the round $r$ is determined to be the process numbered ($r \mod |\Pi|$). Alternatively, if the set $\Pi$ of processes is ordered, the round identifier can be made a pair, where the second component is the leader’s identifier; the ordering on the rounds is then the standard lexicographic ordering.

The leader-based algorithms inevitably assume that the algorithm will eventually end up with a stable leader process for a long enough period of time. A stable leader must not be crashed, and all other correct processes must believe it to be the leader and not try to topple it. In practice, this means the leader must be able to communicate with the other processes, and in a timely manner. In the HO model, the stable leader assumption for a round is expressed by two kinds of round formulas, depending on whether the leader is sending or receiving messages.

$$\phi_{ls}(r) \triangleq \forall p. \ ldr(r) \in HO_p^r$$

$$\phi_{lr}(c,r) \triangleq HO_{ldr(r)}^r > c \cdot |\Pi|$$

Another limitation of the original fault detector/HO model was that it could only model message loss, but not message corruption. As a consequence, Byzantine failures and transmission faults could not be modeled. To address this, Biely et al. [BWCB+07] extend the HO model with the notion
4. Consensus and Models of Distributed Systems

of safe heard-of (SHO) sets. Like HO sets, SHO sets are defined per-process and per-round, and are in fact subsets of the corresponding HO sets. Process \( q \) is defined to be in processes \( p \)'s SHO set in round \( r \) if \( p \) received a message from \( q \) in that round, and the received message corresponded to the message that \( q \) should have sent following the protocol. Biely et al. also propose consensus algorithms in the presence of transmission, but not Byzantine faults. This work was extended by Milosevic and others in [MHS14] to cover Byzantine faults as well. However, we do not consider Byzantine failures in this thesis.

4.4 Consistency in Distributed Systems

If a distributed system is to be viewed as a single system and not a disparate collection of unrelated entities, the system’s processes must be able to coordinate their actions and provide *consistent* outputs to the system’s clients. Intuitively, consistency can be seen as a measure of similarity to a centralized system. There exist many notions of consistency in the literature [VV15, Ady99], aimed at different settings and types of systems.

In this thesis, we are interested in fault tolerance, which is generally achieved using replication: each node runs a copy of a centralized system, including the entire state of the system. It is natural to expect such a system to be virtually indistinguishable from a centralized one to the clients, and thus strongly consistent. The strongest relevant consistency notion is *linearizability* [HW90]. We note that linearizability does come at a cost: it is impossible to achieve in fault-tolerant systems without sacrificing availability, as proved in the now famous CAP theorem [GL12], and it comes with a performance penalty [BFF+14] compared to less consistent systems. Still, its strong semantics still often make it the preferred choice for application developers [CDE+13]. Roughly, a system execution is linearizable if there is a way to reorder the clients’ requests and the system’s responses such that (1) each response immediately follows the corresponding request; (2) the order of non-overlapping operations of any two clients is preserved; and (3) the resulting sequential execution respects the sequential semantics of the system.

A prominent method for creating linearizable replicated systems is *state-machine replication*, first proposed by Lamport in [Lam78]. Here, the centralized (sequential) system is modeled as a deterministic state (possibly infinite state) machine, driven by a series of client requests. In the distributed version, each replica has a copy of this machine. If the exact same stream of requests (without perturbations) is fed to every replica, the system’s execution is guaranteed to be linearizable. The problem of determining a common stream of values is known as *total-order broadcast* or *atomic broadcast* [CGR11].
4.4. Consistency in Distributed Systems

Total-order broadcast can be further reduced to the problem of determining a common value for every position in the input stream. This problem of determining a common value out of a set of proposed ones is known as the **consensus** problem. We now describe it in detail.

### 4.4.1 The consensus problem

The consensus problem assumes a fixed set $\Pi = \{1, \ldots, n\}$ of communicating processes. Typically, we want a solution that works for any $n > 0$, i.e., an algorithm parameterized by $n$. Each process in $\Pi$ gets some input from the value domain $\mathcal{V}$, and the goal is to have all processes decide on a common output. There exist multiple definitions of the problem, depending on the system model used. We base our definition on the one in [CBS09], and we say that a system solves the consensus problem if it guarantees:

**(Uniform) agreement:** No two processes ever decide on two different values.

**Termination:** Every process eventually decides on a value.

**Non-triviality:** Any value decided upon was input to some process.

**Stability:** Once a process decides, it never reverts to an undecided state.

Note that, contrary to what its name might suggest, the termination requirement says nothing about stopping the execution. In fact, as mentioned in Section 4.3.3, we assume that all processes run forever. Furthermore, as there exists no notion of a failed process in the HO model, none of the requirements (including termination) make any provisions or exemptions for failed processes.

We now define the requirements formally over traces of HO systems. We assume that the state of each process is a record, with fields $\text{inp}$ (the input received by the process) of type $\mathcal{V}$ and $\text{dec}$ (the decision made by the process) of type $\mathcal{V}_\perp$. Given a global state $s$, we abuse notation to let $s(p, f)$ refer to the field $f$ of process $p$’s state. Then, a trace $\tau$ satisfies:

- **agreement** iff
  $$\forall i, j \in \text{dom}(\tau). \forall p, q. \forall v, w \in \mathcal{V}. \tau(i)(p, \text{dec}) = v \land \tau(j)(q, \text{dec}) = w \implies v = w;$$

- **termination** iff:
  $$\forall p. \exists r \in \text{dom}(\tau). \tau(r)(p, \text{dec}) \neq \bot;$$

- **non-triviality** iff:
  $$\forall r \in \text{dom}(\tau). \forall p. \tau(r)(p, \text{dec}) \neq \bot \implies \exists q. \tau(r)(p, \text{dec}) = \tau(0)(q, \text{inp});$$
4. Consensus and Models of Distributed Systems

- stability iff:

$$\forall r, r' \in \text{dom}(\tau). \forall p. \tau(r)(p, \text{dec}) \neq \bot \land r' \geq r \implies \tau(r')(p, \text{dec}) \neq \bot.$$ 

An algorithm in the HO model then solves consensus if all infinite traces that arise from any HO collection that respects the communication predicate of the algorithm respect satisfy all four of the above properties. Except for termination, all of the above properties are safety, and for them it suffices to look at all finite prefixes of such traces.

We reiterate here that our definition of consensus comprises agreement on a single value. These days it is also not unusual to see algorithms such as Multi-Paxos [Lam01a] or Zab [JRS11], or even systems such as Bitcoin [Nak08], being referred to as consensus algorithms. Our usage of the term consensus will, however, always be consistent with the above definition; Multi-Paxos and Zab are total-order broadcast algorithms in our terminology.
Chapter 5
Explaining and Unifying Consensus Algorithms

In the previous chapter, we explained the consensus problem, and described how it is fundamental for distributed computing. We have also seen an example consensus algorithm. However, we provided no explanation as to why the algorithm solves the problem. In general, such explanations are non-trivial, due to the complicated setting that the algorithms operate in. In this chapter, we leverage abstraction to provide a simple explanation of the basic ideas underlying several different consensus algorithms. Furthermore, we classify the algorithms according to how they implement these ideas.

We start with an overview of the key points of our approach. In Section 5.2, we describe the framework we use for modeling the algorithms and their abstractions. Sections 5.3–5.7 then explain the ideas behind the different classes of consensus algorithms, following the tree structure of Figure 5.1; each section covers a single class, together with a sample concrete algorithm, where applicable. In Section 5.8, we describe a new consensus algorithm to answer an open question from [CBS09]. The chapter ends with a discussion of the related work in Section 5.9.

5.1 Our Approach

In Section 1.1.2, we described the challenges in understanding consensus algorithms and their relationships. This set us on search for an approach that could provide (1) abstraction and simplification, (2) unification and classification of consensus algorithms, while also providing (3) precision and correctness guarantees.

We address these issues by combining three elements. First, we describe consensus algorithms using the stepwise refinement approach outlined in Section 2.2.3. Our abstractions describe the system using non-local steps that depend on the states of multiple processes, removing the need for communi-
5. Explaining and Unifying Consensus Algorithms

![Consensus Family Tree](image)

Figure 5.1: The consensus family tree. Boxes contain models of concrete algorithms.

cation. These abstractions allow us to focus on the main algorithmic ideas, without getting bogged down in details, thereby providing simplicity. We then gradually introduce details in successive, more concrete models that refine the abstract ones. In order to be implementable in a distributed setting, the final models must use strictly local steps, and communicate only by message passing. The simulation relations precisely describe the link between the abstract and concrete models. Furthermore, we implement the same abstract models by different concrete ones. The end result is a refinement tree of models, where branching corresponds to different implementations as illustrated in Figure 5.1. This tree captures the relationships between the different consensus algorithms, found at its leaves, providing a natural classification of the algorithms. The use of refinement thus addresses the points (1) and (2) raised above.

Second, since the concrete algorithms derived operate in communication-closed rounds (as defined in Section 4.3.2), we employ the HO model to describe these algorithms in a lockstep fashion. This simplifies our models of these algorithms, and it also further increases the abstraction level in our initial models. It allows our non-local steps to not only depend on, but also change the state of multiple processes. Hence, this choice further helps with point (1). Yet our results translate to the asynchronous setting of the real world, thanks to the preservation result established in [CSCBM09] and described in Section 4.3.3.

Finally, we formalize all the models we present in the theorem prover Isabelle/HOL [NPW02], using the Isabelle formalization of the Heard-Of model by Debrat and Merz [DM12]. We use Isabelle to prove the desired properties of our models and establish refinement relations between them without assuming any fixed bounds on either the number of processes or rounds. This provides us with strong guarantees about the precision and correctness of our results, addressing point (3). A version of the formalization is available in the Isabelle Archive of Formal Proofs (AFP) [MS15]. However, since the presentation in this chapter slightly differs from the AFP formalization, we also provide an adjusted version of the theories [Mar16].
5.2 The Specification and Verification Framework

As in Section 4.3.3, we assume a fixed set $\Pi$ of $N$ processes, a set of messages $M$, and for each $p \in \Pi$, a set $S_p$ of $p$’s local states. In the rest of the chapter, we adopt the convention that $p$ and $q$ range over $\Pi$, and $r$ over $\mathbb{N}$.

All of our models have a semantics in terms of unlabeled transition systems, described in Section 2.2. Furthermore, all of the models are round-based and lockstep, in the sense that all processes execute their local steps simultaneously. However, since we derive algorithms by stepwise refinement, we take the liberty of working with two types of lockstep models with different communication mechanisms, which give rise to two abstraction levels and associated views:

1. **global view**: lockstep models with direct access to all processes’ states. These models clearly exhibit the central ideas underlying the algorithms and simplify reasoning about them. However, implementing such models in a distributed fashion requires further refinement.

2. **local view**: lockstep models with message passing communication, including the possibility of message loss. These models can directly serve as a basis for a distributed implementation.

We specify the first type of models directly as event-based transition systems introduced in Section 2.2.1. Guards can refer to the state variables of any process. Likewise, a state update may affect any process. This type of systems is used in all non-leaf models of the tree in Figure 5.1; all of these models are closed under stuttering. For the local view (used in the leaf models) we adopt the HO model. To prove refinement between the models, we use forward simulation. When relating two event-based models, we will decompose the step proof obligation into guard strengthening and action refinement, as described in Section 2.3.

Recall from Section 4.4.1 that the consensus problem is specified by four properties: agreement, termination, non-triviality and stability. The latter two are usually straightforward to prove, and we do not discuss them further in this chapter. The difficult part is achieving both agreement and termination.

Ideally, we would like to show that our abstract models already guarantee both termination and agreement, and conclude from the refinement proof that the implementations inherit these guarantees. As discussed in Section 2.2.3, this works for unconditional properties like agreement, but not for termination, which is usually only fulfilled by a subset of traces that satisfies liveness-like communication predicates. While we will consider termination conditions informally for all of our models, we take the easy way out and prove termination individually for each concrete algorithm formulated in the HO model. Fortunately, assuming a suitable communication predicate, this is fairly simple.
5. Explaining and Unifying Consensus Algorithms

<table>
<thead>
<tr>
<th>Process</th>
<th>$HO^r_{p_i}$</th>
<th>Messages received: $\mu^r_{p_i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>${p_1, p_2, p_3}$</td>
<td>${p_1 \mapsto m_1, p_2 \mapsto m_2, p_3 \mapsto m_3}$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>${p_1, p_2}$</td>
<td>${p_1 \mapsto m_1, p_2 \mapsto m_2}$</td>
</tr>
<tr>
<td>$p_3$</td>
<td>${p_1, p_3}$</td>
<td>${p_1 \mapsto m_1, p_3 \mapsto m_3}$</td>
</tr>
</tbody>
</table>

Figure 5.2: An example of filtering by HO sets within a round, for $N = 3$. We assume that the process broadcast messages in this round, i.e., $send^r_{p_i}(s_{p_i}, \cdot) = m_i$

5.3 Voting, Quorums and Defection

All the consensus algorithms we consider share a few basic ideas. These ideas are captured by our most abstract model, which we call Voting and describe in this section. To motivate these ideas, let us first consider some other, more obvious candidate solutions to the consensus problem, and see what their shortcomings are.

The first candidate is to have all processes mutually exchange their inputs, and pick the result deterministically, for example, by taking the smallest input. Unfortunately, in the presence of failures, this scheme can violate agreement. Consider the example in Figure 5.2. The processes $p_2$ and $p_3$ do not see each other’s messages (for example, because the direct link between them is temporarily dysfunctional). The above scheme could then have them pick different values.

Another obvious candidate is to have one distinguished process, the leader, collect the inputs, pick one, and announce its decision to the others. Two-phase commit protocols are based on this idea. This guarantees agreement, but the leader is a single point of failure for termination. If it fails, there is no way of proceeding; we do not know if it decided anything, and whether it announced its decision to the other processes. Trying again, with a different leader, could violate agreement.

We thus need to revert to a decentralized approach. All the algorithms we consider achieve this by voting, based on simple counting. Each process picks a value to vote for, and announces the vote to all other processes. Processes then count the votes: if a process sees that some value received an absolute majority (more than $\frac{N}{2}$) of the votes, it decides on that value. Clearly, two different values cannot both get a majority of votes, ensuring agreement. We can also obtain the same effect by taking a slightly more abstract view, which will be useful later, where we require a value to receive votes from a quorum of processes instead of a majority. A set of processes is a quorum if it is a member of a quorum system $QS \subseteq 2^\Pi$, where, to ensure agreement, we require:

$$\forall Q, Q' \in QS. Q \cap Q' \neq \emptyset.$$  \hspace{1cm} (Q1)

In contrast to the leader-based approach, voting has no single point of
failure. However, while it is possible to terminate and reach a decision even when any non-quorum of processes fails, there are no guarantees. For example, if all the processes vote for different values, no value will receive a quorum of votes. Furthermore, even if a value does receive a quorum of votes, message loss can prevent processes from learning this. To address these problems, we iterate voting in multiple rounds, and allow processes to switch their votes between the rounds. Switching allows us to eventually form a quorum of votes for the same value within a round. Voting is iterated until such a quorum is formed, and until all the processes become aware of the quorum and decide; to simplify, we assume that it is iterated forever.

5.3.1 Formalizing Voting

We now have the basic ingredients of our most abstract model. Its system state is represented by the record:

```
record v_state =
  next_round : N
  votes : N → Π → V
  decisions : Π → V
```

where we recall that \( V \) is the set of possible input values. The fields’ names suggest their purpose:

- **next_round** is the next round to be run. It is a natural number, initially 0.
- **votes** is a (curried) function that, given a round number and a process, tells us which vote, if any, the process cast in that round. In other words, **votes** is the system’s voting history. Initially, no votes are cast.
- **decisions** records the current decision, if any, of the given process. Initially, no decisions are made.

In the rest of this chapter, we treat partial functions \( g : A \rightarrow B \) as total functions \( g : A \rightarrow B_\perp \), where \( g(x) = \bot \) if \( x \notin \text{dom}(g) \). Recall that, by the convention from Section 2.1, \( \bot \notin B \); in particular, \( \bot \notin V \). With this, we formalize the voting principle for decisions in a single round, where **r-decisions** and **r-votes** are partial functions of type \( \Pi \rightarrow V \):

```
\text{d\_guard}(\text{r-decisions}, \text{r-votes}) \triangleq \forall p. \forall v \in V.
\text{r-decisions}(p) = v \Longrightarrow \exists Q \in QS. \text{r-votes}[Q] = \{v\}.
```

A process can decide on any value \( v \) that receives a quorum of votes. We always allow the processes not to decide, even if such a \( v \) exists, to anticipate the possibility of message loss in the implementations.

The voting principle ensures agreement within a single round, but the rounds are iterated. This, together with vote switching, gives us some hope
5. Explaining and Unifying Consensus Algorithms

of achieving termination, but we must now also ensure agreement across the rounds. The basic property we must establish is that if a value receives a quorum of votes in some round, then no other value ever receives a quorum of votes in any other round. Formally:

$$\forall r, r', \forall v, v' \in V. \forall Q, Q' \in QS. \\text{votes}(r)(Q) = \{v\} \land \text{votes}(r')(Q') = \{v'\} \implies v = v'.$$

This formulation of the property, however, leaves open how to implement it. We thus replace it by a slightly stronger and more operational property: forbidding defection. That is, once a quorum for a value is formed, no process from that quorum may ever vote for any other value. To anticipate the unreliability of the distributed setting, we always allow a process not to vote, modeled as a vote for $\bot$. We formalize this as the following predicate, where $\text{v-history}: \mathbb{N} \to (\Pi \to V)$ is a voting history:

$$\text{no_defection}(\text{v-history}, r\text{-votes}, r) \triangleq \forall r' < r. \forall v \in V. \forall Q \in QS. \\text{v-history}(r')[Q] = \{v\} \implies r\text{-votes}[Q] \subseteq \{\bot, v\}.$$

We can now clearly see the tension between agreement and termination present in all voting-based consensus algorithms. To achieve termination, processes may need to switch their votes between the rounds; but in doing so, they must not defect, if agreement is to be preserved.

We now have all the ingredients for the sole event of this model: a round of voting. Its parameters are the current round $r$ and the round votes and decisions, both of type $\Pi \to V$.

Event $\text{v\_round}(r, r\text{-votes}, r\text{-decisions})$:

Guard

$$r = \text{next\_round}$$

$$\text{no_defection}(\text{votes}, r\text{-votes}, r)$$

$$\text{d\_guard}(r\text{-decisions}, r\text{-votes})$$

Action

$$\text{next\_round} := r + 1$$

$$\text{votes} := \text{votes}(r := r\text{-votes})$$

$$\text{decisions} := \text{decisions} \sqcup r\text{-decisions}$$

Recall that $g \sqcup h$ stands for the update of the partial function $g$ with the partial function $h$. The event’s guards and actions directly formalize the previous discussion. There must be no defection in the round votes, and the decisions are made by the voting principle. The next state is obtained by increasing the round, and updating the voting history and decisions.
5.3. Voting, Quorums and Defection

5.3.2 Voting ensures agreement

We have seen a formal definition of the agreement property for the HO model in Section 4.4.1. On our abstract model, the formalization is almost identical.

A trace $\tau$ over states of type $\text{v.state}$ satisfies agreement if:

$$\forall i, j \in \text{dom}(\tau). \forall p, q. \forall v, w \in V. \tau(i).\text{decisions}(p) = v \land \tau(j).\text{decisions}(q) = w \implies v = w.$$  

Here $s.\text{decisions}$ denotes the value of the field $\text{decisions}$ in the state $s$.

Proving agreement for the Voting model is straightforward. The $\text{d.guard}$ predicate, combined with the quorum property (Q1), ensures that agreement is preserved within a round. The $\text{no_defection}$ guard ensures it across the different rounds.

The agreement property will be inherited by all the subsequent models, since we will prove that they refine Voting. As discussed in Section 5.2, this is not the case for termination, and we impose no termination conditions on the Voting model. We do, however, informally discuss termination next.

5.3.3 Towards an implementation

The Voting model uses a global view of the system. To make the model implementable, we must reconstruct this view using a combination of strictly local process actions and communication. More concretely, the processes need to exchange their votes and voting histories using messages. However, due to failures, the view that a process can reconstruct this way might be only partial, reflected in the filtering by HO sets (Figure 5.2). As an intermediate informal step towards an implementation, we now consider what happens if we try to perform the steps globally, but based on a partial view.

Consider the scenario shown in Figure 5.3 where, after one round of voting, the votes of processes $p_1$–$p_4$ are visible to us, but the vote of process $p_5$ is not. The example demonstrates a vote split, with $p_1$ and $p_2$ voting 0, and with $p_3$ and $p_4$ voting 1. Define quorums to be simple majorities. As neither 0 nor 1 receives a quorum of visible votes, we cannot make a decision based on these votes. We could make one in the next round by changing some of the votes. However, we cannot distinguish between the following three possibilities:

1. The process $p_5$ voted 0, forming a quorum of three votes for 0. Although this quorum is not visible to us, it does exist, and we must not change the votes of the processes voting for 0 if we are to preserve the no defection property. Hence we should change the votes of the processes voting for 1 (to 0).

2. This case is the same as the last, but swapping 0 and 1.
5. Explaining and Unifying Consensus Algorithms

<table>
<thead>
<tr>
<th>Process</th>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>( p_3 )</th>
<th>( p_4 )</th>
<th>( p_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vote</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>?</td>
</tr>
</tbody>
</table>

Figure 5.3: A possible partial view of histories after 1 round of voting

3. The process \( p_5 \) did not vote at all, or it voted for some value other than 0 or 1. In this case, we may freely change the votes of the other processes.

The partial information we receive is thus ambiguous and prevents us from achieving termination while preserving safety. We have two options: either (1) remove the ambiguity so that we can change the votes of some of the processes or (2) prevent the situation from occurring in the first place. These options represent the next design choice (after using voting), and correspond to the two branches from the root in Figure 5.1.

5.4 Fast Consensus: Enlarging Quorums

The failed example from Figure 5.3 is suggestive. Both the votes for 0 and the votes for 1 could be extended to a quorum of three votes by including the vote of the fifth process. But there is an easy way to prevent the confusion: require four votes to reach a decision instead of three. In other words, we change the definition of a quorum to mean all sets of size four or larger, instead of simple majorities. It is not hard to see that, for any split of the four visible votes, this enables us to determine at least one vote that we can safely change.

This solution works for the concrete problem above, but how do we generalize it? First, note that this solution does not work when our partial view includes only three or fewer processes. Thus, we assume a lower bound on the number of visible processes. More precisely, we assume a set of guaranteed visible sets of processes. The assumption is that eventually at least one such set will be visible. In the implementations, the guaranteed visible sets will be realized by guaranteed HO sets, where the guarantee comes from the communication predicates (such as \( \exists r. \phi_{th}(c,r) \), where \( \phi_{th} \) is as defined in Section 4.3.3) and thus relies on network and failure assumptions. Once such a set is visible, we should be able to change the votes and make progress.

What was blocking this progress in Figure 5.3 was a partitioning of the visible set \( S = \{ p_1, \ldots, p_4 \} \) into two sets of processes \( S_0 = \{ p_1, p_2 \} \) and \( S_1 = \{ p_3, p_4 \} \), respectively voting for 0 and 1, such that:

\[
S_0 \cup \overline{S} \in QS \land S_1 \cup \overline{S} \in QS,
\]

where \( \overline{S} \) is \( S \)'s complement and \( QS \) is the quorum system. That is, both \( S_0 \) and \( S_1 \) could have been extended to a quorum using the invisible votes.
Since \( S_0 \cap S_1 = \emptyset \), set theory gives us:
\[
S_0 \cup \overline{S} \in QS \land S_1 \cup \overline{S} \in QS
\implies \exists Q_0, Q_1 \in QS, \ Q_0 \cap Q_1 \subseteq \overline{S}
\iff \exists Q_0, Q_1 \in QS, \ Q_0 \cap Q_1 \cap S = \emptyset.
\]

This leads to an obvious strengthening of (Q1), requiring that for all quorums \( Q \) and \( Q' \), and all guaranteed visible sets \( S \):
\[
Q \cap Q' \cap S \neq \emptyset.
\] (Q2)

This ensures that, in case of a vote split, only one subset of votes from a guaranteed visible set can be extended to a quorum. Thus, we can switch all the other ones. To ensure that we can also decide based on any guaranteed visible set, we also stipulate that for every such set \( S \), there exists a quorum \( Q \) such that:
\[
Q \subseteq S.
\] (Q3)

Interestingly, conditions (Q2) and (Q3) define a dissemination quorum if we interpret \( S \) as a set of Byzantine processes [MR98]. As these conditions strengthen (Q1), the Voting model will still guarantee agreement under them. They also allow us to achieve termination, but the termination guarantees might not be inherited by refinement, and hence we do not formalize this discussion in an abstract model. We will explain how it is reflected in the concrete algorithms later. Moreover, all these algorithms employ an optimization to the Voting model that avoids exchanging the entire voting histories. We formalize and describe this optimization next.

### 5.4.1 Optimizing Voting

This optimization is based on two observations. First, a process can clearly never defect by repeating its last non-\( \bot \) vote. Second, when changing its vote, it is enough to check for defection against the last non-\( \bot \) votes of the other processes, rather than checking against their entire voting histories. That is, if a process is not defecting with respect to the last votes of the other processes, it will not defect with respect to the votes in any previous round. To see why, assume that in round \( r \) a quorum \( Q \) of processes all voted for a value \( v \). By the no defection property, in rounds between \( r \) and the current round, no process \( p \) in \( Q \) can change its vote to one different from \( v \). Therefore, \( p \)'s last non-\( \bot \) vote must remain \( v \). So if \( Q \) was a quorum of processes voting for \( v \) in round \( r \), after this round, all members of \( Q \) will always retain \( v \) as their last vote.

The state of the system is thus changed to record just the last non-\( \bot \) vote of each process instead of the entire voting history. We still use \( \bot \) to denote that a process never voted.
5. Explaining and Unifying Consensus Algorithms

1: **Initially**: $\text{inp}_p$ is $p$’s input value,
2: $\text{dec}_p = \perp$
3: $\text{send}_p^r$
4: send $\text{inp}_p$ to all
5: $\text{next}_p^r$
6: if $|H_{OP}^p| > \frac{2n}{3}$ and all received messages equal some $v$ then
7: $\text{dec}_p := v$
8: if $|H_{OP}^p| > \frac{2n}{3}$ then
9: $\text{inp}_p := \text{smallest most often received value}$

Figure 5.4: The HO model of the OneThirdRule algorithm

```
record opt_v_state =
    next_round : $\mathbb{N}$
    last_vote : $\Pi \rightarrow V$
    decisions : $\Pi \rightarrow V$
```

The guard for checking defection now looks at just the last votes $lvs : \Pi \rightarrow V$ instead of the entire voting history:

$$\text{opt_no_defection}(lvs, r-votes) \triangleq \forall v \in V, \forall Q \in QS. \ lvs[Q] = \{v\} \Rightarrow r-votes[Q] \subseteq \{\perp, v\}.$$ 

The voting round is then modified as follows:

**Event** $\text{opt_v_round}(r, r-votes, r-decisions)$:

**Guard**

- $r = \text{next_round}$
- $\text{opt_no_defection}(\text{last_vote}, \text{r-votes})$
- $\text{d_guard}(\text{r-decisions}, \text{r-votes})$

**Action**

- $\text{next_round} := r + 1$
- $\text{last_vote} := \text{last_vote} \triangleleft \text{r-votes}$
- $\text{decisions} := \text{decisions} \triangleleft \text{r-decisions}$

5.4.2 Implementations: fast consensus

The optimized model abstracts several variants of consensus algorithms found in the literature. We prove that it is refined by the OneThirdRule [CBS09] algorithm shown in Chapter 4. It is also refined by the generalization of OneThirdRule called $A_{T,E}$ [BWCB+07] (assuming no Byzantine processes). Moreover, it also describes the algorithms used in the first round of the protocol from [BGMR01] and in the fast rounds of Fast Paxos [Lam06]. For ease of reference, we again show the HO model of OneThirdRule in Figure 5.4.
In OneThirdRule, quorums are sets of more than \( \frac{2n}{3} \) processes. Hence, the decision rule in lines 7–8 ensures the \texttt{d\_guard} from the (optimized) Voting model, where the refinement relation relates the state variables of each process \( p \) to the \( p \)-values of the fields in the abstract model as follows: the \texttt{opt\_no\_defection} guard holds for \texttt{last\_vote} and \texttt{inp}, and \texttt{dec}_p equals \texttt{decisions}(p).

Given the new quorum size, defining guaranteed visible sets to also be all sets of size greater than \( \frac{2n}{3} \) ensures conditions (Q2) and (Q3). As OneThirdRule is a concrete, fully distributed algorithm, we replace guaranteed visible sets by guaranteed HO sets. This is reflected in the communication predicate required for termination, which we already encountered in Section 4.3.3:

\[
\exists r_1, r_2 > r_1. \phi_{th}(\frac{2}{3}, r_1) \land \phi_{uf}(r_1) \land \phi_{th}(\frac{2}{3}, r_2).
\]

The most interesting part of the algorithm are the lines 9–10, which guarantee no defection and at the same time direct the votes such that they eventually converge to a common value. By (Q2) and the cardinality condition on the HO set (line 9), we know that only one received value could have been voted for by a quorum; the greater than \( \frac{2n}{3} \) requirement on quorums and HO sets ensures that it is the one that received the most votes. If there is a tie in the number of votes, no value could have received a quorum of votes, and processes may switch their votes freely. Either way, no process will defect by choosing a value that received the most votes. Choosing the smallest such value provides the required vote convergence. The communication predicate ensures the existence of a round \( r_1 \) in which all processes adopt the same vote (due to \( \phi_{uf} \) and \( \phi_{th}(\frac{2}{3}) \)), and of a later round \( r_2 \) in which the processes receive enough votes to decide and terminate.

In OneThirdRule, a round of voting requires one round of communication. This also applies to the other algorithms of this type, earning them the name Fast Consensus. If all the processes start with the same value \( v \), the algorithm can terminate within a single failure-free round. Otherwise, the algorithm still terminates within two rounds that satisfy the above communication predicate. The speed comes at a price though. The communication predicate of OneThirdRule requires the existence of rounds where \( \phi_{th}(\frac{2}{3}) \) holds. Hence \( n_{\text{fail}} < \frac{n}{3} \), where \( n_{\text{fail}} \) is, as before, the number of tolerated process failures. It is not difficult to see that this is optimal, given conditions (Q2) and (Q3). It is, however, possible to implement the Voting model without the additional requirements on quorum sets; this will only require \( n_{\text{fail}} < \frac{n}{2} \). We will show how to do this in the next section. The price paid is that the algorithms become more complicated and require multiple communication steps to perform one round of voting.
5.5 Same Vote

Fast Consensus resolved the situation from Figure 5.3 by disambiguating the vote split. In this section, we take the other approach, corresponding to the other branch from Figure 5.1: we prevent the split from ever happening, thus eliminating the problematic example completely. Banning vote splits immediately implies that all the votes cast within a round must be the same. We can still allow the possibility that some processes do not cast a vote. Formally, we will replace the Voting round event $v\_round(r, r\text{-}votes, r\text{-}decisions)$ by a Same Vote round $sv\_round(r, S, v, r\text{-}decisions)$, where the processes in $S$ vote for a value $v \in V$, and the others vote $\bot$.

This requires vote agreement: all processes must agree on the value of $v$. But this seems like a paradoxical, circular way to solve consensus: having all processes agree on a single value is exactly what consensus is about! There is, however, a subtle difference between vote agreement and consensus. Because we allow processes to vote $\bot$, unlike for consensus, it is not necessary that every process gets an (non-$\bot$) output from vote agreement. Thus, vote agreement does not share the termination requirement of consensus. However, to make progress, we cannot drop this requirement completely, but we instead relax it: we require that enough processes get an output in some round. The relaxed termination requirement is now collective. Each voting round contains one instance of vote agreement; it is not necessary that all instances terminate, but at least one must do so. Moreover, the outcomes of the different vote agreement instances are independent; we do not require that they match.

The consequence of the laxer termination requirements is that some of the ideas that we described at the start of Section 5.3, and which failed to solve consensus because of their weak termination properties, can now be recycled to successfully solve the vote agreement problem. The two ideas are non-iterated voting, which we will henceforth refer to as simple voting, and the leader-based approach. Before we put either of them to use in the implementations, we must ensure (and this is the tricky part of the algorithms) that any agreed upon vote preserves the no defection property of the Voting model.

5.5.1 Formalizing Same Vote

The system state remains the same as in the Voting model. As before, we require that there is no defection in the votes. Since each process will now vote for either $v$ or $\bot$, voting for $v$ must not cause any process to defect; we say that $v$ must be safe. If there previously existed a quorum for a value $w$, we must have $v = w$. Otherwise, the processes that previously voted for $w$
could defect by voting for $v$. Formally:

$$\text{safe}(v\text{-hist}, r, v) \triangleq \forall w. \forall Q \in QS. v\text{-hist}(r)[Q] = \{v\} \implies v = w.$$ 

In a Same Vote round $r$, the processes in some set $S$ receive an output $v$ from vote agreement and vote for $v$, while the others vote for $\perp$. If $S = \emptyset$ then $v$ is unused and unconstrained, otherwise it must be safe. Formally:

**Event sv\_round($r, S, v, r\text{-decisions}$):**

**Guard**

$r = \text{next\_round}$

$S \neq \emptyset \implies \text{safe}(\text{votes}, r, v)$

$d\_guard(r\text{-decisions}, [S \mapsto v])$

**Action**

next\_round := $r + 1$

votes := $\text{votes}(r := [S \mapsto v])$

decisions := $\text{decisions} \triangleleft r\text{-decisions}$

Note that since in this chapter we identify partial functions to $\mathcal{V}$ with total functions to $\mathcal{V} \perp$, $[S \mapsto v]$ maps all processes from $S$ to $v$, and the others to $\perp$. The refinement relation between Voting and Same Vote is just the identity. The refinement proof hinges on the fact that $\text{safe}$ implies $\text{no\_defection}$ with $r\text{-votes} = [S \mapsto v]$.

### 5.5.2 Towards an implementation

As a step towards an implementation of the Same Vote model in a distributed setting, we again look at some possible scenarios with only partial information. As we wish to improve the fault tolerance to $n_{\text{fail}} < \frac{n}{2}$ (process failures (over $\frac{3}{4}$ for Fast Consensus)), we restrict our view to just over $\frac{n}{2}$ processes.

It was the combination of partial information and vote splits that prevented us from changing the votes without causing defection in the example of Figure 5.3. That particular situation is now eliminated, as vote agreement ensures that such vote splits within a single round can no longer occur. However, vote agreement prevents neither vote splits across multiple rounds, nor hiding of quorums by a partial view. Consider the example in Figure 5.5: it is not obvious which values are safe for round 3. A priori, it may be that 0 received a quorum of votes in round 0 (if process $p_4$ or $p_5$ voted for 0), or that 1 received a quorum in round 1 (if $p_4$ and $p_5$ both voted for 1), resembling the ambiguity present in the Voting model and Figure 5.3. However, the situation can be resolved, and the next two sections describe two ways to do so. They correspond to the two branches from the Same Vote model in Figure 5.1.
5. Explaining and Unifying Consensus Algorithms

Figure 5.5: Same Voting: a possible partial view of histories after three voting rounds.

5.6 Observing Quorums

Figure 5.5 demonstrates the difficulty of detecting vote quorums and finding safe values based on a partial view of the voting history. The main idea behind the solution introduced in this section is that each process maintains a vote candidate value \( v \in V \) that is safe to vote for by construction. Maintaining the candidates’ safety requires each process to detect (with potential false positives) when a quorum of votes is formed for some value. For this, each process must observe the votes of the other processes. We now describe this scheme in more detail.

Initially, all values are safe. Thus, processes can initialize their candidates to arbitrary values; in particular, they can use their input values. Furthermore, all values will remain safe until the first time a quorum is formed for some value, at which point all processes must update their candidates to this value. To ensure that this happens, we require each process \( p \) to try to update its candidate in every round, based on the votes it observes in the round. More precisely, consider an arbitrary process \( p \) and a round \( r \). Due to the Same Vote principle, there is some value \( v \in V \) such that each vote cast in \( r \) is either for \( v \) or for \( \bot \). We say that process \( p \)'s observation in round \( r \) is \( v \) if \( p \) receives a vote for \( v \) from at least one process in \( r \) and \( \bot \) if it receives only votes for \( \bot \). If \( p \) observes \( v \) (i.e., not \( \bot \)) then it updates its candidate to \( v \).

Assume now that \( r \) is in fact the first round in which a quorum of votes for some value \( v \) is formed. If \( p \) observes \( v \), it will update its candidate to \( v \), and safety will be guaranteed. However, if \( p \) observes \( \bot \), it fails to update the candidate, which may violate safety. To avoid this possibility, we require that \( p \) waits to receive votes from some quorum \( Q \) of processes before it makes its observation and moves on to the next round. By (Q1), \( Q \) intersects with the set of processes voting for \( v \), which ensures that \( p \) will receive at least one vote for \( v \), and thus update its candidate to \( v \). Thus, after round \( r \), the candidates of all processes will become \( v \). Since we assume that the votes are always selected from the set of candidates, \( v \) is the only value that can be voted for, and hence observed after this point. Further updates based on observations will thus not change the candidates and therefore preserve
5.6. Observing Quorums

As an example, interpret Figure 5.5 as if it were showing the observations that the processes make in each round, instead of the votes they cast. The candidates after round 2 are:

\[[p_1 \mapsto 0, p_2 \mapsto 0, p_3 \mapsto 1, p_4 \mapsto ?, p_5 \mapsto ?]\]

that is, processes \( p_1 \) and \( p_2 \)'s candidate is 0 and \( p_3 \)'s candidate is 1, while \( p_4 \) and \( p_5 \)'s candidates are unknown. We immediately see that both 0 and 1 are safe for round 3, as they are among the candidates. Moreover, we can even conclude that all values are safe. Otherwise, the set of candidates would be a singleton, containing only the unique value that has received a vote quorum.

5.6.1 Formalizing Observing Quorums

First, we extend the state record \( v_{state} \) with following field to record the processes’ candidates:

\[
cand : \Pi \rightarrow \mathcal{V}.
\]

The safety of a new vote \( v \) is now determined based on the candidates. With \( cs : \Pi \rightarrow \mathcal{V} \) we define this as follows:

\[
cand_safe(cs, v) \triangleq v \in \text{ran}(cs).
\]

We represent the observations made in each round by a partial function \( obs : \Pi \rightarrow \mathcal{V} \). According to the discussion above, \( obs \) is of the form \([OS \mapsto v]\), where \( v \) is the round vote, and \( OS \) is the set of processes observing \( v \). If \( v \) receives a quorum of votes, we require \( OS = \Pi \). We can however generalize this by allowing processes to observe not only votes but also each other’s candidate values, i.e., we only require \( \text{ran}(obs) \subseteq \text{ran}(cand) \). From the previous discussion we know that all old candidate values remain safe if \( v \) does not receive a quorum of votes. Otherwise, we still require \( obs = [\Pi \mapsto v] \). This adoption of others’ candidates will prove useful for termination. We formalize these considerations in the following round event.

Event \text{obsv\_round}(r, S, v, r\_decisions, obs):

Guard

\[
\begin{align*}
\text{Guard} & \quad r = \text{next\_round} \\
& \quad S \neq \emptyset \Rightarrow \text{cand\_safe(cand, v)} \\
& \quad \text{ran}(obs) \subseteq \text{ran(cand)} \\
& \quad S \in QS \Rightarrow obs = [\Pi \mapsto v] \\
& \quad \text{d\_guard}(r\_decisions, [S \mapsto v])
\end{align*}
\]

Action

\[
\begin{align*}
\text{Action} & \quad \text{next\_round} := r + 1 \\
& \quad \text{cand} := \text{cand} \triangleleft obs \\
& \quad \text{decisions} := \text{decisions} \triangleleft r\_decisions
\end{align*}
\]
5. Explaining and Unifying Consensus Algorithms

The guard $S \in QS \implies obs = [\Pi \mapsto v]$ ensures that quorums of votes are reflected in all processes’ observations. Since no guard consults the voting history and only the current round’s votes are needed to make decisions, there is, in contrast to the Same Vote model, no need to record votes. We therefore drop the field votes from the state.

The refinement relation between the Observing Quorums and Same Vote models relates the fields votes in Same Vote and cand in Observing Quorums by requiring that

$$\text{votes}(r)[Q] = \{v\} \implies \text{cand} = [\Pi \mapsto v]$$

holds for all values $v \in V$, quorums $Q \in QS$, and rounds $r$ preceding the current one. The common fields next_round and decisions are related by the identity. Based on this relation, we can prove that cand_safe(cand, v) implies safe(votes, r, v).

5.6.2 Implementing Observing Quorums

The previous model captures several algorithms from the literature. We prove that it is refined by Ben-Or’s algorithm [BO83] and the UniformVoting algorithm [CBS09]. It also captures the generic algorithm from [GR04], although we do not formally prove this. The model, however, only tells us how to pick safe values in each round. In the implementation, the processes must use a vote agreement scheme to agree on one such value. We have already mentioned two candidate schemes: the leader-based scheme and simple voting. Either can be used here. As an example, we show the UniformVoting algorithm (Figure 5.6), which uses simple voting.

In the algorithm, a round of voting requires two rounds of communication. To be consistent with the HO model nomenclature [CBS09], we call the communication rounds just “rounds” in the concrete model, and we call the collection of such rounds that correspond to an abstract model round a phase.\footnote{“Round” and “phase” are also used in this sense in [Lyn96]; unfortunately, their meanings are inverted in some other works, e.g. [Lam01a, CGR11].} To clearly disambiguate between rounds of the abstract model and the multiple communication rounds which implement them, we refer to the communication rounds as sub-rounds in this chapter.

Vote agreement takes place in the first sub-round, while casting and observing votes take place in the second sub-round. Each process $p$ stores its candidate in the inp$_p$ field; the candidates are the inputs to vote agreement (line 6); picking any one of them will satisfy the cand_safe guard. The output, recorded in the variable vote$_p$, is generated by simple voting, and corresponds to the parameter $v$ of the abstract model’s event obsv_round.

The voting principle of simple voting is encoded in the combination of the check in line 10 and the assumed communication predicate $\forall r. \phi_{th}(\frac{1}{2}, r)$, with
5.6. Observing Quorums

1: Initially: \( \text{inp}_p \) is \( p \)'s input value
2: other fields are \( \bot \)
3:
4: Round \( r = 2\phi \): // vote agreement
5: send\( \text{inp}_p \) to all
6:
7: next\( \text{r}_p \):
8: \( \text{inp}_p := \) smallest value received
9: if all the values received equal \( v \) then
10: \( \text{vote}_p := v \)
11: else
12: \( \text{vote}_p := \bot \)
13: Round \( r = 2\phi + 1 \): // casting and observing votes
14: send\( \text{inp}_p, \text{vote}_p \) to all
15:
16: next\( \text{r}_p \):
17: if at least one \((\ _, v) \) with \( v \neq \bot \) received then
18: \( \text{inp}_p := v \)
19: else
20: \( \text{inp}_p := \) smallest \( w \) from \((w, \bot) \) received
21: if all received equal \((\ _, v) \) for \( v \neq \bot \) then
22: \( \text{dec}_p := v \)
23: else
24: \( \text{dec}_p := \bot \)

Figure 5.6: The HO model of UniformVoting

\( \phi_{th} \) as defined in Section 4.3.3. The same predicate is used in the second sub-round to ensure the guards \( S \in QS \Rightarrow obs = [\Pi \mapsto v] \) (lines 19–20) and \( d\_guard \) (lines 23–24).

Processes update their candidates either to the round vote (line 20) or to a candidate of some other process (lines 9 and 22). This corresponds to making a non-\( \bot \) observation and satisfies the guard \( ran(obs) \subseteq ran(cand) \). Moreover, the adoption of other processes’ candidates helps the convergence to a common vote candidate, necessary for termination of vote agreement. This termination is guaranteed by the additional communication predicate \( \exists r \phi_{uf}(r) \). In the sub-round satisfying \( \phi_{uf}(r) \), all processes will adopt the same candidate, and thus agree on a vote (line 11) and decide (line 24).

The refinement relation relates by equality, for each \( p \), the value \( \text{inp}_p \) to the value of \( \text{cand}(p) \) in the abstract model and the value of \( \text{dec}_p \) to \( \text{decisions}(p) \). The refinement proof follows the above remarks. The first sub-round refines the \text{skip} event, while the second sub-round refines the \text{obsv\_round} event.
The algorithms in this section tolerate $n_{\text{fail}} < \frac{n}{2}$ process failures and thus exhibit better fault tolerance than the fast consensus algorithms from Section 5.4.2. Depending on the scheme used for vote agreement, they can also terminate within two fault-free communication rounds, as shown in [GR04]. However, as discussed in Section 4.3.3, the use of waiting (e.g., visible in the communication predicate of UniformVoting) requires a more complicated communication layer, since retransmission is necessary, and it hides the fact that additional messages are required for acknowledgments. The same level of fault tolerance can be achieved without waiting, as the algorithms in the next section show.

### 5.7 Most Recently Used Vote

In the algorithms of the previous section, processes maintain a safe vote candidate at every point in time. In this section, we show how to generate such candidates only when they are needed, based on just partial views of voting histories and without resorting to waiting.

Going back to the definition of a Same Voting round, we observe that any state of Same Voting satisfies the invariant:

\[ \text{votes}(r, p) = v \implies \text{safe}(\text{votes}, r, v) \]

for any $v \in \mathcal{V}$. Moreover, such a $v$ is the only value receiving votes in round $r$. Hence, no other value can receive a quorum of votes in $r$, and $v$ is also safe in round $r + 1$. Formally:

\[ \text{votes}(r, p) = v \implies \text{safe}(\text{votes}, r + 1, v). \]

Returning to the example from Figure 5.5, by the above we conclude that the value 1 is safe in round 2. Moreover, we see a quorum of $\bot$ votes in round 2. By the intersection property (Q1), no value whatsoever could have received a quorum of votes in that round; value 1 is therefore still safe in round 3, and can be chosen as the next vote. In this way we have in fact generated, on the fly, the same candidate that a hypothetical process running the Observing Quorums scheme would do, if its observations were based on the votes shown in Figure 5.5.

It is straightforward to generalize this solution. If we see the voting history of a quorum $Q$ after multiple rounds, the most recently used (MRU) vote will still be safe. This value is in fact unique, since all votes cast within the same round are the same. If nobody in $Q$ ever voted, we define the MRU vote to be $\bot$. In this case, by (Q1) no value ever received a quorum of votes
and all values are safe. Formally:

\[
\text{the\_mru\_vote}(v\text{-hist}, Q) = \begin{cases} \\
\bot & \text{if } \forall r. v\text{-hist}(r)[Q] = \{\bot\} \\
v & \text{otherwise, where} \\
  r = \max \{r \mid v\text{-hist}(r)[Q] \neq \{\bot\}\} \\
v = \epsilon w. w \in v\text{-hist}(r)[Q] \cap V
\end{cases}
\]

where \(v\text{-hist}\) is a voting history and \(Q\) is a quorum. We then say that \(Q\) is an MRU guard for \(v\) given \(v\text{-hist}\) if:

\[
mru\_guard(v\text{-hist}, Q, v) \triangleq Q \in QS \land \text{the\_mru\_vote}(v\text{-hist}, Q) \in \{\bot, v\}.
\]

Following the discussion above, we prove that:

\[
mru\_guard(votes, Q, v) \implies safe(votes, next\_round, v),
\]

for any \(Q \in QS\) and \(v \in V\). Replacing \(safe\) with \(mru\_guard\) in the event \(sv\_round\) thus yields a correct refinement of Same Voting. As the MRU scheme works even with just partial information, we are now ready to move to a distributed implementation.

### 5.7.1 Optimizing MRU Vote

Like Fast Consensus, the MRU scheme can also be optimized to avoid transmitting the entire voting histories of all the processes. The histories were only used in the \(mru\_guard\), to determine the MRU vote of a quorum of processes. This can obviously also be done by just looking at the MRU vote of each individual process in the quorum, together with its associated \(timestamp\), the round number of its last update. The optimized state of the system is thus:

\[
\text{record opt\_v\_state =} \\
  \text{next\_round : } \mathbb{N} \\
  \text{mru\_vote : } \Pi \rightarrow (\mathbb{N} \times V) \\
  \text{decisions : } \Pi \rightarrow V
\]

The guard is changed in the obvious fashion. It now takes a parameter \(mrus : \Pi \rightarrow (\mathbb{N} \times V)\):

\[
\text{opt\_mru\_guard}(mrus, Q, v) \triangleq Q \in QS \land \text{opt\_mru\_vote}(mrus[Q]) \in \{\bot, v\}.
\]

The definition of \(opt\_mru\_vote\) is straightforward but tedious, so we do not show it here. The voting round is changed as expected, and the refinement proof is straightforward.
5. Explaining and Unifying Consensus Algorithms

**Event** `opt_mru_round(r, S, v, Q, r-decisions)`:

**Guard**
- `r = next_round`
- `S ≠ ∅`\(\implies\) `opt_mru_guard(mru_vote, Q, v)`
- `d_guard(r-decisions, [S \mapsto v])`

**Action**
- `next_round := r + 1`
- `mru_vote := mru_vote \triangleleft [S \mapsto (r, v)]`
- `decisions := decisions \triangleleft r-decisions`

This gives us a method of picking candidates. The processes must also agree on exactly one such candidate, and, as in the previous section, we must choose a vote agreement scheme. The Paxos [Lam98] and Chandra-Toueg [CT96] algorithms opt for a leader-based scheme. We show the Paxos algorithm next.

### 5.7.2 Implementing MRU Vote

Figure 5.7 shows a HO model of the Paxos algorithm. We model the variant of Paxos where a voting round is split into three sub-rounds, but the version with four sub-rounds also refines the Optimized MRU Vote model.

The algorithm relies on an a priori known leader for each phase (see Section 4.3.3). In the first sub-round, each process sends its MRU vote to the leader; the leader then finds the MRU value over an entire quorum of processes (Lines 9-10). Here, we split the `mru_vote` field into a timestamp `ts` and a value `cand`, and use a modeling trick due to Charron-Bost and Schiper [CBS09], which simplifies the formalization: the timestamp 0 denotes that the process never voted (corresponding to the \(\bot\) value of `mru_vote`). This simplifies the leader’s rule for picking a safe candidate value, compared to the definition of `opt_mru_vote`; the rule now chooses an arbitrary element with the lowest timestamp (denoted by Hilbert’s choice operator in Line 10). If the leader fails to receive sufficiently many messages, it sets its candidate to \(\bot\). Based on the leader’s candidate, the second sub-round implements leader-based vote agreement. To compensate for the MRU modeling trick above, the processes for which vote agreement succeeds (i.e., the condition in Line 17 holds) must set their `ts` fields to a non-zero value; this explains the addition of 1 to the phase number in Line 18. The last sub-round then deals with ordinary voting, with the decision rule in lines 26–27 implementing the `d_guard`.

The refinement relation equates the `dec_p` variable of each process `p` with the `decision(p)` field of the abstract model. The `mru_vote(p)` of the abstract model is \(\bot\) when `ts_p = 0`, and `(ts_p - 1, inp_p)` otherwise. The termination
5.8 A New Algorithm

In [CBS09], Charron-Bost and Schiper posed the question whether there exists a leaderless consensus algorithm tolerating $n_{fail} < \frac{n}{2}$ failures, whose proof is straightforward, once we add the communication predicate:

$$\exists \phi. \phi_{lr}(\frac{1}{2} \cdot 3\phi) \land \phi_{ts}(3\phi + 1) \land \phi_{th}(\frac{1}{2} \cdot 3\phi + 2),$$

recalling the definitions ($\phi_{lr}$) and ($\phi_{ts}$) from Section 4.3.3. The $\phi_{lr}$ and $\phi_{ts}$ formulas guarantee a stable leader in the phase $\phi$, which causes the vote agreement to succeed. Moreover, the voting itself must also succeed within the same phase, as reflected in the $\phi_{th}$ formula of the predicate.

Figure 5.7: The HO model of Paxos

1: Initially: $ts_p$ is 0, $inp_p$ is $p$’s input value, other fields are ⊥
2:
3: Round $r = 3\phi$: // finding a safe vote candidate
4: send $r_p$:
5: send ($ts_p$, $inp_p$) to the phase leader
6:
7: next $r_p$: // only executed if $p$ is the phase leader
8: if $|HO_p^r| > \frac{n}{2}$ then
9: let max-ts = max $\{t \mid \exists v. (t, v) \text{ received}\}$
10: cand$_p$ := ev. (max-ts, v) received
11: else cand$_p$ := ⊥
12: Round $r = 3\phi + 1$: // vote agreement
13: send $r_p$: // only executed if $p$ is the phase leader
14: send cand$_p$ to all
15:
16: next $r_p$:
17: if ldr($r$) $\in HO_p^r$ and $\mu_p^r(\text{ldr}(r)) = v \neq ⊥$ then
18: $ts_p$ := $\phi + 1$
19: vote$_p$ := $inp_p$ := $v$
20: else vote$_p$ := ⊥
21: Round $r = 3\phi + 2$: // voting proper
22: send $r_p$:
23: send vote$_p$ to all
24:
25: next $r_p$:
26: if received some $v \neq ⊥$ more than $\frac{n}{2}$ times then
27: dec$_p$ := the received message
5. Explaining and Unifying Consensus Algorithms

1. **Initially:** $ts$ is 0, $\text{inp}_p$ is $p$’s input value, other fields are $\perp$.

2. **Round** $r = 3\phi$:  // *finding safe vote candidates*
3. send$_p^r$:
4. send $(ts, \text{inp}_p)$ to all
5. next$_p^r$:
6. if $\lfloor HO_p r \rfloor > \frac{n}{2}$ then
7. let $max-ts = \max \{t | \exists v. (t, v) \text{ received}\}$
8. cand$_p := \min \{v. (max-ts, v) \text{ received}\}$
9. else cand$_p := \perp$
10. **Round** $r = 3\phi + 1$:  // *vote agreement*
11. send$_p^r$:
12. send cand$_p$ to all
13. next$_p^r$:
14. if received $> \frac{n}{2}$ non-$\perp$ values, and all equal some $v$ then
15. $ts_p := \phi + 1$
16. $\text{vote}_p := \text{inp}_p := v$
17. else $\text{vote}_p := \perp$
18. **Round** $r = 3\phi + 2$:  // *voting proper*
19. send$_p^r$:
20. send $\text{vote}_p$ to all
21. next$_p^r$:
22. if received some $v \neq \perp$ more than $\frac{n}{2}$ times then
23. $\text{dec}_p := v$

Figure 5.8: The HO model of the New Algorithm

Safety does not depend on waiting (and, more generally, without an “invariant” communication predicate of the form $\forall r. P(HO_p r)$ for some predicate $P$). Leaderless algorithms are useful in settings with no stable leader and can decrease latency [Lam06], while the drawbacks of waiting were already mentioned in Section 5.6.2.

The classification we propose in this chapter provides us with sufficient guidance to find such an algorithm. The requirement to tolerate $n_{\text{fail}} < \frac{n}{2}$ failures disqualifies the Fast Consensus algorithms (which handle only less than $\frac{n}{3}$ failures). We thus turn to the Same Vote mechanism. Since the Observing Quorums model requires waiting in order to provide safety, we are left with the MRU model as the only candidate in the hierarchy. The only remaining question is whether we need a leader. Since we only use leaders for vote agreement, the answer is no: we can implement vote agreement using
5.9. Related Work

The technique of stepwise refinement is well-known in the distributed systems community, and has already been successfully applied to consensus algorithms before. Lampson [Lam01b] derives three variants of Paxos from an abstract model, with a manual proof of invariants and refinement. Notably, his models were used as blueprints for the consensus portion of the
Zab atomic broadcast algorithm [JRS11]. Lamport [Lam11] develops two abstract models into a variant of Paxos, which he then refines into a Byzantine variant of Paxos. All models are formalized in TLA [Lam02], as are most of the proofs. Bryans formally develops the Floodset algorithm in Event-B [Bry11]. The algorithm is simpler than the ones we consider, as it uses the standard synchronous model. While van Rensess et al. [vRSS14] derive families of replication algorithms at a relatively high abstraction layer, somewhat surprisingly, the application of refinement to consensus algorithms has been limited to variants of single algorithms, and there appears to be no work that derives entire families of different consensus algorithms. Moreover, there is also no prior work that combines refinement with lockstep models. Both [Lam01b] and [Lam11] use the standard asynchronous model, which makes their models and proofs more involved than ours.

Several works [MR99, HMR02, MRR02, GR04, GR07, SRSD08, RMS10] provide generic consensus algorithms that can be instantiated with different parameters and primitives to yield known consensus algorithms. Mostéfaoui and Raynal [MR99] unify consensus algorithms with two sub-rounds and a coordinator based on Chandra and Toueg’s unreliable failure detectors $S$ and $\triangleleft S$ [CT96] in a single generic algorithm. Together with Hurfin [HMR02], they extend their generic algorithm to accommodate different message exchange patterns. In [MRR02, GR04] the work is further extended with an abstraction they call $\lambda$, that corresponds to vote agreement in our terminology. They instantiate it using combinations of $\triangleleft S$, the leader detection oracle $\Omega$ [CHT96], and the random oracle [BO83]. Guerraoui and Raynal [GR07] then propose a simple consensus algorithm that completely abstracts from the message passing communication and clearly separates termination (handled by the leader oracle $\Omega$) from safety including agreement (handled by a new abstraction, called Alpha). They give implementations based on shared registers, replicated disks, message passing, and active disks. Two papers cover both benign and Byzantine algorithms. Song et al. [SRSD08] propose a skeletal algorithm based on an extended form of quorum systems, which they instantiate to Paxos, Chandra-Toueg, and Ben-Or as examples. Rütti et al. [RMS10] propose a generic algorithm formulated in the lockstep model, consisting of a single three-round phase. They instantiate it to different algorithms, where some instantiations drop one round of the generic algorithm. Then, they identify three classes of algorithms based on their numbers of rounds and state variables and their allowed numbers of faulty and Byzantine processes.

However, with the exception of [GR07], these generic algorithms do not abstract communication away, and thus are significantly more detailed and complicated than our abstract models. Furthermore, their scope is limited. They each cover at most one of our three classes of algorithms (i.e., the leaf nodes in Figure 5.1) with the exception of [SRSD08] and [RMS10], which, when limited to benign failures, each cover algorithms from two classes. An-
other limitation of these generic algorithms is that they have limited power to explain the relationships between the different algorithms. The only classification of algorithms is offered in [RMS10]. However, it is more technical and less focused on design choices than ours. Finally, none of these works have been fully formalized, and they contain numerous parts with missing proofs or just proof sketches.
Chapter 6

Cutoff Bounds for Consensus Algorithms

In the previous chapter, we saw several examples of consensus algorithms, and we precisely showed how they implement similar basic ideas, consequently proving the algorithms’ correctness. In this chapter, we take advantage of the similarity of their implementations instead, and define ConsL, a simple language that can express many consensus algorithms for the asynchronous and partially synchronous settings with benign failures. Our main result is the proof of the small scope hypothesis for ConsL. Namely, given a ConsL algorithm $A$, we compute its cutoff bound: the number $B$ such that $A$ solves consensus for any number of processes if and only if it solves consensus for exactly $B$ processes. Moreover, the bounds we compute for realistic algorithms are indeed small.

Our proof consists of two main steps. The first step is to prove the zero-one principle for $A$. Namely, we prove that $A$ solves consensus if and only if it solves consensus when the value domain $V$ is $\{0, 1\}$. We then prove the cutoff bound for $A$ run on this binary $V$.

We start with an exposition of ConsL (Section 6.1). We motivate its design in Section 6.1.1, describe its syntax in Sections 6.1.2 and 6.1.3 and its semantics in Section 6.1.4. In Section 6.2 we show some further examples of algorithms expressed in ConsL. Then, we sketch the proofs: for the zero-one principle in Section 6.3, and for the cutoff bounds in Section 6.4. The full proofs are available in the Appendices B and C (Appendix A contains their shared prerequisites). The theorems enable the construction of our verification tool for ConsL algorithms. We briefly discuss the obtained results and the tool construction in Section 6.5. We conclude the chapter with a review of the related work in Section 6.6.
6. Cutoff Bounds for Consensus Algorithms

**Initially:** $\text{inp}_p$ is $p$’s input value,

$\text{dec}_p = \perp$

**send**$_p^r$:

send $\text{inp}_p$ to all

**next**$_p^r$:

if $|\text{HO}_p^r| > \frac{2n}{3}$ and all received messages equal some $v$ then

$\text{dec}_p := v$

if $|\text{HO}_p^r| > \frac{2n}{3}$ then

$\text{inp}_p :=$ smallest most often received value

Figure 6.1: The HO model of the OneThirdRule algorithm

6.1 The Specification Language $\text{ConsL}$

The HO model (Section 4.3.3) leverages the round structure present in many distributed algorithms to create a simple model for them. However, similarities between consensus algorithms for the asynchronous setting with benign failures run deeper than just their round structure. In this section, we exploit these similarities to define $\text{ConsL}$, a simple language that captures many algorithms for this setting.

6.1.1 Structural commonalities between algorithms

To motivate the syntactic choices for $\text{ConsL}$, let us return to OneThirdRule, which we reproduce once again in Figure 6.1 for ease of reference. We point out several characteristics of the algorithm, which are also shared by other consensus algorithms:

1. The processes are symmetric: in any given round, they all execute the same $\text{send}$ and $\text{next}$ functions.

2. The state of each process $p$ contains two distinguished fields $\text{inp}$ (the input received by $p$) and $\text{dec}$ (the decision made by $p$). Initially, $\text{dec}$ is set to the distinguished value $\perp$, indicating that no decision has been made.

3. The $\text{send}$ function always sends the value of a single state field.

4. In the $\text{next}$ function, each state field is either left unchanged or is updated to some received value. No new values are produced; instead, values are simply propagated between fields. Moreover, their origins are irrelevant. The map $\mu_p^r : \Pi \rightarrow M$ of received messages can hence be replaced by the multiset $R_p^r = \#[\mu_p^r]$. A field $f$ is then updated to a value $v$ from $R_p^r$ if:
6.1. The Specification Language ConsL

\[
\langle \text{rspec} \rangle ::= \langle \text{send-field} \rangle \langle \text{cp} \rangle \langle \text{instr} \rangle^* \\
\langle \text{send-field} \rangle ::= \text{identifier} \\
\langle \text{cp} \rangle ::= \ltimes | \rightrightarrows | \leftleftarrows \\
\langle \text{instr} \rangle ::= \langle \text{guard} \rangle \triangleright \langle \text{upd-field} \rangle \\
\langle \text{guard} \rangle ::= \langle (\text{th}), (\text{pred}) \rangle \\
\langle \text{th} \rangle ::= \text{fraction} \\
\langle \text{pred} \rangle ::= \text{any} | \text{smor} | \text{min} | \text{all=} | \text{maxts} \\
\langle \text{upd-field} \rangle ::= \text{identifier}
\]

Figure 6.2: Syntax of ConsL round specifications. The greyed out parts are extensions (Section 6.1.3)

(a) \(|R_p^r|\) is strictly larger than some threshold, expressed as a fraction of the total number of processes; in the example, this fraction is \(\frac{2}{3}\) for the updates to both inp and dec, and

(b) \(v\) fulfills a particular predicate with respect to the set of received messages. In the example, the predicate for the dec update is that all messages in \(R_p^r\) equal \(v\), and for the inp update that \(v\) is a value with the highest multiplicity in \(R_p^r\) and that it is the smallest such value.

6.1.2 Syntax

The above observations motivate the syntax for the basic building block of ConsL, the specification of a single round (Figure 6.2). Here, we focus on the core language, typeset in normal font; the greyed out parts are extensions (Section 6.1.3). A round specification starts with the state field that is sent in the round, followed by the communication pattern. In the \(\ltimes\) pattern, all process pairs exchange messages. The specification ends with a list of update instructions.

An instruction instr consists of a guard and the updated field. We assume that each upd-field appears at most once in the instruction list. The guard consists of a threshold th, which is a fraction \(c \in [0,1)\), and a predicate \(\text{pred}\). Intuitively, if messages are received from more than the given threshold of processes, the target field is updated with some value satisfying the predicate. The predicates are any (any received value), smor (the smallest most often received value), min (the smallest received value), and all= (satisfied by \(v\) if all the received values equal \(v\)). We will use the grammar symbols as projections where convenient; e.g., given a guard \(G\), we write \(\text{th}(G)\) for its threshold. Figure 6.3 shows the (single) round specification of
6. Cutoff Bounds for Consensus Algorithms

\[
\text{inp} \preceq_{\Sigma} (\frac{2}{3}, \text{all}=) \triangleright \text{dec} \ 
(\frac{2}{3}, \text{smor}) \triangleright \text{inp}
\]

Figure 6.3: The OneThirdRule algorithm in ConsL

the OneThirdRule algorithm.

While the OneThirdRule algorithm repeats the same round indefinitely, we have also seen (Section 5.6.2) other algorithms whose units of repetitions are phases: finite sequences of rounds. Hence, a ConsL algorithm \( \mathcal{A} \) consists of a finite set of phases, a phase sequence, specified by an infinite word \( w \) over this set, and a communication predicate, specified as below. The phase sequence determines the infinite sequence of round specifications to execute, reflecting our assumption that processes run forever. While our theorems also hold for arbitrary phase sequences, to obtain finite-state systems and enable model checking, we require \( w = uv^\omega \), for finite words \( u \) and \( v \).

Communication predicates. As we use an HO model semantics, ConsL algorithms must express their environment assumptions using communication predicates. Arbitrary predicates could make cutoff bounds unobtainable, so we provide a specification mechanism that is restricted, but still sufficient for expressing the predicates used by practical algorithms in our target setting. The building blocks are the round formulas \( \phi_{th}(c, r) \) and \( \phi_{uf}(r) \) from Section 4.3.3. Abusing notation, we associate the round labels \( \phi_{th}(c) \) and \( \phi_{uf} \) with the corresponding round formulas. Let \( L = \{ \phi_{uf} \} \cup \{ \phi_{th}(c) \mid c \in [0, 1) \} \) be the set of all round labels. A ConsL communication predicate is then specified by a language of infinite words over the alphabet \( \Sigma = \mathcal{P}(L) \). Again, to ensure finite representation, we require the language to be \( \omega \)-regular. As an example, recall the communication predicate of the OneThirdRule algorithm:

\[ \exists r_1, r_2 > r_1. \phi_{th}(\frac{2}{3}, r_1) \land \phi_{uf}(r_1) \land \phi_{th}(\frac{2}{3}, r_2). \]

In ConsL, this predicate is specified as \( \Sigma^* \Lambda_1 \Sigma^* \Lambda_2 \Sigma^\omega \), with \( \Lambda_1 = \{ \phi_{th}(\frac{2}{3}), \phi_{uf} \} \) and \( \Lambda_2 = \{ \phi_{th}(\frac{2}{3}) \} \).

Restrictions. To ensure that cutoff bounds exist, we enforce several syntactic restrictions on ConsL. Their motivation is technical; we will show some examples in this chapter, but we leave the details for the appendix.

We first define a partial order \( \sqsubseteq_P \) on the predicates by any \( \sqsubseteq_P \) \( P, P \sqsubseteq_P P \), and \( P \sqsubseteq_P \text{all}= \), for all predicates \( P \). Intuitively, if a value \( v \) satisfies a predicate \( P_1 \), then it also satisfies all predicates \( P_2 \) where \( P_2 \sqsubseteq_P P_1 \). We extend this order to guards such that \( G_1 \sqsubseteq G_2 \) if and only if \( \text{th}(G_1) \leq \text{th}(G_2) \) and \( \text{pred}(G_1) \sqsubseteq_P \text{pred}(G_2) \).

Next, we constrain the data flow within a phase. Intuitively, a consensus protocol phase is a single attempt to reach a decision on one of the input
6.1. The Specification Language *ConsL*

![Diagram](image)

Figure 6.4: The phase graph of OneThirdRule

values. We exploit this by assuming that all data within a phase originates from the *inp* field, and that *inp* and *dec* are updated at most once. We formalize this using the notion of a phase graph. First, given a phase \( \Phi = [r_{s_1}, r_{s_2}, \ldots, r_{s_n}] \), and a field \( f \), let \( f \)'s latest update before \( i \), denoted \( lu(f, i) \), be the largest \( j \), with \( j < i \), such that \( f \) is updated in \( r_{s_j} \), and 0 if no such \( j \) exists. The phase graph is then a directed graph whose nodes are pairs \((i, f)\) such that either the field \( f \) is updated in \( r_{s_i} \), or \( i = 0 \) and \( f \) is sent in some \( r_{s_j} \) with \( lu(f, j) = 0 \). There is an edge \((i, f) \rightarrow (j, g)\) in the graph if and only if \( f \) is sent in \( r_{s_j} \), \( g \) is updated in \( r_{s_j} \), and \( i = lu(f, j) \). Figure 6.4 shows \( \frac{1}{3} \)-rule’s phase graph. Our restrictions are then as follows:

1. **(R1)** The phase graph of each phase is a tree rooted at \((0, \text{inp})\). For \( f \in \{\text{inp}, \text{dec}\} \), at most one node \((i, f)\) with \( i > 0 \) exists, and it must be a leaf.
2. **(R2)** The set of all guards used in the algorithm is totally ordered.
3. **(R3)** min and smor predicates only appear in instructions where send-field is *inp*.
4. **(R4)** If \( th(G) = 0 \), then \( pred(G) = \text{any} \).

Note that the terminal branches of the phase graph that end in fields other than *inp* and *dec* describe data flows that cannot affect the consensus properties. The corresponding steps can be simplified or removed, which we formalize as Lemma A.2. As a consequence, we can assume that each phase has at most one round specification with two update instructions. As such a specification corresponds to a fork between the paths \((0, \text{inp}) \sim (i, \text{inp})\) and \((0, \text{inp}) \sim (j, \text{dec})\) in the phase graph, we call the corresponding rounds fork points. These fork points are crucial for the consensus agreement property. Their design must ensure that whenever a value \( v \) is decided on in one phase, the *inp* field is modified so that only \( v \) can be decided on in later phases. Dealing with fork points is the most challenging part of our proofs, as we will see in Sections 6.3 and 6.4.

### 6.1.3 Extensions

To cover additional algorithms, we increase the expressiveness of *ConsL* by including three additional features: leaders (\(l\)), timestamps (\(t\)), and randomness (\(r\)). We write *ConsLE* for a given set \( E \subseteq \{l, t, r\} \) to denote the
6. Cutoff Bounds for Consensus Algorithms

language with the corresponding extensions. An algorithm must specify the
extensions it uses. As we do not know of any algorithms combining ran-
domness and timestamps, for simplicity we assume \( \{r, t\} \not\subseteq E \). The leaders
and timestamp extensions are subject to some further syntactic restrictions,
needed in our proofs.

**Leaders.** We have discussed the use of leaders in consensus algorithms in
Section 4.3.3 and Chapter 5, where we also saw an example of a leader-
based algorithm, namely Paxos [Lam98]. To model leader-based algorithms,
we add two new communication patterns:

- \( \langle \rangle \), where only the leader broadcasts a message in a round, and
- \( \geq \rangle \), where all processes send a message exclusively to the leader (a
  so-called convergecast).

We assume the following leader policy: leaders can switch arbitrarily be-
tween phases, but they remain constant within a phase. As discussed in Sec-
tion 4.3.3, leader-based algorithms assume that some leader will eventually
become stable, an assumption expressed using \( \phi_{ls} \) and \( \phi_{lr} \) formulas. We thus
extend the set \( L \) of transition labels with the set \( \{\phi_{ls}\} \cup \{\phi_{lr}(c) \mid c \in [0, 1)_{\mathbb{Q}}\} \).

Lastly, leaders introduce some further syntactic restrictions on \( ConsL \).
Give a phase \( \Phi = [rs_1, rs_2, \ldots, rs_n] \) and its phase graph, for \( 1 \leq i \leq n \) we define:

\[
\text{spans}(\Phi, i) = \{(f, k) \mid \exists j, g. (j, f) \rightarrow (k, g) \land j < i \leq k\},
\]

i.e., \( (f, k) \in \text{spans}(\Phi, i) \) means that field \( f \) has been updated previously (in
some step \( j \)) and is still to be sent (in step \( k \)). We call the field \( f \) active in
step \( i \) of phase \( \Phi \). Then we define:

\[
\text{next-ldrf}(\Phi, i) = \{f \mid (f, k) \in \text{spans}(\phi, i) \land k \text{ is a } \langle \rangle \text{ step}\},
\]

i.e., the set of active fields that will be sent in a \( \langle \rangle \) step. We also abuse nota-
tion and, given an algorithm, write \( \text{next-ldrf}(rs) \) instead of \( \text{next-ldrf}(\Phi, i) \)
if \( rs \) is the \( i \)-th step of the phase \( \Phi \).

Then, we add the restrictions:

(R5) Only \( (0, \text{any}) \) guards may appear in \( \langle \rangle \) steps.
(R6) \text{inp} and \text{dec} do not appear in the update instructions of \( \geq \rangle \) steps.
(R7) given a round specification \( rs_i \), no field from \( \text{next-ldrf}(rs) \) is simul-
taneously updated with \text{dec} or \text{inp}.
6.1. The Specification Language \textit{ConsL}

\begin{align*}
\text{inp} & \not\rightarrow (\frac{1}{2}, \text{maxts}) \triangleright \text{cand} \\
\text{cand} & \not\rightarrow (0, \text{any}) \triangleright \text{inp} \\
\text{vote} & \not\rightarrow (\frac{1}{2}, \text{any}) \triangleright \text{dec}
\end{align*}

Communication predicate: \((\Sigma^3)^* \{\phi_{lr}(\frac{1}{2})\} \{\phi_{ls}\} \{\phi_{th}(\frac{1}{2})\} \Sigma^\omega\)

Figure 6.5: Paxos written in \textit{ConsL}

\textbf{Timestamps.} In Section 5.7.1 we saw how timestamps can be useful for consensus algorithms. To cater for timestamps, we extend \textit{ConsL} with \textit{timestamped fields}, which store a value together with the time of its last update. Time is logical, expressed by round numbers. When sending out a timestamped field, both the value and the timestamp are transmitted. A new predicate, \texttt{maxts}, then selects a value with the highest timestamp; to break ties, the smallest such value is selected. In \textit{ConsL}, timestamps only make sense with the \texttt{inp} field, since the other fields are either never sent out (in particular \texttt{dec}), or, as we will shortly see, do not persist between phases (in particular, fields different from \texttt{inp} and \texttt{dec}).

Timestamped algorithms add another syntactic restriction to \textit{ConsL}:

(R8) For a guard \(G\), \(\text{pred}(G) = \text{maxts}\) if and only if the send-field of the step where it appears is \texttt{inp} and the algorithm is timestamped.

To demonstrate the use of leaders and timestamps, recall the Paxos algorithm (Figure 5.7 from Section 5.7.2). Figure 6.5 shows the \textit{ConsL} (or more precisely, \textit{ConsL}\{\texttt{l,t}\}) program for it. The single three-round phase is repeated forever. The use of timestamps is visible in the \texttt{maxts} predicate. The communication predicate is the \textit{ConsL} encoding of the predicate from Section 5.7.2, where \(\Sigma^3\) is an abbreviation for \(\Sigma \Sigma \Sigma\), as usual. Note that the semantics of the \textit{ConsL} version is slightly different (formally defined in the next section) from the semantics of the algorithm shown in Figure 5.7: the Hilbert’s choice operator is replaced by a deterministic selection of the smallest value, as \texttt{maxts} breaks ties by choosing the smallest values.

\textbf{Randomness.} Randomization is an alternative to partial synchrony for making consensus solvable [BO83]. Randomized algorithms typically assume that \(\mathcal{V} = \{0, 1\}\), and we adopt this assumption. They also guarantee termination probabilistically: all processes eventually decide with probability 1. The termination proof usually relies on an almost-sure “lucky toss”, where all processes draw the same favorable randomness. We turn this into a standard termination guarantee by (1) modeling randomness as non-determinism: processes non-deterministically choose a bit for the fallback values; (2) providing a way to specify lucky tosses, inspired by the Ben-Or algorithm [BO83]; and (3) extending the set \(L\) of transition labels with a special
6. Cutoff Bounds for Consensus Algorithms

\[
\begin{align*}
\text{inp} & \not\supseteq \left(\frac{1}{2}, \text{all=}\right) \triangleright \text{vote} \\
\text{vote} & \not\supseteq \left(\frac{1}{2}, \text{all=}\right) \triangleright \text{dec} \\
(0, \text{any}) & \triangleright \text{inp}
\end{align*}
\]

A randomized algorithm.

Communication predicate: \(\{\phi_{\lambda_0}(\frac{1}{2})\} \cdot \{\lambda, \phi_{\lambda_0}(\frac{1}{2})\} \cdot \{\phi_{\lambda_0}(\frac{1}{2})\}\)

\[L(t_0, t_1) \triangleq \begin{cases} 
0, & \text{if } t_0 \geq 0 \\
1, & \text{otherwise}
\end{cases}\]

Figure 6.6: Ben-Or

label \(\lambda\), indicating that a lucky toss occurred. The communication predicate then express the requirements on when luck tosses must occur.

The lucky toss is defined as a slight generalization of the definition of a lucky toss for the Ben-Or algorithm. There, the criterion is that, whenever some process updates its \(\text{inp}\) value to a bit \(b\), all processes draw the randomness \(b\). For illustration, the \(\text{ConsL}\) program for the Ben-Or algorithm is shown Figure 6.6. The lucky toss is specified by the \(Lt\) function, whose parameters \(t_b\) are the counts of processes that update \(\text{inp}\) by \(b\). More precisely, these counts are expressed as fractions of the total number of processes, as we will see in the next section.

6.1.4 Semantics

The semantics of \(\text{ConsL}\) is defined by a translation into the Heard-Of model. Thus, as in Sections 4.3.3 and 4.4.1, we assume that the system is parameterized by a set \(\Pi\) of processes and a set \(V\) of values, which we now additionally assume to be totally ordered. The message space is then defined to be \(M \triangleq V \cup \{\bot, *\}\) for non-timestamped algorithms, and \(M \triangleq (N \times V) \cup V \cup \{\bot, *\}\) for timestamped algorithms. The most complicated part of the translation involves formalizing the intuitive description of the guards (Section 6.1.2). We thus address it first.

Guards. Given a multiset \(M\) of elements from \(M\), define \(vs(M) \triangleq M \setminus \{\bot, *\}\). Then, given a guard \(G = (t, p)\), a multiset \(M\) (of received messages), and a value \(v \in V\), we write \(M \models G(v)\) if \(|vs(M)| > t \cdot |\Pi|\), and one of the following four conditions holds:

- \(p = \text{any}\) and \(vs(M)(v) > 0\),
- \(p = \text{all=}\) and \(vs(M)(v) = |vs(M)|\),
- \(p = \text{min}\) and \(v\) is the smallest value in \(vs(M)\), or
6.1. The Specification Language ConsL

- \( p = \text{smor} \) and \( v \) is the smallest most frequent value in \( M' = \text{vs}(M) \), i.e., \( \forall v'. M'(v) \geq M'(v') \land (M'(v) = M'(v') \implies v \leq v') \).

- \( \text{pred}(G) = \text{maxts}, (t, v) \in M \) for some \( t \), and whenever \((t', v') \in M, t' \leq t \) or \( t' = t \) and \( v' \geq v \).

Note the connection with the guard ordering (Section 6.1.2): if \( G_1 \subseteq G_2 \) then \( M \models G_2(v) \) implies \( M \models G_1(v) \). Furthermore, note that if \( M \) is a multiset over \( \mathbb{N} \times \mathcal{V} \), no value from \( \mathcal{V} \) can satisfy anything other than the maxts predicate. However, restriction (R8) ensures that we do not need to consider such cases in practice.

**States and initial states.** The state space of all processes is a record that consists of inp and dec fields, and additionally all other fields that appear as send-field or upd-field in the algorithm’s round specifications. The type of inp is \( \mathcal{V} \) for non-timestamped, and \( \mathbb{N} \times \mathcal{V} \) for timestamped algorithms, and the type of all other fields is \( \mathcal{V} \). The set of initial states is the same for each process: inp takes an arbitrary value from \( \mathcal{V} \), paired with the timestamp 0 for timestamped algorithms, while all other fields are \( \bot \).

**Send and next functions.** As mentioned earlier, the phase sequence of a ConsL algorithm uniquely determines a round specification \( rs(r) \) for each round \( r \in \mathbb{N} \) to be executed. We translate each round specification \( rs(r) \) into a pair \((\text{send}^r_p, \text{next}^r_p)\) as follows:

- \( \text{send}^r_p(s_p, q) \) returns:
  1. if \( cp \) is \( \searrow \nearrow \) or \( cp \) is \( \nearrow \) and \( q = \text{ldr}(r) \), or \( cp \) is \( \searrow \) and \( p = \text{ldr}(r) \), then \( s_p.sf \), where \( sf \) is the send-field of \( rs(r) \);
  2. otherwise, the dummy value \( \star \).

- \( \text{next}^r_p(s_p, \mu^r_p) \) updates process \( p \)'s state by selecting new values for all fields in the instruction list of \( rs(r) \). Let \( R^r_p = \#[\mu^r_p] \). Given an instruction \( G \triangleright f \), the set of possible new values of the field \( f \) of process \( p \) is determined as follows:
  1. If \( cp \) of the step is \( \searrow \) and \( p \neq \text{ldr}(r) \), then the only possible value is \( \bot \).
  2. For all values \( v \in \mathcal{V} \) such that \( R^r_p \models G(v) \), a possible new value for \( f \) is either \( v \) if \( f \neq \text{inp} \) or the algorithm is not timestamped, or \((r + 1, v) \) otherwise.
  3. If no such value \( v \in \mathcal{V} \) exists, the only possible value is the fallback value, which is:
     - \( \bot \) if \( f \notin \{\text{inp, dec}\} \),
6. Cutoff Bounds for Consensus Algorithms

- the old value of \( f \) of process \( p \) if one of the following holds:
  * \( f = \text{inp} \) and the algorithm is not randomized
  * \( f = \text{dec} \)
- any value from \( V \) if \( f = \text{inp} \) and the algorithm is randomized
  (note that, by assumption, \( V = \{0, 1\} \) in this case).

We call fields other than \( \text{inp} \) and \( \text{dec} \) ephemeral since their fallback value \( \perp \) and the restriction (R1) jointly imply that they do not keep state between successive phases. Moreover, the semantics ensures that the \( \text{dec} \) field never reverts from a value in \( V \) to \( \perp \). Hence, the stability requirement of consensus holds by construction for all \( \text{ConsL} \) algorithms. We therefore do not further discuss this requirement.

**Labeled transition system semantics.** In Section 4.3.3 we introduced the unlabeled transition system semantics of the HO model. To restrict reasoning to those traces satisfying the communication predicates, we now label the traces with the round labels of Sections 6.1.2 and 6.1.3. The \( r \)-th unlabeled step \( s \rightarrow s' \) of a trace generated by an HO collection \( \{HO_p^r\}_{p \in \Pi}^{r \in \mathbb{N}} \) gives rise to a set of labeled steps \( s \xrightarrow{\Lambda} s' \), where \( \Lambda \in \Sigma \), such that:

(L1) \( \phi_{uf} \in \Lambda \) implies that the formula \( \phi_{uf}(r) \) holds for \( \{HO_p^r\}_{p \in \Pi}^{r \in \mathbb{N}} \).

(L2) \( \phi_{ls} \in \Lambda \) implies that the \( cp \) of the step is \( <_p \) and \( \phi_{ls}(r) \) holds.

(L3) \( \phi_{th}(c) \in \Lambda \) (respectively \( \phi_{lr}(c) \in \Lambda \)) implies that \( \phi_{th}(c, r) \) holds and that the step type is \( >_p \) (respectively \( \phi_{lr}(c, r) \) holds and that the step type is \( \preceq_p \)), and that \( c \) appears as the threshold of some guard in the algorithm. For technical reasons, we also require that for all guards \( G \) in the step, \( th(G) = 0 \lor th(G) = c \).

(L4) \( \lambda \) in the label \( \Lambda \) of a step \( s \xrightarrow{\Lambda} s' \) implies that

(a) the \( \text{inp} \) field is updated in the step \( s \rightarrow s' \)
(b) all the processes in the step draw the randomness \( Lt(t_0, t_1) \) where \( t_0 = \gamma_n(\#[U(\text{inp})](0)) \) and \( t_1 = \gamma_n(\#[U(\text{inp})](1)) \), where \( \#[\cdot](\cdot) \) is as defined in Section 2.1, \( n = |\Pi| \) and:

- the global update \( U(f)(p) \) is the value that process \( p \) used to update the field \( f \) in the \( s \rightarrow s' \), and \( \perp \) if \( p \)'s update of \( f \) failed, that is, a fallback value was used. We define \( U \) more formally in Appendix A.
- \( \gamma_n(c) \) is the largest protocol threshold \( t \) such that \( c > t \cdot n \). We define \( \gamma \) more formally in Section 6.4.1.
The semantics of a \(\text{ConsL}\) algorithm \(\mathcal{A}\) is the set of infinite traces whose labels form a word in the communication predicate of \(\mathcal{A}\). Property satisfaction is relative to this semantics. However, as noted in Section 4.4.1, all the consensus properties except for termination are safety, so for them it suffices to consider all finite prefixes of such traces.

Note that the conditions (L1)–(L4) are implications, and not equivalences. This allows us to omit formulas from a label even when they hold for a given step. We make use of this in our proofs as follows: when turning a counterexample trace \(\tau_{cex}\) into a trace \(\tau_P\) with some desired property \(P\) (for example, turning a \(\tau_{cex}\) with \(|\mathcal{V}| > 2\) into a \(\tau_P\) with \(\mathcal{V} = \{0, 1\}\) in the proof of the zero-one principle), we must preserve labels to ensure that \(\tau_P\) is allowed by the communication predicates. It could, however, happen that \(\tau_P\) satisfies more formulas than those in the label of \(\tau_{cex}\). If the above conditions were equivalences, the labels of \(\tau_{cex}\) and \(\tau_P\) would differ. Implications avoid such problems, without requiring further constraints on the communication predicates (e.g., some form of upward closure).

### 6.2 Further ConsL Examples

We have already seen how to express OneThirdRule, Paxos and Ben-Or algorithms in ConsL. We now also show how to do this for the other two algorithms we saw in Chapter 5, UniformVoting [CBS09] and the new algorithm from [MSB15]. We showed the HO model of the latter in Figure 5.8 (Section 5.8). Its corresponding ConsL program is:

\[
\begin{align*}
\text{inp} & \rightarrow (\frac{1}{2}, \text{maxts}) \triangleright \text{cand} \\
\text{cand} & \rightarrow (\frac{1}{2}, \text{all}=) \triangleright \text{inp} \\
\text{vote} & \rightarrow (\frac{1}{2}, \text{any}) \triangleright \text{dec}
\end{align*}
\]

and its communication predicate is expressed as

\[
(\Sigma^3)^3 \cdot \{\phi_{th}(\frac{1}{2}), \phi_{uf}\} \cdot \{\phi_{th}(\frac{1}{2})\} \cdot \{\phi_{th}(\frac{1}{2})\} \Sigma^\omega.
\]

The HO model of the UniformVoting algorithm was presented in Figure 5.6 (Section 5.6.2). However, the original version of this algorithm cannot be expressed in ConsL for two reasons:

1. in the second round, the values of two fields are sent out. ConsL allows only a single field to be sent out.
2. \text{inp} is updated twice within a phase, conflicting with the restriction (R1).

Fortunately, the first problem is merely an optimization: the algorithm works just as well when only a single field is sent out in the second round, with a
slight change to the communication predicate. The second problem can be worked around by splitting the first round into two, where we first update \( inp \) and then update \( vote \). Unfortunately, this adds an extra message delay to the algorithm. We then artificially split the algorithm into two phases \( \Phi_1 \) and \( \Phi_2 \), such that \( inp \) is updated only once per phase. The splitting into phases does not affect the round sequence, since we use \((\Phi_1 \Phi_2)^\omega\) as the phase sequence. The resulting ConsL program is then:

\[
\begin{align*}
\Phi_1 & \quad \text{inp} \not\rightarrow \left(\frac{1}{2}, \text{min}\right) \triangleright \text{inp} \\
\Phi_2 & \quad \text{inp} \not\rightarrow \left(\frac{1}{2}, \text{all}\right) \triangleright \text{vote} \\
& \quad \text{vote} \not\rightarrow \left(\frac{1}{2}, \text{all}\right) \triangleright \text{dec} \\
& \quad (0, \text{any}) \triangleright \text{inp}
\end{align*}
\]

We also need to adjust the communication predicate to account for the extra round. The predicate is now an intersection of two predicates, the first of which:

\[ (\Sigma^2\{\phi_{bf}(1_2)\})^\omega \]

is required for safety, while the second one:

\[ (\Sigma^3)^+\{\phi_{uf}, \phi_{bf}(1_2)\}\{\phi_{bf}(1_2)\}\Sigma^\omega \]

additionally ensures termination. Further examples of ConsL algorithms are distributed together with our verification tool for ConsL, described in Section 6.5.

### 6.3 Zero-One Principle for Consensus Verification

The zero-one principle for sorting networks [Knu73] is a well-known result stating that a sorting network correctly sorts all sequences of inputs if and only if it correctly sorts all sequences of elements from \( \mathbb{B} \triangleq \{0, 1\} \). We prove an analogous result for our language and the consensus problem. We call the consensus problem for the binary domain \( V = \mathbb{B} \) the binary consensus problem. Since the randomization extension already assumes this domain, we restrict our attention here to non-randomized algorithms. We also need a further restriction on ConsL:

(\text{RT}) \text{ min} \text{ and all= guards do not appear simultaneously in the same step.}

The restriction is necessary for the termination part of the consensus property in the following theorem.

**Theorem 3.** An algorithm expressed in ConsLE (with \( r \notin E \)) that additionally obeys (RT) solves the consensus problem for an arbitrary value domain \( V \) if and only if it solves the binary consensus problem.
6.3. Zero-One Principle for Consensus Verification

<table>
<thead>
<tr>
<th>Phase $\Phi_1$</th>
<th>Phase $\Phi_2$</th>
<th>Phase $\Phi_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{inp} \not\leftrightarrow (\frac{1}{2}, \text{min}) \triangleright \text{inp}$</td>
<td>$\text{inp} \not\leftrightarrow (\frac{1}{2}, \text{min}) \triangleright \text{vote}$</td>
<td>$\text{inp} \not\leftrightarrow (\frac{1}{2}, \text{all=} \triangleright \text{dec}$</td>
</tr>
<tr>
<td>$\text{vote} \not\leftrightarrow (0, \text{any}) \triangleright \text{dec}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Phase sequence: $\Phi_1 \Phi_2 \Phi_3^\omega$; communication predicate: $\{\phi_{\mathcal{B}}(\frac{1}{2})\}^\omega$.

Figure 6.7: Example showing the necessity of (RT)

There are compelling intuitive reasons why the principle should hold. Since we assumed $r \notin E$, $\text{ConsL}$'s semantics immediately implies that all algorithms guarantee non-triviality (in addition to stability). We thus only have to consider agreement and termination, for which we prove that their violations are preserved when restricting $\mathcal{V}$ to $\mathbb{B}$. By definition, agreement requires only two values to disprove. We combine this with the earlier observation that $\text{ConsL}$ algorithms simply propagate values between the processes’ fields. Then it suffices to ensure that whenever two different values can be propagated (in particular, to the $\text{dec}$ field) in a multi-valued agreement counterexample, both 0 and 1 can be propagated when $\mathcal{V} = \mathbb{B}$. This is in general possible as the values themselves are irrelevant and only their relative ordering matters. Disproving termination requires showing that, whenever it is possible for updates (in particular, updates of $\text{dec}$) to fail in a multi-valued setting, the same can happen in the binary setting. From the language semantics (Section 6.1.4), there are two ways for an update to fail. The first way is to have the process receive insufficiently many non-$\perp$ messages. As this is independent of the size of $\mathcal{V}$, we can mimic this cause of failure in the binary setting. The second way is to have the process receive different values when the update is guarded by an $\text{all=} \triangleright$ predicate. In this case, two values also suffice.

Unfortunately, the proof is significantly more involved than this intuition might suggest. One example of its intricacy is the restriction (RT). The algorithm in Figure 6.7 shows why the restriction is necessary. This algorithm violates termination in a three-valued, but not in the binary setting. To see why the algorithm always terminates in the binary setting, consider the first execution of the phase $\Phi_3$. To violate termination, some process must fail to update $\text{dec}$ in $\Phi_3$. Since the communication predicate requires all processes to receive messages from more than one half of all the processes, the only way a process can fail to update $\text{dec}$ in this round is to have it receive at least one 0 and at least one 1; thus some processes $p$ and $q$ must have their $\text{inp}$ fields set to 0 and 1 respectively at the start of $\Phi_3$. Since the communication predicate forces $q$ to update $\text{inp}$ in the phase $\Phi_2$, there exists a set $\Pi_1$ of processes, with $|\Pi_1| > \frac{|\Pi|}{2}$, such that all processes in $\Pi_1$ had their $\text{inp}$ fields set to 1 at the start of $\Phi_2$. All processes from $\Pi_1$ must then have set $\text{inp}$ to 1 in the phase $\Phi_1$. However, then they also must have set $\text{vote}$ to 1 in $\Phi_1$. 

101
6. Cutoff Bounds for Consensus Algorithms

This is the crucial difference to the multi-valued setting, where two processes \( p \) and \( q \) can update \( \text{inp} \) to two different values \( v \) and \( v' \), while the updates to \( \text{vote} \) fail at both \( p \) and \( q \). In the binary setting, since all processes from \( \Pi_1 \) set \( \text{vote} \) to 1 and \( |\Pi_1| > \frac{|\Pi|}{2} \), taking the communication predicate into account, we then see that all processes must update \( \text{dec} \) in \( \Phi_1 \).

Note, however, that the algorithm in Figure 6.7 also violates agreement in both settings. We leave open the question of whether there exists an algorithm in the language without restriction (RT) that violates consensus, but does not have a binary trace violating consensus.

In the rest of the section, we sketch the main proof ideas and refer to Appendix B for the details. We fix an arbitrary ConsL^{E}E algorithm \( A \) (again, \( r \not\in E \)) and prove the theorem separately for agreement and termination. For simplicity of exposition, we assume in the rest of the section that \( A \) is not timestamped. We start with some definitions used in both proofs.

6.3.1 Definitions

Recall from Section 4.3.3 that the global state \( s \) of the HO transition system consists of a round number and a product of process states. Also recall that we assume process states to be records. For convenience, we introduce an abuse of notation on the states. Let \( \mathcal{F} \) be the set of field names used in \( A \). We denote the round component of \( s \) as \( \text{rnd}(s) \), and treat \( s \) itself as a function, such that \( s(f)(p) \) returns the \( p \)'s value of the field \( f \in \mathcal{F} \) in the state \( s \). Thus, \( \#s(f) \) is the multiset containing the values of the field \( f \) of all the processes. We define binary multisets to be multisets over \( \{0, 1, \perp\} \), and binary states to be all states \( s \) for which \( \#s(f) \) is a binary multiset for all \( f \in \mathcal{F} \). A binary trace is then any trace where all states are binary. We say that a state or trace is of width \( n \) if it is generated by an instance of the system with \( |\Pi| = n \).

Our goal is to show that for any trace \( \tau \) violating agreement or termination there exists a binary trace \( \tau_b \) with identical labels (and thus allowed by the communication predicate whenever \( \tau \) is) that also violates the same property. The \( \tau_b \) we construct will actually have the same width as \( \tau \). For technical reasons, we assume that \( \tau \) is of a sufficiently large width, an assumption justified by a trace inflation lemma, which turns any counterexample into a suitable wide one. In the remainder of this section, we thus implicitly assume that the cardinality of all multisets and the width of all states and traces mentioned are the same.

We define constraints to be guards combined with a field name, that is, triples \( (f, \text{th}, \text{pred}) \), and extend the ordering \( \sqsubseteq \) to constraints. All the guards in this and the next section are assumed to appear in the specification of the algorithm \( A \), and for all constraints \( (f,G) \) the guard \( G \) must appear in the list of instructions in a step in which \( f \) is a send-field. We extend the satisfaction relation of Section 6.1.4 to states \( s \) and constraints \( C \) as follows.
First, we define \(s, f, W \models G(v)\) if \(#s(f)[W] \models G(v)\). Then, \(s \models C(v)\) if there exists a \(W\) such that \(s, f(C), W \models (\text{th}(C), \text{pred}(C))(v)\). Intuitively, the constraints capture the possibility of propagation: if \(s \models C(v)\) and \(f(C)\) is the send-field of the step \(s \rightarrow s'\), then \(v\) can be propagated from \(f\) to any upd-field of \(s'\) whose update is guarded by \(G\) such that \(C = (f, G)\). To capture simultaneous propagation, we say that \(M\) simultaneously satisfies \(G_1(v_1)\) and \(G_2(v_2)\), written \(M \models G_1(v_1) \land G_2(v_2)\), if \(M \models G_1(v_1)\) and \(M \models G_2(v_2)\). We define \(s \models C_1(v_1) \land C_2(v_2)\) for states and constraints analogously.

### 6.3.2 Agreement

As already mentioned, the main idea behind the agreement proof is to ensure that whenever two different values can be propagated from one field to another in the multi-valued counterexample \(\tau\), both 0 and 1 can be propagated in \(\tau_b\). Thus, given any multiset \(M\), we must find a binary multiset \(M_b\) such that, for all \(G_1, G_2\), and \(v_1 \neq v_2\), whenever \(M \models G_1(v_1) \land G_2(v_2)\), then also \(M_b \models G_1(1) \land G_2(0)\) or \(M_b \models G_1(0) \land G_2(1)\). When this and some additional technical conditions (detailed in Appendix B) are met, we say that \(M_b\) is a binary companion of \(M\). Our first result shows that every multiset has a binary companion of the same size.

We next define the critical round \(r_c\) of \(\tau\) to be the first round in which some process makes a decision, that is, updates its \(\text{dec}\) field to a value \(v_1 \in \mathcal{V}\). Since \(\tau\) is assumed to be a counterexample to agreement, \(r_c\) exists. The critical phase is the phase to which \(r_c\) belongs to. The proof then consists of three main parts:

1. Let \(r_s\) be the first round of the critical phase. Recall that \(\tau|_{r_s}\) is the prefix of \(\tau\) of length \(r_s + 1\). We show that there exists a trace \(\tau'_b\) of length \(r_s + 1\), such that the labels on \(\tau'_b\) match the labels on \(\tau|_{r_s}\), and such that \(\#[\tau'_b(i)(\text{inp})]\) is a binary companion of \(\#[\tau(i)(\text{inp})]\), for all \(i \in \{0, \ldots, r_s\}\). This allows us to propagate both 0 and 1 in \(\tau_b\) to and from the \(\text{inp}\) field of \(\tau'_b(r_s)\).

2. (a) If a value \(v'_1 \neq v_1\) was also decided on in round \(r_c\), we show that we can extend \(\tau'_b\) to a trace \(\tau_b\) of length \(r_c + 1\) such that both 0 and 1 are decided on in \(r_c\), and such that the labels on \(\tau_b\) match the labels of \(\tau|_{r_c}\). Then, \(\tau_b\) is the desired binary counterexample. More precisely, it can (easily) be extended to an infinite trace that is a counterexample for agreement.

(b) Otherwise, let \(r_n\) be the first round after the critical phase. We extend \(\tau'_b\) to \(\tau''_b\), such that the labels on \(\tau''_b\) and \(\tau|_{r_n}\) match, and such that there exists a value \(b\) such that (1) some process decides on \(b\) in \(\tau''_b(r_n)\); and (2) \(1 - b\) can be propagated from the \(\text{inp}\) field of \(\tau''_b(r_n)\) whenever a value \(v_2 \neq v_1\) can be propagated from the
6. Cutoff Bounds for Consensus Algorithms

\[
\text{inp field in } \tau(r_n). \text{ More formally, let } C_n \text{ be the strongest (relative to } \sqsubseteq) \text{ constraint with } f(C_n) = \text{inp}, \text{ such that } \tau(r_n) \models C_n(v) \text{ and } v \neq v_1. \text{ Note that this is well defined by the restriction (R2). Then, we prove } \tau''_b(r_n) \models C_n(1-b). \text{ This step, and in particular handling the fork point of the critical phase (if it exists), is the most difficult part of the proof.}
\]

3. If the case (b) applied in the previous step, we show that we can extend \( \tau''_b \) to a trace \( \tau_b \) such that \( 1-b \) is decided on in some round after \( r_n \). Since \( b \) was decided on previously, \( \tau_b \) violates agreement.

6.3.3 Termination

To turn a termination counterexample \( \tau \) into a binary counterexample \( \tau_b \), we must prevent at least one process from ever updating its \( \text{dec} \) field in \( \tau_b \). In fact, in our proof we prevent all processes from doing so. The central observation is that, given ConsL’s semantics, there are two ways an update can fail. The first way is to have the process receive insufficiently many non-\( \perp \) messages. The second way is to have it receive two different values when the update is guarded by an \( \text{all=} \) predicate. This observation is reflected in our strategy for producing \( \tau_b \) from \( \tau \), which proceeds phase-by-phase and has two main components:

1. As soon as some process uses a fallback value for an update of a field \( f \) in \( \tau \)’s phase that we are currently considering, we force all the processes in \( \tau_b \) to use the fallback value for updating \( f \). If \( f = \text{dec} \), this ensures that no decisions are made within the phase; if \( f \) is an ephemeral field, this ensures that all further updates to the descendants of \( f \) in the phase graph will fail.

2. Otherwise, no updates have failed so far in the current phase of \( \tau \). Still, to preserve possible causes of future \( \text{all=} \) update failures, we must ensure that whenever two different values \( v_1, v_2 \in \mathcal{V} \) are present in \( \#[\tau(r)(f)] \), where \( f \) is used as a send-field in some step \( r \) of the phase, then both 0 and 1 are also present in \( \#[\tau_b(r)(f)] \). Assume both \( v_1 \) and \( v_2 \) can be propagated from the field \( f \) of \( \tau_b(r) \). Thus, there exist constraints \( C_1 \) and \( C_2 \) such that \( \tau_b(r) \models C_i(v_i) \) and \( f(C) = f \), for \( i = 1, 2 \). Letting \( C = \min(C_1, C_2) \), we conclude that \( \tau(r) \models C(v_1) \land C(v_2) \). Let \( G \) be the maximum guard of \( A \) (well defined by (R2)) such that there exists a binary multiset \( M_b \) with \( M_b \models G(0) \land G(1) \). To ensure that both 0 and 1 can also be propagated, we construct \( \tau_b \) such that \( \#[\tau_b(r)(f)] \models G(0) \land G(1) \).
6.4 Cutoff Bounds for Binary Algorithms

The zero-one principle shows that it suffices to verify consensus algorithms for the binary domain $V = \mathbb{B}$. We now complete our proof of the small scope hypothesis by proving it for the binary case. For an algorithm $A$ with the set of guards $G$, let $T_A = \{\text{th}(G) \mid G \in G \land \text{pred}(G) \neq \text{smor}\} \cup \{\frac{\text{th}(G)}{d} \mid G \in G \land \text{pred}(G) = \text{smor}\}$. The different treatment of $\text{smor}$ guards is technical, motivated by Lemma C.4.

Theorem 4. Let $A$ be an algorithm written in $\text{Cons}L^E$ for some $E$. Let $d$ be the least common denominator of the fractions (in reduced form) in $T_A$. Then, $A$ solves binary consensus for any number of processes if and only if $A$ solves binary consensus for $2d + 1$ processes.

As an example, Theorem 4 yields a cutoff bound of 5 for Paxos (Figure 6.5) and a cutoff bound of 7 for the OneThirdRule algorithm (Figure 6.3). Like with the zero-one principle, we only sketch the main proof ideas here and refer to the Appendix C for the details.

We start by giving an overview of our proof technique and providing intuition for the choice of our cutoff bound $B = 2d + 1$. The details differ slightly depending on the consensus property considered. We first explain the general approach, which is the same for all the properties, and focus on the differences afterwards. Analogous to the proof of the zero-one principle, we show that we can turn any large counterexample of width $k > B$ into a small counterexample of width $B$. The trace inflation lemma allows us to ignore systems of sizes below $B$, by inflating small counterexamples. Our proof technique are simulations on transition systems (Section 2.3). We thus must find a relation $R$ relating similar states $s_s$ and $s_l$ of the small and large systems respectively. To motivate $R$, we recall the main condition for forward simulation, the step condition. For any $s_s$, $s_l$, $s'_l$, and $\Lambda$, we must prove:

$$(s_s, s_l) \in R \Rightarrow s'_l \Rightarrow \exists s'_s, s_s \Rightarrow s'_s \wedge (s'_s, s'_l) \in R. \quad \text{(s-step)}$$

To define the relation $R$, we observe that guards, and thus also steps, are agnostic to absolute numbers of processes; they only use fractional thresholds and compare relative frequencies of values. Hence, we relate states of different sizes based on the frequencies of values from $V_\perp$, expressed as fractions of the number of processes. We discretize these fractions into size-independent slots $\{0, \frac{1}{d}, \frac{2}{d}, \ldots, \frac{d-1}{d}\}$, since only $d$-denominated fractions appear in the algorithm’s guards. The state $s_s$ must then be wide enough to accommodate the $s_l$-slot of each value from $V_\perp$. In Appendix C we show that $2d + 1$ is the smallest such width. We now give more details of the simulation relation and our proof; we assume $V = \mathbb{B}$ in this section.
6. Cutoff Bounds for Consensus Algorithms

6.4.1 Core elements of the simulation relation

Given two natural numbers \( n \) (the system’s width) and \( d \) (with \( d \geq 2 \)), we define two sets \( T \triangleq \{0, \frac{1}{d}, \ldots, \frac{d-1}{d}\} \) and \( T_0 \triangleq T \cup \{-\frac{1}{3d}\} \), and a function \( \gamma_n : \{0, \ldots, n\} \to T_0 \) defined as

\[
\gamma_n(c) = \begin{cases} \left\lceil \frac{c}{d} \right\rceil \frac{1}{d}, & \text{when } c > 0 \\ -\frac{1}{3d}, & \text{when } c = 0. \end{cases}
\]

The function \( \gamma_n \) maps process counts to slots, where \( \gamma_n(c) \) yields the smallest threshold in \( T_0 \) exceeded by the count \( c \). For example, for a state \( s \) of width \( n \), \( \gamma_n(\#[s(f)](v)) \) denotes the slot of processes holding value \( v \) in field \( f \).

Given two multisets \( M_l \) and \( M_s \) of sizes \( k \) and \( B \) respectively, we define:

\[
(M_s, M_l) \in \text{cntMS} \triangleq \forall v \in V_{\bot}. \gamma_B(M_s(v)) = \gamma_k(M_l(v))
\]

\[
(M_s, M_l) \in \text{cntMS}_\geq(W) \triangleq \forall v \in W. \gamma_B(M_s(v)) \geq \gamma_k(M_l(v))
\]

\[
(M_s, M_l) \in \text{cntMS}_{\sum\geq}(W) \triangleq \gamma_B(\sum_{v \in W} M_s(v)) \geq \gamma_k(\sum_{v \in W} M_l(v)).
\]

The first relation requires the slot of each value from \( V_{\bot} \) to be exactly the same in both multisets. Sometimes this will be too strong a requirement, and we will switch to the other two relations, which are weaker (the first two relations can be expressed in terms of the last one, but we retain them for convenience). For example, for the OneThirdRule algorithm, we have \( d = 3 \), \( B = 7 \) and \( T = \{0, \frac{1}{3}, \frac{2}{3}\} \). To help intuition, we provide several examples of the above relations. Take \( k = 13 \) and take:

\[
M^1_s = \{4 \times 0, 3 \times 1\}
\]

\[
M^1_l = \{5 \times 0, 8 \times 1\}
\]

\[
M^2_s = \{5 \times 0, 7 \times 1, 1 \times \bot\}
\]

\[
M^3_s = \{4 \times 0, 9 \times 1\}.
\]

Then, we have the following distribution of \( \gamma_{M_l}(M(v)) \):

<table>
<thead>
<tr>
<th>( M )</th>
<th>( \gamma_{M_l}(M(0)) )</th>
<th>( \gamma_{M_l}(M(1)) )</th>
<th>( \gamma_{M_l}(M(\bot)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M^1_s )</td>
<td>( \frac{1}{3} )</td>
<td>( \frac{1}{3} )</td>
<td>( -\frac{1}{9} )</td>
</tr>
<tr>
<td>( M^1_l )</td>
<td>( \frac{1}{3} )</td>
<td>( \frac{1}{3} )</td>
<td>( -\frac{1}{9} )</td>
</tr>
<tr>
<td>( M^2_s )</td>
<td>( \frac{2}{3} )</td>
<td>( \frac{2}{3} )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( M^3_s )</td>
<td>( 0 )</td>
<td>( \frac{2}{3} )</td>
<td>( -\frac{1}{9} )</td>
</tr>
</tbody>
</table>

Thus, \( (M^1_s, M^1_l) \in \text{cntMS}_{\geq}; \ (M^1_s, M^2_l) \in (\text{cntMS}_{\geq}(V) \cap \text{cntMS}_{\sum\geq}(V)) \) but \( (M^1_s, M^3_l) \notin \text{cntMS}_{\geq} \). We also have \( (M^1_s, M^3_l) \in \text{cntMS}_{\sum\geq}(V) \) but \( (M^1_s, M^3_l) \notin \text{cntMS}_{\geq}(V) \).
These relations form the basis of our simulation relation $R$. As the goal of this section is to provide some intuition about the proofs, we limit our exposition of the relation to some salient points. For example, for all $(s_s, s_l) \in R$ we require:

\[(\#[s_s(\text{inp})], \#[s_l(\text{inp})]) \in \text{cntMS}_{\geq}(V).\]  

(ip-rel)

Similar conditions relate the other fields. The exact relation used depends on both the property we are proving, and on the field’s position in the phase graph. The next subsection provides some more details, focusing on the core language $\text{ConsL}$ (without extensions) for simplicity.

### 6.4.2 Simulating steps

Given a step $s \xrightarrow{\Lambda} s'$ in a trace, define $U$ to be the set of all upd-fields appearing in the step’s instructions. Recall from Section 6.1.4 that $U$ is the global update associated with the step. We let $u_p(f) = U(f)(p)$ and call $u_p$ the local update of the process $p$. Our strategy is to prove (s-step) in three main stages.

1. **Simulate local updates**: for any local update $u_p$ possible from $s_l$, prove that there exists a set $W$ such that any process whose HO set is $W$ can also perform the local update $u_p$ from $s_s$.

2. **Simulate global updates**: given any global update $U_l$ associated with a step $s_l \xrightarrow{\Lambda} s'_l$, combine the local updates from the previous step to construct a global update $U_s$ associated with a step $s_s \xrightarrow{\Lambda} s'_s$, such that $U_s$ is related to $U_l$ by a particular multiset relation. For example, for updates before the fork point, we require that, for all fields $f$ that are updated in the step,

\[(\#[U_s(f)], \#[U_l(f)]) \in \text{cntMS}_{=}.\]

3. **Simulate update applications**: given $s_l \xrightarrow{\Lambda} s'_l$, the associated update $U_l$, and the similar small update $U_s$ possible from $s_s$, show that we can permute the local updates in $U_s$ such that, when applied to $s_s$, they yield an $s'_s$ with $(s'_s, s'_l) \in R$, thus ensuring (s-step). The permutation is necessary when the updated field $f$ is $\text{inp}$, since then the update itself does not completely determine the new values of $f$, as the old values are used as fallback. For instance, if $\#[s_s(\text{inp})] = M_s^1 = \{4 \times 0, 3 \times 1\}$, we can construct two global updates $U_1$ and $U_2$ with $\#[U_1(\text{inp})] = \#[U_2(\text{inp})] = \{4 \times \bot, 3 \times 0\}$, such that applying $U_1$ to $s$ yields a state $s_s'$ with $\#[s'_s(\text{inp})] = \{7 \times 0\}$, and $U_2$ leaves $s_s$ intact.

Stage (1) is relatively straightforward, while the next two stages are significantly more involved. Stage (2) is complicated by the fork points, which make constructing similar global updates a non-trivial combinatorial problem. The restriction (R2) is crucial in solving it. In Stage (3), a problem
arises when the inp field is updated. For example, take states $s_s$ and $s_l$ such that $\#[s_s(\text{inp})] = M_s^1$ and $\#[s_l(\text{inp})] = M_l^1 = \{5 \times 0, 8 \times 1\}$. There exists an update $U_l$ with $\#[U_l(\text{inp})] = \{11 \times \bot, 1 \times 0, 1 \times 1\}$, that yields a state $s'_l$ with $\#[s'_l(\text{inp})] = M_l^3 = \{5, 8\}$. However, the reader can check that no update $U_s$ with $(\#[U_s(\text{inp})], \#[U_s(\text{inp})]) \in \text{cntMS}_= \text{cntMS}_\Sigma$ can yield a state $s'_s$ when applied to $s_s$, such that $(\text{inp-rel})$ holds for $s'_s$ and $s'_l$. Hence, we might be forced to use a $U_s$ with $(\#[U_s(\text{f})], \#[U_s(\text{f})]) \in \text{cntMS}_= \text{cntMS}_\Sigma$ for other fields $f$ updated by $U_s$. After the fork point, we hence weaken the Stage (2) relation to

$$(\#[U_s(f)], \#[U_l(f)]) \in \text{cntMS}_=(W) \cap \text{cntMS}_\Sigma=(W),$$

for an appropriate $W \subseteq V_\bot$. For ephemeral fields, this also implies that the simulation relation between $s'_s$ and $s'_l$ must replace $\text{cntMS}_=$ by $\text{cntMS} \geq (W)$ and $\text{cntMS}_\Sigma \geq (W)$. The choice of $W$ depends on the property whose violation we want to preserve.

**Agreement and non-triviality.** Preserving agreement and non-triviality violations requires the small system to make decisions whenever the large system makes them. Thus, we choose $W = V$. This suffices to show that whenever a process in the large system successfully updates a field with some value $v \in V$, the processes in the small system can also perform that update. Our choice of $W$ might force updates to happen in the small system where none happened in the large system, but this is acceptable for the violations we wish to preserve.

**Termination.** Preserving termination violations requires exactly the opposite: whenever a process in the large system fails to update a field and uses a fallback value, the same must be possible in the small system. Recalling the semantics of $\text{ConsL}$, updates fail for two reasons: an insufficient number of non-$\bot$ messages have been received, or different values have been received and the guard uses an $\text{all=}$ predicate. Choosing $W = \{\bot\}$ preserves the first cause of failure, but not the second one. Choosing $W = V$ preserves the second cause, but not the first one. Thus, the correct choice depends on the step from $s_l$ to $s'_l$, and cannot be determined in advance, which is a problem for forward simulation [LV95]. To overcome the problem, we resort to a backward-forward simulation (Section 2.3), which enables us to switch between the two choices of $W$ on-the-fly. As our transition systems are all finitely-branching, by Theorem 2 backward-forward simulation ensures the inclusion of infinite traces, crucial for termination.
6.5 Experimental Results

We combine Theorems 3 and 4, the finite representations of the phase sequence and the communication predicates, and the techniques from [TS07] for handling unbounded timestamps to turn model checking into a decision procedure for ConsL algorithms and consensus. One simply encodes the HO model of a ConsL algorithm for $\Pi = \{1, \ldots, B\}$ and $V = B$ in the input language of the model checker and verifies it. We have built a tool that automatically generates the appropriate Promela model and LTL properties for a ConsL algorithm for the Spin model checker [Hol04]. As case studies, we generated models of different algorithms found in the literature (Table 6.1). Our verification times confirm that the above decision procedure is applicable in practice, with modest resources.

The tool and the generated models are available online [Mar16]. For simplicity, our tool handles only a subset of phase sequence and communication predicate specifications described in Section 6.1.2. To improve performance, the tool implements several optimizations. First, it reduces the model’s branching factor. In a naive modeling approach, in every round in which the uniformity formula $\phi_{uf}$ does not hold, each of the $B$ processes first chooses its HO set independently and then performs a local update based on this HO set, yielding a branching factor of $2^{B^2}$. Instead, the tool-generated models first calculate the possible local updates and let each process pick one of them, lowering the branching factor to typically 2 or 3. Second, the tool reduces the state space by exploiting the symmetry in the system and applying a counter abstraction. The abstraction is sound and complete. For leaderless algorithms this is immediate since guard satisfaction (Section 6.1.4) is defined exactly on multisets; for leader-based algorithms, we need an additional variable to track the state of the leader process in the abstraction. Lastly, the tool optimizes the algorithm to reuse ephemeral fields where possible, thus further shrinking the state space.

### Table 6.1: Experimental results. Time is shown in seconds.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Bound</th>
<th>Agreement</th>
<th>Termination</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time States</td>
<td>Time States</td>
</tr>
<tr>
<td>Algorithm from [MS15]</td>
<td>5</td>
<td>1.17 1,368,956</td>
<td>1.19 1,370,414</td>
</tr>
<tr>
<td>Paxos</td>
<td>5</td>
<td>0.89 1,135,730</td>
<td>0.93 1,151,691</td>
</tr>
<tr>
<td>Chandra-Toueg</td>
<td>5</td>
<td>0.85 1,032,371</td>
<td>0.89 1,048,332</td>
</tr>
<tr>
<td>Paxos (3 rounds)</td>
<td>5</td>
<td>0.70 853,003</td>
<td>0.73 866,917</td>
</tr>
<tr>
<td>OneThirdRule</td>
<td>7</td>
<td>0.04 67,578</td>
<td>0.04 70,070</td>
</tr>
<tr>
<td>Ben-Or</td>
<td>5</td>
<td>0.03 42,478</td>
<td>0.03 45,348</td>
</tr>
<tr>
<td>Coordinated Uniform Voting</td>
<td>5</td>
<td>0.02 39,650</td>
<td>0.02 39,948</td>
</tr>
<tr>
<td>Simplified Coordinated</td>
<td>5</td>
<td>0.01 27,304</td>
<td>0.01 27,616</td>
</tr>
<tr>
<td>Uniform Voting</td>
<td>5</td>
<td>0.01 17,238</td>
<td>0.01 17,385</td>
</tr>
</tbody>
</table>

6.5 Experimental Results
6. Cutoff Bounds for Consensus Algorithms

6.6 Related Work

The general version of the parametric verification problem was shown to be undecidable by Apt and Kozen [AK86]. Suzuki [Suz88] showed that this holds also when the parameter is the number of replicated processes, each having a fixed state space. The small scope hypothesis is folklore, and was implicitly formulated by Jackson and Damon [JD96], and empirically investigated for data structures written in Java by Andoni et al. [ADKM03], for answer-set programs by Oetsch et al. [OPP+12], and for distributed systems by Yuan et al. [YLZ+14].

Cutoff bounds. Cutoff bounds have been devised for several classes of algorithms and properties: for token-ring systems by Emerson and Namjoshi [EN03]; for systems with existential and systems with universal guards by Emerson and Kahlon [EK00]; for cache coherence protocols by Emerson and Kahlon [EK03]; for rectangular hybrid automata by Johnson and Mitra [JM12]; for software transactional memories by Guerraoui et al. [GHJS08]. Kaiser et al. [KKW10] devise a method for determining cutoff bounds for the thread-state reachability problem dynamically, by performing a partial search of the state space. Abdulla et al. [AHH15] use similar ideas, but their results apply to a larger class of systems. None of the aforementioned results applies to consensus algorithms or other types of fault-tolerant distributed systems. The only cutoff result that we are aware of in this area is by Delzanno et al. [DTT14]. They derive cutoff bounds for the proposer and learner roles of Paxos, but not the acceptor role, for which they perform only bounded verification. We adopt the more common model where all processes play all the roles.

Other (semi-)automated methods. Some parametric systems can be modeled as well-quasi-ordered systems [ACJT96] for which the reachability problem is decidable. However, fault-tolerant distributed systems rely on threshold guards for which no suitable well-quasi-ordering is known to exist. Two recent works have explored alternative approaches for the parametric verification of fault-tolerant distributed systems. John et al. [JKS+13] introduce a threshold abstraction for systems based on a type of threshold guards, loosely similar to ConsL guards. Combined with a counter abstraction, their technique yields a sound, but incomplete verification procedure for next-free LTL properties. They apply it to several simpler fault-tolerant algorithms, but the method does not apply to consensus algorithms. The main limitation of the approach is that the protocol’s message space must be finite. Thus, round-based algorithms (where messages are tagged by their round numbers) are out of scope. Furthermore, in consensus algorithms with an unbounded value domain, the message space grows with the width of the system, even when only input values are sent around. A follow-up
work [KVW14], shows that bounded model checking can be used as a sound verification method for reachability properties of their systems, which provides a more efficient verification procedure for such properties. Another subsequent improvement [KVW15] relies on SMT solvers and a search space pruning technique, conceptually similar to partial-order reduction, to improve the method further. The use of SMT solvers also eliminates some, but not all sources of imprecision in the abstraction. However, neither of these improvements removes the limitations mentioned above.

Drăgoi et al. [DHV⁺14] introduce the consensus logic $\mathcal{CL}$, aimed at verifying the properties of round-based consensus algorithms, and a domain specific language for it [DHZ16]. $\mathcal{CL}$ is strictly more expressive than $\text{ConsL}$, and can encode algorithms for the synchronous and Byzantine settings. They provide a semi-decision procedure for invariant checking, which performs well in their experiments, and a full decision procedure for invariant checking for a fragment $\mathcal{CL}_{\text{dec}}$ whose expressive power is incomparable to $\text{ConsL}$. Their method is only semi-automated, since the user must find the appropriate invariants, and is not guaranteed to give an answer for $\mathcal{CL}$ (outside of $\mathcal{CL}_{\text{dec}}$), since it is based on a semi-decision procedure.
Part III

Conclusions and Future Work
In this thesis, we have investigated the problem of formally verifying fault-tolerant systems. In the first part, we looked at the problem of ensuring the consistency of persistent memory in the presence of crashes and memory faults, using an industrially deployed persistent memory manager as our driving case study. The main challenges for this system’s correctness (and thus its verification) stem from the rampant non-determinism caused by the combination of possible crashes and restarts and hardware failures. The latter have not been considered in the literature before, and they greatly increase the system’s complexity, forcing us to develop a verification approach which could scale appropriately.

The key points of the approach are as follows. We use a structured (rather than event-based) model. This helped us keep the models understandable, eased discussions with IBM researchers, and enabled compositional reasoning. Modeling restarts synchronously significantly reduced the number of cases we had to consider in the proofs. We prove two useful proof rules that allow us to decompose the verification problem in a very natural fashion, along the boundaries of the logical system layers. The same proof rules were recently independently rediscovered in [SBTW16], testifying to their general usefulness in the verification of crash-prone systems.

The second part of the thesis investigated the consensus problem in fault-tolerant distributed systems. Although we do not explicitly leverage the results of the first part here, we note that there does exist a connection: the algorithms for the crash-recovery setting almost universally require the correct recovery of their persistent state after a restart. Our first contribution in this part was a unified description of a number of consensus algorithms found in the literature: OneThirdRule, $A_{T,E}$, Paxos, Chandra-Toueg, Ben-Or, UniformVoting, and the generic algorithm of [GR04]. By using refinement, we could (1) describe the main algorithmic ideas behind them in simple terms and (2) create a taxonomy of the algorithms based on these ideas. We hope that we have also shed light on why the algorithms are con-
7. Conclusions and Future Work

structed the way they are. Finally, the insights gained from the taxonomy helped us develop a new algorithm, which is leaderless, tolerates $n_{\text{fail}} < \frac{n}{2}$ failures, and does not employ waiting to guarantee safety. This answered a question posed in [CBS09] asking whether such an algorithm exists.

Even though the consensus problem has been thoroughly studied, we believe that this development provides both a useful synthesis of existing knowledge about the algorithms we cover and a novel way of understanding and relating them. In particular, the Voting model and the no defection property provide a simple basis for describing the different algorithms, a basis we have not seen in the literature before. The development also provides a verification framework for consensus algorithms that fit our abstractions. For example, the verification of the new algorithm presented in Section 5.8 was made much simpler by the verification of its parent abstract model, compared to a verification from scratch.

Nevertheless, this verification task still involved a non-negligible manual effort. In practice, many users instead opt for fully automated verification approaches (such as model checking) whose soundness hinges on the informal small scope hypothesis. The second contribution of the second part of the thesis is the specification language $ConsL$ for consensus algorithms, for which we derive a zero-one principle and cutoff bounds for verifying consensus properties. The language covers a relevant and nontrivial class of consensus algorithms. Our bounds are algorithm-dependent, but fairly small, either 5 or 7 for our case studies. This formally proves the small scope hypothesis for this class. Furthermore, viewing our result as an empirical data point, we believe that it also lends additional credibility to the small scope hypothesis for fault-tolerant distributed algorithms in general. Finally, the bounds are small enough to be within the reach of standard model checking methods, yielding the first fully automated verification procedure for consensus algorithms.

**Future Work**

Our PMM development lacks an executable implementation. However, we believe that deriving one from our concrete model would only require a modest effort, leveraging modern Isabelle tools for C code [GAK12]. Another option for obtaining executable code would be to move to the framework proposed in [SBTW16]. The functionality of the PMM closely corresponds to the first layer of their file system, which they verify fully automatically. However, this layer provides no resilience to storage failures, and it is significantly simpler than the PMM, since it uses a standard write-ahead log. It would be interesting to see if their approach scales to the PMM’s complexity.

Idempotence is an important notion not only in the context of storage systems, but also in distributed systems [Hel12]: intuitively, since responses
to requests might get lost, duplicated requests are possible and their application should be idempotent. A lot of our proof effort was spent precisely on proving the idempotence of the startup handler and its parts. Our approach to this proof, as well as the one of [SBTW16], boils down to a brute-force search of the state space. While [SBTW16] achieve full automation, we are unsure how this scales to larger systems. In [CZC+15], the idempotence is phrased by exhibiting a crash invariant that holds at any program state. We think that this is also sub-optimal, as such invariants tend to be cumbersome in our experience. Thus, finding better methods of verifying idempotence is an interesting question. The only work that we are aware of in this area is [RV13], but it assumes the existence of a transactional key-value store.

Finally, while error resilience is reflected in our requirements, our formalization does not quantify it, offering no way to compare it in two systems. One possibility to address this would be to switch to a probabilistic model. However, it is unclear if this would be a feasible undertaking at this point, given the current state of tools for probabilistic verification.

One obvious limitation of the work presented in the second part of the thesis is the focus on benign failures. We believe that our refinement framework can be extended to a Byzantine failure model with a reasonable effort, as refinement has been used to adapt consensus algorithms to the Byzantine setting before [Lam01b, Lam11]. For the cutoff bounds, the situation is less clear. On one hand, the ability to arbitrarily change messages would give us more freedom in constructing the small state; on the other hand, the large system would also have more behaviors that must be considered. Furthermore, the resulting bounds would likely be higher than for the benign setting. Since Byzantine consensus is impossible when a third or more processes are corrupted [AW04], Byzantine-tolerant algorithms use thresholds with a denominator of at least 3, and sometimes also 5 or 7. Besides Byzantine faults, there also exist algorithms for the benign setting which fit neither our refinement development nor ConsL. An example is the Fast Paxos [Lam06] algorithm. The algorithm can be seen as a combination of $A_{T,E}$ and standard Paxos, which are placed in different parts of our refinement tree. We believe that our development could be extended to accommodate the algorithm, perhaps by turning it into a graph instead of a tree. For the cutoff bounds, the situation is less clear. The algorithm could fit ConsL syntactically, using the $\text{maxts}$ and all= predicates, with a slight redefinition of all=’s semantics and the removal of the restriction (R8). However, our efforts to extend the simulation relation sketched in Section 6.4 to cover this combination of predicates failed. Namely, we could not come up with a method to preserve the eligibility of both 0 and 1 after an update. More precisely, we were unable to show $s'_0 \models C(b)$ whenever $s'_1 \models C(b)$, for any $b \in B$ and $\text{pred}(C) \in \{\text{maxts}, \text{all=}\}$, where $s'_0$ and $s'_1$ are as in (s-step). Still, as we do not have a proof that such a combination would violate the small scope hypothesis, we leave the question of combining these predicates open.
Another possible direction is to extend the scope of our results to target higher-level primitives that build on consensus algorithms, such as total-order broadcast. While these primitives can be completely reduced to consensus, for efficiency reasons they often use consensus in a white-box fashion. For example, in Lamport’s total-order broadcast extension of Paxos [Lam01a], the same message is sometimes used to execute multiple consensus instances. Another example is Egalitarian Paxos [MAK13], which intertwines the different conceptual consensus instances by attaching constraints to values.

Next, the abstract models in our refinement development capture only the safety guarantees of the target algorithms. It is unclear whether and how this could be extended to termination in a simple fashion. This is not a problem for our development, as proofs of termination in the HO model tend to be simple, thanks to the communication predicates. However, there exists no systematic way of implementing messaging layers that provably ensure these communication predicates. The two previous constructions of such layers that we are aware of, [HS07] and [DHZ16], are ad-hoc and targeted at a particular consensus algorithm each. A modular approach to implementing useful HO messaging layers is an open question. In contrast, while it is also easy to define failure detectors that have no reasonable implementation (and there can thus also be no systematic way of implementing arbitrary detectors), the situation there is still better: there exists a fair number of easily implementable detectors (such as those from [CT96]) that can be modularly used to build easily implementable algorithms.

The last direction for future work that we consider is the further investigation of the zero-one principle. This principle appears natural for any kind of agreement property, as disproving agreement requires only two values, and since no new values need to be computed. While we manage to prove the principle for a substantial class of algorithms, our proofs are anything but elegant, and they require several restrictions on the basic ConsL syntax. A possible alternative to prove the principle would be to encode the transition relation using a logic, and prove a small-model theorem for a relevant fragment of the logic. Unfortunately, the transition relation involves quantification over sets, as well as cardinality constraints, so the desired fragment would probably lie quite far from the classic fragments which have known small-model theorems [BGG97]. Still, recent decidability results for logics such as [YPK10] give some hope that such a fragment could exist. Whatever the approach to proving it might be, the zero-one principle would be useful for other verification approaches, even without the associated cutoff bounds.

The lack of such a result is precisely the reason why the threshold abstraction of [JKS+13] cannot be applied to consensus algorithms: the local state space of each process becomes parametric in the number of processes in a way that the method cannot handle, since more processes mean that more inputs are possible. Extending the zero-one principle could then immediately increase the scope of the method [JKS+13] as well.
Appendix A

Proofs for ConsL: Common Definitions and Results

Since our algorithms describe distributed systems, we will also interchangeably refer to them as protocols. We fix an algorithm $\mathcal{A}$ written in ConsL throughout, and denote by $\mathcal{F}$ the set of all fields that appear as a send-field or upd-field in the round specifications of $\mathcal{A}$. Recall from Section 6.3.1 that constraints are triples $C = (f, th, pred)$ where $f \in \mathcal{F}$, and $th$ and $pred$ are as for guards. We note that constraints are isomorphic to pairs $(f, G)$, where $G$ is a guard; we will implicitly convert between the two views where convenient. A constraint $C$ is an algorithm constraint for $\mathcal{A}$ if there exists a step and a guard $G$ such that the send-field of the step is $f$, $G$ appears in the instructions of the step, and $C = (f, G)$. We denote the set of all algorithm constraints of $\mathcal{A}$ by $C(\mathcal{A})$.

In the remainder, we will often refer to algorithm states of different sizes, that is, with a different number of processes. Given a state $s$, we will use $\Pi(s)$ to denote the set of processes which appear in $s$. Furthermore, we will often abuse notation, discarding the round number from a (global) state $s$ and treating $s$ like a function, either of type $\Pi(s) \to \mathcal{F} \to \mathcal{V} \cup (\mathbb{N} \times \mathcal{V}) \cup \{\perp\}$ or of type $\mathcal{F} \to \Pi(s) \to \mathcal{V} \cup (\mathbb{N} \times \mathcal{V}) \cup \{\perp\}$, whichever order of arguments is more suitable at the moment. The types of arguments will suffice to distinguish which order is meant.

Given a state $s$, a set $W \subseteq \Pi(s)$, a constraint $C = (f, G)$, and a value $v \in \mathcal{V}$, we say that $s$ and $W$ satisfy $C(v)$, written $s, W \models C(v)$ if

$$\#[s(f) \downarrow W] \models G(v)$$

where $g \downarrow S$ denotes the partial function that is the restriction of the function $g$ to the set $S$. We note that $s \models C(v)$ iff there exists a $W$ such that $s, W \models C(v)$.

For two constraints on the same field, we extend the guard ordering. Note that, as for guards, if $C_1 \sqsubseteq C_2$, then $s, W \models C_2(v)$ implies $s, W \models C_1(v)$. 

119
A. Proofs for ConsL: Common Definitions and Results

For the fixed algorithm, recall the definition of $T$ from Section 6.4.1. For a fraction $t \in T$, and a number $n$, define $\alpha_n : T \rightarrow \{0, \ldots, n\}$ by:

$$\alpha_n(t) = \begin{cases} 0 & \text{if } t < 0 \\ \lfloor t \cdot n \rfloor + 1 & \text{if } 0 \leq t \end{cases}$$

Intuitively, the function $\alpha$ calculates the smallest number of processes in an $n$-process system that exceeds the threshold given by $t$.

Assume we are given a step $s \rightarrow s'$ of a HO transition system, where the set of (syntactically) updated fields is $U = \{f_i \mid i \in I\}$, with the corresponding guards $G_i$. The step is determined by some HO collection and ldr function. Assume we are also given a process $p \in \Pi(s)$, and that $p$ received messages $\mu^r_p$ in the step $s \rightarrow s'$. Then, we say that $p$ performed the local update $u_p : U \rightarrow \{0, 1, \perp\}$ if for all $f_i \in U$:

- $u_p(f_i) = s'(f_i, p)$, when $\#[\mu^r_p] \models G_i(s'(f_i, p))$; and
- $u_p(f_i) = \perp$, otherwise.

It is easy to see that $u_p$ is uniquely defined, given the HO collection. The same holds for any labeled step $s \xrightarrow{\Lambda} s'$. Moreover, given the set $\{u_p \mid p \in \Pi(s)\}$ of local updates performed by the processes in a step $s \rightarrow s'$ or $s \xrightarrow{\Lambda} s'$, we say that the global update $U : \mathcal{F} \rightarrow \Pi(s) \rightarrow \{0, 1, \perp\}$ happened in the same step, if $U(f_i)(p) = u_p(f_i)$ for all fields $f_i$ in the update instructions of the step and all $p \in \Pi$.

Given just the step but not the HO collection, there can be multiple possible global updates associated with a step. We adopt the following convention. Whenever we prove a statement quantified over all steps, we automatically also quantify over all HO collections that could give rise to the step. In this context, we will say “the” global update associated with a step where we mean “some” global update. However, we will never rely on the update being unique. Furthermore, we will also say “the” global update when we need to prove the existence of a step; in this case, we also prove the existence of the HO collection which can give rise to that particular global update.

Note that a local update of $\perp$ does not always imply that the new value of the field is $\perp$. The relation between a state transition $s \rightarrow s'$ and its associated global update $U$ is described, for each process $p$ and updated field $f \in U$, as follows:

$$s'(p, f) = \begin{cases} U(p, f) & \text{if } U(p, f) \in \{0, 1\} \text{ or } f \text{ is ephemeral} \\ s(p, f) & \text{if } U(p, f) = \perp \text{ and either} \\ & f = \inp \text{ and } \mathcal{A} \text{ is not randomized or} \\ & f = \dec \\ \text{any } x \in \{0, 1\} & \text{if } U(p, f) = \perp, f = \inp, \text{ and} \\ & \text{the algorithm } \mathcal{A} \text{ is randomized} \end{cases}$$
Finally, we use the notation $\tau(r) \sim \tau(r')$ to denote an execution fragment $\tau$ that starts with a state $\tau(r)$ and ends with a state $\tau(r')$.

**Lemma A.1** (Dropping and permuting updates). Assume $U$ is the global update associated to a step $s \xrightarrow{\Lambda} s'$, where $cp$ is not $\sim_3$. Given any other global update $U'$ such that the set of local updates used in the global update is a subset of the set of local updates used in $U$, there exists a step $s \xrightarrow{\Lambda} s''$ such that $U'$ is the associated global update.

*Proof.* To make a process $p$ perform the same local update as a process $q$, simply use $q$’s $HO$ set as the $p$’s $HO$ set in the step $s \xrightarrow{\Lambda} s''$. By an enumeration of the possible round labels in $\Lambda$, it is easy to see that this step can also be labeled by $\Lambda$. \hfill $\Box$

By the restriction (R1), we note that, within each phase, only the fields that lie on the path from $inp$ to $dec$ in the phase graph can affect the consensus properties. Moreover, only the fields on the path from $inp$ to $inp'$ can affect the later phases. The following lemma then helps us simplify the type of algorithms we need to consider.

**Lemma A.2** (Stripping the phase graph.). *An algorithm $A$ solves consensus if and only if the algorithm $A'$ solves consensus, where $A'$ is the algorithm created by removing:*

1. *all steps where the send-field of the step lies neither on the path $inp \rightarrow inp'$ nor the path $inp \rightarrow dec$ in the graph of the phase where the step appears.*
2. *all instructions where the upd-field lies neither on the path $inp \rightarrow inp'$ nor the path $inp \rightarrow dec$ in the graph of the phase where the instruction appears.*

Thus, WLOG, we may assume the following properties of $A$:

(PS1) The set of instructions of any step must have either one or two elements.

(PS2) At most one set of instructions within the same phase can have two elements.

(PS3) All fields used within a phase lie on the path $inp \rightarrow inp'$ or on the path $inp \rightarrow dec$ (or possibly on both)

Given a step in which two fields $f_1$ and $f_2$ are updated, we will often use the notation $(v_1 v_2)$ to denote the local update $\{f_1 \mapsto v_1, f_2 \mapsto v_2\}$. We refer to local updates written in this way as *tiles.*
A.1 Inflating Trace Sizes

All our results will use the same strategy: converting a “complicated” counterexample to a consensus property (e.g., an example that uses a $V \neq \{0, 1\}$, or an example of a large size) into a simpler one. For this, it will often be useful to assume that the complicated example is of a large enough size. We use $\Sigma_n$ to denote a system state consisting of $n$ processes.

**Definition A.1** (Local traces). Given a labeled trace $\tau$, the local trace of a process $p$ is the projection of $\tau$ onto $p$’s state, removing the labels from $\tau$.

**Definition A.2** (Locally equivalent traces). Two traces $\tau$ and $\tau'$ (not necessarily on the states of the same size) are **locally equivalent** if they have an identical set of local process traces.

**Lemma A.3** (Trace inflation). Given a trace $\tau$ on $\Sigma_n$, we can “inflate” it to a trace $\tau'$ on $\Sigma_{i\cdot n}$, for any integer $i$. More precisely, for any trace $\tau$ on $\Sigma_n$ and non-zero $i \in \mathbb{N}$, there exists a trace $\tau'$ on $\Sigma_{i\cdot n}$ which is locally equivalent to it. Moreover, $\tau'$ has the same labels as $\tau$.

**Proof.** The trace $\tau'$ consists of $i$ copies of each process in $\tau$. If the algorithm uses a leader, in each phase use some arbitrary copy of the leader as the $\tau'$ leader.

In $\vartriangleleft$ rounds, have the copies hear from the leader iff the original heard from the leader in $\tau$. For other rounds, given the HO sets $HO_p^r$ of the $n$-process system, construct the HO sets $HO'_p^r$ of the $i \cdot n$ process system to consist of the $i$ copies of each process in $HO_p^r$. Clearly:

1. for any witness $W$ of the $n$-process system, $|W| > th(C) \cdot n \iff i|W| > th(C) \cdot (in)$
2. the relative ordering of values in $W$ and their relative frequencies remain the same in $W'$

Thus, we conclude

$$\tau(v), HO_p^r \models C(v) \iff \tau'(v), HO'_p^r \models C(v).$$

So the processes in the $(i \cdot n)$-process system can perform the exact same local updates as the processes in the $n$-process system; we let each $\tau'$ process copy performs the same update that its original performed in $\tau$ (by selecting the same value, or drawing the same randomness).

It’s easy to see that this construction yields the same labels in $\tau'$ as in $\tau$. \[ \square \]

**Lemma A.4** (Trace inflation to $j \cdot d + 1$). Given a trace $\tau$ on $\Sigma_n$, we can construct a locally equivalent trace $\tau'$ on some $\Sigma_m$, such that $\gcd(d, m) = 1$, and $m \geq n$. 

122
Proof. Let \( z = \min_{t \in T_1} (\alpha_n(t) - tn) \). Note that \( z = 1 - \max_{t \in T_1} \frac{tn}{\alpha_n(t)} \). We apply Lemma A.3 with \( i = \lceil \frac{z}{2} \rceil d \) to obtain a trace \( \tau'' \) with \( in \) processes that is locally equivalent to \( \tau \), with sets \( HO_p' \) corresponding to sets \( HO_p \) in \( \tau \). Let \( m = in + 1 \). Obviously, \( \gcd(d, m) = 1 \) and \( m \geq n \). Now obtain the trace \( \tau' \) with \( m \) processes by duplicating an arbitrary local trace from \( \tau'' \) (together with the corresponding \( HO \) sets and randomness). Do not include the newly created process in any \( HO \) sets, and have every process retain the same \( HO \) sets as in \( \tau \).

We now show that the satisfaction of guards is not affected by the addition of the new process. Let \( s \) be such that \( 0 \leq s \leq n \) and let \( W \) be a set of processes such that \( |W| = is \). Furthermore, let \( t \in T_1 \). Note that if \( |W| \leq tin \), then clearly also \( |W| \leq t(in + 1) \). Suppose now that \( |W| > tin \). We have to show that also \( |W| > t(in + 1) \). First, observe that

\[
|W| > tin \iff is - tin > 0 \iff s > tn
\]

so by our choice of \( z \) we get \( s - tn \geq z \) and thus by our choice of \( i \)

\[is - tin = i(s - tn) \geq iz \geq 2 > 1.\]

Therefore, since \( 0 \leq t < 1 \), we have

\[|W| = is > tin + 1 > t(in + 1)\]

as required. \( \square \)
Appendix B

Proof of the 0-1 Principle for ConsL

We now show the zero-one principle for ConsL programs: if there exists a trace violating agreement or termination, then there also exists a 0-1, or binary trace violating agreement or termination. A trace is 0-1 if all values appearing in the trace are either 0 or 1. The name is derived from an analogous result for sorting networks.

In the remainder of this section, we assume that we have fixed a program $A$ in ConsL. Additionally, we assume that the algorithm is not randomized. We also assume that all states are of size $k = m \cdot d + 1$, for some $m$. This assumption is justified by Lemma A.4. Lastly, for every field $f$ of the algorithm, we add a constraint $(f, 0, \text{any})$ to $C(A)$, even if such a constraint is not used in the algorithm. This addition cannot violate restriction (R2).

B.1 Preliminaries

Definition B.1 ($\Sigma^b$). $\Sigma^b$ is the set of global states for which the values of all fields of all processes are either 0, 1, or $\bot$.

Definition B.2 (Binary trace). A binary trace is a trace consisting of states from $\Sigma^b$.

Definition B.3 (twovals). For a multiset $M$, the predicate twovals holds if there exist two values $v_1, v_2 \in V$ such that $v_1 \neq v_2$ and $M(v_1) > 0$ and $M(v_2) > 0$.

Note that twovals($M$) is false for any multiset over $\mathbb{N} \times V$.

Lemma B.1 (Constraint monotonicity). Assume $s \models C(v)$ and let $f = f(C)$. Then, for any set $P$, and any state $s'$ such that $\text{rnd}(s') \geq \text{rnd}(s)$,
B. Proof of the 0-1 Principle for \(\text{ConsL}\)

\[ ldr(s') = ldr(s) \text{ and } \]

\[ s'(f)(p) = \begin{cases} 
  v & \text{if } p \in P, \text{ and } (f \neq \text{inp or inp} \in V) \\
  \text{rand}(s'), v & \text{if } p \in P, f = \text{inp and inp} \in (\mathbb{N} \times V) \\
  s(f)(p) & \text{otherwise} 
\end{cases} \]

we also have \(s'| = C(v)\).

**Proof.** By a case distinction on \(\text{pred}(C)\) and the definition of constraint satisfaction. \(\Box\)

**Definition B.4** (Simultaneous satisfaction). We say that \(s\) simultaneously satisfies \(C_1(v_1)\) and \(C_2(v_2)\), written \(s \models C_1(v_1) \land C_2(v_2)\), if \(s \models C_1(v_1)\) and \(s \models C_2(v_2)\).

**Definition B.5** (Trivial constraint). For any field \(f\) used in the algorithm, we define a special constraint \(\text{True}_f\), such that \(\text{True}_f \sqsubseteq C\) for any other constraint \(C\), and \(s \models \text{True}_f(v)\) for any state \(s\) and any value \(v \in V\).

Since the field \(f\) will typically be clear from the context, we will often write \(\text{True}_f\) as simply \(\text{True}\).

**Definition B.6** (Modified constraint set). We define \(C'(A)\) to be the set:

\[ C(A) \cup \left( \bigcup_{f \in F} \text{True}_f \right). \]

**Definition B.7** (Maximum constraint of a value). For a state \(s\), a field \(f\), and a value \(v\), define the maximum constraint of \(v\), written \(\text{max}_C(s, f, v) = C\) such that:

- \(C \in C'(A)\)
- \(s \models C(v)\)
- \(f(C) = f\)
- \(\forall C' \in C'(A). \ s \models C'(v) \implies C' \sqsubseteq C\)

Note that this is well-defined by restriction (R2).

**Definition B.8** (Constraint pair ordering). We define a partial ordering on pairs of constrains as the product order:

\[ (C_1, C_2) \sqsubseteq (C'_1, C'_2) \iff C_1 \sqsubseteq C'_1 \land C_2 \sqsubseteq C'_2. \]

**Definition B.9** (Strongest simultaneously satisfied constraints). For a state \(s\) and a field \(f\), define the strongest simultaneously satisfied constraints on \(s\) and \(f\), written \(\text{max}_{\text{sim}}(s, f)\), to be a pair of constraints \((C_1, C_2) \in (C'(A))^2\) such that:
B.1. Preliminaries

- \( f(C_1) = f(C_2) = f \)
- \( C_1 \sqsupseteq C_2 \)
- there exist values \( v_1, v_2, v_1 \neq v_2 \), such that \( s \models C_1(v_1) \land C_2(v_2) \)
- for all values \( w_1, w_2, w_1 \neq w_2 \), and all constraints \( C'_1, C'_2 \) such that \( f(C'_1) = f(C'_2) = f, s \models C'_1(w_1) \land C'_2(w_2) \) and \( C'_1 \sqsupseteq C'_2 \), we have that \( (C'_1, C'_2) \sqsubseteq (C_1, C_2) \).

In general, it does not make sense to define \( \text{max}_\text{sim} \) for multiple states; it is possible to create two states \( s \) and \( s' \) such that:

- \( \text{max}_\text{sim}(s, \text{inp}) = (C_1, C_2) \)
- \( \text{max}_\text{sim}(s', \text{inp}) = (C'_1, C'_2) \)
- \( C'_1 \sqsubset C_1 \), but \( C'_2 \sqsubseteq C_2 \)

For example, take constraints (all on the input field) \( C_1 = (0, \text{any}) \), \( C_2 = (\frac{1}{3}, \text{all} \*) \) and \( C_3 = (\frac{2}{3}, \text{all} \*) \). We have \( C_1 \sqsubset C_2 \sqsubset C_3 \), and a state can simultaneously satisfy \( C_1 \) and \( C_3 \), or \( C_2 \) and \( C_2 \), but not \( C_2 \) and \( C_3 \).

We introduce a further abuse of notation. For a multiset \( M \) and a constraint \( C \) (or a guard \( G \)), and a value \( v \), we will write \( M \models C(v) \) if there exists a state \( s \) such that \( s \models C(v) \) and \( \#s(f(C)) = M \). We extend the notational abuse to \( M, W \models C(v), M \models C_1(v_1) \land C_2(v_2) \) and \( \text{max}_\text{sim}(M, f) \) in the obvious way. Note that if \( M \models C(v) \) and \( \#s(f(C)) = M \), we have \( s \models C \).

We say that \( M \) is a binary multiset if it only contains (possibly timestamped) values 0, 1 and \( \perp \).

B.1.1 Preserving updates

Lemma B.2 (Preserving failed local single-field updates). Assume \( s \xrightarrow{sf} s' \).
Let \( sf \) be the send-field of the step, let \( U \) be the global update corresponding to the step, and let \( U \) be the list of updated fields, and assume \( f \in U \). Let \( cp \) be the cp of the step.
Assume \( t \) is a state such that \( \text{rnd}(t) = \text{rnd}(s) \), and:

1. \( \#t(sf)(\perp) \geq \#s(sf)(\perp) \)
2. if twovals(\( \#s(sf) \)), then also twovals(\( \#t(sf) \))

if \( cp \) was not \( \neq \), and:

1. \( s(sf)(\text{ldr}(s)) = \perp \) implies \( t(sf)(\text{ldr}(t)) = \perp \)
otherwise.

Then, if $cp$ was not $\leq$, and some process performed a local update $\{ f \mapsto \bot \}$ in the step $s \xrightarrow{\Lambda} s'$, or if $cp$ was $\leq$ and $U(f)(ldr(s)) = \bot$, then there exists a set $W_\bot$ of processes such that using $W_\bot$ as a HO does not violate any threshold of leader formulas in $\Lambda$, and for any step $t \xrightarrow{\Lambda} t'$, setting the HO set of any process $q$ to $W_\bot$ results in $q$ performing a local update with $u_q(f) = \bot$.

**Proof.** Assume that it was the update $u_p$ (of process $p$) that failed in the step $s \rightarrow s'$. By the semantics of $\text{next}$, we conclude that, if $cp \neq \bot$, either:

1. too few non-$\bot$ messages are received by $p$; say only $j$, with $|HO(p)| - j$ messages being $\bot$. The conditions of our lemma ensure that we can construct a set $W_\bot$ of the same size as $HO(p)$, such that setting $HO(q)$ to $W_\bot$ results in only $j$ non-$\bot$ values being received by $q$.

2. the failed guard was $\text{all} = \bot$, and $p$ received messages with two different values, $v_1$ and $v_2$. The conditions of the lemma ensure that we can form $W_\bot$ of the same size, such that setting $HO(q)$ to $W_\bot$ results in both 0 and 1 being received by $q$.

As $|W_\bot| = |HO(p)|$, this does not violate any threshold or leader receive formulas in $\Lambda$, and leader send formulas do not appear since we assumed $cp \neq \bot$.

If $cp = \bot$, the result is immediate from the semantics of $\text{next}$.  

**Definition B.10** ($\bot$-extensions). Given a function $h : V \rightarrow \{0,1\}$, its $\bot$-extension is the function $h_\bot : V_\bot \rightarrow \{0,1,\bot\}$ defined as:

$$h_\bot(x) = \begin{cases} 
\bot, & \text{if } x = \bot \\
h(x), & \text{otherwise}
\end{cases}$$

**Lemma B.3** (Preserving successful local single-field updates). Let $s \xrightarrow{\Lambda} s'$ be a step and denote the associated send-field by $sf$, the associated set of updated fields by $U$, the associated global update by $U$, and the cp by $cp$.

Assume $f \in U$ and the associated constraint was $C$, and assume $U(f)(p) = v \neq \bot$, for some $p$ and $v$. Furthermore, assume $\text{rnd}(t) = \text{rnd}(s)$ and $t \models C(b)$; if $cp = \leq$, also assume $s(sf)(ldr(s)) = b$.

Then, there exists a set $W$ such that, in any step from $t$:

- any process, if $cp \neq \leq$
- the coordinator, if $cp = \leq$

whose HO set is set to $W$ can perform a local update that sets $\{ f \mapsto b \}$; moreover, using $W$ as the HO set cannot violate any threshold or coordinator predicates in $\Lambda$.  

128
Proof. By semantics of next, noting (L2), (L3), and (R4). □

Lemma B.4 (Single field global updates). Let \( s \) be a state and \( s_b \) be a binary state, with \( \text{rnd}(s_b) = \text{rnd}(s) \). Let \( \mathcal{U} \) be the global update associated with a step \( s \xrightarrow{A} s' \), let \( sf \) be the send-field of the step, let \( U \) be the list of fields updated in the step, and let \( cp \) be the cp of the step. If \( cp \neq \bot \), assume \( \#(s_b(sf))(\bot) \geq \#(s(sf))(\bot) \), and assume \( \text{twovals}(\#(s(sf))) \) implies \( \text{twovals}(\#(s_b(sf))) \).

Let \( f \in U \), and assume its corresponding constraint is \( C \). Assume that \( h : \mathcal{V} \to \{0, 1\} \) is such that:

1. if \( cp \neq \bot \), for all \( v \in \#(U(f)) \), \( s_b \models C(h(v)) \);
2. if \( cp = \bot \), \( s_b(sf)(ldr(s_b)) = h_\bot(s(sf))(ldr(s)) \),

where \( h_\bot \) is the \( \bot \)-extension of \( h \).

Then, there exists a step \( s_b \xrightarrow{A} s_b' \), such that for the associated update \( \mathcal{U}_b \) we have \( \mathcal{U}_b(f)(p) = h_\bot(\mathcal{U}(f)(p)) \).

Proof. We perform a case distinction on whether \( \phi_{of} \in \Lambda \).

If \( \phi_{of} \notin \Lambda \), we simply use Lemmas B.3 and B.2 to yield the required HO sets.

If \( \phi_{of} \in \Lambda \), by the semantics of next we have that \( \#(U(f))(\bot) > 0 \iff \#(U(f))(\bot) = k \). If \( \#(U(f))(\bot) > 0 \), we apply Lemma B.2 to obtain a set \( W \), and use this as the HO set of all the processes. Otherwise, \( \#(U(f))(\bot) = 0 \).

If \( \text{twovals}(\#(U(f))) \) holds, then \( \text{pred}(C) = \text{any} \) (since the other constraints are deterministic) and \( \text{twovals}(\#(s(sf))) \). Then, if for some values \( v_1 \) and \( v_2 \) we have \( h(v_1) \neq h(v_2) \) and \( \#(U(f))(v_i) > 0 \), for \( i \in \{1, 2\} \), by the condition on \( h \) we conclude that two different values also appear in \( s_b \), and we let the HO set of all the processes be \( \Pi \). Otherwise, there exists a \( b \) such that \( h(U(f)(p)) = b \), for all \( p \); obtain a set \( W \) from Lemma B.3, and set the HO sets of all the processes to \( W \). □

Lemma B.5 (Binarizing ephemeral fields is easy). Let \( f(C) \) be an ephemeral field, and assume \( M, W \models C(v) \). Let \( h : \mathcal{V} \to \{0, 1\} \) be an arbitrary function, let \( h_\bot \) be its \( \bot \)-extension, and let \( M' \) be the multiset image of \( M \) under \( h_\bot \). Then, \( M', W \models C(h(v)) \).

Proof. Follows from \( M'(h_\bot(v)) \geq M(v), h(x) \neq \bot \), guard satisfaction relation, and (R3). □

Lemma B.6 (Mixed tile preservation). Assume a local update \( (v_1) \), where \( v_1 \neq v_2 \), occurs in a step \( s \xrightarrow{A} s' \) where the send-field is \( sf \) and the updated fields are \( f_1 \) and \( f_2 \). Denote the constraints corresponding to the guards by \( C_1 \) and \( C_2 \). Assume \( s_b \models C_1(b_1) \land C_2(b_2) \) for a binary state \( s_b \), where \( b_1 \neq b_2 \), and \( \text{rnd}(s_b) = \text{rnd}(s) \). Then, there exists a set \( W \) such that:
B. Proof of the 0-1 Principle for ConsL

\begin{itemize}
  \item any process, if the cp was not \succ
  \item the coordinator, if the cp was \preceq
\end{itemize}

that has \( W \) for its HO set can perform the local update \( (b_1', b_2') \), for some \( b_1' \neq b_2' \), and such that using \( W \) as the HO set cannot violate any threshold or leader formulas in \( \Lambda \).

**Proof.** First, we note that since \( (v_1, v_2) \) appeared, the step was not a \( \preceq \) step. Furthermore, neither \( C_1 \) nor \( C_2 \) can be \( \min \), and not both are selections (since selections are deterministic).

If both \( C_1 \) and \( C_2 \) are \( \text{any} \), then the result is easy. Otherwise, one of them is a selection, and the other one is \( \text{any} \). WLOG, assume \( C_1 \) is a selection. Since \( s_b \models C_1(b_1) \), there exist at least \( \alpha_k(\text{th}(C_1)) \) non-\( \bot \) values in \( s_b(sf) \). We take \( W \) to be \( \Pi \); this contains both \( b_1 \) and \( b_2 \) in \( s_b(sf) \), as \( s_b \models C_1(b_1) \land C_2(b_2) \). The selection will choose one of the values, and we let \( \text{any} \) choose the other. \( \square \)

**Lemma B.7** (Preserving \( \bot \) in simultaneous updates). Assume two fields \( f_1 \) and \( f_2 \) are updated in a step \( s \xrightarrow{\Lambda} s' \) with \( sf \) as the send-field; denote the associated constraints by \( C_1 \) and \( C_2 \), and the cp by \( cp \).

Assume a \( (v, \bot) \), \( v \neq \bot \) tile appears as the local update of a process \( p \), and assume \( \text{rnd}(s_b) = \text{rnd}(s) \). Given some \( b \in \{0, 1\} \), assume \( s_b \models C_1(b) \). If \( cp \neq \preceq \), assume \( \#[s_b(sf)](\bot) \geq \#[s(sf)](\bot) \), and assume that twovals(\( \#[s_b(sf)] \)) implies twovals(\( \#[s(sf)] \)). Lastly, assume \( \text{pred}(C_1) \neq \min \) or \( b = 0 \).

Then, there exists a set \( W \) such that, for any step from \( s_b \):

\begin{itemize}
  \item any process, if \( cp \neq \preceq \)
  \item \( \text{ldr}(s_b) \), if \( cp = \preceq \)
\end{itemize}

that has \( W \) for its HO set in the step can perform the local update \( (b) \), and that using \( W \) as the HO set cannot violate any threshold or coordinator formulas in \( \Lambda \).

**Proof.** Denote the process that performed the update \( (v, \bot) \) by \( p \), and denote its HO set in the step by \( HO_p \). Denote the subset of \( HO_p \) from whom \( p \) received non-\( \bot \) messages by \( V \). We note that \( |V| \geq \alpha_k(\text{th}(C_1)) \). Furthermore, the existence of \( (b) \) implies that:

1. by (R5), this cannot be a \( \preceq \) step, and hence also \( \phi_{ls} \notin \Lambda \)
2. by (R8), neither of \( C_1 \) and \( C_2 \) are \( \max \).
3. \( C_1 \sqsubset C_2 \).

We perform a case distinction:
1. $|V| < \alpha_k(\text{th}(C_2))$. Since $s_b \models C_1(b)$, we can find a set $W'$ of size $\alpha_k(\text{th}(C_1))$, so that $s_b, W' \models C_1(b)$. If using $W'$ as a HO set does not violate any threshold or coordinator formulas in $\Lambda$, we are done. Otherwise, since $\text{th}(C_1) < \text{th}(C_2)$, from (L3) we conclude that $\text{th}(C_1) = 0$, and then from (R4) that $C_1 = (0, \text{any})$. We select as $W$ any set of size $|\text{HO}_p|$ such that $s_b(sf)[W](b) > 0$ and $s_b(sf)[W](\bot) = s(sf)[\text{HO}_p](\bot)$.

2. $|V| \geq \alpha_k(C_2)$. From the semantics of $\text{pred}$, we conclude that $\text{pred}(C_2) = \text{all}^=$, that two values exist in $s(sf)$ (and thus also in $s_b(sf)$) and that $\text{pred}(C_1) \neq \text{all}^=$. Performing a case distinction on $\text{pred}(C_1)$, we conclude that, when $b = 0$ or $\text{pred}(C) \neq \min$, there exists a witness $W$ such that $s_b, W \models C_1(b)$ and $s_b(\text{inp})[W](1 - b) > 0$.

We note that both choices of $W$ enable it to be used as a HO set without violating any threshold or coordinator formulas in $\Lambda$. □

**Lemma B.8** (Homogeneous tile preservation). Assume two fields $f_1$ and $f_2$ are updated in a step $s \xrightarrow{\Lambda} s'$ with $sf$ as the send-field; denote the associated constraints by $C_1$ and $C_2$, and the cp by $cp$. Further, assume a $(t_s)$ tile, for some $v \in V$ appears as the local update of a process $p$. Let $C = \max(C_1, C_2)$.

Then, given any $s_b$ and any $b$ such that $s_b \models C(b)$ and $\text{rnd}(s_b) = \text{rnd}(s)$, there exists a set $W$ such that, for any step from $s_b$:  

- any process, if $cp \neq \not\supseteq$
- $ldr(s_b)$, if $cp = \not\supseteq$

that has $W$ for its HO set in the step can perform the local update $(t_s)$, and that using $W$ as the HO set cannot violate any threshold or coordinator formulas in $\Lambda$.

*Proof.* WLOG, assume $C_1 \sqsubset C_2$. Use Lemma B.3 to find the set $W$ that allows an update $\{C_1 \mapsto b\}$ from $s_b$, while respecting $\Lambda$. Since $C_1 \sqsubset C_2$, the same set $W$ also allows the update $(t_s)$. □

**Lemma B.9** (Empty tile preservation). Assume $s \xrightarrow{\Lambda} s'$. Let $sf$ be the send-field of the step, let $U$ be the global update corresponding to the step, and let $cp$ be the cp of the step.

Assume $t$ is a state such that $\text{rnd}(t) = \text{rnd}(s)$, and:

1. $\#(t(sf))(\bot) \geq \#(s(sf))(\bot)$
2. if $\text{twovals}(\#(s(sf)))$, then also $\text{twovals}(\#(t(sf)))$

if $cp$ was not $\not\subset$, and:

1. $s(sf)(ldr(s)) = \bot$ implies $t(sf)(ldr(t)) = \bot$
B. Proof of the 0-1 Principle for ConsL

otherwise.

Then, if some process \( p \) performed an empty local update in the step \( s \xrightarrow{\Lambda} s' \), where \( p \) is \( \text{ldr}(s) \) if \( \text{cp} = \rightarrow \) and any process otherwise, then there exists a set \( W_\bot \) of processes such that using \( W_\bot \) as a HO set does not violate any threshold predicates in \( \Lambda \), and for any step \( t \xrightarrow{\Lambda} t' \), setting the HO set of any process \( q \) to \( W_\bot \) results in \( q \) performing an empty local update.

Proof. Take the field with the weakest constraint in the list of fields updated in the step, and apply Lemma B.2 to yield the set \( W_\bot \).

**Lemma B.10** (Multiple tiles with \( \phi_{uf} \)). Assume two fields \( f_1 \) and \( f_2 \) are updated in \( s \xrightarrow{\Lambda} s' \), and denote the associated constraints by \( C_1 \) and \( C_2 \). Furthermore, assume \( \phi_{uf} \in \Lambda \) and two tiles \((x,y)\) and \((w,z)\) appear in the associated global update. Then, from any state \( t \) such that \( \text{rnd}(t) = \text{rnd}(s) \), there exists a HO set \( W \) such that for any two values \( w_1 \) and \( w_2 \) such that \( t \models C_1(v_1) \) and \( t \models C_2(v_2) \), the tiles:

- \((v_1,y)\), if \( x \neq w \) and \( y = \perp \)
- \((x,v_2)\), if \( y \neq z \) and \( x = \perp \)
- \((v_1,v_2)\), if \( x \neq w \) and \( y \neq z \)

can be used, together with \((x,y)\), by any process whose HO set is \( W \).

Proof. From \( \phi_{uf} \in \Lambda \), the semantics of labels and next, and the existence of \((x,y)\) and \((w,z)\) with \( w \neq x \), we conclude that the \( \text{cp} \) was \( \right\uparrow \), and that \( \text{pred}(C_1) = \text{any} \); the result then follows from the semantics of constraint satisfaction. The proof of the other items is similar.

**B.2 Agreement**

The idea: ensure that the two strongest constraints on the input field satisfied by some two different values at the start of the critical phase of a concrete, multi-valued trace are also satisfied at the start of the critical phase of the binary trace. Depending on the scenario, we then choose to decide 0 or 1 (or both) in the critical phase, and ensure that the state of the input field at the end of the critical phase enables us to decide the other bit (that is, the bit that has not been yet decided on, if it exists) later.

**B.2.1 Binary companions**

**Definition B.11** (Binary companions). Given a multiset \( M \) and a field \( f \), let \( (C_1,C_2) = \max_{\sim}(M,f) \). A binary multiset \( M_b \) is a binary companion for \( M \) with respect to \( f \) if \( |M_b| = |M| \) and:

---

132
1. $M_b \models C_1(1) \land C_2(0)$

2. if $M$ and $M_b$ are multisets over $\mathcal{V} \perp \ (\text{and not } \mathbb{N} \times \mathcal{V})$, for all $v \neq \perp$, $M_b(1) \geq M(v)$

3. $M_b(\perp) = M(\perp)$

**Lemma B.11** (Binarizing $\text{max}_{\text{sim}}$). For any multiset $M$, any field $f$, and any $ts \in \mathbb{N}$, there exists a binary companion $M_b$ to $M$ with respect to $f$, such that all timestamps in $M_b$ are $ts$.

**Proof.** Let $(C_1, C_2) = \text{max}_{\text{sim}}(M, f)$. Perform a case distinction:

1. There exists a $C_i$, $i \in \{1, 2\}$, such that $C_i$ is a selection. By (R3) we conclude that $f = \text{inp}$, and that there are no $\perp$ values in $M$, and item 3 of the binary companion definition is satisfied. Perform another case distinction on the type of $\text{pred}(C)$:
   - min. If the other constraint, $C' = C_{3-i}$ is True, set all values in $M$ to 1 to obtain $M_b$. Otherwise, set a single value in $M_b$ to 0, and set all the other values to 1. It is easy to see that this satisfies both items 1 (due to our choice of $k$, $\alpha_k(\text{th}(C)) < k$) and 2 (since $M$ contains at least two values).
   - smor. Perform a further case distinction, on the type of the other constraint. By (R2), the only possibilities are:
     - True. Create $M_b$ by setting all values in $M$ to 1.
     - any. Create $M_b$ by using a single 0 and setting the other values to 1. Since $s$ is of size $k$, there exists a witness for $C_1(1)$ that does not include the 0, and item 1 is satisfied. Obviously, item 2 is satisfied as well.
     - smor. Create $M_b$ by setting $\lceil \alpha_k(\text{th}(C_2)) \rceil$ values to 0, and the rest to 1. For any $s_b$ with $s_b(\text{inp}) = M_b$ and any set of processes $W$ of size $\alpha_k(\text{th}(C_1))$ that contains all the processes that hold 0’s, $s_b, W \models C_2(0)$. Take any $W'$ of size $\alpha_k(\text{th}(C_1))$ that leaves out at least one process that holds 0. This is always possible by our assumption on the state width, and since $\text{th}(C_1) < 1$. Then, since $\text{th}(C_1) \geq \text{th}(C_2)$, $s_b, W' \models C_1(1)$. This ensures item 1. For item 2, notice that it cannot be that for some $v$, $\#s(\text{inp})(v) > \#s_b(\text{inp})|(1)$, since otherwise $s \not\models C_2(v')$ for any $v' \neq v$.
     - all=, Use the same $M_b$ as in the previous case. Notice that since, for all $v$, $M_b(1) \geq M(v)$, $M \models C_1(v)$ implies $M \models C_1(1)$.
   - maxs. By (RS) and the definition of constraint satisfaction on multisets, $C_{1-i} \in \{\text{True}, \text{maxs}\}$. If True, we set all the elements
B. Proof of the 0-1 Principle for ConsL

to \((ts, 1)\). Otherwise, create \(M_b\) by taking a single \((ts, 0)\), and
taking \((ts, 1)\) for the other elements.

2. None of \(C_i\) are a selection. Of \(v_1, v_2\), map the one with the higher count
in \(M\) to 1, and the other one to 0. Map all the other non-\(\perp\) values to 1.
It is easy to see that for the resulting \(M_b\), \(M_b|C_1(1); C_2(0)\), and
that \(M(1)b \geq M(v)\) for any \(v \neq \perp\).

\[
\text{Definition B.12 (Best companion). Let } M \text{ be a multiset over } \mathcal{V} \text{ or } \mathbb{N} \times \mathcal{V}
\text{(that is, it does not include } \perp \text{ values) Its best companion is the multiset}
B(M) \text{ such that:}
\]
- \(B(M)\) is a binary companion to \(M\) with respect to \(\text{inp}\)
- for any other binary companion \(M_b\) of \(M\), \(B(M)(0) \geq M_b(0)\).

B.2.2 Intra-phase preservation of binary companions

We note that, by Lemma B.11, \(B\) is well-defined. We also extend \(B\) to a
relation on states, where \(s_b \in B(s)\) if \(\text{rnd}(s_b) = \text{rnd}(s)\) and:

- \(\text{inp}\) is of type \(\mathcal{V}\), and \#\[s_b(\text{inp})]\] = \(B(\#[s(\text{inp})])\) and
- \(\text{inp}\) is of type \(\mathbb{N} \times \mathcal{V}\), and \#\[s_b(\text{inp})]\] is any binary companion of \#\[s(\text{inp})]\).

\[
\text{Lemma B.12 (0 resilience). Let } M = \{i \times v, (k-i) \times v'\}, \text{ for some } v, v' \in \mathcal{V}
\text{ and some } i \text{ and } j, \text{ and assume } M \models C(1-b). \text{ Let } M' = \{i \times 1, (k-i) \times 0\}. Then, } M' \models C(0).
\]

Proof. By case distinction on the type of \(C\) and semantics of constraint satisfaction.

\[
\text{Lemma B.13 (1 resilience). Let } M = \{i \times v, (k-i) \times v'\}, \text{ for some } i \text{ and } j,
\text{ assume } M \models C(v), \text{ and assume } \text{pred}(C) \neq \text{min}. \text{ Let } M' = \{i \times 1, (k-i) \times 0\}. Then, } M' \models C(1).
\]

Proof. By case distinction on the type of \(C\) and semantics of constraint satisfaction. For \(\text{smor}\), note that our choice of \(k\) guarantees that \(\text{th}(C)\) is an
odd number, and thus \(i \geq \lceil \frac{\alpha(k(\text{th}(C)))}{2} \rceil \).

\[
\text{Lemma B.14 (Increasing } C_1 \text{ count). In the following, assume all the con-
straints are on the input field. Assume } s \rightarrow s', \text{ denote the associated global}
update by } U, \text{ and let } \text{max}_{\text{sim}}(s, \text{inp}) = (C_1, C_2), \text{ and } \text{max}_{\text{sim}}(s', \text{inp}) = (C'_1, C'_2). \text{ Assume } s \models C_1(v). \text{ Furthermore, assume that } \#U(\text{inp})(x) > 0
\text{ implies } x \in \{v, \perp\}.
\]

Then:
1. for any value $w \neq v$ and any constraint $C$, $s' \models C(w)$ implies $C \sqsubseteq C_2$. 

2. $C'_1 \sqsubseteq C_1$

3. if $\text{inp}$ is of type $V$, then $B(\#s'(\text{inp}))(1) \geq B(\#s(\text{inp}))(1)$

Proof. If the algorithm is timestamped, the claims are immediate from the semantics of $\text{next}$ and $\text{maxts}$. Since the algorithm is not randomized, and $v$ is the only value written, by the semantics of $\text{next}$ we conclude, for any set $S$, $\#s'[S](v) \geq \#s[S](v)$, and for $w \neq v$, $\#s'[S](w) \leq \#s[S](w)$.

We prove item 1 by contradiction. Assume $w \neq v$, $s', W \models C(w)$ and $C \sqsubseteq C_2$. Proceed by a case distinction on $\text{pred}(C)$:

- any or $\text{all}= -$ obviously contradicting, since there are no $\bot$ values in $\text{inp}$, only $v$ gets written, and $s, W \not\models C(w)$.

- a selection. Then, since, there are no $\bot$ values in the $\text{inp}$ field and $|W| > \alpha(\theta h(C))$, there must exist a value $v'$ such that $s, W \models C(v')$. Since $C_2 \sqsubseteq C$, it must be $v' = v$. By Lemma B.1 we conclude $s', W \models C(v)$. Since selections are deterministic, this contradicts the assumption $w \neq v$.

For item 2, we note that, since there exists a $W$ such that $s, W \models C_1(v)$, by Lemma B.1 we also have $s', W \models C_1(v)$ and thus $C'_1 \sqsubseteq C_1$. Item 3 then follows from $C'_1 \sqsubseteq C_1$ and the definition of $B$. \hfill $\square$

**Definition B.13** (Last two different values). Given an execution fragment $\tau(r) \leadsto \tau(r')$, such that $r' > r$ and the rounds $r \ldots (r' - 1)$ belong to the same phase, and given a field $f$ such that $f$ is updated in some step $\tau(r_f) \rightarrow \tau(r_f + 1)$, where $r \leq r_f < r'$, denote by $l2v(\tau(r) \leadsto \tau(r'), f)$ some pair of values $(v_1, v_2)$, $v_1 \neq v_2$ such that:

1. if there exists a round $r_2$, $r \leq r_2 < r'$, such that for the update $\mathcal{U}_2$ associated to the step $\tau(r_2) \rightarrow \tau(r_2 + 1)$, a field $f_2$ that lies on the path to $f$ is updated in the step, and $\text{twovals}(\#\mathcal{U}_2(f))$ holds; then letting $r_2$ be the highest such round, $v_1, v_2 \in \#\mathcal{U}_2(f_2)$.

2. no such round exist, but there exists a round $r_2$, $r \leq r_2 < r'$, such that for the send field $s f_2$ associated to the step $\tau(r_2) \rightarrow \tau(r_2 + 1)$, a field $f_2$ that lies on the path to $f$ is updated in the step, and $\text{twovals}(\#\tau(r_2)(s f_2))$ holds; then letting $r_2$ be the highest such round, $v_1, v_2 \in \#\tau(r_2)(s f_2)$.

3. $v_1$ and $v_2$ are undefined otherwise

**Definition B.14** (Values on a path). Let $\tau(r) \leadsto \tau(r')$ be an execution fragment, such that $r' > r$ and $r \ldots (r' - 1)$ all belong to the same phase. Let $f$ be a field that is updated in some step $\tau(r_f) \rightarrow \tau(r_f + 1)$ of the fragment,
with the send field $sf$ and update $U_f$. Then, denote by $pvals(\tau(r) \leadsto \tau(r'), f)$ the set of all values $v$ such that either:

1. $v \in \#[U_f(f)]$
2. $r_f > r$ and $v \in pvals(\tau(r) \leadsto \tau(r_f), sf)$

**Lemma B.15** (Single-path update preservation). Assume $r$ and $r' \geq r$ belong to the same phase, and let $\tau$ be an execution fragment. Let $U$ be the global update of the step $\tau(r') \rightarrow \tau(r' + 1)$, and let $U'$ be list of fields updated in this step. Let $sf$ be the send-field of the step $\tau(r) \rightarrow \tau(r + 1)$, and let $cp$ be the cp of the same step.

Then, given any field $f \in U$, any $h : \mathcal{V} \rightarrow \{0,1\}$, and any binary state $s_b$, such that:

1. $\text{ rnd}(s_b) = r$
2. (a) if $cp \notin \mathcal{C}'_i$:
   i. twovals($\#[\tau(r)(sf)]$) implies twovals($\#[s_b(sf)]$)
   ii. $\#[s_b(sf)](\bot) \geq \#[\tau(r)(sf)](\bot)$
   iii. if $C$ is the constraint on $sf$ on the path from $sf$ to $f$, for any value $v \in pvals(\tau(r) \leadsto \tau(r' + 1), f)$, $s_b \models C(h(v))$.

(b) if $cp = \mathcal{C}'_i$: $s_b(sf)(ldr(s_b)) = h_\bot(\tau(r)(sf)(ldr(\tau(r))))$

3. given the largest round $r''$, $r \leq r'' \leq r'$, such that $sf''$ is the send-field of the step $\tau(r'') \rightarrow \tau(r'' + 1)$ and twovals($\#[\tau(r'')(sf)]$) holds, we have twovals($\#[h_\bot \circ \tau(r'')(sf)]$)

it is possible to create an execution fragment $\tau_b$ such that:

- $\tau_b(r) = s_b$
- the labels on $\tau_b(r) \leadsto \tau_b(r' + 1)$ match the labels on $\tau(r) \leadsto \tau(r' + 1)$
- for the update $U_b$ corresponding to the step $\tau_b(r') \rightarrow \tau_b(r' + 1)$, we have $\#[U_b(f)] = \#[h_\bot \circ U(f)]$.

**Proof.** By induction on the difference $r' - r$. If $r = r'$, the result follows from Lemma B.4. If $r' > r$, given $\tau(r) \leadsto \tau(r' - 1)$ and $\tau(r' - 1) \rightarrow \tau(r')$, let $U''$ be the global update associated with $\tau(r' - 1) \rightarrow \tau(r')$, let $sf'$ be an ephemeral field, and hence $\tau(r')(sf') = U''(sf')$ and $\#[\tau_b(r')(sf')] = \#h_\bot \circ U''(sf')]$. Hence, $\#[\tau(r')(sf')](\bot) = \#[\tau(r')(sf')](\bot)$ and we can additionally permute $U''$ to ensure $\tau_b(r')(sf')(ldr(\tau_b(r'))) = h_\bot(\tau(r')(sf')(ldr(\tau(r')))).$
By the condition on \( h \), we further have that twovals(\(|\tau(r')(sf')|\)) implies twovals(\(|\tau(r')(sf')|\)). Lastly, is \( cp \neq \angle^r \), from Lemma B.5 we have \( \tau(r') \models C_f(v) \) implies \( \tau_b(r') \models C_f(h(v)) \). Thus, the result again follows from Lemma B.4.

**Corollary B.1** (Single path update preservation from \( B \)). Assume the round \( r \) starts a phase, assume \( r' \geq r \) belongs to the same phase as \( r \), and let \( \tau(r) \sim \tau(r'+1) \) be an execution fragment. Let \( cp_1 \) be the \( r \)-th phase. Let \( U \) be the global update of the step \( \tau(r') \rightarrow \tau(r'+1) \), and let \( U \) be the list of fields updated in this step.

Then, given any field \( f \in U \), any \( h : V \rightarrow \{0,1\} \), and any binary state \( s_b \), such that:

1. \( s_b \) is a \( B(\tau(r)) \)

2. given the constraint \( C \) on \( inp \) on the path from \( inp \) to \( f \), for any value \( v \in \text{vals}\( (\tau(r) \sim \tau(r'+1), f) \), \( s_b \models C(h(v)) \).

3. given the largest round \( r'' \), \( r \leq r'' \leq r' \), such that \( sf'' \) lies on the path to \( f \) in the phase graph and that it is the send-field of the step \( \tau(r'') \rightarrow \tau(r''+1) \) and twovals(\(|\tau(r'')(sf'')|\)) holds, we have that also twovals(\(|\tau(r'')(sf')|\)) holds.

it is possible to create an execution fragment \( \tau_b(r) \sim \tau_b(r'+1) \) such that:

1. \( \tau_b(r)(f) = s_b(f) \) for all \( f \in F \)

2. the labels on \( \tau_b(r) \sim \tau_b(r'+1) \) match the labels on \( \tau(r) \sim \tau(r'+1) \)

3. for the update \( U_b \) corresponding to the step \( \tau_b(r') \rightarrow \tau_b(r'+1) \), we have \( \#[U_b(f)] = \#[h \circ \tau(r'')(sf'')]. \)

4. \( U_b(f)(ldr(\tau_b(r'))) = h \circ (U(f)(ldr(\tau(r')))) \).

**Proof.** Condition 1 guarantees the conditions 1 and 2(a)i of Lemma B.15, and condition 2(a)ii of Lemma B.15 does not apply since the send-field of the step \( \tau(r) \rightarrow \tau(r+1) \) is \( inp \). Condition 2 ensures condition 2(a)iii of Lemma B.15 when \( cp_1 \neq \angle^r \). When \( cp_1 = \angle^r \) by (R1) there can be only one value \( v \in \text{vals}\( (\tau(r) \sim \tau(r'+1), f) \}; we choose the coordinator in \( \tau_b(r) \) such that \( \tau_b(r)(inp)(ldr(\tau_b(r))) = h(v) \), ensuring condition 2b of Lemma B.15.

**Corollary B.2** (Single path ephemeral field preservation from \( B \)). Assume the round \( r \) starts a phase, assume \( r' > r \) belongs to the same phase as \( r \), and let \( \tau(r) \sim \tau(r'+1) \) be an execution fragment. Let \( sf' \) be the send-field of the step \( \tau(r') \rightarrow \tau(r'+1) \)

Then, given any \( h : V \rightarrow \{0,1\} \), and any binary state \( s_b \), such that:

1. \( s_b \) is a \( B(\tau(r)) \)
B. Proof of the 0-1 Principle for ConsL

2. given the constraint $C$ on inp on the path from inp to $f$, for any value
   $v \in pvals(\tau(r) \leadsto \tau(r' + 1), f)$, $s_b \models C(h(v))$.

3. given the largest round $r''$, $r \leq r'' \leq r'$, such that $sf''$ lies on the
   path to $f$ in the phase graph and that it is the send-field of the step
   $\tau(r'') \rightarrow \tau(r'' + 1)$ and $twovals(\#[\tau(r'')(sf)])$ holds, we have that also
   $twovals(\#[h \circ \tau(r'')(sf)])$ holds.

it is possible to create an execution fragment $\tau_b(r) \leadsto \tau(b(r' + 1))$ such that:

1. $\tau_b(r)(f) = s_b(f)$ for all $f \in F$
2. the labels on $\tau_b(r) \leadsto \tau(b(r' + 1))$ match the labels on $\tau(r) \leadsto \tau(r' + 1)$
3. $\#[\tau_b(rf)](sf') = \#[h \circ \tau(rf)](sf')$.
4. $\tau_b(r)(sf')(ldr(\tau_b(r'))) = h(\tau(r')(sf')(ldr(\tau_b(r'))))$.

Proof. Immediate from Corollary B.1 and update semantics for ephemeral fields. \hfill \Box

Lemma B.16 (Label-preserving extension). Let $\tau(r) \leadsto \tau(r')$ be a fragment.
Then, given any starting state $\tau_b(r)$, there exists a fragment $\tau_b(r) \leadsto \tau_b(r')$
such that the labels on $\tau_b(r) \leadsto \tau_b(r')$ match the labels on $\tau(r) \leadsto \tau(r')$.

Proof. Set all HO sets in the fragment to $\Pi$. \hfill \Box

B.2.3 Preserving counterexamples until the critical phase

Lemma B.17 (Phase B-extension). Let $r_s$ be a round that starts a phase
and $r_n > r_s$ the round that starts the successive phase. Let $\tau(r_s) \leadsto \tau(r_n)$
be an execution fragment. Then, given any $s_b$ such that $s_b$ is a $B(\tau(r_s))$,
there exists an execution fragment $\tau_b(r_s) \leadsto \tau_b(r_n)$ such that:

1. $\tau_b(r_s)(f) = s_b(f)$, for all $f \in F$
2. $\tau_b(i)$ is a $B(\tau(i))$, for all $i \in \{r_s, \ldots, r_n\}$.
3. the fragment $\tau(r_s) \leadsto \tau(r_n)$ and $\tau_b(r_s) \leadsto \tau_b(r_n)$ have the same labels.

Proof. We proceed by a case distinction on whether inp’ belongs to the phase
graph of the phase of $p$. If it does not, item 2 holds regardless of the updates
performed in the fragment. Thus, any extension obtained from Lemma B.16
suffices.

Thus, assume that inp is updated in round $r_i$ such that $r_s \leq r_i < r_n$.
The difficult part is to create the fragment $\tau_b(r_s) \leadsto \tau_b(r_i + 1)$ such that the
lemma holds; for the fragment $\tau_b(r_i + 1) \leadsto \tau_b(r_n)$, we take any extension
from Lemma B.16.
Let \((v_1, v_2)\) be \(\lnot 2\forall(\tau(r_s) \sim \tau(r_i + 1), \mathbf{inp})\) if defined, and two arbitrary different values from \(\mathcal{V}\) otherwise. Let \(\mathcal{U}\) be the update associated with the step \(\tau(r_i) \rightarrow \tau(r_i + 1)\). We let \(\max_{\text{sim}}(\tau(r_s), \mathbf{inp}) = (C_1, C_2)\). Finally, let \(C\) be the constraint associated with the update on the input update path in the step \(\tau(r_s) \rightarrow \tau(r_i + 1)\).

We perform a case distinction on the number of non-\(\perp\) values in \(\#\mathcal{U}(\mathbf{inp})\):

- \(\geq 2\); this necessarily includes \(v_1\) and \(v_2\). This implies \(\tau(r_s) \models C(v_1) \land C(v_2)\), and by the definition \(\mathcal{B}\), then also \(\tau_b(v_s) \models C(0) \land C(1)\). Invoke Corollary B.1 with \(h(v_1) = 1\), \(h(x) = 0\) for \(x \neq v_1\), to yield the fragment \(\tau_b(r_s) \sim \tau_b(r_i + 1)\) and update \(\mathcal{U}_b\) such that \(\#\mathcal{U}_b(\mathbf{inp})\) is created by taking a binary companion of \(\#\tau(r_n)(\mathbf{inp})\) such that all its timestamps are \(t_s = r_n\) (possible by Lemma B.11), and removing all the timestamps.

- \(0\). Then, \(\tau(r_i)(\mathbf{inp}) = \tau(r_s)(\mathbf{inp})\). Invoke Corollary B.1 with \(h(v_1) = 1\), \(h(x) = 0\) for \(x \neq v_1\), to yield the fragment \(\tau_b(r_s) \sim \tau_b(r_i + 1)\) and update \(\mathcal{U}_b\) such that \(\mathcal{U}_b(\mathbf{inp})(p) = \perp\).

- \(1\); denote the value by \(v\). We conclude \(C \subseteq C_1\). Rotate \(v_1\) and \(v_2\) such that \(v \neq v_2\). We will choose a \(b\) such that \(\tau_b(r_s) \models C(b)\) and apply Corollary B.1 with \(h(v_1) = h(v) = b\) and \(h(x) = 1 - b\) to yield the fragment \(\tau_b(r_s) \sim \tau_b(r_i)\) and the update \(\mathcal{U}_b\) for the step \(\tau_b(r_i) \rightarrow \tau_b(r_i + 1)\) such that \(\#\mathcal{U}_b(\mathbf{inp})\) is created by taking a binary companion of \(\#\tau(r_n)(\mathbf{inp})\) such that all its timestamps. This guarantees item 3. We will further show how to ensure item 2 using Lemma A.1 on \(\mathcal{U}_b\).

If type of \(\mathbf{inp}\) is \(\mathcal{V}\), let \(\delta = \mathcal{B}(\#\tau(r_i + 1)(\mathbf{inp}))(1) - \mathcal{B}(\#\tau(r_s)(\mathbf{inp}))(1)\).

We choose \(b\) as follows:

- if \(\#\mathcal{U}(\mathbf{inp})(v) = k\), choose \(b = 1\). It is easy to see that item 2 is satisfied.

- \(\#\mathcal{U}(\mathbf{inp})(v) < k\). Thus, we also have \(\#\mathcal{U}(\mathbf{inp})(\perp) > 0\). Perform a further case distinction:

  * Type of \(\mathbf{inp}\) is \(\mathcal{N} \times \mathcal{V}\). Note that then, by (R8), the fact that \(\max\text{ts}\) is a selection, and by the definition of \(\mathcal{B}\), for any constraint \(C_{\mathbf{inp}} \subseteq C(A)\) with \(f(C_{\mathbf{inp}}) = \mathbf{inp}\) we have \(\tau_b(r_s) \models C_{\mathbf{inp}}(1)\). If \(C \subseteq C_2\), take \(b = 0\), and invoke Lemma A.1 to yield an update \(\mathcal{U}'_b\) such that \(\#\mathcal{U}'_b(\mathbf{inp})(\perp) = k - 1\) and \(\#\mathcal{U}'_b(\mathbf{inp})(0) = 1\); it is easy to see that we can permute \(\mathcal{U}'_b\) such that that \(\tau_b(r_i) \models C_{\mathbf{inp}}(x)\) for an arbitrary \(C_{\mathbf{inp}}\) and any \(x \in \{0, 1\}\), and that item 2 holds. If \(C \supseteq C_2\), take...
B. Proof of the 0-1 Principle for ConsL

\[ b = 1 \] and invoke Lemma A.1 to yield an update \( U'_b \) with \( \#[U'_b(\text{inp})](\bot) = k \). It is easy to see that this ensures 2. Both constructions also preserve item 3.

* Type of \( \text{inp} \) is \( V \), \( \tau(r_s) \models C_1(v) \) and \( C_2 \sqsubset C_1 \). Then, by Lemma B.14, \( \max_{\text{sym}} \tau(j) = (C'_1, C'_2) \), where \( C'_1 \sqsupseteq C_1 \) and \( C'_2 \sqsubseteq C_2 \), and \( \delta \geq 0 \). We take \( b = 1 \) and apply Lemma A.1 to \( U_b \) to yield an update \( U'_b \) such that \( \#[U'_b(\text{inp})](1) = \delta \), and such that it yields \( B(\#[\tau(r_i + 1)(\text{inp})])(1) \) when applied to \( \tau(r_i)(\text{inp}) \).

* Type of \( \text{inp} \) is \( V \), and either \( \tau(r_s) \not\models C_1(v) \) or \( C_2 = C_1 \). This case is analogous to the previous one, but we can now chose whether to map \( h(v) \) to 1 or to 0. If \( \delta \geq 0 \), we choose \( h(v) = 1 \), otherwise we choose \( h(v) = 0 \).

\[ \square \]

**Theorem B.1.** Given a trace prefix \( \tau|_{l+1} \), such that the round \( l \) starts a phase and twovals(\#[\tau(l)(\text{inp})]) holds, there exists a binary trace \( \tau_b \) such that, for all \( i \in \{0, \ldots, l - 1\} \):

1. \( \tau_b(i) \) is a \( B(\tau(i)) \)
2. the fragments \( \tau(0) \leadsto \tau(l) \) and \( \tau_b(0) \leadsto \tau_b(l) \) have the same labels.

**Proof.** By generalized induction on \( l \).

For \( l = 0 \), for all fields \( f \neq \text{inp} \) all values in \( \tau(0)(f) \) are \( \bot \); we let \( \tau_b(0)(\text{inp}) = B(\tau(0))(\text{inp}) \).

Inductive step. We assume that the round \( l \) starts a phase. Let \( p \) be the round which starts the phase preceding the phase of \( l \). We apply Lemma B.17 to extend \( \tau_b \).

\[ \square \]

**B.2.4 Preserving counterexamples through the critical phase**

**Definition B.15 (Critical round).** Given a trace that violates agreement, the **critical round** of the trace is the first round where some process makes a decision.

**Definition B.16 (Critical phase).** Given a trace that violates agreement, the **critical phase** of the trace is the phase to which the critical round belongs to.

**Lemma B.18 (Propagating dominating values intra-phase).** Assume \( r \) and \( r' \) belong to the same phase \( \phi \), \( r' > r \). Let \( sf \) be the send-field of the step \( r \rightarrow r + 1 \), and assume \( \#[s(sf)](x) = k \). Furthermore, assume that \( f \) is updated in the step \( r' - 1 \rightarrow r' \), let \( cp \) be the cp of that step, and assume that there is a path from \( sf \) to \( f \) in the phase graph of the phase of \( \phi \). Lastly, assume \( \text{rad}(s) = r \).

140
Then, given any fragment $\tau(r) \leadsto \tau(r')$, there exists a fragment $\tau'(r) \leadsto \tau'(r')$, such that $\tau'(r) = s$, the labels on the two fragments match, and for the update $U$ associated to the step $\tau'(r' - 1) \rightarrow \tau'(r')$:

1. if $cp \neq \exists x, U(f)(p) = x$ for all $p$
2. if $cp = \exists x, U(f)(ldr(\tau'(r')))) = x$

Proof. For each step between $r$ and $r'$, we set the HO set of all the processes to $\Pi$. By the semantics of next and constraints, it is easy to see that this yields the desired $U$. \qed

**Lemma B.19** (Propagating a constraint-satisfying value). Assume $r$ and $r'$ belong to the same phase $\phi$, $r' > r$. Let $sf$ be the send-field and $cp_1$ be the cp of the step $r \rightarrow r + 1$. Furthermore, assume that $f$ is updated in the step $r' - 1 \rightarrow r'$, let $cp_2$ be the cp of that step, and assume that there is a path from $sf$ to $f$ in the phase graph of the phase of $\phi$. Let $C$, $f(C) = sf$ be the constraint on the path from $sf$ to $f$ in round $r$, and assume $\tau(r) \models C(v)$ if $cp_1 \neq \exists x$, and $ldr(\tau(r))(sf) = v$ otherwise. Lastly, assume $\text{rand}(s) = r$.

Then, given any fragment $\tau(r) \leadsto \tau(r')$, there exists a fragment $\tau'(r) \leadsto \tau'(r')$, such that $\tau'(r) = \tau(r)$, the labels on the two fragments match, and for the update $U$ associated to the step $\tau'(r' - 1) \rightarrow \tau'(r')$:

1. if $cp_2 \neq \exists x, U(f)(p) = v$ for all $p$
2. if $cp_2 = \exists x, U(f)(ldr(\tau'(r')))) = v$

Proof. Set the HO set of each process in the step $\tau'(r) \xrightarrow{\Lambda} \tau'(r' + 1)$ to some $W$ of the appropriate size such that $\tau'(r), W \models C(v)$ (always possible due to (R4) and (L3)) and set it to $\{ldr(\tau'(r))\}$ if $cp_1$, and then apply Lemma B.18 if $r' > r + 1$. \qed

**Lemma B.20** (Propagating $\perp$’s from the coordinator). Assume $r$ and $r'$ belong to the same phase $\phi$, $r' > r$. Let $sf$ be the send-field of the step $r \rightarrow r + 1$, and assume the cp of the step is $\perp$. Furthermore, assume that $f$ is updated in the step $r' - 1 \rightarrow r'$, and assume that there is a path from $sf$ to $f$ in the phase graph of the phase of $\phi$. Lastly, assume $\text{rand}(s) = r$, and $s(sf)(ldr(s)) = \perp$.

Then, given any fragment $\tau(r) \leadsto \tau(r')$, there exists a fragment $\tau'(r) \leadsto \tau'(r')$, such that $\tau'(r) = s$, the labels on the two fragments match, and for the update $U$ associated to the step $\tau'(r' - 1) \rightarrow \tau'(r')$, $U(f)(p) = \perp$ for all $p$.

Proof. Similar to the proof of Lemma B.18. \qed

**Theorem B.2** (Post-critical round constraint preservation). Given a trace $\tau$ that violates agreement, any value $v$ written to $\text{dec}$ in the critical round
r_d, any \( v' \neq v \), and any algorithm constraint \( PC \in C(A) \) such that \( \tau(r_n) \models PC(v') \), where \( r_n \) is the first round of the phase following the critical phase, there exist a binary trace \( \tau_b \) and a value \( b \) such that:

1. some process decides on \( b \) in \( \tau_b(r_d) \)
2. \( \tau_b(r_n) \models PC(1 - b) \).
3. labels on \( \tau_b|_{r_n+1} \) match the labels on \( \tau|_{r_n+1} \)

Proof. Let \( r_s \) be the first round of the critical phase. We apply Theorem B.1 to \( \tau|_{r_s+1} \) to obtain a prefix \( \tau_b|_{r_s+1} \) that matches the labels on \( \tau|_{r_s+1} \), and where:

\[
\tau_b(r_s)(\text{inp}) = B(\tau(r_s)(\text{inp})).
\]

If \( \text{inp}' \) is not in the phase graph of the critical phase, the result follows easily. If \( \text{max}_C(\tau(r_s), \text{inp}, v) \supseteq \text{max}_C(\tau_b(r_s), \text{inp}, 0) \), let \( b = 1 \), otherwise let \( b = 0 \); this guarantees \( \tau_b(r_s) \models PC(1 - b) \), and since \( \text{inp} \) is not updated in the phase, then also \( \tau_b(r_n) \models PC(1 - b) \). We obtain the result by invoking Corollary B.1 with an appropriate \( h \) such that \( h(v) = b \).

We thus assume \( \text{inp} \) is updated in some round \( r_i \) of the critical phase. Note that, to show item 2, it is necessary and sufficient to construct \( \tau_b \) such that \( \tau_b(r_i + 1) \models PC(1 - b) \), since \( \tau_b(r_n)(\text{inp}) = \tau_b(r_i + 1)(\text{inp}) \).

Furthermore, since \( \text{dec} \) is also in the list of updated fields in some step, by (R1) there exists a fork round \( r_f \), with \( r_f \leq r_i \) and \( r_f \leq r_d \), in which two fields are updated. Denote the field on the input update path by \( f_{\text{inp}} \), and the one on the decision update path by \( f_{\text{dec}} \) (possibly \( f_{\text{inp}} = \text{inp} \) and \( f_{\text{dec}} = \text{dec} \)). Let a tile \( (\tilde{f}) \) denote the local update \( \{ f_{\text{dec}} \mapsto x, f_{\text{inp}} \mapsto y \} \).

Denote the constraints associated with the updates to \( f_{\text{dec}} \) and \( f_{\text{inp}} \) by \( C_{\text{dec}} \) and \( C_{\text{inp}} \) respectively, and denote the send-field of the round \( r_f \) by \( sf \), and the corresponding global update by \( U_f \). Since \( v \) was decided on in this round, we conclude:

\[
\tau(r_f) \models C_{\text{dec}}(v) \quad \text{ (C_{\text{dec}}-v)}
\]

\[
\#[U_f(f_{\text{dec}})(v)] > 0. \quad \text{ (U_f-v)}
\]

If there exists a HO set such that a tile \( (\tilde{v}') \) is possible given \( \tau(r_f)(sf) \), the proof is easy. We choose a \( h \) such that \( h(v) \neq h(v') \), and then if \( r_f > r_s \) we use Corollary B.2 extend \( \tau_b \) with a fragment \( \tau_b(r_s) \bowtie \tau_b(r_f) \) such that \( \tau_b \models \text{max}_C(\tau(r_f), sf, v)(b) \land \text{max}_C(\tau(r_f), sf, v')(1 - b) \), for some \( b \in \{0, 1\} \). We then apply Lemma B.6 to yield a set \( W \) such that, from the state \( \tau_b(r_f) \), any process whose HO set is \( W \) can perform an update \( (b_i') \) for some \( b_1 \neq b_2 \). We let the HO set of all processes to be \( W \) in the step \( \tau_b(r_f) \rightarrow \tau_b(r_f + 1) \). It is easy to see (possibly by applying Lemma B.18 or Lemma B.19 if needed) that \( \tau_b \) can then be extended to \( r_n \) such that for all
\( p \in \Pi \) we have \( \tau_b(r_n)(\text{dec})(p) = b_1 \) and \( \tau_b(r_n)(\text{inp})(p) = b_2 \), and such that item 3 holds.

Hence, in the remainder we assume that such a tile is not possible from \( \tau(r_f)(s_f) \). Let \( \Lambda_f \) be the label of the step \( \tau(r_f) \xrightarrow{\Lambda_f} \tau(r_f + 1) \); respecting the label \( \Lambda_f \), we will choose a \( b \) and construct an update \( U_f^b \) such that:

\[
\#[U_f^b](f_{\text{dec}})(b) \geq \#[U_f](f_{\text{dec}})(v) \quad (f_{\text{dec}-b})
\]

\[
\sum_{x \neq \bot} \#[U_f^b](f_{\text{dec}})(x) \geq \sum_{x \neq \bot} \#[U_f](f_{\text{dec}})(x) \quad (f_{\text{dec}-\sum})
\]

This suffices to show item 1: either directly, if \( r_d = r_f \), or if \( r_d > r_f \) by noting that for all \( C \), \( \tau(r_f + 1) \models C(v) \) implies \( \tau_b(r_f + 1) \models C(b) \) (possibly by permuting \( U_f \) if \( f(C) \in \text{next-ldr}(r_f + 1) \); this will always be possible by (R7)) and using Lemma B.19. We thus reduce showing item 1 in the rest to showing \( (f_{\text{dec}-b}) \) and \( (f_{\text{dec}-\sum}) \).

Furthermore, there are several properties on which we will often rely in the proofs, so we give them names here. Note that we will have to prove the properties when we use them; we just name them here for the reference. They are, for any \( w \neq v \), and for any \( C \in \{\text{C}_{\text{inp}}, \text{C}_{\text{dec}}\} \):

\[
\tau(r_f) \models C(v) \implies \tau_b(r_f) \models C(b) \quad (r_f\text{-prop.1})
\]

\[
\tau(r_f) \models C(w) \implies \tau_b(r_f) \models C(1 - b) \quad (r_f\text{-prop.2})
\]

\[
\#[\tau_b(r_f)(s_f)](\bot) = \#[\tau_b(r_f)(s_f)](\bot) \quad (r_f\text{-prop.3})
\]

\[
\tau_b(r_f)(s_f)(\text{ldr}(\tau_b(r_f))) = h(\tau_b(r)(s_f)(\text{ldr}(\tau(r_f)))) \quad (r_f\text{-prop.4})
\]

If \( r_f = r_s \), we will always show the properties directly. If \( r_f > r_s \), we first extend \( \tau_b \) with a fragment \( \tau_b(r_s) \sim \tau_b(r_f) \) obtained from Corollary B.2 with \( h(v) = b \) and \( h(x) = 1 - b \) for \( x \neq v \). Condition 1 of Corollary B.2 is then satisfied by (B-s), and by \( (C_{\text{dec}}\text{-v}) \) and our choice of \( h \), so is condition 3. We will still have to show the condition 2 of the Lemma. The properties \( (r_f\text{-prop}) \) will then follow.

We proceed by a case distinction:

1. Some value \( w \notin \{v, \bot\} \) was written to \( \text{inp} \) in the critical phase. This implies:

\[
\tau(r_f) \models C_{\text{inp}}(w) \quad (C_{\text{inp}}\text{-w})
\]

\[
\#[U_f](f_{\text{inp}})(w) > 0 \quad (U_f\text{-w})
\]

\[
twovals(\#[\tau(r_f)(s_f)]) \quad (\text{twovals-sf})
\]

From \( (C_{\text{inp}}\text{-w}) \) and \( (C_{\text{dec}}\text{-v}) \), we conclude that the condition 2 of Corollary B.1 holds for any choice of \( b \). From \( (U_f\text{-w}) \) and \( (U_f\text{-v}) \) we also
conclude that the step \( r_f \rightarrow r_f + 1 \) cannot be a \( \prec \) step. Since we also assumed that a tile \( \binom{r}{w} \) does not appear in \( \mathcal{U}_f \), we conclude that two separate tiles, \( \binom{r}{y} \) and \( \binom{r}{w} \) appear in \( \mathcal{U}_f \), so the step is also not an \( \preceq \) step.

When \( r_i > r_f \), instead of showing item 2 directly, we will instead show that the \( b \) and \( \mathcal{U}_f^b \) we construct also satisfy:

\[
\#[\mathcal{U}_f^b(f_{\text{inp}})](1-b) \geq \#[\mathcal{U}_f(f_{\text{inp}})](w) \quad (f_{\text{imp}}-b') \\
\sum_{x \neq \perp} \#[\mathcal{U}_f^b(f_{\text{inp}})](x) \geq \sum_{x \neq \perp} \#[\mathcal{U}_f(f_{\text{inp}})](x) \quad (f_{\text{imp}}-\Sigma)
\]

This suffices in the case \( r_i > r_f \); for any constraint \( C \) with \( f(C) = f_{\text{inp}} \), \( \tau(r_f + 1) \models C(w) \) will imply \( \tau_b(r_f + 1) \models C(1-b) \) (possibly after permutation if \( f(C) \in \text{next-1drf}(r_f+1) \), by (R7)). Then, we can use Lemma B.19 to set \( \#[\tau_b(r_s)(\text{inp})](1-b) = k \) while ensuring item 3. This clearly also guarantees item 2. We will treat item 2 separately in case \( r_i = r_f \).

Perform a further case distinction:

(a) Assume \( \binom{r}{w} \) was not used in the step \( \tau(r_f) \rightarrow \tau(r_f + 1) \). Since we also assumed that \( \binom{r}{x} \) for \( x \notin \{\perp, v\} \) was not used in this step, from \( \mathcal{U}_f-v \) we have that \( \binom{r}{w} \) must have been used. Thus, we have:

\[
\tau(r_f) \models C_{\text{inp}}(v) \\
\tau_b(r_f) \models C_{\text{inp}}(0) \wedge C_{\text{inp}}(1). 
\]

We choose \( b = 1 \) (and hence \( 1-b = 0 \)). If \( r_s = r_f \), \( (r_f, \text{prop}) \) are easily seen to hold from the previous and (B-s). If \( r_s > r_f \), it is easy to check that the condition 2 of Corollary B.2 is satisfied. We conclude

\[
\tau_b(r_f) \models C_{\text{inp}}(0) \wedge C_{\text{inp}}(1). 
\]

Let \( c \) be the count of \( \binom{r}{w} \) tiles in \( \mathcal{U}_f \). We construct \( \mathcal{U}_f^c \) as follows:

- take \( c \) tiles \( \binom{r}{b} \)
- if \( \tau_b(r_f) \models C_{\text{dec}}(1-b) \) and \( \binom{r}{x} \) with \( x \neq v \) appears in \( \mathcal{U}_f \), take \( k-c \) tiles \( \binom{r}{1-b} \); otherwise, take \( k-c \) tiles \( \binom{r}{1-b} \).

It is straightforward to check that this choice of \( \mathcal{U}_f^c \) satisfies \( (f_{\text{dec}}-b) \), \( (f_{\text{dec}}-\Sigma) \) (thus guaranteeing item 1), \( (f_{\text{imp}}-b') \) and \( (f_{\text{imp}}-\Sigma) \). This then suffices to show item 2 when \( r_i > r_f \).

If \( r_f = r_i \), we directly show that this choice of \( \mathcal{U}_f^c \) guarantees item 2. If the type of \( \text{inp} \) is \( N \times V \), this is immediate from (R8), \( b = 1 \) and the definitions of \text{next} and constraint satisfaction. Otherwise, since \( \tau(r_i + 1) \models PC(v') \), from the definition of \( c \)
and Lemma B.1, we conclude that for any state $s$ such that $\#[s(\text{inp})] = \{c \times v, (k-c) \times v'\}$ we also have $s \models PC(v')$. Thus, since $\#[\tau_0(r_i + 1)(\text{inp})] = \{c \times b, (k-c) \times (1-b)\}$, from Lemma B.12 we conclude also $\tau_0(r_i + 1) \models PC(1-b)$, proving item 2.

The only thing that we have left to show is that $U_f^b$ is possible in a step $\tau(r_f) \xrightarrow{\Lambda_f} \tau(r_f + 1)$. This follows from (simsat-$r_f$), ($C_{\text{dec-v}}$), ($r_f$-prop) and Lemmas B.8, B.7 and B.10, and thus item 3 is also guaranteed.

(b) Otherwise, since we assumed that no $\left((v)\right)$ tile was possible (or used), it must be that $\left((v)\right)$ was used in the step $\tau(r_f) \xrightarrow{\Lambda_f} \tau(r_f + 1)$. This implies $C_{\text{inp}} \supset C_{\text{dec}}$. Moreover, since both $\left((v)\right)$ and $\left((x)\right)$ (for some $x$) appear in $U_f$, we conclude $\phi_{uf} \notin \Lambda_f$.

We choose $b = 0$ (and thus $1-b = 1$). It is easy to check that the properties ($r_f$-prop) hold, directly when $r_f = r_s$, and by showing Condition 3 if $r_f > r_s$.

Let $c_1$ be the count of the tile $\left((v)\right)$ in $U_f$, and $c_2$ be the count of the tile $\left((x)\right)$ in $U_f$, and let $c = c_1 + c_2$.

If $r_i > r_f$, we construct $U_f^b$ by taking:

- $c_1$ tiles $\left((b)\right)$
- $c_2$ tiles $\left((b)\right)$
- $k - c$ tiles $\left((1-b)\right)$

This immediately guarantees ($f_{\text{dec-b}}$) and ($f_{\text{dec-\sum}}$). It also guarantees ($f_{\text{inp-b'}}$) and ($f_{\text{inp-\sum}}$), since we assume the tile $\left((v)\right)$ is not possible, and thus item 2 is guaranteed if $r_i > r_f$.

If $r_i = r_f$, we construct $U_f^b$ by taking:

- $c$ tiles $\left((b)\right)$
- $k - c$ tiles $\left((1-b)\right)$

This again immediately guarantees ($f_{\text{dec-b}}$) and ($f_{\text{dec-\sum}}$) (and thus item 1). We show that it also guarantees item 2, noting that $k - c > 0$. If the type of $\text{inp}$ is $\mathbb{N} \times \mathbb{V}$, this is immediate by (R8) and the semantics of $\text{next}$ and constraint satisfaction. Otherwise, we perform a case distinction on $\text{pred}(PC)$. If it is any, the result is immediate since $\#[U_f^b(\text{inp})](1-b) > 0$. If it is a selection, notice that for some value $w'$ we already had $\tau(r_s) \models PC(w')$, and hence $\tau_0(r_s) \models PC(1-b)$, by ($B$-s) and our choice of $b$. Thus also $\tau_0(r_f) \models PC(1-b)$. The claim then follows from Lemma B.1 and our construction of $U_f^b$. If it is $\text{all*}$, we note that:

$$\#[\tau(r_i + 1)(\text{inp})](v') \leq \min(k - c, \#[\tau(r_i)(\text{inp})](v') + \#[U_f(\text{inp})](v'))$$

145
and note that we can permute $\mathcal{U}_{r_i}^{d}$ (using Lemma A.1) so that

$$
\#[r_0(r_i+1)](\inp)(1-b) \leq \min(k - c, \\
\#[r_0(r_i)](\inp)(1-b) + \#[\mathcal{U}_{r_i}^{d}(\inp)](1-b)).
$$

The claim then follows from:

$$
\#[r_0(r_i)](\inp)(v') \leq \#[r_0(r_i)](\inp)(1-b) \quad \text{(by (B-s))}
$$

$$
\#[\mathcal{U}_{r_i}^{d}(\inp)](v') \leq \#[\mathcal{U}_{r_i}^{d}(\inp)](1-b) \quad \text{(construction of $\mathcal{U}_{r_i}^{d}$, $v \neq v'$)}
$$

Lastly, both when $r_i = r_f$ and when $r_i > r_f$, we need to show that the appropriate $\mathcal{U}_{r_i}^{d}$ is possible in a step $r_0(r_f) \xrightarrow{\Lambda_f} r_0(r_f + 1)$.

This follows from our choice of $b$, the fact that $\phi_{uf} \notin \Lambda_f$, properties (rf-prop.1), (rf-prop.2), (rf-prop.3) and Lemmas B.8 and B.9. Thus, item 3 holds as well.

2. No $w \notin \{\bot, v\}$ was written to $\inp$ in the critical phase. Letting $(C_1^a, C_2^a) = \max_{\sim}(\tau(r_s), \inp)$, we claim $PC \sqsubseteq C_1^a$. In case that $\max_{\sim}(\tau(r_s), \inp, v) = C_1$, this follows from Lemma B.14. Otherwise, if $\text{pred}(PC) \in \{\text{any}, \text{all}^a\}$, this follows since no $w \notin \{\bot, v\}$ was written to $\inp$; and if $PC$ is a selection, we note that some value $v'$ must have already satisfied $PC$ in $\tau(r_s)$, and thus $C_1^a \sqsubseteq PC$.

Furthermore, let $(C_1^{s,b}, C_2^{s,b}) = \max_{\sim}(\tau_0(r_s), \inp)$. We note that $(C_1^{s,b}, C_2^{s,b}) \sqsubseteq (C_1^a, C_2^a)$, by (B-s). Let $C_s$ be $C_{\text{dec}}$ if $r_s = r_f$ and the (unique, by (PS2)) constraint associated with the step $r_s \mapsto (r_s + 1)$ otherwise.

Lastly, let $\mathcal{U}_i$ be the global update that is associated with the step $\tau(r_i) \xrightarrow{\Lambda_i} \tau_0(r_i + 1)$. Since we assume only $\bot$ and $v$ get written, from $\tau(r_i + 1) = PC(v')$ we conclude that $\#[\mathcal{U}_i(\inp)](\bot) > 0$.

Proceed by a further case distinction.

- $(w)$, for some $w \neq \bot$ appears in $\mathcal{U}_{r_i}$. This implies $st \neq c_{w};$ where $st$ is the $cp$ of the step.
  - If $C_s \sqsubseteq C_2^{s,b}$, we let $b = 0$, otherwise, $C_s \sqsupset C_2^{s,b}$ and we let $b = 1$.
  - We claim the conditions of Lemma B.7 are met.
    - For $b = 0$, this follows from (B-s) if $r_f = r_s$. When $r_f > r_s$, we note that condition 2 of Corollary B.2 holds by (B-s), and by previous discussion, also (rf-prop) hold; this suffices for Lemma B.7.
    - For $b = 1$, the reasoning is identical, except for the case where $\text{pred}(C_{\text{dec}}) = \text{min}$. But this case not possible since then $r_s = r_f$ (by (R3)) and then $C_s = C_{\text{dec}}$, contradicting $C_s \sqsubseteq C_2^{s,b}$ and (B-s).
Lemma B.7 then yields a set $W$ such that setting the HO set of all processes to $W$ ensures that either $ldr(\tau_b(r_f))$ (if $st = \text{ldr}$) or all processes (if $st \neq \text{ldr}$) can perform the update $\left(\frac{1}{w}\right)$. This clearly preserves all the labels in $\Lambda_f$. It is straightforward to check that both possibilities ensure $(\sum \text{dec}-b)$ and $(\sum \text{dec})$. As discussed previously, this suffices for item 1.

The construction ensures that $\tau_b(r_i)(\text{inp}) = \tau_b(r_{\text{st}})(\text{inp})$, when $r_i = r_f$; but also when $r_i > r_f$, by Lemmas B.18 and B.20. This also ensures item 3.

What is left is to show item 2. If $b = 0$, this follows since $PC \subseteq C_1^b$. If $b = 1$, then $C_s \sqsupseteq C_2^{s,b} \sqsubseteq C_2^{s,b}$ and by Lemma B.14, $PC \subseteq C_2^{s,b}$; thus, item 2 follows again.

• no $\left(\frac{w}{x}\right)$, for any $w \neq \bot$ appears in $\mathcal{U}_f$, and $PC \subseteq C_2^{s,b}$. By $(\mathcal{U}_f{-}\text{v})$, and since we assumed that no tiles $\left(\frac{w}{x}\right)$ for $\bot \neq x \neq y \neq \bot$ appeared in $\mathcal{U}_f$, a tile $\left(\frac{w}{x}\right)$ then must appear in $\mathcal{U}_f$. Furthermore, all tiles that appear in $\mathcal{U}_f$ must be of one of the forms: $\left(\frac{w}{w}\right)$, $\left(\frac{1}{w}\right)$, or $\left(\frac{1}{1}\right)$, for some $w \neq \bot$.

Let $b = 1$. Given $\mathcal{U}_f$, define $\mathcal{U}_f'$ to be the update that results from replacing all $\left(\frac{w}{w}\right)$ tiles from $\mathcal{U}_f$ by $\left(\frac{w}{x}\right)$. Letting $h(v) = b$ and $h(x) = 1 - b$, we then create create an update $\mathcal{U}_f'$ such that $\mathcal{U}_f'(p)(f) = h(p)(\mathcal{U}_f'(p)(f))$ for $f \in \{\text{inp}, \text{dec}\}$. We show that the update $\mathcal{U}_f'$ is possible while preserving $\Lambda_f$, using Lemmas B.8 (for $\left(\frac{w}{w}\right)$ and, if necessary, $\left(\frac{w}{1}\right)$ B.9 (for $\left(\frac{1}{w}\right)$ if necessary) and B.7 (for $\left(\frac{1}{1}\right)$ if necessary), and using Lemma B.10 if $\phi_{\text{af}} \in \Lambda_f$. When $r_f = r_{s}$, it is simple to directly check that the conditions of the Lemmas are satisfied, using (B-s) and our choice of $b$. When $r_f > r_s$, as before, they follow from $(\text{r}_f{-}\text{prop})$; our choice of $b$ also clearly satisfies the condition 2 of Corollary B.2, as required.

We note that our choice of $\mathcal{U}_f'$ ensures $(\text{dec}-b)$ and $(\sum \text{dec})$, and thus, as discussed previously, item 1. We show that items 2 and 3 also hold.

If $r_i = r_f$, by assumption, $\#(\mathcal{U}_f(\text{inp}))(w) = 0$, for $w \notin \{v, \bot\}$. Let $c_1 = \#(\mathcal{U}_f(\text{inp}))(v) + \#(\mathcal{U}_f(\text{inp}))(b)$. If inp is of type $\mathbb{N} \times \mathcal{V}$, by (R8) we conclude that $\text{pred}(PC) = \text{maxts}$, and then by the semantics of $\text{next}$ and constraint satisfaction, that $k - c_1 \geq \alpha_b(\text{th}(PC))$. From this and (B-s) we conclude that we can then use Lemma A.1 to ensure item 2. Next, consider the case where inp is of type $\mathcal{V}$. Let $c_2 = \#(\tau_b(r_i)(\text{inp}))(1-b)$. If $k-c_1 \geq c_2$, we can clearly permute $\mathcal{U}_f'$ to ensure $\#(\tau_b(r_i+1)(\text{inp}))(1-b) = c_2$, and thus since $\tau_b(r_i) \models PC(1-b)$ (by $PC \subseteq C_2^{s,b}$ and (B-s)), also $\tau_b(r_i+1) \models PC(1-b)$ and item 2 holds. If $k-c_1 < c_2$,
we can permute $U^f_b$ to ensure $\#[\tau_b(r_i + 1)(\inp)](1 - b) = k - c_1$. Since $\tau(r_i + 1) \models PC(v')$, by Lemma B.1 we conclude that also $\{c_1 \times v, (k - c_1) \times v'\} \models PC(v')$, and thus by Lemma B.12 also $\tau_b(r_i + 1) \models PC(1 - b)$ as needed.

If $r_i > r_f$, from $U^f_b(f_{\inp}) = h_{\perp} \circ U_f(f_{\inp})$ we conclude that we can use Lemma B.15 to extend $\tau_b$ to $\tau_b(r_i + 1)$ such that $\#[U^b_{\inp}](\inp) = \#[h_{\perp} \circ U_b(\inp)]$; since $\#[U_b(\inp)](\perp) > 0$, we use Lemma A.1 to create an update $U^b_s$, such that $\#[U^b_{\inp}](\perp) = k$. Thus, $\tau_b(r_i + 1)(\inp) = \tau_b(r_s)(\inp)$, and hence $\tau_b(r_i + 1) \models PC(1 - b)$, since $PC \sqsubseteq C_2^{s,b}$.

• no $(\tau^t)$, for any $w \neq \perp$ appears in $U_f$, and $PC \sqsubseteq C_2^{s,b}$. The latter, implies that $\max_C(\tau(r_s), \inp, v) \sqsubseteq C_2$, and thus, if $\#[U_b(\inp)](v) = 0$, or together with Lemma B.14 if $\#[U_b(\inp)](v) > 0$. We choose $b = 0$.

If $r_f = r_s$, we define $U'_f$ to be the update that results from replacing all tiles from $U_f$ of the form $(\perp, v)$ for some $v \notin \{\perp, v\}$ by tiles $(w)$; if $r_f > r_s$, we define $U'_f = U_f$. As in the previous case, we then let $h(v) = b$ and $h(x) = 1 - b$ for $x \neq v$, and define $U_b$ such that $U_b(f)(f) = h_{\perp}(U_b(p)(f))$ for $f \in \{f_{\inp}, f_{\dec}\}$.

We first show that $U'_f$ is possible from $\tau_b(r_f)$ while preserving $\Lambda_f$. We show that using Lemmas B.8 (for $(\tau^t)$ and, if necessary, $(\tau^{t-b})$), B.9 (for $(\tau^t)$ if necessary), and B.7 (for $(\tau^{t-b})$ if necessary), and Lemma B.10 if $\phi i_a \in \Lambda_f$. If $r_f = r_s$, the conditions of the lemmas follow from (B-s) and $\max_C(\tau(r_s), \inp, v) \sqsubseteq C_2$. If $r_f > r_s$, they follow from (r-f-prop), which hold since the condition 2 of Corollary B.2 is satisfied, since $\max_C(\tau(r_s), \inp, v) \sqsubseteq C_2$.

The choice of $U'_f$ guarantees item 3, following the same reasoning as in the previous case. For items 2 and 3, if $r_i > r_f$, we perform the same proof as in the previous case. If $r_i = r_f$, we perform a case distinction on $\pred(\PC)$:

- any or min. Contradiction from (B-s), since then $C_2^{s} \sqsubseteq PC$.
- maxts. Contradiction again, from (R8), $PC \sqsubseteq C_2^{s,b}$ and (R8).
- all=. Since by (B-s) we have

$$\#[\tau_b(r_i)(\inp)](1 - b) \geq \#[\tau(r_i)(\inp)](v')$$

and since $\#[U_f(\inp)](b) = \#[U_f(\inp)](v)$, we conclude we can permute $U^f_b$ to ensure

$$\#[\tau_b(r_i + 1)(\inp)](1 - b) \geq \#[\tau(r_i + 1)(\inp)](v')$$

and thus $\tau_b(r_i + 1) \models PC(1 - b)$. 148
– smor. Let \( c = \#[\mathcal{U}_f(\text{inp})](v) \). Since \( \tau(r_i + 1) \models PC(v') \), we also get that \( \{c \times v, (k - c) \times v'\} \models PC(v') \) by Lemma B.1. We also have \( \#[\mathcal{U}_f^1(\text{inp})](b) = c \) by the construction of \( \mathcal{U}_f^1 \).

If \( k - c \geq \#[\tau_0(r_i)(\text{inp})](1 - b) \), we use Lemma A.1 to permute \( \mathcal{U}_f^1 \) to ensure \( \tau_0(r_i + 1)(\text{inp}) = \tau_0(r_i)(\text{inp}) \) and thus since, by (B-s), \( \tau_0(r_i)(\text{inp}) \models PC(1 - b) \), also \( \tau_0(r_i + 1)(\text{inp}) \models PC(1 - b) \).

If \( k - c < \#[\tau_0(r_i)(\text{inp})](1 - b) \), we permute permute \( \mathcal{U}_f^1 \) so that \( \#[\tau_0(r_i)(\text{inp})](1 - b) = k - c \); Lemma B.13 and the previous fact that \( \{c \times v, (k - c) \times v'\} \models PC(v') \) then ensure that \( \tau_0(r_i + 1) \models PC(1 - b) \).

\[ \Box \]

### B.2.5 Putting it all together

**Theorem B.3** (0-1 principle for agreement). Assume that a trace \( \tau \) violates the agreement property. Then, there exists a binary trace \( \tau_b \) that also violates agreement, such that the labels on \( \tau_b \) match the labels on \( \tau \).

**Proof.** Let \( r_s \) be the first round of the critical phase of \( \tau \), and \( r_d \) the critical round. Perform a case distinction:

- Two different values \( v_1 \) and \( v_2 \) were decided on in the critical phase. Invoke Theorem B.1 to yield a binary trace \( \tau_b \) of length \( r_s + 1 \), with \( \tau_b(r_s) = \mathcal{B}(\tau(r_s)) \). Let \( f_{\text{dec}} \) be the (unique) field updated in the step \( \tau_b(r_s) \) such that it lies on the path between \( \text{inp} \) and \( \text{dec} \) on the phase graph. The result follows from Corollary B.1, invoked with \( h(v_1) = 1 \) and \( h(x) = 0 \) for \( x \neq v_1 \).

- Only one value \( v_1 \) was decided on in the critical phase. A different value \( v_2 \) was decided in \( r_{d2} \). Let \( r_n \), where \( r_n \leq r_{d2} \), be the first round of the phase following the critical phase. Let \( v' \) be the value such that \( v' \neq v \) and that \( \max_C(\tau(r_n), \text{inp}, v') \supseteq \max_C(\tau(r_n), \text{inp}, v'') \) for all \( v'' \neq v_1 \); let \( PC = \max_C(\tau(r_n), \text{inp}, v') \). Invoke Theorem B.2 with \( \tau, PC, v_1 \) and \( v' \) to yield \( b \) and \( 1 - b \) and the trace \( \tau_b \) of length \( r_n + 1 \). Let \( PC_b = \max_C(\tau(r_n), \text{inp}, 1 - b) \); we have \( PC \subseteq PC_b \).

Let \( r_v \) be the first round such that \( r_v \geq r_n \), the send-field of the step \( \tau(r_v) \rightarrow \tau(r_v + 1) \) is \( \text{inp} \), and that for some field \( f \) in the list of updated fields and the associated constraint \( C_f \) we have \( C_f \subseteq PC_b \). Such a round exists since \( v_2 \) was decided on in \( r_{d2} \geq r_n \); necessarily, \( r_v \leq r_{d2} \). We also conclude that \( r_v \) must be the first round of a phase, since \( \text{inp} \) was sent in it.

We first prove that we can extend \( \tau_b \) to \( \tau_b \) having \( \max_C(\tau_b(r_v), \text{inp}, 1 - b) = PC_b \), such that the labels match the labels on \( \tau \), and such that
for any value \( v'' \neq v_1 \), \( \max_C(\tau(r_v), inp, 1 - b) \subseteq PC_b \). We prove this for each phase \( \phi \) between \( r_n \) and \( r_v \) in which \( inp \) is updated. If \( inp \) is not updated in \( \phi \), the result follows immediately from Lemma B.16. If \( inp \) is updated in \( \phi \) in round \( r_i \), by our choice of \( r_v \) and \( PC \) we conclude that the only values written to \( inp \) must be either \( v_1 \) or \( \perp \); and we conclude \( v_1 \max_C(\tau(r_i + 1), inp, w) \subseteq PC_b \) for all \( w \neq v_1 \) (by Lemma B.14). Moreover, at least one \( \perp \) must be written every time, for otherwise all processes would hold \( v_1 \) in their \( inp \) field and \( v_2 \) could not be decided on in \( r_{d2} \). We (possibly repeatedly) apply Lemma B.2 for the fields on the input update path to preserve the possibility of choosing \( \perp \). Note that the condition 2 is only relevant on \( inp \) (since, by our assumption, only \( v_1 \) can be chosen for an update in the phase), and the condition holds since we assume \( \tau_b(r_i) \models PC_b(1 - b) \), which implies \( \#[\tau_b(r_i)(inp)](1 - b) > 0 \); we also have \( \#[\tau_b(r_i)(inp)](1 - b) < k \) since otherwise \( PC_b \) would be the strongest constraint of the algorithm. We then apply Lemma A.1 to ensure that all the processes apply the \{\( inp \rightarrow \perp \)\} update in the step \( \tau_b(r_i) \rightarrow \tau_b(r_i + 1) \). Since by the inductive hypothesis \( \tau_b(r_i) \models PC_b(1 - b) \), we also have \( \tau_b(r_i + 1) \models PC_b(1 - b) \).

We thus have \( \tau_b(r_v) \models PC_b(1 - b) \). If there exists a field \( f_{dec} \) updated on the path from \( inp \) to \( dec \) in the step \( \tau(r_v) \rightarrow \tau(r_v + 1) \), such that for the associated constraint \( C_{dec} \) we have \( C_{dec} \subseteq PC_b \); let \( f = f_{dec} \) and \( C = C_{dec} \). Otherwise, there must exist another field \( f_{inp} \) updated in the step, with the associated constraint \( C_{inp} \subseteq PC_b \), and \( f_{inp} \) must be on the path from \( inp \) to \( inp \) in the phase graph of \( r_v \); let \( f = f_{inp} \) and \( C = C_{inp} \). Apply Lemma B.3 to yield a set \( W \) that enables the update \{\( f \rightarrow 1 - b \)\}, and have all the processes apply the update. This completes our result if \( f = dec \); if \( f \neq dec \), we obtain the result by (possibly repeatedly) applying Lemma B.18.

\[ \square \]

### B.3 Termination

**Definition B.17** (\( C \)). Let \( C \) be the maximum constraint of the algorithm such that there exists a binary state \( s_b \) with \( s_b \models C(0) \wedge C(1) \).

With \( C \) defined this way, we define \( C_f \) as the constraint:

- \((f, th(C), pred(C))\), if \( pred(C) = \text{maxts} \) and type of \( f \) is \( \mathbb{N} \times V \), or if \( pred(C) \neq \text{maxts} \) and type of \( f \) is \( V \)
- \((f, th(C), \text{min})\), if \( pred(C) = \text{maxts} \) and type of \( f \) is \( V \)
It is simple to check that for any $C_f$, there exists a binary state $s_b$ with $s_b \models C(0) \land C(1)$. As the field $f$ will usually be clear from the context, we will abuse the notation and write simply $C$.

Next, we define a simulation relation used to prove the 0-1 principle for termination. For a multiset $M$ over $\mathcal{V}$ or $\mathbb{N} \times \mathcal{V}$, define $\text{twovals}'(M) = \text{twovals}(M')$ where $M'$ is the multiset created by projecting the values of $M$ onto $\mathcal{V}$. Next, for two multisets $M_1$ and $M_2$, define $(M_1, M_2) \in \rho_{\text{step}}$ if one of the following holds:

- $M_2(\bot) = k$ or
- $M_1(\bot) = 0$, and if $\text{twovals}'(M_1)$ holds, then also $\text{twovals}'(M_2)$ holds, and $M_2 \models C(0) \land C(1)$.

Next, overloading $\rho_{\text{step}}$, define $(s, s_b) \in \rho_{\text{step}}$ if $\text{rnd}(s) = \text{rnd}(s_b)$, and for all fields $sf$ that are going to be sent in some round $r' \geq \text{rnd}(s)$, and will not be updated between $\text{rnd}(s)$ and $r'$, we have:

- if $sf \notin \text{next-ldrf}(r)$, then $(\#[\tau(r)(sf)], \#[\tau_b(r)(sf)]) \in \rho_{\text{step}}$
- if $sf \in \text{next-ldrf}(r)$, and if $\tau(r)(sf)(\text{ldr}(\tau(r))) = \bot$ holds, then also $\tau_b(r)(sf)(\text{ldr}(\tau_b(r))) = \bot$.

Lastly, we define:

$$(s, s_b) \in \rho_{\text{term}} \triangleq (s, s_b) \in \rho_{\text{step}} \\
\land \#[\text{dec}](\bot) > 0 \implies \#[s_b(\text{dec})](\bot) = k \\
\land (\#[s(\text{inp})], \#[s_b(\text{inp})]) \in \rho_{\text{step}}$$

**Theorem B.4** (Simulating non-termination). $\rho_{\text{term}}$ is a forward simulation between $A$ run on an arbitrary value domain, and $\Lambda A$ run on the binary value domain.

**Proof.** Recalling the definition of the set of initial states for ConsL algorithms, the base case is easy. Next, given a concrete step $\tau(r) \xrightarrow{\Lambda} \tau(r + 1)$ with the cp cp, the send-field $sf$, the list of updated fields $U$, and the global update $\mathcal{U}$, and with $\tau(r), \tau_b(r) \in \rho_{\text{step}}$, we show that we can create an abstract step $\tau_b(r) \xrightarrow{\Lambda} \tau_b(r + 1)$ such that for the associated update $\mathcal{U}_b$, for every $f \in U$ we have:

- if $f \notin \text{next-ldrf}(r + 1)$, $(\#[\mathcal{U}(f)], \#[\mathcal{U}_b(f)]) \in \rho_{\text{step}}$, and if $f \neq \text{dec}$, also $(\#[\tau(r + 1)(f)], \#[\tau_b(r + 1)(f)]) \in \rho_{\text{step}}$
- if $f \in \text{next-ldrf}(r + 1), \mathcal{U}(f)(\text{ldr}(\tau(r))) = \bot$ implies $\mathcal{U}(f)(\text{ldr}(\tau_b(r))) = \bot$
We claim that this suffices to show that \((\tau(r+1), \tau_0(r+1)) \in \rho_{\text{term}}\). The corresponding proof obligation for ephemeral fields from \(U\) follows immediately from the semantics of \(\text{next}\). If \(\text{inp} \in U\), this result still follows since:

1. the type of \(\text{inp}\) does not allow \(\bot\) values;

2. if \(\text{twovals}'(\#[\tau(r+1)(\text{inp})])\) holds, then either:
   
   (a) \(\text{twovals}'(\#[\tau(r)(\text{inp})])\) and \(\#[U(\text{inp})](\bot) > 0\). Then we have \(\#[U_b(\text{inp})](\bot) = k\) and \(\tau_0(r) \models C(0) \land C(1)\) (from the simulation assumption) implying \(\tau_0(r+1) \models C(0) \land C(1)\).

   (b) \(\text{twovals}'(\#[U(\text{inp})])\) holds. Then, \(\#[U_b(\text{inp})] \models C(0) \land C(1)\) holds, and then also \(\tau_0(r+1) \models C(0) \land C(1)\).

Lastly, if \(\text{dec} \in U\) and \(\#[\tau(r+1)(\text{dec})](\bot) > 0\), then \(\#[\tau(r)(\text{dec})](\bot) > 0\) and \(\#[U(\text{dec})](\bot) > 0\). But then \(\#[\tau_0(r)(\text{dec})](\bot) = k\) and \(\#[U_b(\text{dec})](\bot) = k\), implying \(\#[\tau_0(r+1)(\text{dec})](\bot) = k\).

If \(sf \notin \text{next-ldrf}(r)\) and \(\#[\tau(r)(sf)](\bot) > 0\), then by the simulation assumption, \(\#[\tau_0(r)(sf)](\bot) = k\); we invoke Lemma B.18, and we are done. Similarly, if \(sf \in \text{next-ldrf}(r)\) and \(\tau_0(r)(sf)(ldr(\tau_0(r))) = \bot\), we invoke Lemma B.20 and we are done. So assume \(\#[\tau(r)(sf)](\bot) = 0\) and \(\tau_0(r)(sf)(ldr(\tau_0(r))) \neq \bot\) respectively. We proceed by a case distinction on the number of fields updated in the step:

1. One, denoted by \(f\). If \(cp \not\succcurlyeq 2\) and \(\#[U(\tau(r))](\bot) > 0\), or if \(cp = \succcurlyeq 3\) and \(U(\tau(r))(ldr(\tau(r))) = \bot\), invoke Lemma B.2 to yield a set \(W\), set the HO set of all the processes to \(W\) and we are done. Otherwise, if \(\text{twovals}'(\#[U(\tau(r))])\) holds, then \(cp \not\succcurlyeq 3\), and \(\text{twovals}'(\#[\tau(r)(sf)])\) also holds, and moreover for the constraint \(C\) associated to the update of \(f\) we have \(C \subseteq C\). Take the two witnesses \(v_1\) and \(v_2\) for \(\text{twovals}'(\#[U(\tau(r))])\) let \(h(v_i) = 0\) and \(h(x) = 1\), for all \(x \neq v_i\), choosing \(i\) from \(\{1, 2\}\) such that \(\tau_0(r)(sf)(ldr(\tau_0(r))) = h(\tau(r)(sf)(ldr(\tau(r))))\), and invoke Lemma B.4. This suffices for the simulation step.

2. Two, denoted by \(f_1\) and \(f_2\), with the associated constraints \(C_1\) and \(C_2\); WLOG (by (R2)) assume \(C_1 \subseteq C_2\). Then, if \(U(f_1)(p) = \bot\) for any \(p\) if \(cp \not\succcurlyeq 3\), and for \(p = ldr(\tau(r))\) otherwise, then also \(U(f_2)(p) = \bot\), and we invoke Lemma B.9 to yield a set \(W\), and then set the HO set of all the processes to \(W\) to yield a \(U_b\) such that \(\#[U_b(f_i)](\bot) = k\), for \(i \in \{1, 2\}\), and we are done. Thus, assume \(U(f_1)(p) \neq \bot\) for all \(p\) if \(cp \not\succcurlyeq 3\), and for \(p = ldr(\tau(r))\) otherwise. Perform a further case distinction following the structure of the \(\rho_{\text{term}}\) relation:

   (a) \(\#[U(f_2)](\bot) > 0\) and \(\text{twovals}'(\#[U(f_1)]\)). From the latter, we conclude \(C_1 \subseteq C\) and \(cp = \succcurlyeq 3\). Invoke Lemma B.7 two times, with \(b = 0\) and then \(b = 1\), to yield HO sets \(W_0\) and \(W_1\) (using
Lemma B.10 to ensure \( W_0 = W_1 \) if \( \phi_{uf} \in \Lambda \) that yield \( (\bot) \) and \( (\bot) \) tiles; we construct the update \( \mathcal{U}_b \) using these two HO sets.

(b) one of the two holds: either (1) \( f_2 \notin \text{next-ldrf}(r + 1) \) and \( \#[\mathcal{U}(f_2)](\bot) > 0 \), but not \( \text{twovals}'(\#[\mathcal{U}(f_1)]) \), or (2) we have both \( f_2 \in \text{next-ldrf}(r + 1) \) and \( \mathcal{U}(f_2)(\text{ldr}(\tau(r))) = \bot \). Invoke Lemma B.2 with \( f = f_2 \) to yield the HO set \( W \) that results in an update setting \( \{ f_2 \mapsto \bot \} \), and use it as the HO set for all the processes.

(c) \( \#[\mathcal{U}(f_2)](\bot) = 0 \) and \( \text{twovals}'(\#[\mathcal{U}(f_2)]) \). Then, we again conclude \( \text{cp} = 
\circ \n\circ \). Then, \( C_2 \subseteq C \). Given the two witnesses \( v_1 \) and \( v_2 \) for \( \text{twovals}'(\mathcal{U}(f_2)) \), set \( h(v_1) = 0 \) and \( h(x) = 1 \), for \( x \neq v_i \). Apply Lemma B.4 with \( f = f_2 \), to yield an update \( \mathcal{U}'_b \); note that since \( C_1 \subseteq C_2 \), whenever \( \mathcal{U}'_b(f_1) = b \) it is possible to set \( \mathcal{U}'_b(f_1) = b \) without changing the HO sets, for any \( b \in \{0, 1\} \). Use Lemma A.1 to create \( \mathcal{U}_b \) from \( \mathcal{U}'_b \) such that \( \#[\mathcal{U}'_b(f_i)] \models C(0) \land C(1) \) for \( i \in \{i, 2\} \).

(d) \( \#[\mathcal{U}(f_2)](\bot) = 0 \) and not \( \text{twovals}(\#[\mathcal{U}(f_2)]) \). If not \( \text{twovals}(\mathcal{U}(f_1)) \), there is nothing to prove. Thus assume \( \text{twovals}(\mathcal{U}(f_1)) \), and obtain some two witnesses \( v_1 \) and \( v_2 \) for that fact. Again, we conclude \( \text{cp} = 
\circ \n\circ \). Then, \( C_1 \subseteq C \), and use Lemma B.4 with \( h(v_i) = 0 \), \( h(x) = 1 \) for \( x \neq v_i \), to obtain an update \( \mathcal{U}'_b \); use Lemma A.1 to create \( \mathcal{U}_b \) from \( \mathcal{U}'_b \) such that \( \#[\mathcal{U}'_b(f_1)] \models C(0) \land C(1) \).
Appendix C

Proof of the Cutoff Theorem

The basic idea is to use a threshold based abstraction. Intuitively, the abstract state is a scaled down version of the concrete one, where each notch on the scale corresponds to a threshold. We will then first show that this scaling relation enables each process to take the same local step in the abstract as in the concrete system. Then, we will show that these local steps can be combined to yield a global abstract update that is again a scaled down version of the global concrete update. Lastly, we show that this scaled down global update can be applied to the scaled down state, to again yield a scaled down version of the new concrete state.

There are two difficult parts to the proof:

1. Show that we can create similar global updates when two fields are simultaneously updated in the same round.

2. Show that the global scaled down update can be applied to the persistent field and yield a scaled down version of the state of the concrete persistent field.

For the first part, we currently perform a long combinatorial proof with a lot of case distinctions (Lemma C.18). For the second part, we rely on simulation. We use two similar but different simulations for agreement and termination. The agreement simulation is a forward simulation; to show that it guarantees preservation of agreement, we use a strengthened simulation result (Lemma C.24) with a somewhat involved proof. The termination simulation is a relatively complicated backward-forward simulation; however, the preservation of termination then follows directly.

For both simulations we might have to further adjust the global update yielded by the first part, to preserve the simulation. In particular, this is needed when the inp field is updated simultaneously with another field.

For the remainder of this section, we assume $\mathcal{V} = \{0, 1\}$. 

155
C. Proof of the Cutoff Theorem

C.1 Calculating the Cutoff Bound

As described in Section A, we assume that we have fixed a ConsL algorithm \( \mathcal{A} \); all the definitions from that section carry over.

We define the function \( \gamma_n : \{0, \ldots, n\} \to T \) such that \( \gamma_n(x) \) denotes the largest threshold \( t \in T \) such that \( x \geq \alpha_n(t) \).

Note that, since \( \lfloor t_1 n \rfloor + \lfloor t_2 n \rfloor \leq \lfloor (t_1 + t_2) n \rfloor \leq \lfloor t_1 n \rfloor + \lfloor t_2 n \rfloor + 1 \):

\[
\alpha_n(t_1 + t_2) \leq \alpha_n(t_1) + \alpha_n(t_2) \quad (\alpha+1)
\]

\[
\alpha_n(t_1) + \alpha_n(t_2) \leq \alpha_n(t_1 + t_2) + 1 \quad \text{for } t_1, t_2 \geq 0 \quad (\alpha+2)
\]

and:

\[
\gamma_n(c) = \left\lfloor \frac{\sum_{i=1}^{3} t_i}{d} \right\rfloor - 1 \quad \text{for } 0 < c \leq n \quad (\gamma-eq)
\]

\[
\gamma_n(c_1) + \gamma_n(c_2) \leq \gamma_n(c_1 + c_2) \quad \text{for } c_1 + c_2 \leq n \quad (\gamma+1)
\]

\[
\gamma_n(c_1 + c_2) \leq \gamma_n(c_1) + \gamma_n(c_2) + \frac{1}{d} \quad \text{for } c_1 + c_2 \leq n \quad (\gamma+2)
\]

As our cutoff bound, we take \( B = 2d+1 \). This is the smallest \( n \) such that, for any three thresholds \( t_1, t_2, t_3 \in T \) with \( \sum_{i=1}^{3} t_i \leq \frac{d-1}{d} \) we have \( \sum_{i=1}^{3} \alpha_n(t_i) \leq n \). To see that this holds for \( B \), we have:

\[
\sum_{i=1}^{3} \alpha_B(t_i) \leq [B \cdot \sum_{i=1}^{3} t_i] + 3
\]

\[
\leq [B \cdot \frac{d-1}{d}] + 3
\]

\[
\leq B - \left\lfloor \frac{B}{d} \right\rfloor + 3
\]

\[
\leq B.
\]

For any smaller \( n \), taking e.g. \( t_1 = 0, t_2 = 0 \) and \( t_3 = \frac{d-1}{d} \) yields a counterexample.

The following properties hold, where \( n \) is arbitrary:

(CO1) \( (\alpha_n, \gamma_n) \) is a Galois connection,

(CO2) \( t \leq \gamma_n(\alpha_n(t)) \), and if \( n \geq d \) then \( \alpha_n(\cdot) \) is injective on \( T \), hence \( \gamma_n(\alpha_n(t)) = t \),

(CO3) \( \alpha_n(\gamma_n(c)) \leq c \) and for \( c < B, c \leq \alpha_B(\gamma_B(c)) + 1 \),

(CO4) for \( c_1, c_2 > 0 \) such that \( c_1 + c_2 = n, \frac{d-2}{d} \leq \gamma_n(c_1) + \gamma_n(c_2) \leq \frac{d-1}{d} \)

(CO5) if \( c_i \geq 0 \) for \( i \in I \) and \( \sum_{i \in I} c_i \leq n \), then \( \sum_{i \in I} \gamma_n(c_i) \leq \frac{d-1}{d} \)
C.2. Simulation Relations

In this section, we define the simulation relations for agreement and termination. We start from basic relations on multisets and then extend these to pairs of “big” and “small” states.

C.2.1 Basic relations on multisets

First, we define three relations on multisets $M_1$ and $M_2$ of values from $\mathcal{V} \cup \{\bot\}$, where we view multisets of elements of a set $D$ as functions $D \rightarrow \mathbb{N}$:

\[
(M_1, M_2) \in \text{cntMS}_{\geq}(V) \triangleq \forall v \in V. \gamma_{|M_1|}(M_1(v)) \geq \gamma_{|M_2|}(M_2(v))
\]

\[
(M_1, M_2) \in \text{cntMS}_{\sum\geq}(V) \triangleq \gamma_{|M_1|}(\sum_{v \in V} M_1(v)) \geq \gamma_{|M_2|}(\sum_{v \in V} M_2(v))
\]

\[
(M_1, M_2) \in \text{cntMS}_{=}(V) \triangleq \forall v \in V. \gamma_{|M_1|}(M_1(v)) = \gamma_{|M_2|}(M_2(v))
\]

Note that $(M_1, M_2) \in \text{cntMS}_{\geq}(\emptyset)$ and $(M_1, M_2) \in \text{cntMS}_{\sum\geq}(\emptyset)$ for all multisets $M_1$ and $M_2$ of the appropriate type.
C. Proof of the Cutoff Theorem

C.2.2 Basic state relations

We next define concepts that help us relate two states of different sizes. Throughout, we will assume that \( k \) is some number larger than \( B \). In particular, we will assume that \( k \) and \( d \) are coprime, that is:

\[
gcd(d, k) = 1 \quad \text{(dk-coprim)}
\]

Both of these assumptions are justified by Lemma A.4.

Next, we use these basic relations on multisets to define the following relations on \( \Sigma_k \times \Sigma_B \).

\[
\begin{align*}
(s_a, s_c) \in \text{cntrel}_1(F, V) & \triangleq \forall f \in F. (|s_a(f)|, |s_c(f)|) \in \text{cntMS}_{\geq}(V) \\
(s_a, s_c) \in \text{cntrel}_2(F, V) & \triangleq \forall f \in F. (|s_a(f)|, |s_c(f)|) \in \text{cntMS}_{\Sigma}(V) \\
(s_a, s_c) \in \rho_{\text{init}} & \triangleq s_c.rnd = 0 \implies (|s_a(\text{inp})|, |s_c(\text{inp})|) \in \text{cntMS}_{\Sigma}(\{0, 1\}) \\
(s_a, s_c) \in \text{decrel}_\bot & \triangleq |s_c(\text{dec})|((\bot)) > 0 \implies |s_a(\text{dec})|(\bot) > 0 \\
(s_a, s_c) \in \text{ldrrel}(F) & \triangleq \forall f \in F. s_c(f, ldr(s_c)) = s_a(f, ldr(s_a)) \\
(s_a, s_c) \in \text{ldrrel}_{\text{val}}(F) & \triangleq \forall f \in F. s_c(f, ldr(s_c)) \neq \bot \implies s_a(f, ldr(s_a)) = s_c(f, ldr(s_c)) \\
(s_a, s_c) \in \text{rndrel} & \triangleq \text{rnd}(s_c) = \text{rnd}(s_a) \\
(s_a, s_c) \in \text{tsrel}(t) & \triangleq \forall v \in \{0, 1\}, s_c \models (\text{inp}, t, \text{maxts})(v) \implies s_a \models (\text{inp}, t, \text{maxts})(v)
\end{align*}
\]

Next, we define:

\[
\rho_{\text{basic}}(F) \triangleq \text{rndrel} \cap \rho_{\text{init}} \cap \text{ldrrel}(F) \cap \text{cntrel}_1(\{\text{inp}\}, \{0, 1\})
\]

if \( A \) does not use timestamps. Otherwise, let \( C_{ts} \) be the set of all constraints \( C \) from \( C(A) \) where \( \text{pred}(C) = \text{maxts} \).

\[
\begin{align*}
\rho_{ts} & \triangleq \bigcap_{C \in C_{ts}} \text{tsrel}(\text{th}(C)) \\
\rho_{\text{basic}}(F) & \triangleq \text{rndrel} \cap \rho_{\text{init}} \cap \text{ldrrel}(F) \cap \rho_{ts}
\end{align*}
\]
Based on these relations, we further define:

\[
\begin{align*}
\rho_{cnt}(F, V_1, V_2) & \triangleq \text{cntrel}_1(F, V_1) \cap \text{cntrel}_2(F, V_2) \\
\rho_{cnt}=(F, V_1) & \triangleq \text{cntrel}_1(F, V_1) \\
\rho_{full}(F) & \triangleq \rho_{cnt}(F, \{0, 1, \perp\}, \{0, 1\}) \\
\rho_{val}(F) & \triangleq \rho_{cnt}(F, \{0, 1\}, \{0, 1\}) \\
\rho_{val, \perp}(F) & \triangleq \rho_{val}(F) \cap \text{decrel}_\perp \\
\rho_{\perp, \perp}(F) & \triangleq \rho_{cnt}(F, \{\perp\}, \emptyset) \cap \text{decrel}_\perp
\end{align*}
\]

Note that we have \(\text{cntrel}_1(\{\text{inp}\}, \{0, 1\}) = \rho_{full}(\{\text{inp}\})\), since the \text{inp} field can only have values in \{0, 1\}.

C.2.3 Simulation relations

Next, we define the simulation relations for agreement and for termination in terms of the state relations above. For this purpose, we need some auxiliary definitions.

Using the notion of activity spans defined in Section 6.1.3, we define the set of ephemeral active fields in state \(s\) by

\[
\text{active-f}(s) = \{f \mid (j, f, k) \in \text{spans}(s) \land f \text{ is ephemeral}\}.
\]

Note that \(\text{next-ldrf}(s) \subseteq \text{active-f}(s) \cup \{\text{inp}\}\). Next, we define the following two sets of fields derived from the phase graph corresponding to \(s.rnd\)’s phase:

- \(\text{inp-path}(s) = \pi_2(N_{\text{inp}})\), where \(N_{\text{inp}}\) is the set of nodes that lie on the path from the root to the leaf containing \text{inp} (if such a leaf exists) or the empty set otherwise. (Note that, by (R1), there can be at most one such leaf.)

- \(\text{dec-path}(s) = \pi_2(N_{\text{dec}})\), where \(N_{\text{dec}}\) is the set of remaining nodes of the phase graph, i.e., those not in \(N_{\text{inp}}\).

Here, \(\pi_2\) is the projection of a pair to its second component.

In the simulation relations, we use the following sets of fields depending on a state \(s\):

\[
\begin{align*}
F_1(s) & = (\text{inp-path}(s) \cap \text{active-f}(s)) \setminus \text{next-ldrf}(s) \\
F_2(s) & = (\text{dec-path}(s) \cap \text{active-f}(s)) \setminus \text{next-ldrf}(s) \\
F_1'(s) & = \text{inp-path}(s) \cap \text{next-ldrf}(s) \\
F_2'(s) & = \text{dec-path}(s) \cap \text{next-ldrf}(s)
\end{align*}
\]

Note that the sets \(\text{inp-path}(s)\) and \(\text{dec-path}(s)\) depend only on the type of phase that \(s\) is part of and remain constant within a phase whereas
C. Proof of the Cutoff Theorem

The cutoff theorem states that active-\textit{f}(s) and next-ldrf(s) (and hence \( F_1(s) \) and \( F_2(s) \)) also depend on the round of the state.

These sets have the following properties:

- Each of \( F_1(s) \), \( F_2(s) \), \( F^c_1(s) \), or \( F^c_2(s) \) is either the empty set or a singleton set. Their union contains at most two fields.

- \( F_1(s) \) and \( F_2(s) \) only contain ephemeral fields, while \( F^c_1(s) \) and \( F^c_2(s) \) may contain inp or ephemeral fields.

- \( sf \in F_1(s) \cup F_2(s) \cup \{\text{inp}\} \) whenever \( \text{rnd}(s) \) is not a \( < \alpha \) round and has a send field \( sf \).

- \( sf \in F^c_1(s) \cup F^c_2(s) \) whenever \( \text{rnd}(s) \) is a \( < \alpha \) round with send field \( sf \).

- \( f \in F_1(s) \cup F_2(s) \cup F^c_1(s) \cup F^c_2(s) \) if \( f \) is ephemeral and \( s \) is the state resulting from a transition where \( f \) is updated.

The simulation relation for agreement is then defined as follows.

\[
(s_a, s_c) \in \rho_{agr} \triangleq (s_a, s_c) \in \rho_{basic}(F^c_1(s_c)) \\
\quad \cap \rho_{full}(F_1(s_c)) \\
\quad \cap \rho_{val}(F_2(s_c)) \\
\quad \cap \text{ldrrel}_{val}(F^c_2(s_c))
\]

Finally, we define our backward-forward simulation used for termination, which is defined on \( \Sigma_k \times A(\Sigma_B) \). In the definition, \( R - 1(\{y\}) \) denotes \( \{x \mid (x, y) \in R\} \).

\[
(S_a, s_c) \in \rho_{term} \triangleq \\
S_a = \rho_{basic}(F^c_1(s_c))^{-1}(\{s_c\}) \\
\quad \cap \rho_{full}(F_1(s_c))^{-1}(\{s_c\}) \\
\quad \cap \rho_{val}(F_2(s_c))^{-1}(\{s_c\}) \\
\quad \cap \text{ldrrel}_{val}(F^c_2(s_c))^{-1}(\{s_c\}) \\
\lor S_a = \rho_{basic}(F^c_1(s_c))^{-1}(\{s_c\}) \\
\quad \cap \rho_{full}(F_1(s_c))^{-1}(\{s_c\}) \\
\quad \cap \rho_{val}(F_2(s_c))^{-1}(\{s_c\}) \\
\quad \cap \text{ldrrel}_{val}(F^c_2(s_c))^{-1}(\{s_c\})
\]
C.3 The Lemmas

C.3.1 Multiset relations

**Lemma C.1** (cntMS is total). Assume $M_c$ is a $k$-element multiset. Then, there exist an $B$-element multiset $M_a$ such that:

$$(M_a, M_c) \in \text{cntMS} = \{0, 1, \perp\}.$$ 

**Proof.** Let $V = \{0, 1, \perp\}$, and let $t_v = \gamma_k(M_c(v))$. If there is only one value $v \in V$ with $t_v \geq 0$ then we set $M_a(v) = B$ and $M_a(v') = 0$ for $v' \neq v$ and we are done. Otherwise, we have $t_v \geq 0$ for $l \in \{2, 3\}$ values $v \in V$. By (CO5), we have $\sum_{v \in V} t_v \leq \frac{d-1}{2}$, and by (CO9) have $\sum_{v \in V} \alpha_B(t_v) \leq B$, i.e., we can fit $\alpha_B(t_v)$ of each value $v$ in $M_a$. Using (CO6), (CO7) and (CO12), we can show that there are no more than $l$ elements missing, i.e., $r = B - \sum_{v \in V} \alpha_B(t_v) \leq l$. Hence, we can increment $M_a(v)$ for the $v$’s with $t_v > 0$ by one as necessary to fill up $M_a$ to $B$ elements, while preserving $\gamma_B(M_a(v)) = t_v$ for all $v \in V$. \qed

**Lemma C.2** (Matching counts). Assume $M_1$ and $M_2$ are multisets of sizes $k$ and $B$, and assume we are given two sets $X$ and $Y$. Let $c_1(Z) = \sum_{z \in Z} M_i(z)$. Assume $c_2(X) > 0$ and $c_2(Y) > 0$, $c_2(X) + c_2(Y) = k$ and $c_1(X) + c_1(Y) = B$. Furthermore, assume $\gamma_B(c_1(Z)) \geq \gamma_k(c_2(Z))$, for $Z \in \{X, Y\}$. Then we have $\gamma_B(c_1(Z)) = \gamma_k(c_2(Z))$ for $Z \in \{X, Y\}$.

**Proof.** Since $\gcd(d, k) = \gcd(d, B) = 1$ by (dk-coprim) and our choice of $B$, we can apply equation (CO6) twice to obtain $\gamma_B(c_1(X)) + \gamma_B(c_1(Y)) = \gamma_k(c_2(X)) + \gamma_k(c_2(Y))$. From the assumptions of the lemma we also have $\gamma_B(c_1(Z)) \geq \gamma_k(c_2(Z))$ for $Z \in \{X, Y\}$. Hence, the result follows by arithmetic. \qed

**Lemma C.3** (cntMS preserves 2-sums). Assume $M_1$ and $M_2$ are multisets of sizes $B$ and $k$ respectively, such that $(M_1, M_2) \in \text{cntMS} = \{V_\perp\}$. Then, for any $x \neq y$, $\gamma_B(M_1(x) + M_1(y)) = \gamma_k(M_2(x) + M_2(y))$.

**Proof.** Let $z \in V_\perp$ be such that $x \neq z \neq y$. If $M_2(x) = 0$, $M_2(y) = 0$, or $M_2(z) = 0$, the result follows from the definitions of cntMS and $\gamma$. Otherwise, $M_2(x) > 0$, $M_2(y) > 0$, and $M_2(z) > 0$, and the result follows from the definition of cntMS, $(\gamma+1)$, $(\gamma+2)$, and (CO6). \qed

C.3.2 Preservation of guards

**Lemma C.4** (Simulation relations preserve guards). Fix $C \in \mathcal{C}(\mathcal{A})$, and let $f = f(C)$. Assume the following hold:

- if $\text{pred}(C) \neq \text{maxts}$, then $(s_a, s_c) \in \text{pred}(\{f\})$,
- if $\text{pred}(C) = \text{maxts}$, then $(s_a, s_c) \in \text{tsrel}(\text{th}(C))$
Then, there exists a set \( W_a \) such that \( s_a, W_a \models C(v) \), and if \( \text{pred}(C) = \text{any} \) then also \( \gamma_B(|W_a|) = \gamma_k(|W_c|) \). Moreover, if \( \text{pred}(C) \neq \text{maxt} \), we can choose \( W_a \) such that \( \#s_a(f)[W_a] \) contains both 0 and 1 values whenever \( \#s_c(f)[W_c] \) does.

**Proof.** If \( \text{pred}(C) = \text{maxt} \), the claim follows directly from the definition of \( tsrel \) and the semantics of guards. Thus, assume \( \text{pred}(C) \neq \text{maxt} \). We first treat the satisfaction of the threshold \( th(C) \) and then the different predicates. Since \( s_c, W_c \models C(v) \), we have \( \#s_c(f)[W_c](0) + \#s_c(f)[W_c](1) \geq \alpha_k(th(C)) \). Hence, \( \gamma_k(\#s_c(f)[W_c](0) + \#s_c(f)[W_c](1)) \geq th(C) \) and by \( cntMS_{\Sigma \geq 1} \), we also have that \( \gamma_B(\#s_a(f)[W_a](0) + \#s_a(f)[W_a](1)) \geq th(C) \). Therefore, \( \#s_a(f)[W_a](0) + \#s_a(f)[W_a](1) \geq \alpha_B(th(C)) > th(C) \cdot B \) as required. Let \( t_1 = \gamma_k(|W_c|) \). We proceed by case distinction on \( \text{pred}(C) \).

- **smor:** by (R3), \( \#s_c(f)[\bot] = 0 \). Since \( s_c, W_c \models C(v) \), at least half of the processes in \( W_c \) must have \( v \) as their value. Then, since \( |W_c| \geq \alpha_k(t_1) \):

\[
\begin{align*}
\#s_c(f)[v] & \geq \frac{\alpha_k(t_1)}{2} \\
\iff \#s_c(f)[v] & \geq \frac{|k \cdot t_1| + 1}{2} \quad \text{(by def. } \alpha \text{ and } t_1 \geq 0) \\
\iff \#s_c(f)[v] & \geq \frac{|k \cdot t_1|}{2} + \frac{1}{2} \\
\iff \#s_c(f)[v] & \geq \left\lfloor \frac{k \cdot t_1}{2} \right\rfloor + \frac{1}{2} \quad \text{(by properties of } \lfloor \cdot \rfloor) \\
\iff \#s_c(f)[v] & \geq \alpha_k\left(\frac{t_1}{2}\right) \quad \text{(since } \#\lfloor \cdot \rfloor \text{ is integral) } \\
\iff \#s_a(f)[v] & \geq \alpha_B\left(\frac{t_1}{2}\right) \quad \text{(by } cntrel_1) \\
\iff \#s_a(f)[v] & \geq \left\lfloor \frac{B \cdot t_1}{2} \right\rfloor + 1 \quad \text{(by def. } \alpha \text{ and } \frac{B}{2} \geq 0) \\
\iff \#s_a(f)[v] & \geq \left\lfloor \frac{B \cdot t_1}{2} \right\rfloor + 1 \quad \text{(by properties of } \lfloor \cdot \rfloor) \\
\iff \#s_a(f)[v] & \geq \alpha_B\left(\frac{t_1}{2}\right) + 1 \quad \text{(by } cntrel_1) \quad \text{(by } \alpha_B(t_1) \text{ odd by } (CO11))
\end{align*}
\]

Hence, \( s_a, W_a \models C(v) \) for an appropriate witness \( W_a \). Moreover, in case \( \#s_c(f)[W_c] \) contains both 0’s and 1’s, then both values also appear in \( \#s_a(f) \) by \( cntrel_1 \). Since \( t_1 > 0 \) (by the language restriction (R4)) and \( \alpha_B(t_1) \) is odd (by (CO11)), we have \( \alpha_B(t_1) \geq 3 \). Therefore, we
Lemma C.5 (Downsizing witnesses). Assume $s$ is of size $n$ and $s,W \models C(v)$. Then, for any $c$ such that $\alpha_n(th(C)) \leq c \leq |W|$, there exists $W' \subseteq W$ such that $|W'| = c$ and $s,W' \models C(v)$.

Proof. We simply remove processes from $W$, starting with the processes $p$ where $s(f(C))(p) \neq v$, respectively $s(f(C))(p) \neq (ts,v)$ for some $ts$, if $f(C)$ is timestamped.

C.3.3 Preservation of updates

Lemma C.6 (Preserving completely successful local updates). Assume that $s_c \xrightarrow{A} s'_c$ is a step with send field $sf$, communication pattern $cp$, and set of update fields $U$. Suppose $\text{rnd}(s_a) = \text{rnd}(s_c)$ and further assume:

1. if $cp = \lessdot_r$, then $(s_a,s_c) \in \text{ldrrel}_{val} \{sf\}$
2. if $cp \neq \lessdot_r$, the protocol is timestamped, and $sf = \text{inp}$, then $(s_a,s_c) \in \rho_{ts}$
3. otherwise, $(s_a,s_c) \in \rho_{val} \{sf\}$
Suppose process $p$ performs the local update $u_p$ in the step and $\bot \notin \#[u_p]$. Then, there exists a set of processes $W \subseteq \Pi(s_a)$ such that, for all processes $q \in \Pi(s_a)$, if

$$cp = \exists z \implies (ldr(s_c) = p \iff ldr(s_a) = q)$$

then $s_a, W \models C(u_p(f(C)))$ for all constraint $C$ related to the step, i.e., using $W$ as its HO set, process $q$ can perform update $u_p$ from $s_a$. Moreover, using $W$ as the HO set does not violate any threshold or leader formulas in $\Lambda$.

**Proof.** If $cp \not= \not<$, the claim follows for $W = \Pi$ straight from (R5), the definitions of the language semantics and of the relation $ldr_{rel}$. Note that $\phi_{ls}$ is the only possible label in this case.

Assume $cp \not= \not<$. Let $HO_p$ be the HO set of $p$ in the step $s_c \xrightarrow{\Lambda} s_c'$. We must find a $W$ such that $s_a, W \models C(u_p(f))$, for all fields $f \in U$ and corresponding constraints $C$. Additionally, the choice of $W$ must respect any $\phi_{lp}$ and $\phi_{lh}$ formulas in $\Lambda$; note that, by (L2), we have $\phi_{ls} \not\in \Lambda$. We first consider constraint satisfaction.

If $U = \{f\}$ for some $f$, then $s_c, HO_p \models C(u_p(f))$. Use Lemma C.4 to obtain the required $W$ and we are done.

Otherwise, $U = \{f_1, f_2\}$. WLOG, assume $C_1 \subseteq C_2$. Perform a case distinction:

- $u_p(f_1) = u_p(f_2) = v$. This is similar to the singleton $U$ case. We know that $s_c, HO_p \models C_1(v)$; invoke Lemma C.4 to obtain the $W$. By constraint ordering, we also have $s_a, W \models C_2(v)$.

- $u_p(f_1) = v_1 \not= v_2 = u_p(f_2)$. Clearly, none of the associated predicates is all$\not=$; and not both are the same selection, since selections are deterministic. Hence, we have $\text{pred}(C_2) = \text{any}$ by (R2) and also $\text{pred}(C_1) = \text{maxts}$ by (R8).

We invoke Lemma C.4 with $C_1$ to obtain a set $W$. Since $HO_p$ is a witness for both $C_1$ and $C_2$, $\#s_c(sf)[HO_p]$ contains both 0 and 1 and, by Lemma C.4, so does $\#s_a(sf)[W]$. Thus, since $\text{pred}(C_2) = \text{any}$ and $C_2 \subseteq C_1$ implies $\text{th}(C_2) \leq \text{th}(C_1)$, also $s_a, W \models C_2(u(f_2))$.

It remains to show that no $\phi_{lp}(c)$ and $\phi_{lh}(c)$ formula in $\Lambda$ is violated. Assume such a formula exist, for some $c$. Perform a case distinction on $\text{pred}(C)$, where $C$ is the constraint that we invoked Lemma C.4 with. If $\text{pred}(C) \not= \text{any}$, then, by (R4) and (L3), $c = \text{th}(C)$, and since $s_a, W \models C(v)$ for some $v$, we have $\gamma_B(|W|) \geq c$. If $\text{pred}(C) = \text{any}$, Lemma C.4 guarantees that $\gamma_B(|W|) = \gamma_k(|HO_p|)$ and hence that $|HO_p| > c \cdot k$ implies $\gamma_B(|W|) \geq c$.

□

**Lemma C.7** (Preserving completely failed local updates). Assume $s_c \xrightarrow{\Lambda} s_c'$, denote the corresponding send field for the step by $sf$, the set of fields in the
instructions of the step by \( U \) and the communication pattern by \( cp \). Assume that the local update \( u_p \) performed by a process \( p \) in the step was empty, that is, \( u_p(f) = \bot \) for all \( f \in U \). Assume \( \text{rnd}(s_a) = \text{rnd}(s_c) \). Then, there exists a condition \( \Psi \), such that:

1. if \( cp = \leq_s \), then \( \Psi = (s_a, s_c) \in \text{ldr}\text{rel}(\{sf\}) \)
2. if \( cp \neq \leq_s \), then either
   
   \[
   \Psi = (s_a, s_c) \in \rho_{\text{cnt}}(\{sf\}, \{0, 1\}, \{0, 1\})
   \]
   
   or
   
   \[
   \Psi = (s_a, s_c) \in \rho_{\text{cnt}}(\{sf\}, \{\bot\}, \emptyset)
   \]

and such that, if \( \Psi \) is satisfied, there exists a set \( W \) of processes in \( s_a \), such that, for any process \( q \) in \( \Pi(s_a) \) such that

\[
cp = \geq_s \implies (p = \text{ldr}(s_c) \iff q = \text{ldr}(s_a)),
\]

and for any \( s' \) such that \( s_a \rightarrow s'_a \), whenever the HO set of \( q \) is \( W \), the process \( q \) performs the empty update \( u_p \) in the step to \( s'_a \). Moreover, setting the HO set of \( q \) to \( W \) cannot violate any leader or threshold formulas in \( \Lambda \).

**Proof.** If \( cp = \geq_s \) and \( p \neq \text{ldr}(s_c) \), then, by the assumptions, \( q \neq \text{ldr}(s_a) \) and the result is immediate from the semantics. Otherwise, take \( C \) to be the weakest constraint associated to the step \( s_c \rightarrow s'_c \) and \( f = f(C) \). Then, we construct a HO set \( W \) such that \( W \) is allowed by the leader and threshold formulas in \( \Lambda \), and for all \( v, s_a, W \neq C(v) \). Since \( C \) is the weakest constraint, we can then perform the empty update for the entire \( U \).

If \( \{\phi \mid \exists c. \phi \in \{\phi_{ls}, \phi_{lr}(c), \phi_{th}(c)\}\} \cap \Lambda = \emptyset \), set \( W = \emptyset \) and we are done (\( \Psi \) can be arbitrary). Note that this is necessarily the case if there exists an associated constraint \( C \) such that \( \text{pred}(C) = \text{maxts} \); otherwise, since \( \text{th}(C) > 0 \) by (R4), by (L3) we conclude that an empty update is not possible. Thus, in the rest, we assume that \( \text{pred}(C) \neq \text{maxts} \) for any of the constraints of the step, and we assume \( \{\phi \mid \exists c. \phi \in \{\phi_{ls}, \phi_{lr}(c), \phi_{th}(c)\}\} \cap \Lambda \neq \emptyset \).

We perform a case distinction on \( cp \):

- if \( cp = \leq_s \), then by (L3), we have \( \phi_{ls} \in \Lambda \). Since \( u_p(f) = \bot \), we conclude \( s_c(sf)(\text{ldr}(s_c)) = \bot \). Then, assuming \( \Psi = (s_a, s_c) \in \text{ldr}\text{rel}(\{sf\}) \), we let \( W = \Pi \).

- otherwise, for some \( c, \phi_{th}(c) \in \Lambda \) or \( \phi_{tr}(c) \in \Lambda \). By restriction (L3), \( \text{th}(C) = 0 \) or \( \text{th}(C) = c \). Perform a further case distinction:

  - \( c = \text{th}(C) \), and \( \#(s_c(sf))(\bot) > 0 \). Then, choose

    \[
    \Psi = (s_a, s_c) \in \rho_{\text{cnt}}(\{sf\}, \{\bot\}, \emptyset)
    \]

    which guarantees that \( s_a(sf,p') = \bot \) for some \( p' \); pick \( W \) to be any set of size \( \alpha_B(c) \) that includes \( p' \).
C. Proof of the Cutoff Theorem

\[ c = \text{th}(C), \text{ and } \#[s_e(s_f)](\bot) = 0. \] Then, we conclude that \( \text{pred}(C) = \text{all} \), and that \( \forall b \in \{0,1\}. \#[s_e(s_f)](b) > 0; \) thus, we pick

\[ \Psi = (s_a, s_c) \in \rho_{\text{cnt}}(\{s_f\}, \{0,1\}, \{0,1\}) \]

and pick \( W \) to be any set of processes in \( s_a \) of size at least \( \alpha_B(\gamma_k([H0_p])) \) such that \( \forall b \in \{0,1\}. \#s_a(s_f)[W](b) > 0; \) this is possible by \( \text{cntMS} \geq \). Using \( W \) as the HO set cannot violate any \( \phi_l \) or \( \phi_r \) in \( \Lambda \).

\[ \text{th}(C) = 0 \neq c. \] Then, by (R4), \( \text{pred}(C) = \text{any} \). We conclude that \( \gamma_k(\#[s_e(s_f)](\bot)) \geq c, \) and pick:

\[ \Psi = (s_a, s_c) \in \rho_{\text{cnt}}(\{s_f\}, \{\bot\}, \emptyset) \]

Thus, \( \#[s_a(s_f)](\bot) \geq \alpha_B(c), \) and we pick the processes holding \( \bot \) in \( s_f \) as our \( W \).

\[ \square \]

Lemma C.8 (Preserving local updates). Assume \( s_c \xrightarrow{\Lambda} s'_c \); denote the corresponding send field for the step by \( s_f \); and the communication pattern by \( cp \). Assume \( \text{rnd}(s_a) = \text{rnd}(s_c) \). Further, assume:

1. if \( cp = s'_c \), then \((s_a, s_c) \in \text{ldrrel}(\{s_f\});\)
2. if \( cp \neq s'_c \), \( s_f = \text{\#inp}, \) and the protocol is timestamped, then \((s_a, s_c) \in \rho_{\text{ts}});\)
3. otherwise, \((s_a, s_c) \in \rho_{\text{cnt}}(\{s_f\}, \{0,1, \bot\}, \{0,1\}).\)

Assume that a process \( p \) performed a local update \( u_p \) in the step to \( s'_c, \) and assume that \( q \) is such that:

\[ cp = s'_c \iff (p = \text{ldr}(s_c) \iff q = \text{ldr}(s_a)).\]

Then, there exists a set \( W \) of processes in \( s_a \), such that, whenever \( s_a \rightarrow s'_a \) and the HO set of \( q \) is \( W \), \( q \) can perform the local update \( u_p \) in the step to \( s'_a \). Moreover, using \( W \) as a HO set does not violate any threshold or leader predicates in \( \Lambda \).

Proof. Denote the set of fields in the instructions of the step by \( U \). By the property (PS1), \( |U| \in \{1,2\}. \) If \( u_p(f) = \bot \) for all \( f \in U \), then invoke Lemma C.7. If \( u_p(f) \neq \bot \) for all \( f \in U \), invoke Lemma C.6. Note that one of the previous two cases must hold whenever \( cp = s'_c \) (by (R5)), or whenever \( sf = \text{\#inp} \) and the protocol is timestamped (by (R8)), or whenever \( |U| = 1 \).

Hence, assume \( cp \neq s'_c \), assume \( U = \{f_1, f_2\} \), and WLOG, \( u_p(f_1) = v \) and \( u_p(f_2) = \bot \). Denote the associated constraints by \( C_1 \) and \( C_2 \). As noted, we assume \( \text{pred}(C_1) \neq \text{maxts} \neq \text{pred}(C_2). \)
We must find a set \( W \) such that \( s_a,W \models C_1(v) \) and \( s_a,W \not\models C_2(v') \), for all \( v' \). Let \( HO_p \) be the HO set of \( p \) in the step \( s_c \xrightarrow{\Lambda} s'_c \), and let \( W_c \subseteq HO_p \) be the set of processes from whom \( p \) received non-\( \perp \) messages. Then, \( s_c,W_c \models C_1(v) \). Since \( u_p(f_2) = \perp \), by the language semantics we have that either:

- \( \gamma_k(|W_c|) < th(C_2) \). Then, \( th(C_1) < th(C_2) \). For any \( c \) such that \( \{\phi_{th}(c),\phi_{tr}(c)\} \cap \Lambda \neq \emptyset \), we then have \( c = th(C_2) \) by (L3). Furthermore, (L3) also implies \( th(C_1) = 0 \) and \( pred(C_1) = any \).

Since \( s_c,HO_p \models C_1(v) \), apply Lemma C.4 with \( C_1 \) and \( HO_p \) to obtain a set \( W_a \), and if \(|W_a| > \alpha_B(th(C_2))\), use Lemma C.5 to trim it to a set \( W'_a \) with \(|W'_a| = \alpha_B(th(C_2))\). If \( s_a(sf)(p) = \perp \) for at least one process in \( W'_a \), we choose \( W = W'_a \) and we are done. Otherwise, first use Lemma C.5 to trim down \( W'_a \) by one further process, and then add one process \( p \) with \( s_a(sf)(p) = \perp \) (it must exist by \#\( s_a(sf) \)(\( \perp \)) \( > 0 \) and the definition of \( cntMS_\perp \)) to the result obtain the desired \( W \) (with \(|W| = \alpha_B(th(C_2))\)).

- \( \gamma_k(|W_c|) \geq th(C_2) \). Then, \( pred(C_2) = alls \) and both 0 and 1 were received by \( p \). Use Lemma C.4 with \( C_1 \) and \( W_c \) to obtain \( W \); we conclude \( s_a,W \models C_1(v) \) and, for all \( v' \), \( s_a,W \not\models C_2(v') \). Moreover, for any \( c \) such that \( \{\phi_{th}(c),\phi_{tr}(c)\} \cap \Lambda \neq \emptyset \), we also conclude \( \gamma_B(\|W\|) \geq c \).

If \( pred(C_1) = any \), this follows from Lemma C.4 and the fact that \( \gamma_k(W_c) \geq c \). Otherwise, this follows since \( s_a,W \models C_1(v) \) and since by (R4) and (L3) we have \( c = th(C_1) \).

\[ \square \]

**Lemma C.9** (Feasibility of various local updates). Assume \( s \xrightarrow{\Lambda} s' \), assume the set of fields in the step’s instructions is \( U = \{f_1,f_2\} \), and assume the corresponding constraints are \( C_1 \) and \( C_2 \). Denote by \( \mathcal{U} \) the associated global update, and denote by \( cp \) the communication pattern of the step.

If \( C_1 \subseteq C_2 \), then:

1. whenever \( \mathcal{U}(f_1)(p) = \perp \), then also \( \mathcal{U}(f_2)(p) = \perp \).

2. if \( \mathcal{U}(f_2)(p) = v_2 \neq \perp \) for some process \( p \), then a process \( q \) with the same HO set as \( p \) could also perform the update \( (v_2)_{v_1} \), where \( q \) is any process if \( cp \neq \perp \), and \( ldr(s) \) if \( cp = \perp \).

3. assume that \((v_1)_{v_2}\) with \( v_1 \neq v_2 \) appears in \( \mathcal{U} \), and also \((v)_{v_2} \). Then, there exists a set \( W \) such that any process whose HO set is \( W \) can perform the update \( (v_1)_{v_2} \). Moreover, using \( W \) as the HO set does not violate any of the formulas from \( \Lambda \).
Proof. (1) and (2) follow from the definition of constraint ordering. For (3), first note that the existence of both \( \nu_1 \) and \( \nu_2 \) implies that \( \phi_{af} \notin \Lambda \). Next, note that the existence of \( \nu_1 \) implies that \( cp \neq \emptyset \) and thus \( \phi_{ts} \notin \Lambda \), by (L2), and that neither of \( C_1, C_2 \) can be \( \text{all}=\); and since selections are deterministic and since from (R2) all selections must be of the same type, we also conclude that not both of \( \text{pred}(C_1) \) and \( \text{pred}(C_2) \) are selections; and since \( C_1 \subseteq C_2 \), we have \( \text{pred}(C_1) = \text{any} \). Take the HO set \( W' \) that yielded the update \( \nu_1 \). If needed, replace one process \( q \) from \( W' \) that had \( s(sf)(q) = v \) by some process \( p \) such that \( s(sf)(p) = v_1 \). This yields the desired \( W \). Moreover, using \( W \) as the HO set does not violate any formula of form \( \phi_{th}(c) \) or \( \phi_{tr}(c) \) in \( \Lambda \).

C.3.4 Preservation of update distributions

The following two lemmas tell us that, for concrete steps in which only one field is updated, we can mimic the distribution of values in the abstract steps.

**Lemma C.10** (Preserving \( \forall \) in 1-field global updates). Assume \( s_c \xrightarrow{\Lambda} s'_c \) is a step with send field \( sf \), communication pattern \( cp \), set of updated fields \( U \), and associated global update \( U_c \). Suppose that \( U = \{f\} \), \( \text{rnd}(s_a) = \text{rnd}(s_c) \), and further assume:

- if \( cp = \emptyset \), then \( (s_a, s_c) \in ldrrel_{val}(\{sf\}) \)
- if \( cp \neq \emptyset \), the protocol is timestamped, and \( sf = \text{inp} \), then \( (s_a, s_c) \in \rho_{ts} \)
- otherwise, assume \( (s_a, s_c) \in \rho_{val}(\{sf\}) \)

Then, there exists a transition \( s_a \xrightarrow{\Lambda} s'_a \) and a corresponding global update \( U_a : U \rightarrow \Pi(s_a) \rightarrow \{0, 1, \perp\} \) such that

\[
(U_a, U) \in \rho_{val}(\{f\}) \cap \text{ldrrel}_{val}(\{f\}).
\]

**Proof.** For any \( b \in \{0, 1\} \) with \( \#(U_c(f))(b) > 0 \), we obtain the corresponding local update \( \{f \mapsto b\} \) from Lemma C.6, respecting all the leader and threshold predicates from \( \Lambda \). We simply take at least \( \alpha_E(\gamma_k(\#(U_c(sf))(b))) \) of each, for every \( b \in \{0, 1\} \), making sure that \( ldr(s_c) \) performs the same update as \( ldr(s_c) \). If \( \phi_{af} \notin \Lambda \), or if \( \#(U_c(f))(b) > 0 \) for at most one \( b \), this is obviously possible. If \( \phi_{af} \in \Lambda \) and \( \#(U_c(f))(b) > 0 \) for all \( b \in \{0, 1\} \), from the language semantics and (L1) we conclude that the guard must be \( \text{any} \) and that \( cp \neq \emptyset \). The result follows from the guard semantics for \( \text{any} \) and Lemma C.4.

**Lemma C.11** (Preserving \( \perp \) in 1-field global updates). Assume \( s_c \xrightarrow{\Lambda} s'_c \). Denote the corresponding send field for the step by \( sf \), the set of fields updated in the step by \( U \), the global update performed in the step by \( U_c \) and the communication pattern by \( cp \). Assume \( U = \{f\} \) and \( \text{rnd}(s_a) = \text{rnd}(s_c) \).

Then, there exists a condition \( \Psi \) such that:
C.3. The Lemmas

- if \( cp = \geq \), then \( \Psi = (s_a, s_c) \in ldrrel(\{sf\}) \)
- if \( cp \neq \geq \), then either:
  \[
  \Psi = (s_a, s_c) \in \rho_{cnt}(\{sf\}, \{0, 1\}, \{0, 1\})
  \]

  or

  \[
  \Psi = (s_a, s_c) \in \rho_{cnt}(\{sf\}, \{\bot\}, \emptyset)
  \]

and such that, if \( \Psi \) is satisfied, there exists a state \( s'_a \) such that \( s_a \xrightarrow{\Lambda} s'_a \) and for the corresponding global update \( U_a \) we have:

\[
(U_a, U_c) \in \rho_{cnt}(\{f\}, \{\bot\}, \emptyset)
\]

and

\[
(U_a, U_c) \in ldrrel(\{f\}).
\]

Proof. By case distinction on \( cp \).

1. Assume \( cp = \geq \). By the language semantics all processes other than the leader perform the empty local update regardless of their HO sets. Then, \( (U_a, U_c) \in \rho_{cnt}(\{f\}, \{\bot\}, \emptyset) \) follows immediately, and the only thing left to find the \( \Psi \) that guarantees \( (U_a, U_c) \in ldrrel(\{f\}) \), and only when \( U_c(f)(ldr(s_c)) = \bot \). We obtain it from Lemma C.7.

2. Assume \( cp \neq \geq \). If \( \#[U_c(f)](\bot) = 0 \), there is nothing to prove. If \( \#[U_c(f)](\bot) > 0 \), we obtain the corresponding \( \Psi \) and \( W \) for the empty update from Lemma C.7. We set the HO set of all the processes to \( W \).

\[\Box\]

Lemma C.12 (Preserving 1-field global updates). Assume \( s_c \xrightarrow{\Lambda} s'_c \). Denote the corresponding send field for the step by \( sf \), the set of fields updated in the step by \( U \), the global update performed in the step by \( U_c \), and the communication pattern by \( cp \). Assume that \( rnd(s_a) = rnd(s_c) \), \( U = \{f\} \) and assume

1. if \( cp = \geq \), then \( (s_a, s_c) \in ldrrel(\{sf\}) \)
2. if \( cp \neq \geq \), and if \( sf = inp \) and if the protocol is timestamped, then \( (s_a, s_c) \in \rho_{ts} \)
3. otherwise, \( (s_a, s_c) \in \rho_{full}(\{sf\}) \)

Then, there exists a transition \( s_a \xrightarrow{\Lambda} s'_a \) and a corresponding global update \( U_a \) such that

\[
(U_a, U_c) \in \rho_{cnt} = (\{f\}, \{0, 1, \bot\}).
\]

169
Proof. The conditions of Lemmas C.6 and C.7 are satisfied (regardless of which \( \Psi \) Lemma C.7 chooses). We use them to obtain the same local updates from \( U_c \), and then redistribute them along the lines of Lemma C.1. It is easy to see that this does not violate any formulas in \( \Lambda \). \( \square \)

For simultaneous updates, the situation gets much more complicated. By the property (PS1), only two fields \( f_1 \) and \( f_2 \) can be simultaneously updated. Given a global update \( U \), we can associate with it a count matrix. This is a \( 3 \times 3 \) matrix with an associated ordering \( o : \{1, 2, 3\} \mapsto \{0, 1, \bot\} \), such that the entry \( c_{ij} \) corresponds to the count of the tile \( (o(i), o(j)) \). Then, with \( i = o^{-1}(v) \), \( \#[U(f_1)](v) = \sum_{j=1}^3 c_{ij} \), and \( \#[U(f_2)](v) = \sum_{j=1}^3 c_{ji} \).

By (R2), we can, WLOG, assume that for the associated update constraints \( C_1 \) and \( C_2 \):

\[ C_2 \sqsubseteq C_1. \]  

(ctr-ord)

This will allow us to simplify the matrix by assuming it has a certain shape. First, we will always assume that \( o(3) = \bot \). We then note that by (ctr-ord), \( (\bot) \) tiles cannot exist. Thus, \( c_{13} = c_{23} = 0 \). Moreover, we show the following lemma:

Lemma C.13 (Removing a tile type). Given a global update \( U \) we can create another update \( U' \) of the same size such that:

1. \( U' \) arises from the same \( HO \) sets as \( U \)
2. \( U' \) contains at most one of the following two types of tiles: \( (0) \) and \( (1) \)
3. \( \#[U'(f_i)](v) = \#[U(f_i)](v) \) for \( i = 1, 2 \) and \( v \in \{0, 1, \bot\} \).

Proof. If not both \( (0) \) and \( (1) \) tiles appear in \( U \), take \( U' = U \) and we are done. Otherwise, by Lemma C.9 item (2), we conclude that \( (0) \) and \( (1) \) tiles are possible using the same \( HO \) sets as in \( U \). Replace each pair of \( (0) \) and \( (1) \) tiles by a pair of \( (0) \) and \( (1) \) tiles until (at least) one type is exhausted. \( \square \)

Since in this section, we will only care about counts of values in the update, we will regularly use Lemma C.13 to assume that for any given update, there exists a \( b \in \{0, 1\} \) such that the tile \( (b) \) does not appear in \( U \). Take \( o(b) = 1 \) and \( o(1 - b) = 2 \). As we already noted that \( c_{13} = c_{23} = 0 \), the resulting count matrix is thus lower triangular.

Next, based on the counts in the big update, \( \#[U_c(f)](x) \) for each updated field \( f \), and value \( x \), we will next determine the target counts in the small system, \( \#[U_a(f)](x) \). Finally, we will construct \( U_a \) such that its count matrix is of a similar shape to the matrix \( U_c \), such that \( U_a \) both achieves the target counts, and that it is possible to create in the small system.

The following lemmas helps us determine these counts.
Lemma C.14 (Small three-slot sums imply positive counts). Assume that 
\( \gcd(n, d) = 1 \), \( \sum_{i=1}^{3} c_i = n \), \( c_i \geq 0 \), and \( \sum_{i=1}^{3} \gamma_n(c_i) = \frac{d-2}{d} \). Then, \( c_i > 0 \), for \( i = 1 \ldots 3 \).

Proof. It is easy to eliminate the case of two \( c_i \)'s being 0. If exactly one \( c_i = 0 \), by (CO6), \( \sum_{j \neq i} \gamma_n(c_j) = \frac{d-1}{d} \), contradicting the assumption of the lemma.

Lemma C.15 (Small three-slot sums imply same two-slot sums). Assume 
\( \gcd(n, d) = 1 \), \( \sum_{i=1}^{3} c_i = n \), \( c_i \geq 0 \), and \( \sum_{i=1}^{3} \gamma_n(c_i) = \frac{d-2}{d} \). Then, for any pair \( i \neq j \),
\[ \gamma_n(c_i + c_j) = \gamma_n(c_i) + \gamma_n(c_j) + \frac{1}{d}. \]

Proof. From Lemma C.14 we get \( c_i > 0 \). The claim then follows from (CO6) (with \( c_{1,j} = c_{i,j} \) and \( c_{2} = c_{k} \)) and (CO7).

Lemma C.16 (Small three-slot sums in \( B \)-process systems). Given any \( c_1, c_2, c_3 \geq 0 \) such that \( \sum_{i=1}^{3} c_i = B \), \( \sum_{i=1}^{3} \gamma_B(c_i) = \frac{d-2}{d} \) if and only if for exactly two values \( i \neq j \), \( c_i = \alpha_B(\gamma_B(c_i)) + 1 \) and \( c_j = \alpha_B(\gamma_B(c_j)) + 1 \).

Thus, letting \( t^X = \sum_{x \in X} \gamma_k(\#[U(f_1)](x)) \) and \( t_X = \sum_{x \in X} \gamma_k(\#[U(f_2)](x)) \),
the first step towards creating the “small” update \( U_a \) will be to select the small counts of a value \( v \) in each field, with the constraint that they be either \( \alpha_B(t^v) \) or \( \alpha_B(t^v) + 1 \) in \( f_1 \), and \( \alpha_B(t_v) \) or \( \alpha_B(t_v) + 1 \).

To ensure that the small update is possible whenever the large update is possible, we have to ensure that the types of the tiles used in the small update are possible. As a first approximation, whenever we have \( c_{i,j} > 0 \) in the large count matrix, Lemma C.8 allows us to have positive values in the same entry of the small matrix. But Lemma C.9 allows us some more freedom:

- if \( c_{21} > 0 \), we can also use the tile \( (\alpha_{(2)}^{(2)}) \) (item (2) of Lemma C.9)
- if \( c_{21} > 0 \) and \( c_{32} > 0 \), we can also use the tile \( (\frac{1}{\alpha_{(1)}}) \) (item (3) of Lemma C.9)

With this in mind, the following combinatorial lemma will help us assemble the count matrix form the small update. Intuitively, the \( b_{ij} \) are the entries of the big count matrix, and \( s_{ij} \) are the entries of the small one. Note that the guarantees of the lemma reflect the above observations – an entry in the small matrix is non-zero either if the corresponding tile existed in the large update, or if we can conclude that it could have existed without disturbing the communication predicates.
Lemma C.17 (Combinatorics of simultaneous updates). Define $R = \{2, 3\}$ and $C = \{1, 2\}$. Given any natural numbers $b_{ij}$, where $i \in R$, $j \in C$, let $br_i = \sum_{j \in C} b_{ij}$, and $bc_j = \sum_{i \in R} b_{ij}$. Assume we are also given natural numbers $sr_i$ and $sc_j$, such that:

1. $\sum_{j \in C} sc_j = \sum_{i \in R} sr_i$
2. $sr_i > 0 \Rightarrow br_i > 0$ and $sc_j > 0 \Rightarrow bc_j > 0$
3. $sr_2 > sc_2 \Rightarrow b_{21} > 0$, and $sr_2 < sc_2 \Rightarrow b_{32} > 0$

Then, there exist natural numbers $s_{ij}$, such that:

1. $s_{21} > 0 \Rightarrow b_{21} > 0$
2. $s_{22} > 0 \Rightarrow b_{21} > 0 \lor b_{22} > 0$
3. $s_{31} > 0 \Rightarrow b_{31} > 0 \lor (b_{21} > 0 \land b_{32} > 0)$
4. $s_{32} > 0 \Rightarrow b_{32} > 0$
5. $\sum_{j \in C} s_{ij} = sr_i$
6. $\sum_{i \in R} s_{ij} = sc_j$

Proof. By Z3.

Finally, we obtain the main result of this section.

Lemma C.18 (Preserving simultaneous global updates). Assume $s_c \xrightarrow{\Lambda} s'_c$ with the associated global update $U_c$; denote the corresponding send field for the step by $sf$, denote the set of fields updated in the corresponding step of $A$ by $U$, and denote the communication pattern by $cp$. Assume $\text{rnd}(a) = \text{rnd}(c)$ and $cp \neq \emptyset$. Further, assume

(i) if $cp = \emptyset$, then $(s_a, s_c) \in \text{ldrrel}(\{sf\})$
(ii) if $cp \neq \emptyset$, if $sf = \text{inp}$, and if the protocol is timestamped, then $(s_a, s_c) \in \rho_{\text{ts}}(\{sf\})$
(iii) otherwise, $(s_a, s_c) \in \rho_{\text{full}}(\{sf\})$

Then, there exists a transition $s_a \xrightarrow{\Lambda} s'_a$ and a corresponding global update $U_a$ such that:

$(U_a, U_c) \in \rho_{\text{cnt}}(U, \{0, 1, \bot\})$
C.3. The Lemmas

Proof. If $U$ is a singleton set, the result follows from Lemma C.12. Thus, we assume that $U = \{f_1, f_2\}$. Moreover, if $\#[U_c(f_i)](v) = k$ for some $i \in \{1, 2\}$ and $v \in \{0, 1, \bot\}$, we are effectively also back in the single field update case. Thus, we assume that $U_c(f_i)$ contains at least two values for all $i \in \{1, 2\}$. As already mentioned, WLOG we also assume (ctr-ord).

Furthermore, we will later show how to reduce the case where one of the fields contains exactly two values while the other one contains three values to the case of equal number of values in each field (either 2 or 3). We thus for the moment assume that both fields contain the same number of values. Since we are only interested in $\#[U_c(f)]$ for $f \in U$, and not in the particular local updates that each process performs, we assume that the count matrix of $U_c$ is lower triangular; this assumption is justified by Lemma C.13. Denote the entries of the count matrix by $b_{ij}$. Let $b = o^{-1}(2)$, where $o$ is the ordering associated to the count matrix. Consequently, $1 - b = o^{-1}(1)$.

If the matrix is diagonal, we’re effectively back in the single field update case. So it is safe to assume that at least one of the off-diagonal terms is larger than 0.

Next, we let $t^X$ and $t_X$ be as previously defined in the section; we alternatively write $t(1)(X) = t^X$, $t(2)(X) = t_X$. Note that the shape of the matrix implies $t^{1-b} \leq t_{1-b}$ and $t_\bot \leq t^\bot$. To create the small update $U_a$, it suffices to determine the entries $s_{ij}$ of its count matrix; the associated ordering is the same as for the count matrix of $U_c$.

We first define small-sum$(i) \triangleq (\sum_{v \in \{0, 1, \bot\}} t(i)(v) = \frac{d-2}{2})$. Note that by Lemma C.14 small-sum$(i)$ implies that all three values are present in $U_c(f_i)$. We proceed by a case distinction.

1. small-sum$(1) = small-sum(2)$. We thus have $\sum B(t_v) = \sum B(t_v')$.

   Let:
   
   \begin{itemize}
   \item $sr_2 = \alpha_B(t^b)$
   \item $sr_3 = \alpha_B(t^\bot) - \alpha_B(t_\bot)$
   \item $sc_1 = \alpha_B(t_{1-b}) - \alpha_B(t^{1-b})$
   \item $sc_2 = \alpha_B(t_b)$
   \end{itemize}

   and use Lemma C.17 to obtain $s_{21}$ to $s_{33}$ (it is easy to verify that the conditions of the lemma are met, taking into account $\sum \alpha_B(t_v) = \sum \alpha_B(t_v')$). Let $x = 1$ if small-sum$(2)$ and 0 otherwise. Let $s_{33} = \alpha_B(t_\bot) + x$, and let $s_{11} = B - (sr_2 + sr_3 + s_{33})$.

2. small-sum$(1)$, but not small-sum$(2)$. By Lemma C.14 we conclude that all three values are present in $f_1$, and thus also in $f_2$ (since both fields have the same number of values). We thus have $2 + \sum \alpha_B(t_v') = \sum \alpha_B(t_v)$. Do a case distinction:
C. Proof of the Cutoff Theorem

(a) \( t^b \neq t_b \). Note that if \( t^b > t_b \), we conclude \( b_{21} > 0 \), and if \( t^b < t_b \), we conclude \( b_{32} > 0 \). Let \( x \) be 1 if \( t^{1-b} < t_{1-b} \), and 0 otherwise. Let:

- \( s_{11} = \alpha_B(t^{1-b}) + x \)
- \( s_{33} = \alpha_B(t_{\perp}) \)
- \( sr_2 = \alpha_B(t^b) + 1 \)
- \( sr_3 = \alpha_B(t_{\perp}) + (1-x) - s_{33} \)
- \( sc_1 = \alpha_B(t_{1-b}) - s_{11} \)
- \( sc_2 = \alpha_B(t_b) \)

and invoke Lemma C.17 to yield \( s_{21} \) to \( s_{32} \).

(b) \( t^b = t_b \). From our assumptions small-sum(1) and \( \neg \)small-sum(2) and the fact \( t_{\perp} \leq t^x \) we conclude that \( t^{1-b} < t_{1-b} \).

From this, we conclude either \( b_{21} > 0 \) or \( b_{31} > 0 \). Let \( x = 0 \) if \( b_{21} = 0 \) and 1 otherwise. Set:

- \( s_{11} = \alpha_B(t^{1-b}) + 1 \)
- \( s_{33} = \alpha_B(t_{\perp}) \)
- \( gr_2 = \alpha_B(t^b) + x \)
- \( gr_3 = \alpha_B(t_{\perp}) + (1-x) - s_{33} \)
- \( gc_1 = \alpha_B(t_{1-b}) - s_{11} \)
- \( gc_2 = \alpha_B(t_b) \)

And invoke Lemma C.17.

3. small-sum(2), but not small-sum(1). Symmetric to the previous case, where the symmetry is on the antidiagonal of the count matrix: we swap up and down (as in \( t_X \) and \( t^X \)) and the roles of \( 1-b \) and \( \perp \).

It is readily seen that the resulting small matrix ensures that \( (\mathcal{U}_a, \mathcal{U}_c) \in \rho_{cnt} = (f_1, \{0, 1, \perp\}) \). To conclude that the formulas in \( \Lambda \) are also satisfied, we must show that whenever \( s_{ij} > 0 \), the corresponding tile can be used without violating any of the formulas in \( \Lambda \). For \( s_{21} \) to \( s_{32} \), this follows from the the guarantees of Lemma C.17 and Lemmas C.8 and C.9. For \( s_{11} \) and \( s_{33} \), whenever \( s_{11} = \alpha_B(t^{1-b}) \) and \( s_{33} = \alpha_B(t_{\perp}) \), this also follows Lemma C.8 and the shape of the count matrix of \( \mathcal{U}_c \). The only danger is when setting them to \( \alpha_B(t^{1-b}) + 1 \) (resp. \( \alpha_B(t_{\perp}) + 1 \)); but this only happens when small-sum(1) (resp. small-sum(2)) holds, which guarantees that all three values are present in the field \( f_1 \) (resp. \( f_2 \)), which guarantees the existence of the corresponding tiles in \( \mathcal{U}_c \).

This completes the proof for the case where an equal number of values appears in both \( f_1 \) and \( f_2 \); we now show how to reduce to it the case of different numbers of values in \( f_1 \) and \( f_2 \). We do this by relabeling a particular type of “victim” tile in \( \mathcal{U}_c \), either \( (\tilde{x} y) \) or \( (\tilde{y} z) \) for some \( x \) and \( y \), to \( (\tilde{z} \tilde{z}) \) such that for the resulting update \( \mathcal{U}_c^\prime \).
(U\textsuperscript{r}1) the relabeling ensures that the same number of values (either two or three) appears in each field. This will be achieved either by adding \( x \) to a field in which it did not previously appear, or by completely removing \( y \) from a field in which it did appear.

(U\textsuperscript{r}2) the tile \( \left( \frac{1}{x} \right) \) does not exist in \( \mathcal{U}_c \). This ensures that the relabeling is a bijection.

Note that \( \mathcal{U}_{c}^{r} \) is still lower triangular. We then produce the update \( \mathcal{U}_{c}^{r} \) from \( \mathcal{U}_{c}^{r} \) as if no relabeling was done, such that \( (\mathcal{U}_{a}^{r}, \mathcal{U}_{c}^{r}) \in \rho_{\text{cnt}} = (\{ f \}, \{0, 1, \bot\}) \), and thus in particular \( (\mathcal{U}_{a}^{r}, \mathcal{U}_{c}^{r}) \in \rho_{\text{cnt}} = (\{ f \}, \{0, 1, \bot\}) \). We claim that undoing the relabeling on \( \mathcal{U}_{c}^{r} \) then yields \( (\mathcal{U}_{a}, \mathcal{U}_{c}) \in \rho_{\text{cnt}} = (\{ f \}, \{0, 1, \bot\}) \). For this, we note that, from (U\textsuperscript{r}2) and the way \( \mathcal{U}_{c}^{r} \) is constructed, there exists an \( i \in \{1, 2\} \) such that any tile in \( \mathcal{U}_{c}^{r} \) that sets \( \{f_{i} \mapsto x\} \) also sets \( \{f_{1-i} \mapsto x\} \). The above method of constructing \( \mathcal{U}_{a}^{r} \) (in particular, Lemma C.17) then guarantees that the resulting \( \mathcal{U}_{a}^{r} \) preserves this property. Thus, the slot of the tile \( \left( \frac{1}{x} \right) \) will be the same in both \( \mathcal{U}_{c}^{r} \) and \( \mathcal{U}_{c}^{r} \); it is not difficult to see that this, together with \( (\mathcal{U}_{a}^{r}, \mathcal{U}_{c}^{r}) \in \rho_{\text{cnt}} = (\{ f \}, \{0, 1, \bot\}) \) and (CO6) and Lemma C.3, suffices for the claim. Moreover, the previous argument for the feasibility of \( \mathcal{U}_{c}^{r} \) from \( s_{a} \) translates to \( \mathcal{U}_{a} \) as well. We determine the relabeling as follows:

1. If a value \( v \in \{0, 1\} \) is missing in \( f_{2} \), then all three values are present in \( f_{1} \), and there must exist a tile \( \left( \frac{1}{1-v} \right) \) in \( \mathcal{U}_{c} \) (since a tile \( \left( \frac{1}{x} \right) \) is not possible by (ctr-ord)). We relabel all such tiles to \( \left( \frac{1}{v} \right) \). Note that a tile \( \left( \frac{1}{1-v} \right) \) must also exist (again since \( \left( \frac{1}{1-v} \right) \) is not possible due to (ctr-ord)), thus the relabeling yields three values in each field and ensures (U\textsuperscript{r}1).

Furthermore, (U\textsuperscript{r}2) holds since no tile \( \left( \frac{1}{v} \right) \) exists in \( \mathcal{U}_{c} \).

2. If \( \bot \) is missing in \( f_{2} \), then three values must be present in \( f_{1} \), and there must exist a tile \( \left( \frac{1}{v} \right) \) for some \( v \in \{0, 1\} \). Perform a further distinction:

(a) There are no \( \left( \frac{1}{v} \right) \) tiles for \( x \neq \bot \). Relabel \( \left( \frac{1}{v} \right) \) to \( \left( \frac{1}{v} \right) \) (this yields two values in each field and ensures (U\textsuperscript{r}1)). By assumption, \( \left( \frac{1}{v} \right) \) does not exist in \( \mathcal{U}_{c} \), ensuring (U\textsuperscript{r}2).

(b) A tile \( \left( \frac{1}{v} \right) \) exists, for \( x \neq \bot \). Relabel \( \left( \frac{1}{v} \right) \) to \( \left( \frac{1}{v} \right) \) (yielding three values in each field and ensuring (U\textsuperscript{r}1)). Since \( \bot \) is missing in \( f_{2} \), \( \left( \frac{1}{v} \right) \) does not exist in \( \mathcal{U}_{c} \), ensuring (U\textsuperscript{r}2).

3. If a value \( v \) is missing in \( f_{1} \), perform a further distinction:

(a) if there exists a tile \( \left( \frac{1}{v} \right) \), we relabel all such tiles to \( \left( \frac{1}{v} \right) \) (this yields three values in each field, since we know that \( \left( \frac{1}{v} \right) \) must exist, and thus ensures (U\textsuperscript{r}1)). The assumption that \( v \) is missing in \( f_{1} \) guarantees (U\textsuperscript{r}2).

(b) Otherwise, there must exist a tile \( \left( \frac{1-\mathcal{U}v}{v} \right) \).
C. Proof of the Cutoff Theorem

i. If no other tiles exist with \(1 - v\) on the top, we relabel such tiles to \((1 - v)\), ensuring \((U'2)\) (and since \((\bot)\) does not exist, this yields two values in each field, and ensures \((U'1)\)).

ii. Otherwise, we relabel \((1 - v)\) to \((v)\), yielding three values in each field and ensuring \((U'1)\). Since \(v\) is missing in \(f_1\), we also have \((U'2)\).

4. Note that by \((\text{ctr-ord})\) it is not possible for \(\bot\) to be missing in \(f_1\) (but not be missing in \(f_2\)).

Corollary C.1 (Leaders and update preservation). Assume the same conditions as Lemma C.18 hold, additionally allowing the possibility that \(cp = \geq 3\). Then, given any set \(C \subseteq F\), there exists a \(U_a\) such that:

\[
U_c(f)(ldr(s'_c)) = U_a(f)(ldr(s'_{a}))
\]

for all \(f \in U \cap C\), and

\[
(U_a, U_c) \in \rho_{\text{cnt}} = (\{f\}, \{0, 1, \bot\})
\]

for all \(f \in U \setminus C\).

Proof. A case distinction on \(cp\):

1. \(cp = \geq 3\). By Lemma C.8, \(ldr(s_a)\) can perform the same local update as \(ldr(s_c)\). All other processes perform the empty update in both concrete and abstract steps. Thus,

\[
(U_a, U_c) \in \rho_{\text{cnt}} = (\{f\}, \{0, 1, \bot\})
\]

for all \(f \in U\). We need to show

\[
U_c(f)(ldr(s'_c)) = U_a(f)(ldr(s'_{a}))
\]

for \(f \in C \cap U\). If \(ldr(s_c) = ldr(s'_c)\), this is immediate. Otherwise, \(U_c(f)(ldr(s'_c)) = \bot\), and we are necessarily at the end of the phase; we switch the leaders in the step \(s_a \rightarrow s'_a\) to ensure this.

2. \(cp \neq \geq 3\). Lemma C.18 applies. By property \((\text{PS1})\), \(|U| \in \{1, 2\}\). If \(U \cap C = \emptyset\), we simply use Lemma C.18 and we are done. If \(U \cap C = U\), we obtain the result from Lemma C.8. Otherwise, \(U = \{f_1, f_2\}\), and WLOG \(f_1 \in C\), while \(f_2 \notin C\). Use Lemma C.18 to obtain an update \(U'_a\); we have \(#[U'_a(f_1)][U_c(f_1)(ldr(s'_c))] > 0\) and there exists a local update in \(U'_a\) that sets \(f_1\) to \(U_c(f_1)(ldr(s'_c))\). To obtain \(U_a\), permute \(U'_a\) so that this local update is applied by \(ldr(s'_a)\).
C.3. The Lemmas

We will use the previous corollary for the majority of our proofs. However, in the presence of inp field updates, it will not be enough to simply mimic the distribution of values in the concrete. Conceptually, the reason is that our abstraction is relatively coarse. We give a concrete example here. Consider an algorithm with only a single threshold, $\frac{1}{2}$. Consider a concrete state $s_c$ of size 19, where $\#[s_c(\text{inp})](0) = 9$. Any related abstract state $s_a$ of size $2d + 1 = 5$ will have either $\#[s_a(\text{inp})](0) = 1$ or $\#[s_a(\text{inp})](0) = 2$. Consider the first case. Now, a concrete update $U_c$ where $\#[U_c(\text{inp})](-) = 10$ and $\#[U_c(\text{inp})](0) = 1$ can push $\#[s_c(\text{inp})](0)$ to 10. To preserve the simulation, we must perform an abstract update $U_a$ where $\#[U_a(\text{inp})](0) \geq 2$.

Now, if another field $f$ was simultaneously updated together with inp, we might have to sacrifice full $cntMS_\geq$ on $f$. The following lemma allows us to do that.

**Lemma C.19** (Tweaking simultaneous updates). Assume $s$ is an $B$-process state. Further, assume $s \xrightarrow{\Lambda} s_a$. Let $U$ be the set of updated fields in the step, let $f_1 \in U$, and denote the corresponding global update by $U$. For $X \in \mathcal{P}(\{0, 1, \perp\})$ let:

$$c_X = \sum_{x \in X} \#[U(f_1)](x)$$
$$t_X = \gamma_B(c_X)$$

where we will omit the brackets when writing the set $X$. If there exists a $f_2 \in U$, $f_2 \neq f_1$, we also write:

$$c_X = \sum_{x \in X} \#[U(f_2)](x)$$
$$t_X = \gamma_B(c_X)$$

Let $b \in \{0, 1\}$, and $b' = 1 - b$. Assume $c^b > 0$ and $c^\perp > 0$.

Then, there exist transitions $s \xrightarrow{\Lambda} s'_1$, $s \xrightarrow{\Lambda} s'_2$ and corresponding global updates $U_1, U_2$, where we write:

$$c_X^{i:i} = \sum_{x \in X} \#[U_i(f_1)](x)$$

and such that:

1. $c^{b_1} = c^{b_2} = \alpha_B(t^b) + 1$
2. For $d'' \in \{c^{v_1}, c^{v_2}\}$, either $d'^b = c^b'$ and $d'^\perp \geq c^\perp - 1$, or $d'^b \geq c^b' - 1$ and $d'^\perp = c^\perp$
C. Proof of the Cutoff Theorem

3. $c^{b,\perp;1} \geq c^{b,\perp}$ and $c^{b,\perp;2} \geq c^{b,\perp}$

4. If $f_2 \in U$, $(U_1, U) \in \rho_{val}(\{f_2\})$

5. If $f_2 \in U$, $(U_2, U) \in \rho_{cut}(\{f_2\}, \{\perp\}, \emptyset)$

Proof. Note that, since $c^{\perp} > 0$ and hence $c^b < B$, by property (CO3), we have either $c^b = \alpha_B(t^b)$ or $c^b = \alpha_B(t^b) + 1$. If $c^b = \alpha_B(t^b) + 1$, we take $U_1 = U_2 = U$. It is then easy to see that items 1-5 hold for this case.

Furthermore, if $U = \{f_1\}$, then since $c^b > 0$ and $c^{\perp} > 0$ we change $U$ by having one of the processes that used the empty local update switch to using the local update $\{f_1 \mapsto b\}$. It is again easy to see that all the requirements are satisfied.

In the rest, we thus assume $U = \{f_1, f_2\}$ and $c^b = \alpha_B(t^b)$. We will construct $U_1$ and $U_2$ by changing a single tile from $U$ (the tile might differ between $U_1$ and $U_2$). We will replace it by another tile that is either from $U$, or that can arise from some HO set that gave rise to the updates in $U$. As we do not introduce new HO sets, this cannot violate any of the formulas from $\Lambda$.

For $U_1$, let $w = b'$; for $U_2$, let $w = \perp$. If in $U$ there exist tiles $\left(\frac{b}{w}\right)$ and $\left(\frac{z}{w}\right)$ for some $z \neq b$, we simply replace one of $\left(\frac{z}{w}\right)$ by $\left(\frac{b}{w}\right)$. It is easy to see that this yields the required $U_1$ and $U_2$.

Thus, assume that no such two tiles exist. Then, fix $\alpha_B(t_w)$ tiles that yield $w$ in $f_2$; we will keep those tiles in the new updates $U_1$ and $U_2$, and change one of the remaining ones. We first show that, in these remaining tiles, there exists some tile $\left(\frac{z}{y}\right)$ with $x \neq b$. By the assumption that not both $\left(\frac{b}{w}\right)$ and $\left(\frac{z}{w}\right)$ tiles with $z \neq b$ exist, we have either:

- all the tiles that we fixed are $\left(\frac{b}{w}\right)$. In this case, since we also assume that $c^{\perp} > 0$, one of the remaining tiles must be some $\left(\frac{z}{y}\right)$, or

- none of the tiles that we fixed are $\left(\frac{b}{w}\right)$. Then by (CO11), we have $\alpha_B(t_w) + c^b = \alpha_B(t_w) + \alpha_B(c^b) < B$. Hence, there is a remaining tile $\left(\frac{z}{y}\right)$ with $x \neq b$.

We proceed to replace the victim tile $\left(\frac{z}{y}\right)$ with $x \neq b$ by:

- $\left(\frac{b}{y}\right)$, if $c_b > 0$. This tile is possible from the HO sets that give rise to $U$, since also $c^b > 0$, and since by (R2) the guards are totally ordered (so either the process that set $f_1 \mapsto b$ could also have set $f_2 \mapsto b$ with the same HO set, or the process that set $f_2 \mapsto b$ could also have set $f_1 \mapsto b$).

- any existing tile $\left(\frac{b}{y}\right)$ otherwise; we know that such a tile exists, since $c^b > 0$. 

178
It is easy to see that the additional update \( f_1 \mapsto b \) in the replacement tiles ensures points 1-3 for both \( U_1 \) and \( U_2 \). Regarding points 4 and 5, observe that, even if we have \( y = w \) and therefore replace one value \( w \), the slot of \( w \)'s in \( f_2 \) remains the same, since \( \gamma_B(\alpha_B(t_w)) = t_w = \gamma_B(c_w) \). This suffices for point 5 and for the \( cnt_{\Sigma_\geq} \)-part of point 4. It remains to establish the \( cnt_{\Sigma_\geq} \)-part of point 4. For the case of \( c_b > 0 \) and the replacement tile \( (b)^{b_0}_{b} \), this holds, since \( c_{b,b';1} \leq \#[U_2(f_2)](b) + \#[U_2(f_2)](b') \). For the case of \( c_b = 0 \) and the replacement tile \( (b)^{b_0}_{b} \), the \( cnt_{\Sigma_\geq} \) part is already covered by the \( cnt_{\Sigma_\geq} \) part.

\[
\text{C.3.5 Simulation: the inductive step}
\]

**Lemma C.20** (All phase updates originate from \( \text{inp} \)). Assume that some process updates some field to the value \( b \in \{0,1\} \) in the step \( s \rightarrow s' \) of a trace \( \tau \). Let \( r = \text{rand}(s) \), i.e., \( \tau(r) = s \), and let \( r_0 \) be the starting round of phase \( \Phi(r) \) and assume that the \( \text{inp} \) field is not updated in any round \( r' \) such that \( r_0 < r' \leq r \). Then \( \#(s(\text{inp}))(b) > 0 \).

**Proof.** Let \( sf \) be the send-field of the step \( s \rightarrow s' \). Note that, by the restrictions on our language, the \( \text{inp} \) field is sent out in round \( r_0 \). We prove the claim by generalized induction on the difference \( r - r_0 \). If this difference is 0, then \( sf = \text{inp} \), and \( b \) must have been selected by one of the guards; thus, it must also appear in \( s(\text{inp}) \). For the inductive case, \( r - r_0 > 0 \), note that \( sf \neq \text{inp} \), since \( \text{inp} \) is only sent once per phase by restriction (R1). Hence, \( sf \) must be an ephemeral field, which some process must have updated to \( b \) in a previous round \( r_1 \) of the phase (i.e., \( r_0 < r_1 < r \)). Since \( \text{inp} \) is not updated in phase between rounds \( r_0 \) and \( r \) by assumption, we have \( s(\text{inp}) = \tau(r_1)(\text{inp}) \) and we obtain the desired conclusion by the induction hypothesis.

**Lemma C.21** (Normal input update simulation). Suppose the protocol is either standard, or it is randomized and \( \lambda \notin \Lambda \) (where \( \lambda \) is the randomness label). Consider a step \( s_c \xrightarrow{\Delta_s} s'_{c} \) with send-field \( sf \), set of updated fields \( U \), and corresponding global update \( U_c \). Assume that \( \text{inp} \in U \) and \( (s_a,s_c) \in \text{randrel} \).

Further assume:

- if \( cp \leq \Lambda_s \), then \( (s_a,s_c) \in \text{ldrrel}(\{sf\}) \cap \rho_{\text{full}}(\{\text{inp}\}) \)
- if \( cp \neq \Lambda_s \), then \( (s_a,s_c) \in \rho_{\text{full}}(\{\text{inp},sf\}) \).

Then, given any distinguished process \( p_\perp \) in \( s_a \), there exist states \( s'_a \) and \( s''_a \) such that \( s_a \xrightarrow{\Lambda} s'_a \) and \( s_a \xrightarrow{\Lambda} s''_a \) and:

1. \( (s'_a,s''_c) \in \rho_{\text{full}}(\{\text{inp}\}) \) and \( (s'_a,s''_c) \in \rho_{\text{full}}(\{\text{inp}\}) \)
2. if there is an ephemeral \( f \in U \) then \( (s'_a,s''_c) \in \rho_{\text{val}}(\{f\}) \).
3. if there is an ephemeral \( f \in U \) then \( (s''_a,s''_c) \in \rho_{\text{cur}}(\{f\},\{\perp\},\emptyset) \)
C. Proof of the Cutoff Theorem

4. \((s'_a, s'_c) \in \ldrrel(F)\) and \((s''_a, s''_c) \in \ldrrel(F)\), where \(F\) is defined as \(\text{next-ldrrel}(s'_c) \cap U\).

5. if \(\text{dec} \in U\), then for the global update \(U'_a\) associated with the step \(s_a \xrightarrow{\lambda} s'_a\), we have \((U'_a, U_c) \in \rho_{\text{out}}(\{\text{dec}\})\).

6. if \(\text{dec} \in U\) and \(#[U_c(\text{dec})](\perp) > 0\), then for the global update \(U''_a\) associated with \(s_a \xrightarrow{\lambda} s''_a\), we have \(U''_a(\text{dec})(p_{\perp}) = \perp\).

Proof. We first sketch the main idea of the proof for which we focus on items 1–3. We will use Lemma C.18, and additionally Lemma C.19 if needed, to obtain abstract updates \(U_1\) and \(U_2\) that preserve all the formulas in \(\Lambda\) and whose value counts are similar (in terms of the desired \(\rho\) relations) to the counts in \(U_c\). For the ephemeral field \(f \in U\) (if it exists), by the semantics of \(\text{next}\) (see Section 6.1.4), the counts of all values in \(s'_a\) (resp. \(s''_a\)) will come directly from \(U_1(f)\) (resp. \(U_2(f)\)), ensuring item 2 (resp. item 3).

The main difficulty comes from the semantics of fallback values for the \(\text{inp}\) field. The value counts in \(s'_a(\text{inp})\) (and \(s''_a(\text{inp})\)) depend not only on the update, but potentially also on the old values in \(s_a(\text{inp})\), again by the semantics of \(\text{next}\). Each permutation of the global update (where the local updates of two processes are exchanged) can thus yield different value counts in \(s'_a(\text{inp})\) (or \(s''_a(\text{inp})\)). We will thus have to carefully permute \(U_1\) (resp. \(U_2\)) using Lemma A.1 in order to create the global update \(U'\) (resp. \(U''\)) that will yield the state \(s'_a\) (resp. \(s''_a\)) respecting item 1. Moreover, since the counts of values in \(f\) depend only on the counts in the update, if an update ensures item 2 (resp. items 3 or 5), then so does any permutation of that update.

We now begin the detailed proof. We treat item 4 upfront, since it is almost trivial, due to our language restrictions. In particular, from (R7), if there exists an \(f \in U \setminus \{\text{inp}\}\), we have that \(f \notin \text{next-ldrrel}(s'_c)\). So the only non-trivial case is that \(\text{inp} \in \text{next-ldrrel}(s'_c)\). But then, since by restriction (R1) \(\text{inp}\) is sent out only once per phase, \(s'_c\) must belong to a new phase. Once we have determined the successor states \(s'_a\) and \(s''_a\) (at the end of our construction), we are thus free to change the leader in \(s'_a\) and \(s''_a\) as we see fit; item 1 of the lemma guarantees that \(s'_a(\text{inp})(\ldr(s'_c))\) exists in both \(s'_a\) and \(s''_a\). Hence, we can choose \(\ldr(s'_a)\) and \(\ldr(s''_a)\) such that item 4 holds.

Next, we introduce some notation. For \(X \in \mathcal{P}(\{0, 1, \perp\})\), let:

\[
\begin{align*}
s_X &= \gamma_k(\sum_{x \in X} \#s_c(\text{inp})(x)) \\
s'_X &= \gamma_k(\sum_{x \in X} \#s'_c(\text{inp})(x)) \\
p_X &= \gamma_k(\sum_{x \in X} \#U_c(\text{inp})(x))
\end{align*}
\]
where we omit the brackets when writing the set \( X \). We note several facts. For all \( b \in \{0, 1\} \), necessarily:

\[
\begin{align*}
p_b &\leq s'_b \quad (p_b) \\
\#[s'_c(inp)](b) &\leq \#[U_c(inp)](b) + \#[U_c(inp)](\bot) \quad (s'b2)
\end{align*}
\]

Also, when the protocol is not randomized, we have:

\[
\#[s'_c(inp)](b) \leq \#[s_c(inp)](b) + \#([U_c(inp)](\bot)) \quad (s'b1)
\]

Lastly, from \((s_a, s_c) \in \rho_{full}(\{inp\})\) and the fact that \( inp \) never contains \( \bot \), by Lemma C.2, whenever both 0 and 1 appear in \( s_c(inp) \), we get:

\[
\gamma_B(\#[s_a(inp)](b)) = s_b. \quad (s_a \gamma b)
\]

Note that, by (R6) and \( inp \in U \), the step’s communication pattern is not \( \geq \). Hence, we can start our construction by applying Lemma C.18 to obtain an update \( U \) satisfying

\[
(U, U_c) \in \rho_{cnt} = (\{inp\}, \{0, 1, \bot\}) \quad (inp-cnt)
\]

and also

\[
(U, U_c) \in \rho_{cnt} = (\{f\}, \{0, 1, \bot\}) \quad (f-full)
\]

if there is an \( f \in U \setminus \{inp\} \). Note that, by (inp-cnt), for all \( X \subset \{0, 1, \bot\} \):

\[
\gamma_B(\sum_{x \in X} \#[U(inp)](x)) = p_X \quad (U \gamma)
\]

If \( \#[U_c(inp)](\bot) = 0 \), then the values of the \( s'_c(inp) \) do not depend on the values of \( s_c(inp) \), i.e., an update fully determines the successor state. If item 6 does not apply (i.e., \( dec \notin U \) or \( \#[U_c(dec)](\bot) = 0 \)), we directly set \( U'_a = U''_a = U \) and apply this update to the state \( s_a \) to obtain \( s'_a = s''_a \). Items 1–3 and item 5 then easily follow from (inp-cnt) and (f-full). If item 6 applies, then there is a process \( q \) such that \( U_c(dec)(q) = \bot \) and we swap the local updates of \( q \) and \( p_\bot \) in \( U \) to obtain \( U' \). We then set \( U'_a = U''_a = U' \) and applying this update to the state \( s_a \) yields \( s'_a = s''_a \). Item 6 holds by construction. Note that (inp-cnt) and (f-full) still hold for \( U' \) instead of \( U \) and therefore items 1–3 and item 5 also hold in this case (items 2 and 3 trivially so).

For the rest, we thus assume \( \#[U_c(inp)](\bot) > 0 \). We note that, in case \( f \in U \setminus \{inp\} \) exists and \( \#[U_c(f)](\bot) > 0 \), by (R2), we can invoke Lemma C.9 (item (1)) and conclude that a tile \( (\bot) \) existed in \( U_c \). We proceed by a case distinction:

- Case 1: there exists a \( b \) such that \( p_b \geq 0 \) and \( s'_b > s_b + p_b \). We conclude:
  - \( s_b \geq 0 \) by Lemma C.20, since \( p_b \geq 0 \).
also $s_{1-b} \geq 0$, since $s_b < s'_b$. Thus $(s_a\gamma b)$ holds.
- $\#[\mathcal{U}(\text{inp})](b) < B$, since $s'_b > p_b$ and using (inp-cnt).
- From $s'_b > s_b + p_b$, if the protocol is not randomized, we have:
  $$s'_b = s_b + p_b + \frac{1}{d}$$
  \text{(s'_b-eq)}

by (s'\text{b1}), monotonicity of $\gamma$, and $(\gamma+2)$.

From (U$\gamma$) and (CO3) we have $\#[\mathcal{U}(\text{inp})](b) \in \{\alpha_B(p_b), \alpha_B(p_b) + 1\}$. We distinguish two cases. (i) If $\#[\mathcal{U}(\text{inp})](b) = \alpha_B(p_b) + 1$, we take $\mathcal{U}_1 = \mathcal{U}_2 = \mathcal{U}$. (ii) Otherwise, $\#[\mathcal{U}(\text{inp})](b) = \alpha_B(p_b)$ and we feed $\mathcal{U}$ into Lemma C.19 to obtain updates $\mathcal{U}_1$ and $\mathcal{U}_2$ such that $\#[\mathcal{U}_i(\text{inp})](b) = \alpha_B(p_b) + 1$. In either case, the following holds (for $i = 1, 2$):

$$(\mathcal{U}_i, \mathcal{U}) \in \rho_{\text{cnt}}(\{\text{inp}\}, \{b\}, \{b, \bot\})$$
\text{(inp-b,⊥)}
\#[\mathcal{U}_i(\text{inp})](b) = \alpha_B(p_b) + 1$$
\text{(inp-p_b+1)}
(\mathcal{U}_1, \mathcal{U}) \in \rho_{\text{cnt}}(\{f\}, \{0, 1\}, \{0, 1\})$$
\text{(f-0,1)}
(\mathcal{U}_2, \mathcal{U}) \in \rho_{\text{cnt}}(\{f\}, \{\bot\}, \emptyset)$$
\text{(f-⊥)}

From (inp-p$_b$+1) we also have:

$$\gamma_B(\#[\mathcal{U}_i(\text{inp})](b)) = p_b$$
\text{(U_i$\gamma$b)}

We now permute $\mathcal{U}_1$ (respectively $\mathcal{U}_2$) to yield updates $\mathcal{U}'$ and $\mathcal{U}''$ that will give rise to our desired states $s'_a$ and $s''_a$ when applied to $s_a$. By the previous discussion, equations (f-full), (f-0,1), and (f-⊥) guarantee items 2, 3, and 5 of the current lemma respectively, if $\mathcal{U}_1$ and $\mathcal{U}_2$ are applied to $s_a$; the guarantees remain intact if we apply any permutation of $\mathcal{U}_1$ and $\mathcal{U}_2$. We choose $\mathcal{U}'$ and $\mathcal{U}''$ to also satisfy items 1 and 6 as follows:

1. Order the set of processes (by labeling them from 1 to $B$), such that all the processes holding $b$ in $s_a(\text{inp})$ appear on the left, i.e., such that there exists a $j$, $1 < j \leq B$, such that for all the processes $p$ labeled by $j$ or less, $s_a(\text{inp}, p) = b$, and for all other processes $q$, $s_a(\text{inp}, q) = 1 - b$. If item 6 applies, choose the ordering such that the distinguished process $p_\bot$ is labeled by either 1 or $B$.

2. Take $l = \alpha_B(s'_b)$ (note that $1 \neq l \neq B$). We thus also have $\gamma_B(l) = s'_b$. Label the set of all the processes to the left of and including the $l$-th process by $L$, and the set of the other processes by $R$. From $s'_b > s_b$ and $(s_a\gamma b)$, we know that all the processes in $R$ hold the value $1 - b$ in $s_a(\text{inp})$, i.e., $l \geq j = \#[s_a(\text{inp})](b)$. 

182
Thus, steps 3 and 4 guarantee that for all
\[ \forall b, \gamma s((\#[\mathcal{U}_i](\text{inp}))(b) = \gamma a''). \]
This follows from \( p_b \leq s_b' \) by applying the (strict) monotonicity of \( \alpha_B \), from \((\text{inp}-p_b+1)\) and our choice of \( l \).

4. For the remaining processes in \( L \), we apply local updates that set \( \text{inp} \mapsto \bot \) to these processes. To see that we have enough such updates, observe that:
\[
\sum_{x \in \{b, \bot\}} \#[\mathcal{U}_i](\text{inp}))(x) \geq \alpha_B(p_b, \bot) \text{ by (inp-b, \bot) and } (\mathcal{U}\gamma)
\]
\[
\geq \alpha_B(s'_b) \text{ by (s'2b)}
\]
and monotonicity of \( \alpha_B, \gamma_k \)
\[
= l
\]
and note that we started from the process labeled by \( l \). Moreover, observe that if item 6 applies and \( p_\bot \) was labeled by 1, we can ensure that we use the empty local update \((\dagger)\) for \( p_\bot \) and thus guarantee item 6.

Next, if the protocol is randomized, use \( b \) as the random value for all the remaining processes in \( L \). If the protocol is not randomized, note that after step 3, by \((\text{inp}-p_b+1)\) exactly \( m = l - (\alpha_B(p_b) + 1) \) processes remain in \( L \); by \((s'_b\text{-eq})\) and \((\Lambda\text{13})\) we have \( l = \alpha_B(s'_b) = \alpha_B(s_b) + \alpha_B(p_b) + 1 \), and thus \( m = \alpha_B(s_b) \). Since \( \alpha_B(s_b) \leq \#[s_a(\text{inp})](b) \) (by \((s_a \gamma b)\)) our ordering guarantees that, for all of these processes, \( s_a(\text{inp}) \) is \( b \). Thus, applying \( \text{inp} \mapsto \bot \) updates to them yields the value \( b \).

5. As remarked earlier, all of the processes in \( R \) have their \( \text{inp} \) set to \( 1-b \) in \( s_a \). Since we have used up all the updates that set \( \text{inp} \mapsto b \) in step 3, the remaining local updates in both \( \mathcal{U}_1 \) and \( \mathcal{U}_2 \) set either \( \text{inp} \mapsto \bot \) or \( \text{inp} \mapsto (1-b) \). If item 6 applies and \( p_\bot \) was labeled by \( B \), use the empty local update \((\dagger)\) for \( p_\bot \), possibly changing \( \mathcal{U}_2 \) (without violating any of the predicates in \( \Lambda \)); note that, if we are doing this, then items 2 and 3 do not apply. This ensures item 6. Otherwise, arrange the remaining local updates arbitrarily among the remaining processes; if the protocol is randomized, use \( 1-b \) as the random value for these processes. In any case, all processes in \( R \) will hold the value \( 1-b \) after the update.

Thus, steps 3 and 4 guarantee that for all \( \forall p \in L \). \( s'_a(\text{inp})(p) = s''_a(\text{inp})(p) = b \), and step 5 guarantees that \( \forall q \in R \). \( s'_a(\text{inp})(q) = s''_a(\text{inp})(q) = 1-b \). Since \( \gamma_B(l) = s'_b \), we conclude \( \gamma_B(\#[s'_a(\text{inp})](b)) = \gamma_B(\#[s''_a(\text{inp})](b)) = s'_b \) and, using Lemma C.2, also \( \gamma_B(\#[s''_a(\text{inp})](1-b)) = \gamma_B(\#[s''_a(\text{inp})](1-b)) = s'_{1-b} \). This establishes item 1.
C. Proof of the Cutoff Theorem

- Case 2: otherwise, for all $b \in \{0, 1\}$, $s'_b \leq s_b + p_b$ or $p_b < 0$. We note that, if $p_b < 0$ and the protocol is not randomized then

$$s'_b \leq s_b \quad (s'_b\text{-shrink})$$

Note that, by (PS1) and since $\text{inp} \in U$, at least one of the items 3 and 6 is trivial. If the latter is trivial (i.e., $\text{dec} \notin U$), we will use the same $U'$ for both $s'_a$ and $s''_a$ (that is, $s''_a = s'_a$), that will be a permutation of $U$. As in Case 1, items 2 and 3 then hold by (I-full) and the initial discussion, and item 5 is also trivial. If item 6 is not trivial (i.e., $\text{dec} \in U$), then item 3 is; we still use a permutation of $U$ as $U'$, which again ensures item 5 and (the now trivial) item 2.

1. Pick a $b \in \{0, 1\}$ such that $s'_b \geq 0$. As before, order the processes such that all the ones holding $b$ are on the left. Furthermore, if item 6 applies, choose the ordering such that $p_{\perp}$ is labeled by 1 or $B$. Let

$$i = \max(\alpha_B(s'_b), \#[U(\text{inp})](b)).$$

We note that by $(U\gamma)$, $\gamma_B(i) = s'_b$. Denote the set of processes to the left and including $i$ by $L$, and the remaining processes by $R$. We have $\gamma_B(|L|) = s'_b$ and $\gamma_B(|R|) = s'_{1-b}$.

2. Starting from the $i$-th process and going left, apply all the updates that set $\text{inp} \mapsto b$ from $U$. This is possible, since $\#[U(\text{inp})](b) \leq i$ by our choice of $i$. If we reach $p_{\perp}$ in this way (hence $i = \#[U(\text{inp})](b)$, and either $p_{\perp}$ was labeled by 1 or $i = B > \alpha_B(s'_b)$ and $p_{\perp}$ was labeled by $B$), and item 6 is not trivial, change $U$ to replace one the updates that set $\text{inp} \mapsto b$ to the empty local update (which, as noted before, was guaranteed to already exist in $U$).

3. Starting from the $i + 1$-th process and going right, apply all the $\text{inp} \mapsto (1-b)$ updates from $U$. We show that this is possible, i.e., we have $\#[U(\text{inp})](1-b) \leq B - i$. If $i = \#[U(\text{inp})](b)$, the claim follows since $\#[U(\text{inp})](b) + \#[U(\text{inp})](1-b) \leq B$. Otherwise, $i = \alpha_B(s'_b)$, and additionally, we conclude $\#[U(\text{inp})](1-b) < B$. Then:

$$\#[U(\text{inp})](1-b) \leq \alpha_B(p_{1-b}) + 1 \quad \text{by (U\gamma) and (CO10)}$$

$$\leq \alpha_B(s'_{1-b}) + 1 \quad \text{by (p_b), monotonicity of } \alpha_B$$

$$= B - \alpha_B(s'_b) \quad \text{by (CO6) and (CO8)}$$

Analogous to the previous case, if we reach $p_{\perp}$ in this way, and we need to show item 6, change $U$ to replace one the updates that set $\text{inp} \mapsto (1-b)$ to the empty local update (which, as noted before, was guaranteed to already exist in $U$).
4. Apply the remaining updates for the remaining processes arbitrarily. If item 6 applies, and we have not yet assigned a local update to \( p_\bot \), assign the empty local update to it.

Hence, we conclude that the updates are distributed as follows: the processes in \( L \) apply local updates of either \( \text{inp} \mapsto b \) or \( \text{inp} \mapsto \bot \), starting with the former at \( i \) (leftwards), and the processes in \( R \) apply local updates of either \( \text{inp} \mapsto (1 - b) \) or \( \text{inp} \mapsto \bot \), again starting with the former at \( i + 1 \) (rightwards).

We show that item 1 of the lemma is satisfied. Note that the part regarding \( \text{cntMS} \sum \geq \) holds trivially for \( \text{inp} \), so we can focus on \( \text{cntMS} \geq \).

If the protocol is randomized, we can have the processes in \( L \) choose \( b \) as their randomness, and have the processes in \( R \) choose \( 1 - b \). The result then follows by the previous remarks about the cardinality of \( R \) and \( L \).

We next treat the case where the protocol is not randomized. If \( \#[U(\text{inp})] = B \), the result is immediate from Step 2 (nothing is done in steps 3 and 4). Otherwise, since then for all \( x \in \{0, 1\} \) either \( s'_x \leq s_x + p_x \) or \( (s'_x \text{-shrink}) \) applies, by \((\alpha + 1)\) and monotonicity of \( \alpha_B \) we have that \( \alpha_B(s'_x) \leq \alpha_B(s_x) + \alpha_B(p_x) \). The result thus follows since in both of \( L \) and \( R \) we first apply the \( \text{inp} \mapsto x \) updates and then \( \text{inp} \mapsto \bot \) updates, and since the processes already holding \( x \) are updated at the end.

\[ \square \]

Lemma C.22 (Randomized input update simulation). Suppose the protocol is randomized and \( \lambda \in \Lambda \) (where \( \lambda \) is the randomness label). Consider a step \( s_c \xrightarrow{\Lambda} s'_c \) with send-field \( sf \), set of updated fields \( U \), and corresponding global update \( U_c \). Assume that \( \text{inp} \in U \) and \( (s_a, s_c) \in \text{rndrel} \). Further assume:

- if \( cp = \emptyset \), then \( (s_a, s_c) \in \text{ldrrel}(\{sf\}) \)
- if \( cp \neq \emptyset \), then \( (s_a, s_c) \in \text{rho}(\{sf\}) \).

Then, given any distinguished process \( p_\bot \) in \( s_a \), there exist states \( s'_a \) and \( s''_a \) such that \( s_a \xrightarrow{\Lambda} s'_a \) and \( s_a \xrightarrow{\Lambda} s''_a \) and:

1. \((s'_a, s'_c) \in \text{rho}(\{\text{inp}\}) \) and \((s''_a, s'_c) \in \text{rho}(\{\text{inp}\}) \)

2. if there is an ephemeral \( f \in U \) then \((s'_a, s'_c) \in \text{rho}(\{f\}) \).

3. \((s'_a, s'_c) \in \text{ldrrel}(F) \) and \((s''_a, s'_c) \in \text{ldrrel}(F) \), where \( F = \text{next-ldrf}(s'_c) \cap U \)

4. if \( \text{dec} \in U \), then for the global update \( U'_a \) associated to the step \( s_a \xrightarrow{\Lambda} s'_a \), we have \((U'_a, U_c) \in \text{rho}(\{\text{dec}\}) \).
C. Proof of the Cutoff Theorem

5. if dec $\in U$ and $\#[\mathcal{U}_c(dec)](\perp) > 0$, then for the global update $\mathcal{U}_a''$ associated to $s_a \xrightarrow{\lambda} s_a''$, we have $\mathcal{U}_a''(dec)(p_{\perp}) = \perp$.

Proof. Let $b = Lt(\gamma_k(\#[\mathcal{U}_c(inp)](0)), \gamma_k(\#[\mathcal{U}_c(inp)](1)))$. Since $\lambda \in \Lambda$, all the processes in $s_c'$ that use the local updates that set $\text{inp} \mapsto \perp$ will set $\text{inp}$ to $b$. Effectively, this turns $\text{inp}$ into an ephemeral field, since the old values of $\text{inp}$ do not matter anymore. Apply Lemma C.18 to yield an update $\mathcal{U}_a$ such that $(\mathcal{U}_a, \mathcal{U}_c) \in \rho_{\text{cnt}}(U, \{0, 1, \perp\})$. We then set $\mathcal{U}_a' = \mathcal{U}_a$ and obtain $\mathcal{U}_a''$ by permuting $\mathcal{U}_a$ to satisfy item 5. For these updates $\mathcal{U}_a'$ and $\mathcal{U}_a''$, items 1, 2, and 4 are satisfied. Finally, we establish item 3 by the same reasoning as in the proof of Lemma C.21, item 4.

Lemma C.23 (Timestamped input update simulation). Suppose that the protocol is timestamped. Consider a step $s_c \xrightarrow{\lambda} s_c'$ with send-field $sf$, set of updated fields $U$, and corresponding global update $\mathcal{U}_c$. Assume that $\text{inp} \in U$ and $(s_a, s_c) \in \rho_{ts} \cap \text{rndrel}$. Further, if $sf \neq \text{inp}$, assume:

- if $cp = s_c$, then $(s_a, s_c) \in \text{ldrrel}(\{sf\})$
- if $cp \neq s_c$, then $(s_a, s_c) \in \rho_{full}(\{sf\})$.

Then, given any distinguished process $p_{\perp}$ in $s_a$, there exists a state $s'_a$ and $s''_a$ such that $s_a \xrightarrow{\Lambda} s'_a$ and $s_a \xrightarrow{\Lambda} s''_a$ and:

1. $(s'_a, s'_c) \in \rho_{ts}$ and $(s''_a, s''_c) \in \rho_{ts}$.

2. if there is an ephemeral $f \in U$ then $(s'_a, s'_c) \in \rho_{full}(\{f\})$.

3. $(s'_a, s'_c) \in \text{ldrrel}(F)$ and $(s''_a, s''_c) \in \text{ldrrel}(F)$, where $F$ is defined as $\text{next-ldrf}(s'_c) \cap U$.

4. if $\text{dec} \in U$, then for the global update $\mathcal{U}_a$ associated to the step $s_a \xrightarrow{\lambda} s'_a$, we have $(\mathcal{U}_a', \mathcal{U}_c') \in \rho_{val}(\{\text{dec}\})$.

5. if $\text{dec} \in U$ and $\#[\mathcal{U}_c(\text{dec})](\perp) > 0$, then for the global update $\mathcal{U}_a''$ associated to $s_a \xrightarrow{\Lambda} s''_a$, we have $\mathcal{U}_a''(\text{dec})(p_{\perp}) = \perp$.

Proof. Similar to (but simpler than) the proof of Lemma C.21; many of the same remarks apply. Item 3 is trivial; by restrictions (R7), (R5) and (R8), and the definition of $\text{next-ldrf}$, we get $\text{next-ldrf}(s'_c) \cap U = \emptyset$. We thus do not consider it further.

Note that since $\text{inp} \in U$, we have $cp \not\geq s_c$. Therefore, we can apply Lemma C.18 to yield a global update $\mathcal{U}_a$ with $(\mathcal{U}_a, \mathcal{U}_c) \in \rho_{\text{cnt}}(U, \{0, 1, \perp\})$. The update $\mathcal{U}_a'$ will be a permutation of $\mathcal{U}_a$. This clearly ensures items 2 and 4. The update $\mathcal{U}_a''$ will also nearly be a permutation of $\mathcal{U}_a$, but perhaps slightly modified to ensure item 5. In the following, we use the abbreviations
\[ c_v = \#[U_a(inp)](v) \] and \[ a_v = \#[U_a(inp)](v) \). We also use primed versions of \( a_v \), e.g., \( a'_v = \#[U'_a(inp)](v) \).

Unlike in the proof of Lemma C.21, there now exist several cases where any permutation of \( U_a \) suffices for item 1:

- either \( c_\perp = 0 \), or both \( c_0 > 0 \) and \( c_1 > 0 \). Let \( C \in \mathcal{C}_{ts} \) be arbitrary. Then:

\[
\begin{align*}
    s'_c \models C(0) &\iff c_0 > 0 \\
    &\iff a_0 > 0 \\
    &\implies s'_a \models C(0)
\end{align*}
\]

and

\[
\begin{align*}
    s'_c \models C(1) &\iff c_1 + c_\perp > th(C) \cdot k \\
    &\iff \gamma_k(c_1 + c_\perp) \geq th(C) \quad \text{by def. } \alpha \text{ and } \gamma \\
    &\iff \gamma_B(a_1 + a_\perp) \geq th(C) \quad \text{by (CO6)} \\
    &\iff a_1 + a_\perp > th(C) \cdot B \quad \text{by def. } \alpha \text{ and } \gamma \\
    &\implies s'_a \models C(1)
\end{align*}
\]

- \( c_\perp = k \), and hence, \( a_\perp = B \). Again, let \( C \in \mathcal{C}_{ts} \) be arbitrary. Since the protocol is timestamped, by the semantics of next, the processes that perform the local update \( inp \mapsto \bot \) retain their old value of \( inp \). But then:

\[
\begin{align*}
    s'_c \models C(b) &\iff s_c \models C(b) \iff s_a \models C(b) \iff s'_a \models C(b)
\end{align*}
\]

Thus, in all these cases it suffices to set \( U'_a = U_a \) and choose \( U''_a \) to be a permutation of \( U_a \) that satisfies item 5, and we are done.

The only remaining case is when only \( b \) and \( \perp \) values exist in \( U_a(inp) \), for some \( b \in \{0, 1\} \), i.e., \( c_0 > 0 \), \( c_\perp > 0 \), and \( c_{1-b} = 0 \). We first show item 1 for \( s'_a \). We start with the more involved case for the value \( 1-b \). Let \( C \in \mathcal{C}_{ts} \) and suppose \( s'_c \models C(1-b) \). We can assume without loss of generality that \( C \) is the strongest such constraint (i.e., the one with the largest threshold). We have to construct an update \( U'_a \) and the associated state \( s'_a \) such that \( s'_a \models C(1-b) \). Assume \( s'_c, W_c \models C(1-b) \) for some set of processes \( W_c \). Since \( c_{1-b} = 0 \), we also have \( s_c, W_c \models C(1-b) \) and all the processes in \( W_c \) apply local updates that set \( inp \mapsto \perp \). Thus, \( \gamma_k(c_{\perp}) \geq \gamma_k(|W_c|) \geq th(C) \). We use Lemma C.4 to obtain a witness \( W'_a \) such that \( s_a, W'_a \models C(1-b) \). Next, we use Lemma C.5 to trim down \( W'_a \) to \( W_a \subseteq W'_a \) such that \( |W_a| = \alpha_B(th(C)) \) and we still have \( s_a, W_a \models C(1-b) \). Since \( \gamma_k(c_{\perp}) = \gamma_B(a_{\perp}) \), we also have \( a_{\perp} \geq |W_a| > th(C) \cdot B \). We construct \( U'_a \) by permuting \( U_a \) such that all the processes in \( W_a \) apply local updates that set \( inp \mapsto \perp \). This ensures \( s'_a \models C(1-b) \) (also with witness \( W_a \)). For the case of the value \( b \), let \( C \in \mathcal{C}_{ts} \)
and assume $s'_{a} \models C(b)$. Then from $c_{b} > 0$, we have $a_{b} > 0$ and thus also $a'_{b} > 0$. Therefore, we obtain $s'_{a} \models C(b)$. This establishes item 1 for $s'_{a}$.

Next, we construct $U''_{a}$ and the associated $s''_{a}$ such that both items 1 and 5 are satisfied. In case $\#(U_{a}(\text{dec})) (\perp) = 0$, we set $U''_{a} = U'_{a}$ and we are done. Hence, suppose $\#(U_{a}(\text{dec})) (\perp) > 0$. Since $c_{\perp} > 0$, we also have $a_{\perp} > 0 \text{ and } \#(U_{a}(\text{dec})) (\perp) > 0$. Since all guards are totally ordered by (R2), we conclude that a local update $(\perp)$ exists in $U_{a}$. For item 1 and the case of $1 - b$, we follow the same reasoning as above to construct the set of processes $W_{a}$ such that $|W_{a}| = \alpha_B(th(C))$, $a_{\perp} \geq |W_{a}| > th(C) \cdot B$, and $s_{a}, W_{a} \models C(1 - b)$ for the strongest constraint $C \in C_{ta}$. We construct $U''_{a}$ in two steps: (1) we permute $U_{a}$ such that all processes in $W_{a}$ apply local updates $\text{inp} \rightarrow \perp$, yielding $U'_{a}$ as above; (2a) If $p_{\perp} \in W_{a}$, we obtain $U''_{a}$ by further permuting $U'_{a}$ to ensure that $p_{\perp}$ applies the update $(\perp)$; (2b) If $p_{\perp} \notin W_{a}$, then since $|W_{a}| = \alpha_B(th(C)) < B - 1$, there exists at least one more process $p \neq p_{\perp}$ such that $p \notin W_{a}$. We construct $U''_{a}$ from $U'_{a}$ such that $p_{\perp}$ applies the update $(\perp)$ and $a'_{b} > 0$ holds. We do this as follows. We permute $U'_{a}$ such that $p$ updates $\text{inp} \rightarrow b$ and then we duplicate the update $(\perp)$ if necessary such that $p_{\perp}$ applies it (where $p_{\perp}$ uses the same HO set as the process from which we duplicate this update). In any case, the resulting update $U''_{a}(\text{dec}) = \perp$. Hence, item 5 holds. Applying $U''_{a}$ to $s_{a}$, we see that also $s''_{a} \models C(1 - b)$ (also with witness $W_{a}$), establishing item 1 for the value $1 - b$. Since $a'_{b} > 0$, it is easy to see that item 1 is also satisfied for the value $b$. □

Lemma C.24 ($\rho_{agr}$ simulation step). Assume $s_{c} \xrightarrow{\Delta} s'_{c}$ and $(s_{a}, s_{c}) \in \rho_{agr}$. Then there exists an $s'_{a}$ such that $(s'_{a}, s'_{c}) \in \rho_{agr}$ and $s_{a} \xrightarrow{\Lambda} s'_{a}$. Moreover, for the set of fields $U$ updated in the step and the global updates $U_{c}$ and $U_{a}$ corresponding to the steps, we have that $(U_{a}, U_{c}) \in \rho_{val}(\{\text{dec}\})$ whenever $\text{dec} \in U$.

Proof. By property (PS1), $|U| \leq 2$. Let $sf$ be the send-field of the step and $cp$ be the communication pattern. By restriction (R1), $sf$ is either $\text{inp}$ or an ephemeral field. If it is ephemeral, again by restriction (R1), it must have been updated in the current phase of $s_{c}$, and thus $sf \in \text{active-f}(s_{c})$. We perform a case distinction:

- $\text{inp} \in U$. Then, we conclude either:
  
  - $cp \xrightarrow{\Lambda}$ and $sf \in F_{1}(s_{c})$. By the definition of $\rho_{agr}$, then $(s_{a}, s_{c}) \in \text{ldrel}(\{sf\})$.
  
  - $cp \neq \xrightarrow{\Lambda}$ and either:
    
    * $sf \in F_{1}(s_{c})$. By the definition of $\rho_{agr}$, $(s_{a}, s_{c}) \in \rho_{fail}(\{sf\})$
    
    * $sf = \text{inp}$, and the protocol is not timestamped. By the definition of $\rho_{agr}$, then $(s_{a}, s_{c}) \in \rho_{fail}(\{sf\})$.

188
* \( sf = \text{inp} \), and the protocol is timestamped. By the definition of \( \rho_{agr} \), then \((s_a, s_c) \in \rho_{ts}\).

Furthermore, by the definition of \( \rho_{agr} \), if the protocol is not timestamped, then \((s_a, s_c) \in \rho_{val}(\{\text{inp}\})\), and otherwise \((s_a, s_c) \in \rho_{ts}\).

We apply:

- Lemma C.23 if the protocol is timestamped;
- Lemma C.22 if the protocol is randomized, and \( \lambda \in \Lambda \);
- Lemma C.21 otherwise.

We obtain a state \( s'_a \) such that \( s_a \xrightarrow{\lambda} s'_a \), such that all of the following hold:

- if the protocol is timestamped, then \((s'_a, s'_c) \in \rho_{ts}\);
- if the protocol is not timestamped, then \((s'_a, s'_c) \in \rho_{full}(\{\text{inp}\})\);
- if there exists an ephemeral field \( f \in U \setminus \{\text{inp}\} \), then \((s'_a, s'_c) \in \rho_{val}(\{f\})\);
- if \( \text{dec} \in U \), \((\mathcal{U}_a, \mathcal{U}_c) \in \rho_{val}(\{\text{dec}\})\).
- \((s'_a, s'_c) \in \text{ldrrel}(\text{next-ldrf}(s'_c))\)

Since by restriction \((\mathbf{R1}) \), \( \text{inp} \) is only updated once in each phase, for any \( f \in U \setminus \{\text{inp}\} \), we have \( f \notin \text{inp-path}(s'_c) \), so if \( f \in F_1(s'_c) \cup F_2(s'_c) \), at most \( f \in F_2(s'_c) \). Thus, we have \((s'_a, s'_c) \in \rho_{agr} \) as required.

- \( \text{inp} \notin U \) and \( U = \{f\} \) is a singleton set. The obligation \((s'_a, s'_c) \in \rho_{agr} \) is non-trivial only if \( f \) is ephemeral. If \( f = \text{dec} \), we only have to show \((\mathcal{U}_a, \mathcal{U}_c) \in \rho_{val}(\{\text{dec}\})\)

  - \( sf \in \text{inp-path}(s_c) \). This is similar to the previous case. By the definition of \( \rho_{agr} \) we again have \((s_a, s_c) \in \rho_{full}(\{sf\}) \) if the \( cp \) was not \( s'_c \), and \((s_a, s_c) \in \text{ldrrel}(\{sf\}) \) otherwise. Applying Corollary C.1 yields a global update \( \mathcal{U}_a \) such that \((\mathcal{U}_a, \mathcal{U}_c) \in \rho_{full}(\{f\}) \) if \( f \notin \text{next-ldrf}(s'_c) \) (which suffices for the case \( f = \text{dec} \)), and a related state \( s'_a \), such that \((s'_a, s'_c) \in \text{ldrrel}(\text{next-ldrf}(s'_c)) \). If \( f \) is ephemeral, we have \( s'_a(f) = \mathcal{U}_a(f) \), and thus \((s'_a, s'_c) \in \rho_{full}(\{f\}) \). This suffices to show \((s'_a, s'_c) \in \rho_{agr} \).

  - \( sf \in \text{dec-path}(s_c) \). Then, if \( cp \) was not \( s'_c \), we have \((s_a, s_c) \in \rho_{val}(\{sf\}) \). Otherwise, we have \((s_a, s_c) \in \text{ldrrel-val}(\{sf\}) \). Clearly, we also have \( f \in \text{dec-path}(s'_c) \) or \( f = \text{dec} \).

If \( f \notin \text{next-ldrf}(s'_c) \), we apply Lemma C.10 to obtain an update \( \mathcal{U}_a \) satisfying \((\mathcal{U}_a, \mathcal{U}_c) \in \rho_{val}(\{f\}) \). This also suffices in the case \( f = \text{dec} \). Otherwise, if \( f \in \text{next-ldrf}(s'_c) \), we use Lemma C.6 to yield the desired local update for \( \text{ldr}(s'_a) \). Thus, we have \((s'_a, s'_c) \in \rho_{agr} \).
Lemma C.25 \(\rho_{\text{term}}\) simulation step. Assume \(s_c' \xrightarrow{\Delta} s_c\) and \((S_a, s_c) \in \rho_{\text{term}}\). Then there exists an \(S'_a\) such that \((S'_a, s_c') \in \rho_{\text{term}}\) and, for all \(s_a' \in S'_a\), there exists an \(s_a \in S_a\) such that \(s_a' \xrightarrow{\Delta} s_a\).

Proof. Let \(sf\) be the send-field of the step, and let \(U\) be the set of fields updated in the step.

Perform a case distinction:

- \(\text{inp} \notin U\) and \(U\) is not a singleton set. Then, \(sf \in \text{inp-path}(s_c)\). We apply Corollary C.1 to obtain a global update \(\mathcal{U}_a\) such that \((\mathcal{U}_a, \mathcal{U}_c) \in \rho_{\text{full}}(U \setminus \text{next-ldr}(s_c'))\) and for the resulting state \(s'_a\) we have \((s'_a, s'_c) \in \text{ldrrel}(U \cap \text{next-ldr}(s'_c))\). Thus also \((s'_a, s'_c) \in \rho_{\text{full}}(\{\text{f}\})\) holds for every ephemeral field \(f \in U \setminus \text{next-ldr}(s'_c)\). This suffices to show \((s'_a, s'_c) \in \rho_{\text{agg}}\) as required. Obviously, if \(\text{dec} \in U\), we also have \((\mathcal{U}_a, \mathcal{U}_c) \in \rho_{\text{agg}}(\{\text{dec}\})\).

C. Proof of the Cutoff Theorem

- \(\text{inp} \notin U\) and \(U\) is not a singleton set. Then, \(sf \in \text{inp-path}(s_c)\). We apply Corollary C.1 to obtain a global update \(\mathcal{U}_a\) such that \((\mathcal{U}_a, \mathcal{U}_c) \in \rho_{\text{full}}(U \setminus \text{next-ldr}(s_c'))\) and for the resulting state \(s'_a\) we have \((s'_a, s'_c) \in \text{ldrrel}(U \cap \text{next-ldr}(s'_c))\). Thus also \((s'_a, s'_c) \in \rho_{\text{full}}(\{\text{f}\})\) holds for every ephemeral field \(f \in U \setminus \text{next-ldr}(s'_c)\). This suffices to show \((s'_a, s'_c) \in \rho_{\text{agg}}\) as required. Obviously, if \(\text{dec} \in U\), we also have \((\mathcal{U}_a, \mathcal{U}_c) \in \rho_{\text{agg}}(\{\text{dec}\})\).

Lemma C.25 \(\rho_{\text{term}}\) simulation step. Assume \(s_c' \xrightarrow{\Delta} s_c\) and \((S_a, s_c) \in \rho_{\text{term}}\). Then there exists an \(S'_a\) such that \((S'_a, s_c') \in \rho_{\text{term}}\) and, for all \(s_a' \in S'_a\), there exists an \(s_a \in S_a\) such that \(s_a' \xrightarrow{\Delta} s_a\).

Proof. Let \(sf\) be the send-field of the step, and let \(U\) be the set of fields updated in the step.

Perform a case distinction:

- \(\text{inp} \notin U\). This implies \(sf \notin F_1(s_c') \cup F_2(s_c')\) or \(sf = \text{inp}\). We take the \(S'_a\) related to \(s_c'\) in the same way as \(S_a\) and \(S_a\) are related (that is, if they were related using the \(\rho_{\perp,\perp}\) "branch", we take that, otherwise we take \(\rho_{\text{val},\perp}\)). We pick an \(s'_a \in S'_a\) and perform a similar proof as in the corresponding case of Lemma C.24. We apply Lemma C.21 (respectively Lemma C.23 if maxts is used in the protocol of Lemma C.22 if \(\lambda \in \Lambda\)) to yield two states \(s^1_a\) and \(s^2_a\). If \(S_a\) and \(s_c\) are related via \(\rho_{\text{val},\perp}\) and \(\text{dec} \notin U\), we take \(s^1_a\) and otherwise we take \(s^2_a\) as the \(s_a\).

If \#\([s_c(\text{dec})](\perp) > 0\), then also \#\([s_c'(\text{dec})](\perp) > 0\), and then for any \(s'_a \in S'_a\), by the definition of \(\text{decrel}_\perp\), there exists a process \(p_\perp\) such that \(s'_a(p_\perp)(\text{dec}) = \perp\). We use this process for item 6 of Lemma C.24 (respectively item 5 of Lemma C.23 or item 5 of Lemma C.22). This guarantees that there exists an \(s_a \in S_a\) such that \(s'_a \xrightarrow{\Delta} s_a\).

- \(\text{inp} \notin U\). For all ephemeral fields \(f \in U\), any permutation of any update will yield the same counts in \(f\). Restriction (R7) guarantees that we can choose a permutation to satisfy \(\text{ldrrel}, \text{ldrrel}_\perp\) or \(\text{ldrrel}_{\text{val}}\) on \(f\) if needed (the same restriction also guarantees that this is not needed if \(\text{dec} \in U\)).

Perform a further case distinction:

- \(sf \in \text{inp-path}(s_c)\) or \(sf = \text{inp}\). Again, for \(S'_a\) we take the same branch of \(\rho_{\text{term}}\) via which \(s_c\) and \(S_a\) are related. We invoke Lemma C.18, or Corollary C.1 if we need to preserve one of the leader relations. Furthermore, if \(\text{dec} \in U\), then by (R7) we know that we do not need to preserve any leader relations for \(U \setminus \text{dec}\) in \(s_c\). If \#\([s_c(\text{dec})](\perp) > 0\), then also \#\([s'_c(\text{dec})](\perp) > 0\),
C.3. The Lemmas

and \( \#[U_{\text{dec}}](\bot) > 0 \). Hence by Lemma C.18, we can create an update \( U_a \) from \( s_a \) such that \( \#[U_{\text{dec}}](\bot) > 0 \). By the definition of \( \rho_{\text{term}} \), there exists a process \( p_\bot \) such that \( s'_a(p_\bot)(\text{dec}) = \bot \). We permute the update \( U_a \) to ensure that \( p_\bot \) performs the local update that sets \( \text{dec} \mapsto \bot \).

– otherwise, \( sf \in \text{dec-path}(s_c) \). In that case, \( U \) is a singleton set.

We perform a case distinction:

* \( U \neq \{ \text{dec} \} \) and \( s_c \) and \( S_a \) are related via \( \rho_{\text{val},\bot} \). Relate \( s'_c \) and \( S'_a \) via \( \rho_{\text{val},\bot} \) as well. For any \( s'_a \in S'_a \), use Lemma C.10 to derive the desired \( s_a \).

* \( U = \{ \text{dec} \} \) or \( s_c \) and \( S_a \) are related via \( \rho_{\bot,\bot} \). Notice that if \( \text{dec} \in U \), then \( F_2(s_c) = \emptyset \), since all fields in in \( \text{dec-path} \) have already been sent out. Use Lemma C.11; if the resulting \( \Psi \) is \( (s_a, s_c) \in \rho_{\text{val}}(\{ sf \}) \) or \( (s_a, s_c) \in \text{ldrrel}_{\text{val}}(\{ sf \}) \), then relate \( s'_c \) and \( S'_a \) via \( \rho_{\text{val},\bot} \). Otherwise, relate \( s'_c \) and \( S'_a \) via \( \rho_{\bot,\bot} \).

Lemma C.11 guarantees that, for any \( s'_a \in S'_a \), there exists a global update \( U_a \) associated to a step \( s'_a \xrightarrow{\Delta} s_a \), such that sufficiently many \( \bot \)'s arise in the update. If \( U = \{ f \} \) for some ephemeral field \( f \), we simply take this \( s_a \).

Otherwise, \( U = \{ \text{dec} \} \), and we show that using a permutation of \( U_a \) we can reach an \( s''_a \) from \( s'_a \) such that \( (s''_a, s_c) \in \text{decrel}_{\bot} \). Note that, if \( \#[s_c](\text{dec})(\bot) > 0 \), then also \( \#[s_c](\text{dec})(\bot) > 0 \) (since \( \text{dec} \) field defaults to the old value) and \( \#[U_{\text{dec}}](\bot) > 0 \). We thus also have \( s'_a(\text{dec})(p_\bot) = \bot \) for some \( p_\bot \); we permute the update \( U_a \) to ensure that \( p_\bot \) applies the empty local update. This suffices to guarantee \( \text{decrel}_{\bot} \).

\[ \square \]

C.3.6 Simulation

Theorem C.1 (\( \rho_{\text{agr}} \) simulation). \( \rho_{\text{agr}} \) is a forward simulation between \( P \) running on \( \Sigma_k \) and \( P \) running on \( \Sigma_B \).

Proof. The base case is easy. The inductive case follows from Lemma C.24.

\[ \square \]

Theorem C.2 (\( \rho_{\text{term}} \) simulation). \( \rho_{\text{term}} \) is a backward-forward simulation between \( P \) running on \( \Sigma_k \) and \( P \) running on \( \Sigma_B \).

Proof. Again, the base case is easy. This time, the inductive case is guaranteed by Lemma C.25.

\[ \square \]
C. Proof of the Cutoff Theorem

C.3.7 Property preservation

Lemma C.26 (Preservation of decisions). Assume that in a $k$-process trace $\tau$ some two processes $p$ and $q$ decide on two values $x, y \in \{0, 1\}$ in rounds $r_1$ and $r_2$ respectively (possibly, $p = q$, $x = y$, and $r_1 = r_2$). Then, there exists a $\rho_{agr}$-related $B$-process trace that performs the same decisions in the same rounds.

Proof. By Theorem C.1 and the definition of forward simulation, there exists an $B$-process state $s$ such that $(s, \tau(0)) \in \rho_{agr}$. WLOG, assume $r_2 \geq r_1$. Create $\sigma$ by iterating Lemma C.24 $r_2$ times, and then extend $\sigma$ to an infinite trace using the HO set $HO_p^r = \Pi$ for all $p$ and all $r > r_2$. The conclusion then follows straight from the guarantees of Lemma C.24.

Theorem C.3 (Cutoff bounds for agreement). Assume that a $k$-process trace $\tau$ that violates agreement. Then, there exists an $B$-process trace $\sigma$ that also violates agreement.

Proof. Take the first two rounds $r_1, r_2$ in which conflicting decisions $v_1$ and $v_2$ appear in $\tau$, and apply Lemma C.26.

Theorem C.4 (Cutoff bounds for integrity). Assume that a $k$-process trace $\tau$ that violates the integrity property. Then, there exists an $B$-process trace $\sigma$ that also violates the integrity property.

Proof. Take the first round $r_1$ in which a non-proposed value $v$ is decided on in $\tau$. Set $x = y = v$ and $r_1 = r_2$ and invoke Lemma C.26. This yields an $B$-process trace $\sigma$ that also decides on $v$ in $r$, that is $\rho_{agr}$-related to $\tau$. By the definition of $\rho_{agr}$ (and in particular $\rho_{init}$), since $\gamma_k(\#[\tau(0)(inp)](v)) < 0$, $\gamma_B(\#[\sigma(0)(inp)](v)) < 0$, and $\sigma$ thus also violates integrity.

Theorem C.5 (Cutoff bounds for termination). Assume that a $k$-process trace that violates termination. Then, there exists an $B$-process trace $\sigma$ that also violates termination.

Proof. From Theorems C.2 and 2, the fact that the binary $B$-process system arising from the protocol has finite non-determinism (indeed, any state has only a finite number of successor states), and the definition of $\rho_{term}$.
Appendix D

Bibliography


[AHC+16] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O’Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying High-Assurance File System Implementations. In *Proceedings of the Twenty-First International Conference on Ar-
D. Bibliography

*Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’16, pages 175–188, New York, NY, USA, 2016. ACM.

[AHH15] Parosh Abdulla, Frédéric Haziza, and Lukáš Holík. Param-

[AK86] Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verifica-


D. Bibliography


D. Bibliography

[142x736]

[142x705]

[142x692]

[142x678]

[142x665]

[142x665]

[142x640]

[142x626]

[142x602]

[142x588]

[142x550]

[142x537]

[142x512]

[142x498]

[142x498]

[142x471]

[142x471]

[142x458]

[142x458]

[142x433]

[142x419]

[142x381]

[142x330]

[142x316]

[142x330]

[142x316]

[142x291]

[142x291]

[142x278]

[142x278]

[142x253]

[142x253]

[142x240]

[142x240]

[142x215]

[142x215]

[142x188]

[142x188]

[142x188]

[142x174]

[142x174]

[142x174]

[142x136]

[142x136]

[142x136]

[142x95]


