FORMAL MODELING LANGUAGES FOR HIGH-ASSURANCE DOMAIN-SPECIFIC SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Haobin Ni

May 2024

© 2024 Haobin Ni

ALL RIGHTS RESERVED

FORMAL MODELING LANGUAGES FOR HIGH-ASSURANCE
DOMAIN-SPECIFIC SYSTEMS

Haobin Ni, Ph.D.

Cornell University 2024

Software systems today struggle to behave as intended in deployment. The problem is made worse by the ever-increasing complexity of systems resulting from the rapid expansion of modern-day software. For instance, almost all systems nowadays utilize concurrency, and unexpected behaviors such as race conditions are quite common. In this case and many others, the number of possible program behaviors suffers from combinatorial explosion, which renders conventional testing insufficient to provide enough assurance at a feasible cost.

This dissertation explores alternative approaches that guarantee high assurance systematically and foundationally by building and analyzing formal models of those systems through domain-specific languages. While real-world systems often involve components of many different domains, as a first step, we choose three mission-critical domains as targets: fault-tolerant distributed protocols, concurrent programs, and parsers. For each of those domains, a new language for formally modeling domain-specific systems is proposed. Meta-theoretic analyses of those languages illustrate the relationships between models built in those languages and the realistic behaviors of the systems those models describe, guaranteeing that all those formal models are adequate for analyzing system behaviors. Case studies of systems modeled and analyzed using those languages are also presented to demonstrate the effectiveness of this approach and the developed tools in constructing high-assurance domain-specific systems.

BIOGRAPHICAL SKETCH

Haobin Ni was born in Yancheng, Jiangsu, China in 1995. He moved to Shanghai in 2010 and graduated from Shanghai High School three years later. He then attended Shanghai Jiao Tong University and earned his Bachelor of Science degree in May 2017. Haobin was an active competitive programming contestant for over a decade until he retired in 2016. He then decided to delve deeper into computer science and joined Cornell to pursue his PhD in June 2017.

For science and my advisors, Robbert van Renesse and Greg Morrisett.

ACKNOWLEDGEMENTS

I am forever grateful for the help and support of a great many people during the seven years of my graduate school life. This dissertation would not have been possible without them.

First, to my advisors, Robbert van Renesse and Greg Morrisett. Looking back, I was clueless in the first year, struggling to convey my naive research ideas in broken English. And yet, you both welcomed me with open arms and guided my steps through the arduous path to knowledge. And it is not only knowledge I have learned – you lead by example what it means to be an academic: dedicated to truth, education, and the greater good of society. I can't put into words how grateful I am for your support and guidance throughout our journey together. I can only hope that one day, I will be able to repay a fraction of what you have given to me through my own contribution to the whole endeavor of science and education.

I would also like to thank my mentor, Andrew Myers. Your wise and witty comments during our research meetings and casual conversations taught me a lot about programming language designs and information flow. Although I am not an official advisee, I have always been generously welcome to join the group meetings, where I could exchange ideas freely with peers with shared research interests (and eat the meeting snacks).

Special thanks go to my internship mentors, Nikhil Swamy and Tahina Ramananandro, and colleagues, Aseem Rastogi, Antoine Delignat-Lavaud, Cédric Fournet, and Jonathan Protzenko. It has been a great pleasure of mine working with you on cutting-edge industrial research in formal verification.

I owe a great deal to my roommates Siqiu Yao and Dietrich Geisler, who suffered from prolonged exposure to me during our Ph.D.s. Your emotional

support has been indispensable during difficult times like deadlines, rejections, and pandemics.

Many thanks go to these faculties members at Cornell: Dexter Kozen, Adrian Sampson, Fred Schneider, Mark Bickford, and Ross Tate; to people from Andrew's research group: Ethan Cecchetti, Mae Milano, Tom Magrino, Isaac Sheff, Yizhou Zhang, Rolph Recto, Drew Zagieboylo, Coşku Acay, Yulun Yao, Silei Ren, and Vivian Ding; to the 456 PL lab friends: Pedro Amorim, Ayaka Yorihiro, and Goktug Saatcioglu; to people from other PL groups at Cornell: Joshua Gancher, John Sarracino, Ryan Doenges, Eric Campbell, Michael Roberts, Andrew Hirsch, Mark Moeller, and Anshuman Mohan; to people from Cornell Systems groups: Ted Yin, Yunhao Zhang, Hongbo Zhang, Burcu Canakci, Xinwen Wang, Soumya Basu, Ali Farahbakhsh, Suraaj Sureshkannan, Yu-Ju Huang, Alicia Yang, Shir Cohen Gahtan; to people from other fields: Wen-Ding Li and Harjasleen Malvai; and to undergraduates I have mentored: Chujun Song and Natalie Neamtu.

Additional thanks go to the administrators: Becky Stewart and Lynette Jordan; to the Cornell High School Programming Contest Organization Committee: Sophie Lanchez, Diane Levitt, Lizette DeJesus, Daniel Fleischman, and Vanessa Maley; and to the Cornell ICPC members.

The work in this dissertation was partially funded by NSF grants CCF-1918396, 1601879, CNS-1704788, and CNS-1704615.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . vii
List of Tables . . . ix
List of Figures . . . x

1 Introduction **1**
1.1 Fault-tolerant Distributed Protocols . . . 3
1.2 Concurrent Programs . . . 6
1.3 Parsing . . . 8
1.4 List of Contributions . . . 10

2 Ironwood **14**
2.1 Background . . . 18
2.2 Syntax and High-Level Semantics for Sync . . . 22
2.2.1 Sync Syntax . . . 22
2.2.2 Typing for Sync Programs . . . 24
2.2.3 Denotational Semantics for Sync . . . 26
2.3 Case Studies . . . 33
2.3.1 Modeling Non-terminating Consensus Protocols . . . 33
2.3.2 Bosco . . . 35
2.3.3 Sequential Paxos . . . 40
2.4 Low-Level Semantics . . . 46
2.4.1 Async Node Semantics . . . 46
2.4.2 Single-use Channel Semantics . . . 50
2.4.3 Composing Labeled Transition Systems . . . 54
2.5 Adequacy Proof . . . 58
2.5.1 Reduction to Aligned Traces . . . 59
2.5.2 Adequacy of Aligned Traces . . . 62
2.6 Related Work . . . 67
2.6.1 Detailed Discussions . . . 71
2.7 Future Work . . . 86

3 Harmony **87**
3.1 Using Harmony . . . 89
3.1.1 Describing Lock Algorithms . . . 90
3.1.2 Testing and Debugging Lock Algorithms . . . 92
3.1.3 Checking Refinement Relations . . . 94
3.2 Design . . . 95
3.2.1 Modeling Concurrency and Memory . . . 96
3.2.2 Harmony Virtual Machine . . . 99

3.2.3 Model checking . . . 102
3.3 Implementation . . . 104
3.4 Evaluation . . . 107
3.4.1 Dining Philosophers . . . 110
3.4.2 Concurrent Queue . . . 111
3.4.3 Concurrent Journaling File Server . . . 115
3.4.4 Other Models . . . 117
3.4.5 Evaluation of Model Checking . . . 119
3.5 Preliminary User Study . . . 121
3.6 Related Work . . . 123
3.7 Future Work . . . 124
3.8 Conclusion . . . 126

4 ASN1⋆ 127
4.1 Background on F* and EverParse . . . 132
4.2 A Brief Primer on ASN.1 and DER . . . 135
4.3 ASN1⋆ . . . 138
4.3.1 Syntax and Well-Formedness of ASN.1 . . . 139
4.3.2 Denoting ASN1⋆ Declarations as F* Types . . . 144
4.3.3 A Constructive Formalization of DER . . . 146
4.3.4 Automata-Based Parser Combinator . . . 151
4.4 Experimental Evaluation . . . 155
4.4.1 X.509 Certificates . . . 156
4.4.2 Certificate Revocation Lists . . . 161
4.5 Related Work and Conclusions . . . 162

5 Conclusion 170
5.1 Future Work . . . 170
5.1.1 A Formal Foundation For New Systems And Optimizations 171
5.1.2 Cross-domain Compositionality . . . 172
5.1.3 Large-scale Development of High-assurance Systems . . . 172
5.2 Conclusions . . . 173

Bibliography 174

LIST OF TABLES

2.1 Lines of Code of Ironwood Components 33

3.1 The number of states, memory usage (GBytes), and running time (seconds) of model checking the concurrent queue test program. The server used all 64 hyperthreads (32 cores), while the laptop used all 8 hyperthreads (4 cores). 114
3.2 Comparing identified and anonymous threads. Time is in seconds and size is in megabytes 120
3.3 Summary of (a) study results; and (b) student feedback for the three tasks (0, 1, 2) 123

4.1 Encoding Unicode points to UTF-8 152
4.2 Transitions for the initial state and the first byte 152
4.3 Results of running extracted X.509 and CRL parsers 159

LIST OF FIGURES

2.1 `SimpleVote` Program 15

3.1 Specification of a lock and a spinlock based on atomic test-and-set 90
3.2 Test programs for locks. (a) is based on invariants, (b) on external behavior checking 93
3.3 A Harmony counter-example 93
3.4 PlusCal/TLA+ implementation of Dining Philosophers (by Murat Demirbas). $\mathit{fork}[i] = N$ means fork i is available; otherwise, $\mathit{fork}[i]$ contains the identifier of the philosopher holding the fork. . . . 108
3.5 Promela/SPIN implementation of Dining Philosophers. $\mathit{fork}[i] = 0$ means fork i is available; otherwise, $\mathit{fork}[i]$ contains the identifier of the philosopher holding the fork. 109
3.6 Harmony implementation of Dining Philosophers where forks are represented as locks. Because Harmony programs must have terminating executions, a diner eats 0 or more times before leaving. 110
3.7 The number of states and times used to detect deadlock as function of the number of philosophers using the TLA+ model checker, SPIN, and Harmony. Harmony-- removes the memoization of computation. Times are measured on the laptop. Very short times are omitted. 112
3.8 Harmony specification of a concurrent queue 112
3.9 (a) A reproduction of the pseudo-code for the 1996 Michael&Scott two-lock concurrent queue implementation [87] and its equivalent in Harmony (within the box); (b) a behavioral test program . 113
3.10 Selection of other models evaluated in Harmony. Time is in seconds. 117
3.11 Speed-up of concurrent model checking as a function of the number of threads on the server. There are two models: Dining Philosophers (using 9 philosophers) and Weak Bisimulation of the Michael/Scott concurrent queue implementation using NOPS=3. 119
3.12 Number of SVM evaluations divided by the number of edges in the Kripke structure for different numbers of Harmony threads. Shown for both dining philosophers (diners) and barrier synchronization (racers) 119

4.1 Informal syntax of ASN.1 137
4.2 An ASN.1 declaration from X.509 137
4.3 Architecture of our development 140
4.4 Formal syntax and well-formedness 166
4.5 Denoting ASN.1 definitions as F* types 167
4.6 The parser denotation of ASN1⋆ (fragments) 168
4.7 Representing X.509 in ASN1⋆ 169

CHAPTER 1
INTRODUCTION

This dissertation proposes new formal modeling languages for software systems of specific domains and show how to build high-assurance systems through formal modeling and analysis using those languages.

Even with state-of-the-art software engineering, software systems today suffer from behaving in ways unintended by their design in deployment. This situation is becoming increasingly dire as our society becomes more dependent on the services provided by those systems. The problem is made even more challenging by the rapid expansion of software systems. Modern-day production systems often reach massive scale, containing millions of lines of code. They include intricate business logic to provide a multitude of functionalities at high performance. All of these complexities result in the disturbing phenomenon that the behaviors of those large systems we have built are escaping the grip of our understanding. This is especially harmful for tasks that require precise control of those behaviors, such as airplane control or geo-distributed network systems.

Similar to the case of any other engineering discipline, we have means that can increase our confidence in the produced systems. Testing – checking if the system behaves as intended in a simulated environment – has been an effective approach. However, conventional testing has become insufficient. One reason is that the number of behaviors in many software systems suffers from combinatorial explosion, rendering it infeasible to test a significant portion of all possible behaviors. Theorizing has also been used as a means to provide assurance for software systems. Instead of analyzing the concrete system, an abstract model of the system is analyzed mathematically. While the properties

of the abstract model can be asserted with high confidence, there is an inherent gap between the abstract model and the deployed system where flaws could be introduced.

A more recently developed methodology that could provide high assurance for systems at the level of implementation is formal verification. A formal model of the system is also needed but is usually derived from the system code itself systematically, closing the gap between the abstract model and the concrete reality. Additionally, it requires a machine-checkable proof object to be built alongside the system, which formally guarantees the system to behave as specified in mathematical logic. Unfortunately, although such a proof object ought to exist in theory if the system being built is actually correct, constructing machine-checkable proofs about a modern-day program's behaviors could require a dozen times more effort than writing the program itself. This additional cost restricts the applicability of formal verification to small and high-stake programs.

This dissertation explores alternative paths to building high-assurance systems systematically and foundationally through building and analyzing formal models of systems. There are three pieces to the plan. First, one builds a model of the system in a domain-specific modeling language that restricts the program structure. Second, the model is connected to a real-world program through a compilation or extraction process. Third, one analyzes the model to validate the system. The effort required is alleviated by the design of the modeling language. The properties are transferred to the real-world program through meta-theorems about the domain-specific language used, which are proven once and for all.

While modern software systems almost all involve components of various domains, as a first step in this direction, the research covered by this dissertation

focuses on systems of a single domain for three mission-critical domains: fault-tolerant distributed protocols, concurrent programs, and parsers. For each domain, a new formal modeling language and the corresponding meta-theoretical analysis are presented. Case studies are also included to demonstrate the effectiveness of the proposed approach and the tools developed for constructing high-assurance domain-specific systems. The rest of the introduction will give more background on each domain and illustrate the challenges more concretely.

1.1 Fault-tolerant Distributed Protocols

Distributed systems are the study of systems in which the components may carry out operations independently and coordinate through communication to jointly complete certain tasks. A common setup is a collection of machines communicating through some network that connects them.

Due to having components performing operations independently, distributed systems are concurrent by nature. Because of this, the ordering of the operations performed by the components is not fixed as in the case of non-concurrent programs. This leads to non-deterministic behaviors and complicates the analysis. Another domain feature is failures. Due to the sheer number of components present in a distributed system, the possibility that a few components might not execute exactly as intended is high, so fault-tolerance, the property that the whole system may still operate (possibly at reduced efficiency) with the remaining non-faulty components, is highly desirable.

As a subfield of computer systems, a primary goal of distributed systems is performance: low latency and high throughput. The additional requirement of

fault tolerance divides the systems into many classes depending on what failures they tolerate. Systems that provide different degrees of fault tolerance are usually not directly comparable, as being more resilient to faults often comes at the cost of being less efficient.

Researchers approach these integrated goals from different angles, two of which are theoretical and experimental. The subject of a theoretical study is usually an abstract distributed system. Either explicitly or implicitly, some abstract "machine" and "network", simplified from reality, are assumed. An algorithm may be proposed on top of those abstract models, and some theoretical analysis may be conducted to reveal certain properties of the algorithm or the abstract models.

For instance, the FLP impossibility result [44] is a famous theoretical result in distributed system. Informally, it states that no terminating distributed program can solve the problem of consensus in an asynchronous network in the presence of even a single failure. Clearly, such a statement goes beyond specific physical machines and networks.

For another example, the infamous original Paxos paper [76] described an abstract, non-terminating, distributed algorithm that solves the problem of consensus in the presence of crash failures in an asynchronous network by telling a tangled story about some hypothetical legislation practice of the Paxos island. The paper would make no sense if taken literally without regarding it as talking about some abstract models of distributed system.

The experimental studies, on the other hand, argue for the superiority of a system design by building and measuring concrete distributed systems. At

the center of the stage is a piece of software, which is referred to as "the system" in its narrow sense. The software is deployed upon some physical setup, usually using off-the-shelf machines, sometimes with specialized hardware. To show the effectiveness of the software, some experiments are conducted, and measurements are taken. The design of the experiment usually involves some kind of workload or benchmark and baselines for comparison. The results of the experiments and some explanations of the results are then reported.

Unsurprisingly, both research schemes do not always produce sound conclusions, given that the actual path from the abstract algorithm to the concrete system goes through many layers of informal and implicit abstractions in addition to the complexity at a specific layer. Errors could happen at the higher abstraction layers: the Zyzzyva consensus protocol [69] proposed a novel speculative Byzantine fault-tolerant consensus protocol, but a counterexample that breaks the safety property of the protocol was later found [1]. Errors can also happen at the layer of implementation or stem from the mismatch between the interfaces the high-level assumes and the low-level provides: the Paxos consensus protocol, although simple in theory, turns out to be challenging to implement correctly in practice.

To provide higher assurance for fault-tolerant distributed systems, Chapter 2 of this dissertation presents Ironwood, a formal verification framework for Byzantine-fault-tolerant distributed systems, based on joint work with Robbert van Renesse and Greg Morrisett.

1.2 Concurrent Programs

A regular sequential program follows a single sequence of commands from start to finish, but a concurrent program has more than one flow of control. For instance, to sum up all elements in an array, instead of doing a scan of the array from the beginning to the end, one can compute the sum of the first half and the second half concurrently and then sum up the results.

Concurrent programs are generally understood as several sequential flows of control, called threads, that share the same memory space. The relative execution order of those threads is non-deterministic by default. Sharing the same memory means those different threads can communicate directly yet often implicitly by writing and reading the shared memory, which is different from the case of distributed systems. Failure is also usually not a concern in concurrent programs because threads are not physically isolated, and there is only a single point of failure. In short, the goal of concurrent programs is to maximize performance by completing tasks as concurrently as possible, assuming all threads will follow the program precisely to collaborate.

What is most challenging about concurrent programs is concurrency control – how to be as fast as possible without having threads clash with each other. The simple example of summing up an array by summing up the two halves mentioned above utilizes some concurrency but may not be the most efficient because different threads can run at different speeds due to hardware, scheduling, or unequal distribution of tasks. An alternative algorithm also using two threads is to have one thread going from the beginning to the end and the other from the end to the beginning and sum up the two partial sums when the two threads

meet somewhere in the middle. This algorithm has the advantage that it is more adaptive to the relative speed of the threads. However, extra care is needed to ensure the two threads meet properly – each element of the array must be summed exactly once in the result.

To better control the concurrent behavior, systems researchers have designed many patterns and algorithms. Peterson's algorithm for mutual exclusion [100] is one of the most classic concurrency control algorithms. The goal of the mutual exclusion problem is to ensure that at most one out of multiple concurrent threads can execute a specific part of the program, called the critical section, at a time. This is useful for avoiding race conditions in which two or more threads read/write the same location in memory simultaneously, which may produce erroneous values.

Although appearing simple, Peterson's algorithm is surprisingly subtle. For one thing, assigning independent values to two different variables one after another can be done in either order in a non-concurrent program with the same results. Many compilers leverage this to optimize performance. In Peterson's algorithm, there is a place where independent values are assigned to two different shared variables. However, the critical session guarantee relies on those two assignments being done in a specific order, and reversing the two assignments results in an incorrect algorithm.

Despite being difficult to implement correctly, concurrency is indispensable for performance in modern systems. Many programming languages aid the construction of concurrent programs by providing higher-level primitives, such as the goroutine in Go, the synchronized keyword in Java, and processes in Erlang. Unfortunately, these higher-level primitives are not restrictive enough to

eliminate concurrency bugs. A recent trend that provides high assurance against classes of memory issues is that of the Rust programming language. The type system of Rust tracks the ownership of memory regions and prevents potentially dangerous race conditions. The main drawback of this approach is that it is overly restrictive, making certain safe operations impossible without explicitly turning off parts of the type system for parts of the program. This makes it undesirable when precise low-level control of concurrent behavior and memory operations is needed.

Chapter 3 of this dissertation introduces the Harmony language, which provides high assurance for concurrent programs through exhaustive model checking. It is easy to use because its surface language is similar to Python, and it has a virtual machine semantics similar to system programming languages. It also provides high performance because the semantics is specialized for model checking and enables many optimizations. Chapter 3 is based on joint work with Robbert van Renesse, Renyu Li, William Ma, Kevin Sun, and Anthony Yang.

1.3 Parsing

Parsing, also called syntactic analysis, is the process of processing a stream of symbols according to a formal grammar. The opposite procedure that produces a stream of symbols according to a formal grammar is called deparsing or serializing and is often studied together. In computer science, parsing is often used in compilers to process the input program. Additionally, together with serializing, parsing is used in applications that load and save files from and to disks and in communication across systems. A typical scenario of communication starts

with the sender of a message serializing the data into a string of bytes, which is transported to the receiver, potentially through some network. The receiver then parses the message and derives the data represented by the received string of bytes.

While parsing sounds like a simple task, it has been a common source of software vulnerabilities. For example, a CVE incident [84] in 2009 discovered that Microsoft's CryptoAPI component would incorrectly handle an X.509 certificate's Common Name field if the field contains a null character, which can lead to man-in-the-middle attacks through spoofed certificates. There are multiple factors that contribute to this issue. Firstly, many parsers present a so-called "open" attack surface, i.e., a malicious attacker can provide any input, such as any string of arbitrary bytes, to the parser program. For instance, a browser needs to parse any webpage coming from the Internet, many of which are totally controlled by attackers trying to gain access to private information. This means the parser program must be able to handle all the corner cases properly. Secondly, the grammars the parsers follow are often complicated due to the conflicting goals of being flexible to accommodate different use cases and being efficient in terms of the number of bytes or symbols used to represent data. For example, the extra information about how many bytes are used to represent an integer can be ignored if the sender and the receiver agree on a number a priori, say 4 bytes. But in order to be flexible, the grammar can allow this extra information on the length of the representation to be added when the number of bytes needed exceeds 4. This leads to the subtle case when the number of bytes needed does not exceed 4, yet the length of the representation is still given. The third challenge for parsing is the fact that they often need to be compatible with long-established legacy standards, which may not be originally designed for the current usage and are

updated iteratively through many versions and fixes.

The state-of-the-art for parsing comes in two categories. In the first category, if the grammar can be clearly defined in a well-studied grammar class, one can most certainly use an existing parser generator such as CUP or Bison to generate a parser with high assurance. The other category is when the grammar does not fit into any of those grammar classes due to ad hoc features. This is almost always the case when the raw sequences of bytes are being dealt with. A custom implementation would be unavoidable, which has been known to be error-prone.

Chapter 4 of this dissertation showcases ASN1⋆, the first formalization of the standardized data description language ASN.1. This language allows one to generate correct-by-construction parsers for the DER encoding of ASN.1 that can be used in security-critical cases such as X.509 certificates. Chapter 4 was previously published as *ASN1*: Provably Correct Non-Malleable Parsing for ASN.1 DER* in the proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23), co-authored with Antoine Delignat-Lavaud, Cédric Fournet, Tahina Ramananandro, and Nikhil Swamy [96].

1.4 List of Contributions

I make the following list of contributions in this dissertation.

Chapter 2 introduces Ironwood. I designed the syntax and the denotational semantics of the high-level language of Ironwood, Sync. I implemented this design as a shallow embedding in the Coq proof assistant using parametric higher-order abstract syntax. I modeled two consensus protocols, BOSCO and

SeqPaxos, using the implemented Sync language and then formally defined and proved the agreement properties of those two protocols. I designed the low-level language of Ironwood, Async, including the Async monad, the single-use channel state machine, and the parametric parallel composition of labeled transition systems. I designed the compilation from Sync to Async and implemented the compilation in Coq. I implemented the extraction mechanism for Async monadic programs and an OCaml shim layer for executing extract programs as a distributed system. I analyzed the relation between Sync and Async and proved with pen and paper that the denotational semantics of Sync is adequate for reasoning about the dynamic behavior of Async systems, that is, the properties proven about the Sync models transfer to the compiled Async systems.

Chapter 3 introduces Harmony. I helped design the Harmony language, including its surface syntax, the Harmony virtual machine semantics, and the Interleaving of Instructions with Strictly Consistent memory (IISC) model. I helped design the Harmony compiler and the Charm model checker and contributed to their implementation. I designed several optimizations, including a counterexample minimization algorithm, concurrent hash tables, and NUMA-aware concurrency control. I help designed some experiments for evaluating Harmony. I designed and conducted a preliminary user study on the usability of Harmony as a debugging tool for beginner programmers.

Chapter 4 introduces ASN1$\star$. I designed the syntax, the AS TYPE semantics, and the AS PARSER semantics of ASN1$\star$ and implemented them in the F* proof assistant. The implementation involved extending the EverParse parser generator framework with new generators, such as the automaton parser. The implementation of the two semantics implies the non-malleability results of the

ASN1⋆ parsers using dependent types. I designed and implemented the compilation of ASN1⋆ declarations into extractable parsers using Futamura projection. I implemented standard-compatible X.509 Certificate declarations and Certificate Revocation Lists declarations in ASN1⋆ and conducted the experiments that evaluate the ASN1⋆ design by running extracted parsers on real-world corpora.

Common Themes

Although the three languages target three different domains, they all serve the common goal of providing higher assurance for systems in those domains. Furthermore, they also share principles and techniques.

A common pattern in all three cases is attributing multiple semantics to the same set of syntax. In Ironwood, Sync programs have both a denotational semantics that models distributed systems as non-deterministic functions and an operational semantics that models them as labeled transition systems. In Harmony, Harmony programs have two related semantics: one is a Kripke structure explored by the model checker for analysis and the other is a non-deterministic operational semantics defined by the Harmony virtual machine and the IISC model. In ASN1⋆, ASN1⋆ declarations are mapped to F* types through the AS TYPE denotational semantics and mapped to EverParse parsers through the AS PARSER semantics.

Those semantics are then connected together through reusable techniques: Our adequacy theorem in Ironwood guarantees that safety properties proven for Sync programs in the denotational semantics are preserved in the Async semantics. The theorem itself is proven through a pen-and-paper proof. The correctness

of Harmony model checking relies on the preservation of program behaviors in the Kripke structure regarding the operational semantics, which is ensured by the properties of the semantics and the design of the optimizations applied to the model checker. ASN1⋆ ensures the non-malleability of its parsers by construction through dependent types – the types of the AS PARSER semantics depends on the definitions of the AS TYPE semantics. While using those techniques themselves requires much effort, they are applied at the level of the modeling languages and thus are one-time costs. The resulting languages are reusable for classes of domain-specific systems.

Additionally, the design and the implementation of those domain-specific languages also have other similarities because they are all designed for program analysis instead of for performance or reducing development overhead. For instance, all three languages provide high-level, abstract primitives and restrict program behavior through syntax and type systems. Both Ironwood and ASN1⋆ are implemented through shallow embeddings of the domain-specific language in a proof assistant host language.

As a whole, this dissertation presents the results of my exploration of the design spaces of domain-specific modeling languages and serves as a stepping stone toward more effective methodologies for building high-assurance systems.

CHAPTER 2
IRONWOOD

Distributed systems are indispensable to modern information infrastructure because they provide scalable performance and fault tolerance. Assurance is crucial for such systems, and a great deal of research has focused on applying formal methods to achieve this goal. Existing work includes those that target abstract distributed protocols such as TLA+ [77] and others that verify more concrete, executable programs, whose scopes vary from one-shot verified system artifacts, such as IronFleet [52], to specialized logics and verification frameworks that target a range of systems, such as Verdi [130]. Most of the existing work relies on trace-based semantics to model the asynchrony and concurrency that naturally occur in distributed systems and uses invariants and refinements for formal reasoning.

In this chapter, we propose Ironwood, a formal verification framework for executable Byzantine fault-tolerant distributed systems built in Coq. Ironwood explores a novel approach that does not use trace-based semantics or invariant-style reasoning. The key design of Ironwood is its high-level language, Sync, which has a relatively simple, functional denotational semantics for describing and reasoning about distributed systems. Sync leverages the fact that (good) nodes in a system share the same code and follow the same flow of control to describe the protocols using a synchronous, data-parallel model. It lightens the verification burden by factoring interleavings out of the reasoning and reduces the set of behaviors to consider. Of course, we must still account for the other inherent complexities of distributed systems, such as message re-ordering and messages sent by Byzantine nodes, but Sync isolates this complexity into its

(a) SimpleVote Program in Sync Syntax

$\texttt{SimpleVote}\ (x : \{L : \mathbb{B}, R : \mathbb{B}\}) : \{L : \texttt{option}\ \mathbb{B}\} :=$
\\ *The input x is a distributed value*
\\ *The leader takes x_L, and replica nodes take x_R*
$\text{let}\ \texttt{cnt} := \texttt{comm}\ c\ x_R\ \lceil 0 \rceil_L\ (\lceil \texttt{fcnteq} \rceil_L\ x_L)\ \text{in}$
\\ *Replicas send their inputs to the leader*
\\ *The leader processes the messages with fcnteq*
$\texttt{ret}\ \{L \mapsto (\lceil \texttt{calc_dec} \rceil_L\ \texttt{cnt}_L\ x_L)\}$
\\ *The leader computes calc_dec*

(b) SimpleVote Compiled into Separate Programs for L and R for Execution

\\ *The leader node executes:*	\\ *Every correct replica node executes:*
$\lambda x.\ \text{let}\ \texttt{cnt} := \text{receive}\ c\ 0\ (\texttt{fcnteq}\ x)\ \text{in}$ $\quad \text{return}\ (\texttt{calc_dec}\ \texttt{cnt}\ x)$	$\lambda x.\ \text{send}\ c\ x$

(c) Where

$\texttt{fcnteq}\ (p : \mathbb{B})(\texttt{cnt} : \mathbb{N})(v : \mathbb{B}) : \mathbb{N} :=\ \text{if}\ (p == v)\ \text{then}\ (\texttt{cnt} + 1)\ \text{else}\ \texttt{cnt}$
$\texttt{calc_dec}\ (\texttt{cnt} : \mathbb{N})(p : \mathbb{B}) : \texttt{option}\ \mathbb{B} :=$
$\quad \text{if}\ (\texttt{cnt} \geq n - 2 * f)\ \text{then}\ (\text{Some}\ p)\ \text{else None}$

Figure 2.1: SimpleVote Program

communication primitive, leaving the rest of the semantics deterministic. Not all protocols can be represented in Sync, but when they can, we get to use its functional semantics to do Hoare-style proofs – a style closer to the informal reasoning used by protocol designers, which focuses on the set of possible messages encountered in separated rounds of communication.

As an example, the top part of Figure 2.1 shows a Sync program SimpleVote. It describes the whole distributed system with a single program. Code that SimpleVote compiles to is shown in part (b) to give some intuition for its meaning. Part (c) specifies some auxiliary functions used throughout. Sync supports different *roles* such as leaders (L) versus replicas (R) and associates different code with different roles, but all nodes of a given role execute the

same code. Additionally, the Sync design prevents nodes for a given role from executing different communication patterns (*e.g.*, through a conditional based on its own local input).

`SimpleVote` operates on two roles: L and R. In this example, we assume there is a single L node and n R nodes, where n is a parameter configurable at execution time. The protocol also has another parameter f, the maximum number of R nodes that may be Byzantine. Both n and f are used in the definition of `calc_dec`. The leader node is assumed to be always correct. The first line of `SimpleVote` definition states that the program takes in a *distributed* value x, which contains a Boolean input, x_L, for the leader, and a Boolean input, x_R, for each of the replica nodes. The return type specifies that the protocol returns an optional Boolean value at the leader node. The following line is a let binding that binds the result of a *communication* action to the distributed variable `cnt`. The communication action specifies replica nodes should send their input values x_R on channel c to the leader node, who would wait for at least $n - f$ messages and process those messages by folding the function $\lceil \texttt{fcnteq} \rceil_L\, x_L$ over the list of received messages with the default value 0. `fcnteq` is defined as a pure function and lifted to the role L to execute. It counts the number of messages that contain a value equal to x_L. The final line specifies the leader should compute $\lceil \texttt{calc_dec} \rceil_L\, \texttt{cnt}_L\, x_L$ to make a decision based on whether there were enough votes containing x_L from the replicas. In particular, if there are at least $n - 2f$ votes containing x_L, the leader should return Some x_L and None otherwise. In practice, this code could be a portion of a larger protocol that would involve more communication actions and iterations until a decision is made.

A safety property of this program is that when $n > 3f$ and the leader and

all correct replicas have the same Boolean value b as their input, the output of the leader must be Some b, no matter the actions of the faulty nodes. We can prove this in Ironwood by directly leveraging the denotational semantics of Sync programs.

Informally, the proof takes three steps. First, because the leader waits for at least $n - f$ messages, among which at most f messages come from faulty nodes, there are at least $n - 2f$ messages received from the correct replica nodes, and all of those contain the value b. Second, by the definition `fcnteq`, this implies $\texttt{cnt}_L$ is at least $n - 2f$. Last, by the definition of `calc_dec`, we derive that the leader's output is Some b.

While the above reasoning is intuitive, its soundness is not apparent because we did not explicitly reason about all the possible interleavings of the leader program and all the replica programs. The insight of Ironwood is that the above denotational-style reasoning is sound when the communication actions of the distributed program respect a logical, linear order. This restriction captures common cases and is enforced syntactically in Sync. We back this claim with a formal pen-and-paper proof that connects our high-level Sync semantics to the classic semantics once and for all.

To this end, we define Async, a low-level language that models a distributed program as a global state transition system. The global system is composed of local systems that model individual nodes, and the semantics permits the interleaving of the steps of different local systems. We specifically prove that for any program p in Sync, any output produced by the state transition system derived from compiling p to Async, is included in the functional semantics of Sync. This guarantees that any safety properties proven in the Sync semantics

also hold for the compilation result in the Async semantics.

To demonstrate the efficacy of Ironwood, we present two case studies in which we build mechanized Coq proofs and extract executable systems: Bosco, a one-step Byzantine-fault tolerant consensus protocol; and Sequential Paxos, a crash-fault tolerant consensus protocol that is a variant of Paxos [76]. Our experience with the case studies suggests the effectiveness of our framework for reasoning about an interesting class of fault-tolerant distributed systems with moderate effort.

2.1 Background

Consensus protocols are the main class of protocols we target in Ironwood. They are important building blocks for providing fault tolerance in distributed systems. Informally, the consensus problem begins with a collection of independent nodes that can only communicate through some network, and each node needs to pick a value out of a set of candidate values. Here, we assume the permissioned setting where every node is aware of all nodes in the system, and each pair can communicate directly. The goal is to have all the nodes pick the same value. The challenges are that some nodes may be faulty, and the network may delay the messages.

Two well-studied fault models are crash faults and Byzantine faults. In the crash fault model, a faulty node will follow the designated protocol until some point and stop the execution, so it no longer sends or receives any message. The Byzantine fault model is the least restrictive model, as a faulty node may deviate from the protocol arbitrarily. This is often too loose as the faulty nodes

may solve any problem, even ones known to be impossible or computationally difficult in general. So extra assumptions that limit the computational power of the faulty nodes are common in proposed protocols. Crash fault behaviors are also Byzantine fault behaviors by definition. In this dissertation, we also assume the faults are never detected, i.e., the non-faulty nodes in the systems may never know which nodes are faulty.

Two network models are synchronous and asynchronous. In a synchronous network, a message sent by a non-faulty node to a non-faulty node is guaranteed to be delivered under a known bound. So a receiver node can observe the absence of a send action. In an asynchronous network, there is no such an upper bound, so the messages can be delayed arbitrarily, and a receiver node may never know if a message is not sent.

An ideal consensus protocol would at least provide agreement, validity, and termination. Agreement means all non-faulty nodes will pick the same value. Validity means the picked value must be a proper candidate value, so the protocol cannot just let all nodes decide on a statically-known value. Termination means that all non-faulty nodes should terminate in finite steps.

Due to the famous FLP impossiblity [44], in an asynchronous network and in the presence of even a single crash failure, it is impossible to have a protocol that satisfies all three properties above. Most protocols designed so far would provide the safety properties, agreement and validity, under all scenarios, and termination only when extra assumptions hold. In this dissertation, we focus on agreement and validity.

Network and Adversary Assumptions

Throughout, we assume any network message contains two parts of information: some message payload x, which is a well-typed value, and an auxiliary header, which includes information such as the sender's identity, the receiver's identity, *etc.* The header information is invisible to the distributed program and is used only by the runtime system to provide specific guarantees.

We assume the network does not duplicate messages, which can be enforced by suppressing the duplicates at the receiving end. We also require the messages to be *authenticated*: if a message with the payload x is received by some non-Byzantine receiver p, the runtime of p can check that the identity of the sender is some node q in the system using the information in the header. Furthermore, if q is non-Byzantine, q must have sent the message with payload x. Cryptographic signatures are a common mechanism used to satisfy these assumptions, but we do not model signatures directly.

For the delivery of messages, we assume the network is asynchronous and reliable. Asynchronous means that there is no bound on the delay between the sender node sending a message and the receiver node receiving and processing this message. Reliable means that a message sent from a correct sender to a correct receiver will eventually arrive, which is necessary for the system to make progress. This eventual arrival can be achieved by resending the messages.

We assume a powerful adversary that overestimates the capabilities of a realistic one. In particular, our adversary has complete control over the set of faulty nodes, has unbounded computational resources, is capable of sending out arbitrary messages, and controls the message delivery schedule.

However, we do not explicitly model the adversary doing any of the following:

- Attacking the liveness of the network, such as in denial-of-service attacks.
- Replaying or modifying messages sent by non-Byzantine nodes as in man-in-the-middle attacks. Our authenticated network assumption excludes the cases where the adversary forges messages that appear to have been sent by a correct node. Additionally, this assumption implies that a message replayed by the adversary will have the same content and auxiliary information, such as the sender, receiver, and channel.
- Sending ill-formed messages, such as incomplete auxiliary information or an ill-typed value. Although a common source of vulnerabilities in reality, this kind of messages can be filtered by a correct parser and dropped. In our case, we do not model parsing nor verify its correctness.
- Sending multiple messages with the same header but different payloads. We assume the receiver simply drops all but one message from a given sender.

Under our assumptions, the Byzantine adversary can be modeled as a stateless entity because any series of events that can happen during the execution does not change its capabilities. Furthermore, because the only ways the adversary can affect the execution of correct nodes are by sending well-formed messages that contain arbitrary contents and manipulating the delivery schedule, we model the adversary's influence through a special action of the network that creates well-formed messages out of thin air and consider all possible schedules of the network.

One limitation of this simple adversary model is that we cannot easily reason about protocols where the Byzantine nodes are meant to be limited—*i.e.,* when they should *not* be able to compute or "guess" arbitrary information, such as a nonce or cryptographic key.

2.2 Syntax and High-Level Semantics for Sync

This section introduces Sync, our high-level language for modeling Byzantine fault-tolerant distributed systems. Sync is in choreography style [90, 135], which means a Sync program describes the whole distributed system and can be compiled into programs for individual nodes. We enforce an order on the communication actions syntactically in the described system. The denotational semantics of a closed Sync program is a *set* of possible distributed value combinations on some set of nodes. The denotational semantics of a Sync program with free distributed variables is a function that transforms input values for those variables to a set of possible output distributed value combinations. Sync is synchronous and data-parallel because it models the distributed computation as executing on all nodes in lockstep.

2.2.1 Sync Syntax

Our Sync syntax uses the *role* abstraction to describe distributed systems with different kinds of nodes (such as a leader or a replica) while abstracting the number of nodes that will be executing the code associated with a given role. We use L and R as example roles.

A Sync program describes an abstract distributed protocol as a high-level functional program that runs the same code for each node in a given role, albeit on different input values. Its syntax is defined as follows:

Definition 2.2.1 (Sync Syntax)**.**

Roles	R, L
Meta-terms	t
Channels	c
Distributed Variables	x

Sync Expressions

$$
\begin{array}{rcll}
e & ::= & x_R & \text{(role-local variables)} \\
 & & \lceil t \rceil_R & \text{(replicated Coq terms)} \\
 & & [t_1, \ldots, t_n]_R & \text{(vector of Coq terms)} \\
 & & e_1\, e_2 & \text{(application)}
\end{array}
$$

Sync Programs

$$
\begin{array}{rcll}
p & ::= & \texttt{ret}\,\{R_1 \mapsto e_1, \ldots, R_n \mapsto e_n\} & \text{(return a record)} \\
 & & \texttt{let}\ x \coloneqq p_1\ \texttt{in}\ p_2 & \text{(sequencing)} \\
 & & \texttt{comm}\ c\ e_m\ e_d\ e_f & \text{(communication)}
\end{array}
$$

Sync is realized via a shallow embedding that inherits all of Coq's terms at the expression level and adds *programs* (p) that operate over a record mapping roles to vectors of values, one vector element for each node in the given role. Critically, the only way for nodes to interact with each other is through communication. Thus, a Sync program p is essentially a sequence of communication steps, where sequencing is accomplished through `let` and terminated with a `ret`. The `ret` operation takes a record of expressions, one for each role, and those expressions

are calculated for each node within that role, by mapping the expression's denotation over a vector of environments, one for each node. Thus, the variables bound in a Sync `let` are records mapping roles to vectors of values. In order to ensure that one node cannot access values from another node, `let`-bound variables are restricted within expressions: a given role can only access its vector of the record, and implicitly, a given node can only access its element of the vector. In our Coq formalism, we use parametric higher-order abstract syntax (PHOAS) [28] to represent variables and binding and enforce these constraints.

The program `comm` $c\ e_m\ e_d\ e_f$ involves a channel c, a sender role, and a receiver role. When executed, all of the sender nodes calculate a message e_m and send that message's value to all of the nodes in the receiver role via the channel c. The receiver role's nodes each run a message handler that accepts the incoming messages and combines them with a folding function e_f and default value e_d. The channel c is used by the sender and the receiver nodes to distinguish messages for different rounds of communication. Our type system enforces that channel names are unique and used in a particular order. At the implementation level, this uniqueness is realized through sequence numbers.

2.2.2 Typing for Sync Programs

Our typing rules are largely standard, but ensure that each expression is situated for a given role and each channel c is used exactly once in a specified order. The typing judgment for expressions has the form $\Gamma \vdash_R e : \tau$ where Γ is a context

mapping variables to record types indexed by roles, and is defined as follows:

$$\textsc{Var}\ \frac{\Gamma(x) = \{R_1 : \tau_1, \ldots, R_n : \tau_n\}}{\Gamma \vdash_{R_i} x_{R_i} : \tau_i} \qquad \textsc{Lift}\ \frac{t : \tau}{\Gamma \vdash_R \lceil t \rceil_R : \tau}$$

$$\textsc{Vector}\ \frac{t_1 : \tau \cdots t_n : \tau}{\Gamma \vdash_R [t_1, \ldots, t_n] : \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash_R e_1 : \tau' \to \tau \qquad \Gamma \vdash_R e_2 : \tau'}{\Gamma \vdash_R e_1\, e_2 : \tau}$$

For variables, we find the record type associated with the variable by the context Γ, and then extract the type associated with the given role. Note that it is not necessary for a given role to be present for all variables, so x_R is only well-formed when x is in $\mathsf{Dom}(\Gamma)$ and R is in $\mathsf{Dom}(\Gamma(x))$. For inherited Coq terms, we simply ascribe them the type that Coq gives them, but situated at a given role. Finally, application is typed as expected.

The typing judgment for programs has the form $\Delta; \Gamma \vdash_{\mathcal{R}} p : \{R_i : \tau_i\}$ where Δ is an *ordered* channel context mapping channel names c to a triple (S, R, τ) of a sending role, a receiving role, and a type for messages to be sent on the channel, and where $\mathcal{R}$ is a set of roles that can be used in the program. The judgment is defined with the following rules:

$$\textsc{Ret}\ \frac{\Gamma \vdash_{R_i} e_i : \tau_i \quad R_i \in \mathcal{R}}{\emptyset; \Gamma \vdash_{\mathcal{R}} \mathtt{ret}\, \{R_i \mapsto e_i\} : \{R_i : \tau_i\}}$$

$$\textsc{Let}\ \frac{\Delta_1; \Gamma \vdash_{\mathcal{R}} p_1 : \tau_1 \quad \Delta_2; \Gamma[x : \tau_1] \vdash_{\mathcal{R}} p_2 : \tau_2 \quad \mathsf{Dom}(\Delta_1) \cap \mathsf{Dom}(\Delta_2) = \emptyset}{\Delta_1 +\!\!+ \Delta_2; \Gamma \vdash_{\mathcal{R}} \mathtt{let}\ x\ \mathtt{:=}\ p_1\ \mathtt{in}\ p_2 : \tau_2}$$

$$\textsc{Comm}\ \frac{\Gamma \vdash_S e_m : \tau_m \quad \Gamma \vdash_R e_d : \tau \quad \Gamma \vdash_R e_f : \tau \to \tau_m \to \tau \quad S, R \in \mathcal{R}}{[c : (S, R, \tau_m)]; \Gamma \vdash_{\mathcal{R}} \mathtt{comm}\ c\ e_m\ e_d\ e_f : \{R : \tau\}}$$

For `ret`, we simply check that each expression in the returned record is well-

typed at the corresponding role. In this case, the channel context is empty since this command does not do any communication. For $\texttt{let}\ x := p_1\ \texttt{in}\ p_2$, we check that p_1 has a (record) type τ_1 and then extend Γ with the assumption that $x : \tau_1$ to check that p_2 has the (record) type τ_2. Note that here, the channel contexts are checked to be disjoint and are appended in order. For $\texttt{comm}\ c\ e_m\ e_d\ e_f$, the channel context has *only* c associated with a sending role S, receiving role R, and message type τ_m. We check that the message expression (e_m) is typed at the sending role S with type τ_m, and that the default (e_d) and combining function (e_f) expressions are typed at the receiving role R as τ and $\tau \to \tau_m \to \tau$ respectively. Finally, the communication step only returns τ values for the nodes of the receiver role, so the resulting record type is the singleton $\{R : \tau\}$.

2.2.3 Denotational Semantics for Sync

Notation:

For any natural number n and type τ, $\mathsf{Vec}\, n\, \tau$ describes a sequence of n elements of type τ. We use $\vec{v}$ to make clear the value is a vector value and write $\vec{v}@i$ for the operation that projects element i from vector $\vec{v}$. The operations $\mathsf{map}\ f\ \vec{v}$, $\mathsf{foldl}\ f\ d\ \vec{v}$, $\mathsf{zip}\ \vec{v}_1\ \vec{v}_2$, $\mathsf{unzip}\ \vec{v}$ are the usual map, fold, zip, and unzip for vectors. The operation $\mathsf{const}\, n\, v$ replicates the value v n times to produce a vector.

For any set or type τ, $\mathcal{P}(\tau)$ denotes the power set of τ. We also use the power

set as a monad with the following notations.

$\mathcal{P}(f) : \mathcal{P}(A) \to \mathcal{P}(B)$	lifting function f into its set version by applying f to all elements
$\mathsf{singleton}(x) = \{x\} : \mathcal{P}(\tau)$	the singleton set, also the monadic return
$\gg\!= : \mathcal{P}(A) \to (A \to \mathcal{P}(B))$	the bind applies the function to all elements and takes the union

For any record e, we write $e@R$ to extract the record field named by R. We also have a shorthand $\Pi : \mathsf{Vec}\, n\, (\mathcal{P}(\tau)) \to \mathcal{P}(\mathsf{Vec}\, n\, \tau)$ that forms a set of vectors from a vector of sets by enumerating all combinations of each element.

Configuration

The denotation of a Sync program is parameterized by a *configuration*, which is defined as follows:

Definition 2.2.2 (Distributed System Configuration)**.** Given a role set $\mathcal{R}$, a configuration $\mathcal{C}$ defines the following for each role $R \in \mathcal{R}$:

- The total number of nodes n_R and an upper bound on the number of faults to tolerate f_R. Those should match the static parameters used to define the protocol.
- A set of g_R nodes that are correct or follow the protocol until crash $\mathsf{Good}_R = \{r_1, \ldots, r_{g_R}\}$.
- A set of b_R Byzantine nodes Byz_R, $b_R \leq f_R$ and $b_R + g_R = n_R$.

Given a configuration, we define a relation Netwk_R between a vector of messages sent by nodes in Good_R to a list of messages that a receiver node may

possibly receive. We use this relation to calculate all of the "bad" things that can happen in the network, such as dropping a message, permuting the order in which messages are received, or injecting Byzantine messages. In our setting, Netwk_R is defined as a predicate (`prop`) on a vector and list of messages as follows:

$$\mathsf{Netwk}_R \coloneqq \lambda(\vec{v} : \mathsf{Vec}\, g_R\, \tau).\ (\mathsf{add_any}\, \vec{v}\, b_R) \gg\!= \mathsf{perm} \gg\!= (\lambda \vec{v}.\ \mathsf{trunc}\, \vec{v}\, (n_R - f_R))$$

The term $\mathsf{add_any}\, \vec{v}\, b_R$ produces the set of all vectors that add up to b_R arbitrary values of type τ to the end of $\vec{v}$. The added messages model those sent by a Byzantine node and can take any value. Note that Byzantine nodes may also send multiple messages, but we assume a correct receiver will only process at most one message from each sender node. The term perm produces the set of all permutations of the input vector and models the arbitrary order in which messages may be received. Finally, the term $\mathsf{trunc}\, \vec{v}\, (n_R - f_R)$ produces all prefixes of $\vec{v}$ whose length is at least $n_R - f_R$. This models a receiver node not receiving some of the messages. Due to the asynchronous network, a node cannot distinguish between a message being delayed and a message never sent. So in practice, a receiver only waits for a certain number of messages before continuing. In our case, the maximum number of messages to wait for is $n_R - f_R$ because a correct node always sends its message, and the total number of faulty nodes, either crashing or Byzantine, is bounded by f_R.

This definition is an over-approximation of the possible behaviors we can observe. For instance, this definition allows Byzantine nodes to generate messages with "secrets" that they may not know. Nevertheless, the over-approximation ensures that we cover all of the cases that need to be considered and is yet precise enough that we can still prove useful properties.

Denotational Semantics

We begin by giving a denotation to Sync program types and in particular, we define:

$$[\![\{R_1 : \tau_1, \ldots, R_n : \tau_n\}]\!] = \{R_1 : \mathsf{Vec}\, g_{R_1}\, \tau_1, \ldots, R_n : \mathsf{Vec}\, g_{R_n}\, \tau_n\}$$

That is, $Sync$ records are translated to records of vectors, where the lengths of the vectors are determined by the parameters of the configuration. Next, we lift the translation to variable contexts by defining:

$$[\![[x_1 : \tau_1, \ldots, x_n : \tau_n]]\!] = \{x_1 : [\![\tau_1]\!], \ldots, x_n : [\![\tau_n]\!]\}$$

Thus, environments are represented as records mapping variable names to values, where the values are records mapping roles to vectors. Now we can define the denotation of (derivations of) terms as a recursively defined meta-function with type:

$$[\![\Delta; \Gamma \vdash_{\mathcal{R}} p : \tau]\!] : [\![\Gamma]\!] \to \mathcal{P}([\![\tau]\!])$$

which is defined as follows:[1]:

Definition 2.2.3 (Denotational Semantics of Sync Programs)**.**

$$\begin{aligned}
[\![\texttt{ret}\, \{R_i \mapsto e_i\}]\!] &= \lambda v.\, \mathsf{singleton}\, \{R_i \mapsto [\![e_i]\!]v\} \\
[\![\texttt{let}\ x \coloneqq p_1\ \texttt{in}\ p_2]\!] &= \lambda v.\, [\![p_1]\!]v \gg\!\!= (\lambda y.\, [\![p_2]\!](v[x \mapsto y])) \\
[\![\texttt{comm}\ c\ e_m\ e_d\ e_f]\!] &= \lambda v.\, \mathsf{let}\ \texttt{msgs} \coloneqq [\![e_m]\!]v\ \mathsf{in} \\
&\quad\ \mathsf{let}\ \texttt{netmsgs} \coloneqq \mathsf{Netwk}_S(\texttt{msgs})\ \mathsf{in} \\
&\quad\ \mathsf{let}\ \texttt{pairs} \coloneqq \mathsf{zip}([\![e_f]\!]v)\ ([\![e_d]\!]v)\ \mathsf{in} \\
&\quad\ \mathsf{let}\ \mathsf{app} \coloneqq \lambda(f, d).\, \mathcal{P}(\mathsf{foldl}\ f\ d)\ \texttt{netmsgs}\ \mathsf{in} \\
&\quad\ \mathcal{P}(\lambda x.\{R \mapsto x\})(\Pi(\mathsf{map}\ \mathsf{app}\ \texttt{pairs}))
\end{aligned}$$

[1] Formally, the definition is over derivations of the typing judgment, but to simplify the presentation, we present the definition over Sync syntax

The definition relies upon a denotation for Sync expressions which has type:

$$[\![\Gamma \vdash_R e : \tau]\!] : [\![\Gamma]\!] \to \mathsf{Vec}\, g_R\, \tau$$

and is defined by:

$$\begin{aligned}
[\![x@R]\!] &= \lambda v.\ v@x@R \\
[\![\lceil t \rceil_R]\!] &= \lambda v.\ \mathsf{const}\ g_R\ t \\
[\![[t_1, \ldots, t_n]_R]\!] &= \lambda v.\ [t_1, \ldots, t_n] \quad (n = g_R) \\
[\![e_1\ e_2]\!] &= \lambda v.\ \mathsf{map}\ (\lambda(f, x).f\ x)\ (\mathsf{zip}\ [\![e_1]\!]v\ [\![e_2]\!]v)
\end{aligned}$$

Working backwards, the denotation of a variable simply extracts that variable from the environment ($v@x$) which yields a record of vectors. We then extract the vector of the corresponding role R at which the expression is situated. The denotation of an embedded Coq term t replicates the term as a vector for each of the good nodes at the given role. The denotation of a vector of Coq terms is just that vector, provided its length agrees with the number of nodes in the given role. In practice, we only use vector literals in the meta-theory so this assumption is easy to discharge. The denotation of an application $e_1\, e_2$ is given by calculating the denotation of e_1 and e_2 respectively, which should return a vector of functions and a vector of values respectively. We zip the two vectors into a vector of pairs, and then map the apply function across the resulting vector.

The denotation of programs is more involved because we return a set of records of vectors. We use the powerset monad to make the definitions a little simpler. The cases for `ret` and `let` straightforwardly translate into the singleton and bind in the powerset monad. For $\mathtt{comm}\ c\ e_m\ e_d\ e_f$, we first compute a vector of messages from all of the good senders. We then use Netwk, which can be treated as a function with the powerset monad, to calculate a set of lists of

possible messages, which models the effect of the network. Recall that this set includes the original vector of messages, but also added Byzantine messages, permutations, and dropped messages. We wish to take each possible list of messages in netmsgs and fold each receiver's combining function and default value over the list of messages. This is accomplished through map app pairs but this returns a vector of sets of values, so we use Π to convert this to a set of vectors of values. Finally, we must place each of these vectors into a record with a field for the receiver role ($\mathcal{P}(\lambda x.\{R \mapsto x\})$).

It is easy to see that the denotation respects monadic laws, so for instance, lets can be flattened:

$$[\![\texttt{let}\ x_1 \coloneqq (\texttt{let}\ x_2 \coloneqq p_2\ \texttt{in}\ p_3)\ \texttt{in}\ p_4]\!] = [\![\texttt{let}\ x_2 \coloneqq p_2\ \texttt{in}\ \texttt{let}\ x_1 \coloneqq p_3\ \texttt{in}\ p_4]\!]$$

In fact, every program is equivalent to one in a "let-comm" normal form:

$$p \ ::= \ \texttt{ret}\ \{R_i \mapsto e_i\} \mid \texttt{let}\ x \coloneqq \texttt{comm}\ c\ e_m\ e_d\ e_f\ \texttt{in}\ p$$

which we will leverage in the proof.

An Example Program

We go through the SimpleVote program in Figure 2.1 to give more intuition. The SimpleVote program is typed by

$$[c : (R, L, \mathbb{B})]; [x : \{L : \mathbb{B}, R : \mathbb{B}\}] \vdash_{\{L,R\}} \texttt{SimpleVote} : \{L : \texttt{option}\ \mathbb{B}\}$$

It requires a single Boolean value as input at all good nodes and produces an option Boolean value at the leader as the output.

For demonstration, we use a concrete configuration where there is a single

leader node and $n_R = 4$ replica nodes, r_1, r_2, r_3, r_B, of which r_B is Byzantine and $f_R = b_R = 1$.

The denotation of `SimpleVote` is thus a function of type

$$\{x : \{L : \mathsf{Vec}\,1\,\mathbb{B}, R : \mathsf{Vec}\,3\,\mathbb{B}\}\} \to \mathcal{P}(\{L : \mathsf{Vec}\,1\,(\texttt{option}\,\mathbb{B})\})$$

We use $\top$ and $\bot$ to represent Boolean true and false. Suppose the concrete input is $\{x \mapsto \{L \mapsto [\top], R \mapsto [\top, \top, \bot]\}\}$, which means the leader's input is $\top$ and the replicas' inputs are $[\top, \top, \bot]$ for r_1, r_2, and r_3 respectively. All possible lists of messages that the leader node could receive are given by the Netwk_R relation. Let $\top(r_i)$ represent a value $\top$ sent by node r_i. The exact set is all permutations of

$$\{\top(r_1), \top(r_2), \top(r_B)\}, \{\top(r_1), \bot(r_3), \top(r_B)\}, \{\top(r_2), \bot(r_3), \top(r_B)\},$$
$$\{\top(r_1), \top(r_2), \bot(r_B)\}, \{\top(r_1), \bot(r_3), \bot(r_B)\}, \{\top(r_2), \bot(r_3), \bot(r_B)\},$$
$$\{\top(r_1), \top(r_2), \bot(r_3), \top(r_B)\}, \{\top(r_1), \top(r_2), \bot(r_3), \bot(r_B)\}, \{\top(r_1), \top(r_2), \bot(r_3)\}$$

Because the input to the leader node is $\top$, `fcnteq` counts the number of $\top$ elements in the received messages, and the order of the elements does not influence its output. Thus, the set of possible communication results is $\{1, 2, 3\}$. `calc_dec` outputs $\mathsf{Some}\ \top$ if there are at least $2 = (n - 2 * f)$ votes for $\top$ and None otherwise. So, the set of possible outputs of the leader node is $\{\mathsf{Some}\ \top, \mathsf{None}\}$ given this particular input. This leads to the set of possible final outputs $\{\{L \mapsto [\mathsf{Some}\ \top]\}, \{L \mapsto [\mathsf{None}]\}\}$.

2.3 Case Studies

In this section, we showcase the usage of Ironwood with two concrete consensus protocols. We choose consensus protocols as our primary target because they are critical building blocks for many distributed system algorithms, such as atomic broadcast, replicated state machines, and blockchains, but also because they are known to be error-prone [1, 89]. The lines of code of our Coq development are summarized in Table 2.1. We sketch the steps we followed in our mechanized

Component	LoC
Sync Definition	167
Sync Utility	83
Bosco	622
SeqPaxos	1359
Async Definition and Compilation	144
Runtime (OCaml)	89[2]

Table 2.1: Lines of Code of Ironwood Components

proof using the semantic definitions.

2.3.1 Modeling Non-terminating Consensus Protocols

In general, the goal of a consensus protocol is to have a collection of nodes agree on a common value, even in the presence of faulty nodes and unpredictable network behaviors. The famous FLP impossibility result [44] states that in a fully asynchronous network, there is no non-trivial consensus protocol that is safe, guaranteed to terminate, and can tolerate even a single crash failure. Because of this, consensus protocols are usually not guaranteed to terminate without making stronger assumptions.

[2]Cryptographic signature and related checks are not implemented

In our framework, we model protocols that execute a Sync program in a loop. Since we only consider safety properties, it is sufficient to consider the finite unwindings of the loop. We assume the loop body is a Sync program that takes in some input and produces an output that contains the input to the next iteration. We use the construct b^k to unroll the loop k times and provide fresh channel names for all iterations.

More formally, a protocol body b is of the form $\nu c_1, \ldots, c_n.\lambda x.p$ where the free channel names in p are bound via the fresh quantifier ν and x is the input to the protocol. Given two protocol bodies b_1 and b_2, their concatenation $b_1 \mathbin{+\!\!+} b_2$ is defined as follows:

$$(\nu c_1, \ldots, c_n.\lambda x.p) \mathbin{+\!\!+} (\nu c'_1, \ldots, c'_n.\lambda x'.p') = \nu c_1, \ldots, c_n, c'_1, \ldots, c'_n.\lambda x.p[\lambda x'.p']$$

where $p[\lambda x'.p']$ is a form of monadic substitution defined by:

$$\begin{aligned} (\texttt{ret}\,\{R_i \mapsto e_i\})[\lambda x'.p'] &= p'[e_i/x'@R_i] \\ (\texttt{let}\ x \coloneqq p_1\ \texttt{in}\ p_2)[\lambda x'.p'] &= \texttt{let}\ x \coloneqq p_1\ \texttt{in}\ (p_2[\lambda x'.p']) \\ (\texttt{comm}\ c\ e_m\ e_d\ e_f)[\lambda x'.p'] &= \texttt{let}\ x' \coloneqq \texttt{comm}\ c\ e_m\ e_d\ e_f\ \texttt{in}\ p' \end{aligned}$$

Note that in the `ret` case, our syntax does not support substituting the record expression for the variable x, so we must project each role R's expression and substitute that for each occurrence of x_R. With this definition in hand, given a protocol body b, we can define b^k as:

$$\begin{aligned} b^0 &= b \\ b^{k+1} &= b \mathbin{+\!\!+} b^k \end{aligned}$$

2.3.2 Bosco

Bosco [114] stands for **B**yzantine **O**ne-**S**tep **CO**nsensus, a consensus algorithm that only performs a single round of communication per iteration. It is a symmetric protocol with only one role, which we denote as R for replica, and all nodes in the system follow the same program. We assume two global parameters: n is the total number of nodes, and f is the number of Byzantine faults to tolerate. For simplicity, we analyze the bi-value version of the protocol, where the goal is to have all nodes decide on either $\top$ or $\bot$. A single iteration of Bosco is defined below:

Definition 2.3.1 (Bi-value Bosco in Sync Syntax).

1 $\texttt{Bosco}\ (v : \{R : \mathbb{B}\}) : \{R : (\mathsf{option}\ \mathbb{B}) * \mathbb{B}\} \coloneqq$

2 $\quad \texttt{let cnts} \coloneqq \texttt{comm}\ c\ v_R\ \lceil(0,0)\rceil_R\ \lceil\texttt{fcntb}\rceil_R\ \texttt{in}$

3 $\quad \texttt{ret}\ (\lceil\texttt{mkdec}\rceil_R\ \texttt{cnts}_R)$

where

4 $\texttt{fcntb} \coloneqq \lambda\ (\texttt{cnt}_\top, \texttt{cnt}_\bot), v.\ \mathsf{if}\ v\ \mathsf{then}\ (\texttt{cnt}_\top{+}1, \texttt{cnt}_\bot)\ \mathsf{else}\ (\texttt{cnt}_\top, \texttt{cnt}_\bot{+}1)$

5 $\texttt{mkdec} \coloneqq \lambda\ (\texttt{cnt}_\top, \texttt{cnt}_\bot).$

6 $\quad \texttt{let}\ (\texttt{newv}, \texttt{cnt}) \coloneqq \mathsf{if}\ \texttt{cnt}_\top \geq \texttt{cnt}_\bot\ \mathsf{then}\ (\top, \texttt{cnt}_\top)\ \mathsf{else}\ (\bot, \texttt{cnt}_\bot)\ \texttt{in}$

7 $\quad \mathsf{if}\ \texttt{cnt} * 2 > n + 3 * f\ \mathsf{then}\ (\mathsf{Some}\ \texttt{newv}, \texttt{newv})\ \mathsf{else}\ (\mathsf{None}, \texttt{newv})$

An iteration of the Bosco protocol requires a single Boolean value at every node as the input. On line 2, every node broadcasts its input value v_R to every other node and counts the number of received $\top$ and $\bot$. On line 3, each node computes the decision and the input to the next iteration with function `mkdec`. On line 6, in `mkdec`, a node compares the numbers of $\top$s and $\bot$s it receives and

records the value with more votes as the `newv` and the number of its occurrences as `cnt`. On line 7, each node produces a decision value `dec`, which is Some `newv` if `cnt` is greater than the threshold $\frac{n+3f}{2}$ or None if otherwise. `newv` to be passed to the next iteration as the input.

Using our semantic definitions, we prove the following two properties for Bosco:

1. (One Step): When $n > 7f$, if all correct nodes have the same input B in some iteration, then all correct nodes decide and output Some B in that iteration.
2. (Agreement): When $n > 3f$, if a correct node outputs Some B_1 in an iteration and another correct node, not necessarily a different one, outputs Some B_2 in a not necessarily different iteration, then $B_1 = B_2$.

Other properties, such as unanimity and validity, can be reasoned in a similar fashion.

Definition 2.3.2 (Bosco Configuration)**.** We assume a configuration $\mathcal{C}$ with one role R, good nodes $\mathsf{Good}_R = \{r_1, \ldots, r_{n-b}\}$, and b Byzantine nodes where $b \leq f$.

Proving the one-step property involves a single iteration, and the property is defined formally as:

Definition 2.3.3 (Strongly one-step)**.** If $n > 7f$, then for all Boolean values B, input vectors $\vec{x}$ such that $\forall r_i, \vec{x}@r_i = B$, and output values $\{R \mapsto \vec{y}\} \in [\![P]\!]\{x \mapsto \{R \mapsto \vec{x}\}\}$, we have $\forall r_i, \vec{y}@r_i = (\mathsf{Some}\ B, B)$.

Proof. We first give a lemma about Netwk_R. For convenience, we define $\#_v(\ell)$ as

the number of occurrences of v in the list ℓ.

$\#_v(\ell) \coloneqq \mathsf{foldl}\ (\texttt{fcnteq}\ v)\ 0\ \ell$ where $\texttt{fcnteq} \coloneqq \lambda v, c, v'.\ \mathsf{if}\ v == v'\ \mathsf{then}\ c+1\ \mathsf{else}\ c$

Lemma 2.3.1. For any role R and any ℓ such that $|\ell| = n_R - b_R$, we have:

$$\forall \ell' \in \mathsf{Netwk}_R(\ell), v, \#_v(\ell) - f_R \leq \#_v(\ell') \leq \#_v(\ell) + b_R$$

This lemma can be proven by following the definition of Netwk_R and $\#_v(\ell)$.

Our main proof follows the structure of the Bosco program's semantics. We start with the precondition $\forall r_i, \vec{x}@r_i = B$. By definition, line 2 applied to $\vec{x}$ reduces to:

$$\Pi(\mathsf{map}\ (\lambda(f,d).\ \mathcal{P}(\mathsf{foldl}\ f\ d)\ \mathsf{Netwk}_R(\vec{x}))\ (\texttt{const}\ (n-b)\ (\texttt{fcntb}\ (0,0)))$$

So for any output of line 2, `cnts`, we have

$$\forall r_i, \texttt{cnts}_R@r_i \in \mathcal{P}(\mathsf{foldl}\ \texttt{fcntb}\ (0,0))\ \mathsf{Netwk}_R(\vec{x})$$

By the definition of the power set monad, $\mathcal{P}(\mathsf{foldl}\ \texttt{fcntb}\ (0,0))$ applies the function to every element of the input. Thus,

$$\exists \ell \in \mathsf{Netwk}_R(\vec{x}), \texttt{cnts}_R@r_i = \mathsf{foldl}\ \texttt{fcntb}\ (0,0)\ \ell$$

By definition, $\mathsf{foldl}\ \texttt{fcntb}\ (0,0)\ \ell = (\#_\top(\ell), \#_\bot(\ell))$. Let $(\texttt{cnt}_\top, \texttt{cnt}_\bot) \coloneqq \texttt{cnts}_R@r_i$. Because of $\forall r_i, \vec{x}@r_i = B$, $\#_B(\vec{x}) = n - b$ and $\#_{\neg B}(\vec{x}) = 0$. By the lemma above, we know:

$$\texttt{cnt}_B = \#_B(\ell) \geq \#_B(\vec{x}) - f = n - b - f$$

$$\texttt{cnt}_{\neg B} = \#_{\neg B}(\ell) \leq \#_{\neg B}(\vec{x}) + b = b$$

For line 3, by the definition of `mkdec`, because $n > 7f$, we have:

$$\texttt{cnt}_B \geq n - b - f \geq n - 2f > 5f \geq b \geq \texttt{cnt}_{\neg B}$$

So, $\texttt{newv} = B$ and $\texttt{cnt} = \texttt{cnt}_B \geq n - 2f > \frac{n+3f}{2}$, which leads to the final output $(\mathsf{Some}\ B, B)$. $\square$

To formalize the agreement property, we first need some auxiliary definitions:

$$\mathsf{Step}^k(\vec{x}) \coloneqq (\llbracket \texttt{Bosco}^k \rrbracket \circ \mathcal{P}(@R) \circ \mathcal{P}(\texttt{unzip}))\{x \mapsto \{R \mapsto \vec{x}\}\}$$

$$\mathsf{Decide}_B(\vec{y}) \coloneqq \exists r_i . \vec{y}@r_i = \mathsf{Some}\ B$$

$$\mathsf{Comply}_B(\vec{y}) \coloneqq \forall r_i . \vec{y}@r_i = \mathsf{Some}\ B \vee \vec{y}@r_i = \mathsf{None}$$

Step^k runs the protocol for $k+1$ steps and extracts the results from the record. Decide_B is a predicate on a vector that holds when there is an element equal to $\mathsf{Some}\ B$, which means a node has decided B, and Comply_B is a predicate on a vector that holds when every element is $\mathsf{Some}\ B$ or None, which means each node either decides B or nothing.

We prove a stronger result that implies the agreement property.

Lemma 2.3.2 (Agreement′)**.** If $n > 3f$, then

$$\forall B, \vec{x}, (\vec{y}, \vec{z}) \in \mathsf{Step}^0(\vec{x}), \mathsf{Decide}_B(\vec{y})$$
$$\Rightarrow [\mathsf{Comply}_B(\vec{y}) \wedge \forall k, (\vec{y}_k, _) \in \mathsf{Step}^k(\vec{z}), \mathsf{Comply}_B(\vec{y}_k)]$$

Proof. Proofs like these revolve around some *univalent condition*. It is a predicate on the system state. All reachable states from a state that satisfies the predicate can only decide on one certain value. In our case, thanks to our synchronous functional semantics, the exact univalent condition is exactly the weakest precondition such that the $\texttt{newv}$ computed on line 6 can only be a certain value for all the possible sets of messages being received, as follows:

$$UC'_B(\vec{x}) \coloneqq \forall \vec{x'} \in \mathsf{Netwk}_R(\vec{x}), \mathsf{snd}(\texttt{mkdec}(\mathsf{foldl}\ \texttt{fcntb}\ (0,0)\ \vec{x'})) = B$$

In comparison, invariant-based verification techniques mostly require such predicates to be found manually.

Due to the asymmetrical nature of the $\leq$ comparison, this is equivalent to:

$$UC'_{\top}(\vec{x}) = \#_{\top}(\vec{x}) \geq \frac{n+f}{2} \quad UC'_{\bot}(\vec{x}) = \#_{\bot}(\vec{x}) > \frac{n+f}{2}$$

The decision procedure for Bosco does not utilize this asymmetry [114]. An optimization is possible by using $\geq$ instead of $>$ in the condition on line 7 only when `newv` is $\top$. Here, for simplicity, we use a slightly stronger symmetric predicate that is still sufficient to prove our result:

$$UC_B(\vec{x}) \coloneqq \#_B(\vec{x}) > \frac{n+f}{2}$$

And we prove the agreement property by proving the following:

$$\forall \vec{x}, (\vec{y}, \vec{z}) \in \mathsf{Step}^0(\vec{x}), \mathsf{Decide}_B(\vec{y}) \Rightarrow UC_B(\vec{x}) \tag{2.1}$$

$$\forall \vec{x}, k, (\vec{y}, \vec{z}) \in \mathsf{Step}^k(\vec{x}), UC_B(\vec{x}) \Rightarrow [\mathsf{Comply}_B(\vec{y}) \land UC_B(\vec{z})] \tag{2.2}$$

We prove (1) by walking through the program backward with $\mathsf{Decide}_B(\vec{y})$. Because our program is nondeterministic, walking through the program backward requires us to infer a predicate that covers all possible input values that can lead to any output values satisfying the postcondition.

- On line 3, we know that for some r_i, $\texttt{dec} = \mathsf{Some}\ B$.
- By the definition of `mkdec`, it is necessary to have $\texttt{cnt}_B > \frac{n+3f}{2}$.
- On line 2, by Lemma 2.3.1, $\#_B(\vec{x}) \geq \texttt{cnt}_B - b > \frac{n+f}{2}$, which is our goal, UC_B.

For (2), we do an induction on k. The only case that needs to be proven is $k = 0$. We walk through the program forward to infer a predicate that covers all possible outputs.

- On line 2, by Lemma 2.3.1 and $UC_B(\vec{x})$, for any r_i and any possible network, we have $\texttt{cnt}_B > \frac{n+f}{2} - f \wedge \texttt{cnt}_{\neg B} \leq \#_{\neg B}(\vec{x}) + b < n + f - \frac{n+f}{2} = \frac{n+f}{2}$. So $\texttt{cnt}_B > \texttt{cnt}_{\neg B}$.
- By the definition of `mkdec`, $\texttt{newv} = B$ and $\texttt{dec} = \textsf{Some}\ \texttt{newv} = \textsf{Some}\ B \vee \texttt{dec} = \textsf{None}$.
- On line 3, we have $\forall r_i,\ (\vec{y}@r_i = \textsf{Some}\ B \vee \vec{y}@r_i = \textsf{None}) \wedge (\vec{z}@r_i = B)$.

$\forall r_i,\ \vec{y}@r_i\textsf{Some}\ B \vee \vec{y}@r_i = \textsf{None}$ implies $\textsf{Comply}_B(\vec{y})$. $\forall r_i,\ \vec{z}@r_i = B$ implies $\#_B(\vec{z}) = n - b \geq n - f$. Because $n > 3f$, we have $\#_B(\vec{z}) > \frac{n+f}{2}$, which is UC_B. $\square$

2.3.3 Sequential Paxos

Sequential Paxos, or SeqPaxos for short, is a variant of the Synod consensus protocol used by Paxos [76]. Similar to the original single-decree Paxos protocol, it is a crash fault-tolerant protocol with a single leader node and an arbitrary number of so-called acceptor nodes, which we will call *replicas*.

Let n be the total number of replicas and f be the number of replica crashes to tolerate. The single leader may also crash. We assume the domain of the consensus value is of type $\mathbb{V}$, for which a decidable equality exists. Additionally, we assume that there is a default value for the leader to propose in each iteration,

denoted as a global variable $\texttt{default}: \mathbb{N} \rightarrow \mathbb{V}$. A single iteration of SeqPaxos defined in Sync is listed below:

Definition 2.3.4. SeqPaxos

1 $\texttt{SeqPaxos}\ (x : \{L : \mathbb{N}, R : (\mathsf{option}\ \mathbb{V}) * \mathbb{N}\})$

$\quad : \{L : (\mathsf{option}\ \mathbb{V}) * \mathbb{N}, R : (\mathsf{option}\ \mathbb{V}) * \mathbb{N}\} :=$

2 $\quad \texttt{let maxv} := \texttt{comm}\ c_1\ x_R\ \lceil(\mathsf{None}, 0)\rceil_L\ \lceil\texttt{fmaxr}\rceil_L\ \texttt{in}$

3 $\quad \texttt{let}\ p := \texttt{ret}\ \{L \mapsto \lceil\texttt{pickp}\rceil_L\ \texttt{maxv}_L\ (\lceil\texttt{default}\rceil_L\ (x_L))\}\ \texttt{in}$

4 $\quad \texttt{let}\ y := \texttt{comm}\ c_2\ (\lceil\texttt{pair}\rceil_L\ p_L\ x_L)\ x_R\ \lceil\texttt{update}\rceil_R\ \texttt{in}$

5 $\quad \texttt{let cnt} := \texttt{comm}\ c_3\ y_R\ \lceil 0\rceil_L\ (\lceil\texttt{fcnteq}\rceil_L\ x_L)\ \texttt{in}$

6 $\quad \texttt{ret}\ \{L \mapsto \lceil\texttt{pair}\rceil_L\ (\lceil\texttt{mkdec}\rceil_L\ \texttt{cnt}_L\ p_L)\ (\lceil\texttt{add}\rceil_L\ x_L\ \lceil 1\rceil_L), R \mapsto y_R\}$

where

7 $\quad \texttt{fmaxr} := \lambda\ (v, r), (v', r').\ \mathsf{if}\ r < r'\ \mathsf{then}\ (v', r')\ \mathsf{else}\ (v, r)$

8 $\quad \texttt{pickp} := \lambda\ (ov, _), d.\ \mathsf{match}\ ov\ \mathsf{with}\ |\ \mathsf{Some}\ v \Rightarrow v\ |\ \mathsf{None} \Rightarrow d\ \mathsf{end}$

9 $\quad \texttt{update} := \lambda\ _, (v, r).\ (\mathsf{Some}\ v, r)$

10 $\quad \texttt{fcnteq} := \lambda\ r, c, (_, r').\ \mathsf{if}\ r == r'\ \mathsf{then}\ c + 1\ \mathsf{else}\ c$

11 $\quad \texttt{mkdec} := \lambda\ c, p.\ \mathsf{if}\ c > f\ \mathsf{then\ Some}\ p\ \mathsf{else\ None}$

In the SeqPaxos protocol, the leader takes a single natural number, which is the current round number. The replicas take a pair of an optional $\mathbb{V}$ value and a natural number. The number is the latest round number ever received, and the value is the proposal of that round. To initiate the protocol, the leader should be given the input 1 and the replicas the dummy values None and 0.

SeqPaxos performs three rounds of communication in each iteration. In the first round of communication on line 2, all the replicas send their local value and

round number to the leader. The leader then finds the largest round number and the proposal associated with the round number. On line 3, the leader computes the proposal of the iteration. If the proposal associated with the largest round number received is $\mathsf{Some}\ v$, the proposal is v. Otherwise, the proposal is set to the default value of that iteration. On line 4, the leader broadcasts the proposal with its round number to all the replicas. For each replica, if it receives the message from the leader, it updates its local value and round number to the new local value and round number; otherwise, it keeps its old local value and round number. On line 5, all the replicas send their local value and round number to the leader again. This time, the leader counts how many replicas have updated their local value and round number to the latest proposal and round number pair. On line 6, if the number of up-to-date replicas is greater than f, the leader decides their proposal as the decision of the iteration; otherwise, no decision is made. The leader returns a pair of the decision and an incremented round number while each replica returns its latest value.

Definition 2.3.5 (SeqPaxos Configurations)**.** SeqPaxos requires a configuration $\mathcal{C}$ to have two roles, L and R. L is the leader role, $\mathsf{Good}_L = \{l\}$, $b_L = 0$. We set $f_L = 1$ to model that the leader may crash. R is the replica role, $\mathsf{Good}_R = \{r_1, \ldots, r_n\}$. We also set $b_R = 0$ and $f_R = f$ to model the assumption that up to f replicas may crash.

While we do not directly model the crash behavior, we claim the above definition covers all possible cases for SeqPaxos. This is because we are only reasoning about finite iterations, and there is no difference between a node that has crashed at some point and a node that has certain messages sent from and to it dropped by Netwk.

The special initial input to SeqPaxos is defined as $\texttt{init} = \{L \mapsto [1]; R \mapsto \vec{x}\}$ where $\vec{x}$ is n copies of $(\mathsf{None}, 0)$.

We formally state and prove the agreement property for SeqPaxos.

Definition 2.3.6 (Agreement)**.** If $n > 2f$,

$$\forall D, i, \{L \mapsto [(d_i, r_i)]; R \mapsto \vec{x}_i\} \in [\![\texttt{SeqPaxos}^i]\!](\{x \mapsto \texttt{init}\}),\;\; d_i = \mathsf{Some}\ D \Rightarrow$$
$$\forall j > i, \{L \mapsto [(d_j, _)]; R \mapsto _\} \in [\![\texttt{SeqPaxos}^{j-i-1}]\!](\{x \mapsto \{L \mapsto [r_i]; R \mapsto \vec{x}_i\}\}),$$
$$d_j = \mathsf{Some}\ D \vee d_j = \mathsf{None}$$

Proof. We have the following lemma about the inputs to any iteration i, which can be proven by induction on i and stepping through the program.

Lemma 2.3.3. $\forall i, \{L \mapsto [(_, r)]; R \mapsto \vec{x}\} \in [\![\texttt{SeqPaxos}^i]\!](\{x \mapsto \texttt{init}\})$, implies:

- $r = i + 2 \wedge$
- $\forall u, \mathsf{snd}(\vec{x}@u) < i + 2 \wedge$
- $\forall u, 0 < \mathsf{snd}(\vec{x}@u) \Rightarrow \exists v, \mathsf{fst}(\vec{x}@u) = \mathsf{Some}\ v \wedge$
- $\forall u, w, \mathsf{snd}(\vec{x}@u) = \mathsf{snd}(\vec{x}@w) \Rightarrow \vec{x}@u = \vec{x}@w$.

We again define the univalent condition UC_D for SeqPaxos as the weakest precondition[3] of line 3 always resulting in value D.

$$UC_D(\vec{x}) \coloneqq \forall \ell \in \mathsf{Netwk}_R(\vec{x}), \mathsf{fst}(\mathsf{foldl}\ \texttt{fmaxr}\ (\mathsf{None}, 0)\ \ell) = \mathsf{Some}\ D$$

[3]We treat `default` as some value that cannot be used outside of the particular round at runtime.

For the agreement property, we prove the following:

$$\forall D, i, \{L \mapsto [(d_i, _)]; R \mapsto \vec{x}_i\} \in [\![\texttt{SeqPaxos}^i]\!](\{x \mapsto \texttt{init}\}),$$
$$d_i = \textsf{Some } D \Rightarrow UC_D(\vec{x}_i) \tag{1}$$
$$\forall D, i, \{L \mapsto _; R \mapsto \vec{x}_i\} \in [\![\texttt{SeqPaxos}^i]\!](\{x \mapsto \texttt{init}\}), UC_D(\vec{x}_i) \Rightarrow$$
$$\forall j > i, \{L \mapsto [(d_j, _)]; R \mapsto \vec{x}_j\} \in [\![\texttt{SeqPaxos}^{j-i-1}]\!](\{x \mapsto \{L \mapsto [r_i]; R \mapsto \vec{x}_i\}\}),$$
$$(d_j = \textsf{Some } D \vee d_j = \textsf{None}) \wedge UC_D(\vec{x}_j) \tag{2}$$

It is worth noting that our UC predicate is weaker than the strongest postcondition of lines 5 and 6 deciding D, which means the protocol is actually safe to decide in more cases. An optimization for the original Paxos protocol that utilizes this is described in [49]. Here, we prove the property with this precise condition, allowing optimizations that weaken the decision condition to be verified compositionally and reuse the proof of (2).

To prove (1), we work backward to find all possible intermediate cases that may lead to $d_i = \textsf{Some } D$ and then prove they entail $UC_D(\vec{x}_i)$.

- By Lemma 2.3.3, the round number input for the round that produces $\vec{x}_i$ is $i + 1$.
- By the definition of `mkdec` on line 11, we know on line 6, for the single leader node l, $\texttt{cnt} > f \wedge p = D$.
- On line 5, we have

$$\exists \ell \in \textsf{Netwk}_R(\vec{x}_i), \textsf{foldl } (\texttt{fcnteq } (i+1))\ 0\ \ell > f$$

- By line 4, Lemma 2.3.1, and $b_R = 0$, $\#_{(\textsf{Some } D, i+1)}(\vec{x}_i) > f - b_R = f$.

- Unfolding the definition of UC_D, we need to prove:

  $$\forall \ell \in \mathsf{Netwk}_R(\vec{x}_i), \mathsf{fst}(\mathsf{foldl}\ \mathtt{fmaxr}\ (\mathsf{None}, 0)\ \ell) = \mathsf{Some}\ D$$

  By Lemma 2.3.1 again, $\#_{(\mathsf{Some}\ D, i+1)}(\ell) > f - f = 0$. So there exists w, $w \in \ell = (\mathsf{Some}\ D, i+1)$. By Lemma 2.3.3,

  $$\forall u, \mathsf{snd}(\vec{x}_i@u) \leq i+1 \wedge \mathsf{snd}(\vec{x}_i@u) = i+1 \Rightarrow \vec{x}_i@u = \vec{x}_i@w = (\mathsf{Some}\ D, i+1)$$

  which means $i+1$ is the largest round number, and every replica with round number $i+1$ has value $(\mathsf{Some}\ D, i+1)$. So by the definition of `fmaxr`,

  $$\mathsf{foldl}\ \mathtt{fmaxr}\ (\mathsf{None}, 0)\ \ell = (\mathsf{Some}\ D, i+1)$$

To prove (2), we perform induction on j. It suffices to prove for a single iteration, and we do a forward pass through the program.

- By Lemma 2.3.3, the round number output for the round that produces $\vec{x}_i$ is $i+2$.
- For lines 2 and 3, by the definition of UC_D, we have $p = D$. This implies $\mathtt{dec} = \mathsf{Some}\ D \vee \mathtt{dec} = \mathsf{None}$ on line 6.
- On line 4, our goal is to prove UC_D holds on the new values of the replicas, $y@R$. By the definition of `update` and Lemma 2.3.3,

  $$\forall u, y@R@u = (\mathsf{Some}\ D, i+2) \vee y@R@u = \vec{x}_i@u$$

  So each node nondeterministically picks between switching to the new value $(\mathsf{Some}\ D, i+2)$ and keeping the old value $\vec{x}_i@u$. This property is preserved by the Netwk relation, more formally:

  $$\forall \ell \in \mathsf{Netwk}_R(y@R), \exists \ell' \in \mathsf{Netwk}_R(\vec{x}_i), \forall u, \ell@u = (\mathsf{Some}\ D, i) \vee \ell@u = \ell'@u$$

  For any $\ell \in \mathsf{Netwk}_R(y@R)$, there are two cases.

- If $(\mathsf{Some}\ D, i+2) \in \ell$, then $\mathsf{foldl}\ \texttt{fmaxr}\ (\mathsf{None}, 0)\ \ell = (\mathsf{Some}\ D, i+2)$.
- Otherwise, all nodes whose messages were received did not update their local values. Thus, $\ell = \ell'$ and $\ell \in \mathsf{Netwk}_R(\vec{x}_i)$. Because of $UC_D(\vec{x}_i)$, $\mathsf{fst}(\mathsf{foldl}\ \texttt{fmaxr}\ (\mathsf{None}, 0)\ \ell) = \mathsf{Some}\ D$. Thus, $UC_D(\vec{x}_{i+1} = y@R)$.

□

2.4 Low-Level Semantics

In this section, we define Async, a labeled transition system that serves as a low-level operational semantics for general distributed systems, and that serves as a target language for compiling Sync programs. Async provides a formal basis for proving the adequacy of the denotational reasoning we propose for Sync programs.

We construct Async by composing multiple instances of local labeled transition systems, Async nodes and Channel. An Async node models the behaviors of an individual non-Byzantine node in the system while a Channel models the influence of the network and Byzantine nodes in a single logical round of communication.

2.4.1 Async Node Semantics

Recall that a labeled transition system $\langle S, \Lambda, \rightarrow \rangle$ includes a set of states S, a set of labels Λ, and a transition relation $\rightarrow$ over $S \times \Lambda \times S$ that relates a starting state, a label, and a next state. A behavior or *trace* of a labeled transition system from

a specific initial state is a sequence of labels and states where each element is permitted by $\to$ from the previous state, given the label. We write $s \xrightarrow{\ell} t$ for when $(s, \ell, t) \in \to$. If L is a sequence of labels, we write $s \xrightarrow{L} t$ to mean if L is empty, then $t = s$, and if $L = \ell :: L'$, then there exists a t' such that $s \xrightarrow{\ell} t'$ and $t' \xrightarrow{L'} t$.

The core of a Async node is a program represented by the following definition in Coq, which we will use to represent states in our transition system for nodes.

Definition 2.4.1 (Async)**.**

```
Inductive Async(T : Type) : Type :=
  | return : T → Async T
  | sendThen : ∀c : Channel, msg_t(c) → Async T → Async T
  | rcvThen : ∀c : Channel, (list(msg_t c) → Async T) → Async T.


Fixpoint bind{T U}(e : Async T)(f : T → Async U) :=
  match e with
  | return v ⇒ f v
  | sendThen c m k ⇒ sendThen c m (bind k f)
  | rcvThen c g ⇒ rcvThen c (λ m. bind (g m) f)
  end.


Definition send c m := sendThen c m (return tt).
Definition receive c (d : T) (f : T → msg_t c → T) :=
  rcvThen c (λm. return (foldl f d m)).
```

$\mathsf{Async}\ T$ is a recursively defined type, which describes three kinds of computations: returning a value of type T, sending a message to channel c and then continuing with another computation, and receiving from a channel c a list of messages, which are fed to a function to produce a continuation.

Some auxiliary operations will be needed to build Async values below. The operation $\mathsf{bind}\ t\ f$ "appends" f onto the leaves of Async t. More properly, the leaves of t must be of the form $\mathsf{return}\ e$, and the bind replaces these leaves with $f\ e$. The definitions for send and $\mathsf{receive}$ build corresponding Async trees with trivial continuations that immediately return. Note that bind and return form a monad.

As an example, Async code for the two roles in the `SimpleVote` program can be written as shown in Figure 2.1 part (b), using the notation $\mathsf{let}\ x := e_1\ \mathsf{in}\ e_2$ to represent $\mathsf{bind}\ e_1\ (\lambda\, x.e_2)$.

To give meaning to the Async, we formulate a labeled transition system $\langle S_t, \Lambda_t, \rightarrow_t \rangle$:

Definition 2.4.2 (Async)**.** The state of Async, S_t is a Async value of some return type T.

Λ_t has two kinds of labels:

$$l_t ::= \mathsf{send}\ c\ v \mid \mathsf{receive}\ c\ \vec{v}$$

The transition relation is defined by:

$$(\textsc{Send})\ \mathsf{sendThen}\ c\ v\ k \xrightarrow{\mathsf{send}\ c\ v} k \qquad (\textsc{Receive})\ \mathsf{rcvThen}\ c\ f \xrightarrow{\mathsf{receive}\ c\ \vec{v}} f(\vec{v})$$

To get the initial state for a given node, we now define a translation from

Sync programs and roles down to Async programs.

Definition 2.4.3 (Compiling Sync to Async)**.** Given a record type $\tau = \{R_1 : \tau_1, \ldots, R_n : \tau_n\}$ we define $(\!| \tau |\!)_{R_i} = \tau_i$. When R is not a field of τ, we take $(\!| \tau |\!)_R$ to be unit. Given a variable context $\Gamma = [x_1 : \tau_1, \ldots, x_n : \tau_n]$ we define $(\!| \Gamma |\!)_R$ to be $[x_1 : (\!| \tau_1 |\!)_R, \ldots, x_n : (\!| \tau_n |\!)_R]$. Then, given a Sync program p such that $\Delta; \Gamma \vdash_{\mathcal{R}} p : \tau$ and a role $R \in \mathcal{R}$ and node i in role R, the compilation of p at role R and node i denoted by $(\!| p |\!)_{R,i}$ yields a term t such that $(\!| \Gamma |\!)_R \vdash t : \mathsf{Async}(\!| \tau |\!)_R$ and is defined by:

$$(\!| \texttt{ret}\,\{R_j \mapsto e_j\} |\!)_{R,i} \quad = \quad \begin{cases} \mathsf{return}\ (\!| e_j |\!)_{R,i} & R = R_j \\ \mathsf{return\ tt} & \forall j, R \neq R_j \end{cases}$$

$$(\!| \texttt{let}\ x \coloneqq p_1\ \texttt{in}\ p_2 |\!)_{R,i} \quad = \quad (\!| p_1 |\!)_{R,i} \gg\!= \lambda x.(\!| p_2 |\!)_{R,i}$$

$$(\!| \texttt{comm}\ c\ e_m\ e_d\ e_f |\!)_{R,i} \quad = \quad \begin{cases} \mathsf{let}\ _ \leftarrow \mathsf{send}\ c\ (\!| e_m |\!)_{R,i}\ \mathsf{in} \\ \quad \mathsf{receive}\ c\ (\!| e_d |\!)_{R,i}\ (\!| e_f |\!)_{R,i} \\ \backslash\!\backslash\ R \text{ sending and receiving} \\ \mathsf{send}\ c\ (\!| e_m |\!)_{R,i} \\ \backslash\!\backslash\ R \text{ sending but not receiving} \\ \mathsf{receive}\ c\ (\!| e_d |\!)_{R,i}\ (\!| e_f |\!)_{R,i} \\ \backslash\!\backslash\ R \text{ receiving but not sending} \\ \mathsf{return\ tt} \\ \backslash\!\backslash\ R \text{ not sending nor receiving} \end{cases}$$

The definition assumes a translation for expressions $(\!| e |\!)_{R,i}$ which simply

removes the R from variables and terms:

$$\begin{aligned} (\!|\, x_R \,|\!)_{R,i} &= x \\ (\!|\, \lceil t \rceil_R \,|\!)_{R,i} &= t \\ (\!|\, [t_1, \ldots, t_n] \,|\!)_{R,i} &= t_i \\ (\!|\, e_1\ e_2 \,|\!)_{R,i} &= (\!|\, e_1 \,|\!)_{R,i}\ (\!|\, e_2 \,|\!)_{R,i} \end{aligned}$$

Finally, assuming a top-level program p is closed, then given a role R and node i for that role, we can form a closed Async program for the initial state of node i by computing $(\!|\, p \,|\!)_{R,i}$.

Of course, the execution of a single Async program requires an environment that includes other nodes and some model that captures the intuitive behavior for communication mentioned above. To realize this, our next step is to build a transition system that models communication channels and then to compose Async node and channel transition systems into a global transition system.

2.4.2 Single-use Channel Semantics

A *channel* is a common abstraction used to describe the network. Instead of being modeled as a single entity, the network is logically divided into separate communication channels, each serving only a fixed set of senders and receivers. Channels simplify reasoning through a separation of concerns. We break down each of the multi-use channels into multiple single-use channels. Each channel models a single logical round of communication, and different channels correspond to different logical communication steps. We associate channels with statically known information and give them a dynamic operational semantics. The list of static information about a channel is listed as follows:

Definition 2.4.4 (Static Channel Information)**.** Assume we are given a role set $\mathcal{R}$, a configuration $\mathcal{C}$ for $\mathcal{R}$, and a channel context Δ. Recall that each channel appears at most once in Δ and is associated with a message type τ and a sender role and a receiver role, which can be the same role. We use $\mathsf{msg_t}_\Delta(c)$ to represent the type τ associated with c in Δ.

Suppose c is a channel associated with sender role S and receiver role R. Then $\mathsf{sender}(c)$ denotes the set of non-Byzantine nodes in configuration $\mathcal{C}$ associated with role S; $\mathsf{Byz_sender}(c)$ denotes the set of Byzantine sender nodes in $\mathcal{C}$ associated with role S; and $\mathsf{receiver}(c)$ denotes the set of non-Byzantine receiver nodes in $\mathcal{C}$ associated with role R.

We also define $\mathsf{Netwk}_{\mathsf{Async}}$, which models the effect of asynchrony of the network in the same language of Netwk_R.

$$\mathsf{Netwk}_{\mathsf{Async}} : \mathsf{list}\ \mathsf{msg_t}(c) \to \mathcal{P}(\mathsf{list}\ \mathsf{msg_t}(c)) \coloneqq \lambda l.\ (\mathsf{perm}\ l) \gg\!= (\lambda l.\ \mathsf{trunc}\ l\ (n_S - f_S))$$

$\mathsf{Netwk}_{\mathsf{Async}}$ is a function that maps a list of messages sent to a receiver node to the set of possible lists of messages the receiver could possibly receive. Under our network assumptions, it only performs two operations: permutation, which models message reordering in transit, and truncation of the list to a prefix of at least the lower bound on the number of correct sender nodes, i.e., the nodes that are neither Byzantine nor crash. The effects of Byzantine nodes are separately captured by the transition relation.

Given a channel c with the static information defined above, we define Channel as the labeled transition system $\langle S_c, \Lambda_c, \to_c \rangle$ below.

Definition 2.4.5 (Channel)**.** A channel state $s \in S_c$ is a tuple of four components:

1. $F_s \subseteq \mathsf{sender}(c)$, the set of non-Byzantine senders who have performed their send action.

2. $F_r \subseteq \mathsf{receiver}(c)$, the set of non-Byzantine receivers who have performed their receive action.

3. $F_b : \mathsf{receiver}(c) \to \mathcal{P}(\mathsf{Byz_sender}(c))$, for each non-Byzantine receiver node, a set of Byzantine sender nodes who have sent the receiver a message.

4. $M : \mathsf{receiver}(c) \to \mathsf{list}\ \mathsf{msg_t}(c)$, for each receiver a list of message payloads that have been sent to it.

Λ_c has three kinds of labels:

$$l_c ::= \mathsf{send}\ s\ v \mid \mathsf{byz_send}\ sb\ r\ v \mid \mathsf{receive}\ r\ \vec{v}$$

where $s \in \mathsf{sender}(c)$, $sb \in \mathsf{Byz_sender}(c)$, $r \in \mathsf{receiver}(c)$, $v : \mathsf{msg_t}(c)$, and $\vec{v} : \mathsf{list}\ \mathsf{msg_t}(c)$.

$\mathsf{send}\ s\ v$ models a non-Byzantine sender node s broadcasting a message v to all the receiver nodes. $\mathsf{receive}\ r\ \vec{v}$ models a non-Byzantine receiver node r receiving a well-formed list of messages $\vec{v}$. $\mathsf{byz_send}\ sb\ r\ v$ models a Byzantine sender node sb sends a single message of content v to a single non-Byzantine receiver node r.

$\rightarrow_c$ has one rule for each kind of label.

$$\text{SEND}\ \frac{s \notin F_s \wedge F_s' = F_s \cup \{s\} \wedge \forall r, M'(r) = M(r) +\!\!+\ [v]}{\langle F_s, F_r, F_b, M \rangle \xrightarrow{\mathsf{send}\ s\ v} \langle F_s', F_r, F_b, M' \rangle}$$

$$\text{BYZ_SEND}\ \frac{\begin{array}{c}\forall r' \neq r, F_b'(r') = F_b(r') \wedge M'(r') = M(r') \\ sb \notin F_b(r) \wedge F_b'(r) = F_b(r) \cup \{sb\} \wedge M'(r) = M(r) +\!\!+\ [v]\end{array}}{\langle F_s, F_r, F_b, M \rangle \xrightarrow{\mathsf{byz_send}\ sb\ r\ v} \langle F_s, F_r, F_b', M' \rangle}$$

$$\text{RECEIVE}\ \frac{r \notin F_r \wedge (r \notin \mathsf{sender}(c) \vee r \in F_s) \wedge F_r' = F_r \cup \{r\} \wedge \vec{v} \in \mathsf{Netwk}_{\mathsf{Async}}\ M(r)}{\langle F_s, F_r, F_b, M \rangle \xrightarrow{\mathsf{receive}\ r\ \vec{v}} \langle F_s, F_r', F_b, M \rangle}$$

Channel's state and transitions are mostly about bookkeeping of which nodes have performed their send and receive actions to prevent duplicates: each non-Byzantine sender node may only broadcast once; each receiver node may only receive once; and each Byzantine sender node can only send to each receiver node at most once. In addition, if the sender role is the same as the receiver role, which means a sender node is also a receiver node, it must first perform its send action.

In the receive transition rule, we account for network-reordering and dropping effects with $\mathsf{Netwk}_{\mathsf{Async}}$. The initial state of the channel is the tuple $\langle \emptyset, \emptyset, \lambda\ _.\emptyset, \lambda\ _.[] \rangle$.

2.4.3 Composing Labeled Transition Systems

We build a *global* transition system from a set of local Async and channel systems using the following notions of interaction composition:

Definition 2.4.6 (Interaction Composition)**.** Let $\mathcal{T}_1 = \langle S_1, \Lambda_1, \rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle S_2, \Lambda_2, \rightarrow_2 \rangle$ be labeled transition systems, and let $\Lambda \subseteq (\Lambda_1 + \mathbb{1}) \times (\Lambda_2 + \mathbb{1})$. Then the interaction composition $\mathcal{T}_1 \bowtie_\Lambda \mathcal{T}_2$ is defined to be $\langle S_1 \times S_2, \Lambda, \rightarrow \rangle$ where $(s_1, s_2) \xrightarrow{(\ell_1, \ell_2)} (s_1', s_2')$ when (a) $\ell_1 = \mathtt{tt}$ and $s_1' = s_1$ or $s_1 \xrightarrow{\ell_1}_1 s_1'$, and (b) $\ell_2 = \mathtt{tt}$ and $s_2' = s_2$ or $s_2 \xrightarrow{\ell_2}_2 s_2'$.

Note that we must specify a subset of permissible labels via Λ. In other words, the choice of Λ delimits the permissible combinations of local labels and thus can be used to model the interactions between the local systems. For instance, consider a system composed of a sender and a receiver. The only label for the sender is $\mathsf{send}\ x$, for any integer x. The only label for the receiver is $\mathsf{receive}\ y$, for any integer y. By only allowing the global labels to be of the form $(\mathsf{send}\ z, \mathsf{receive}\ z)$ for the same integer z, we can model synchronous communication between the sender and the receiver.

Given the definition of interaction composition, a special case is when the sub-systems do not interact:

Definition 2.4.7 (Non-Interacting Composition)**.** Let $\mathcal{T}_1 = \langle S_1, \Lambda_1, \rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle S_2, \Lambda_2, \rightarrow_2 \rangle$ be labeled transition systems. Then the non-interacting composition $\mathcal{T}_1 \otimes \mathcal{T}_2$ is given by $\mathcal{T}_1 \bowtie_\Lambda \mathcal{T}_2$, where the labels in Λ are either of the form $(\ell_1, \mathtt{tt})$ or $(\mathtt{tt}, \ell_2)$.

In other words, the non-interacting composition only allows local transitions.

When we have an indexed sequence of transition systems $[\mathcal{T}_1, \ldots, \mathcal{T}_n]$, we write $\otimes_i \mathcal{T}_i$ to represent $\mathcal{T}_1 \otimes \cdots \otimes \mathcal{T}_n$. If $s = (s_1, \ldots, s_n)$ is a state in a composed system, we write $s@i$ to represent s_i. Similarly, if ℓ is a label in the product, then $\ell@i$ represents the label's i^{th} component.

Definition 2.4.8 (Global Compilation)**.** Given a well-typed program $\Delta; \emptyset \vdash_{\mathcal{R}} p : \tau$, and a configuration $\mathcal{C}$, we define the *global* compilation of p to be:

$$(\!| p |\!) = (\otimes_{R, i \in R} (\!| p |\!)_{R,i}) \bowtie_{\Lambda} (\otimes_{c \in \mathsf{Dom}(\Delta)} (\!| c |\!))$$

where $(\!| c |\!)$ for a channel c is the initial channel state of the transition system described in Section 2.4.2, and where we write $i \in R$ to mean i is the node identifier in the set of good nodes for role R specified by configuration $\mathcal{C}$. Additionally, we must specify Λ, the valid global labels for the composed system. A global label ℓ must be of one of the three forms:

- Send: for some c, p, and v, $\ell@c = \mathsf{send}\ p\ v$, $\ell@p = \mathsf{send}\ c\ v$, and for all j, $j \neq p \wedge j \neq c$, $\ell@j = \mathtt{tt}$.
- Byzantine: for some c, p, q, and v, $\ell@c = \mathsf{byz_send}\ p\ q\ v$, and for all $j \neq c$, $\ell@j = \mathtt{tt}$.
- Receive: for some c, p, and $\vec{v}$, $\ell@p = \mathsf{receive}\ c\ \vec{v}$ and $\ell@c = \mathsf{receive}\ p\ \vec{v}$, and for all j, $j \neq p \wedge j \neq c$, $\ell@j = \mathtt{tt}$.

This choice of global labels assumes that no more than one node and exactly one channel can take a local step in each discrete global step. The send label and receive label model the interaction between a single non-Byzantine node and a single channel. The two local labels must agree on the payload of the message(s) being sent/received. The byz_send label models a stand-alone Byzantine action

where an arbitrary message is sent from a Byzantine node to a non-Byzantine receiver through a channel. The asynchronous communication between two nodes is modeled this way as two separate synchronous steps between a node and a channel.

The state and the transition relation of this compiled system follow from the construction. We also have a unique initial state s_0 for the global system: Async node components map to their initial Async program, while Channel components map to the empty channel state.

We say an Async system state s_f is *completed* if all its Async node instances have all been reduced to a return.[4] We write $s{\downarrow}$ as the partial function which extracts these values and returns them as a record of vectors $\vec{v}$, such that $\vec{v}@i = v$ iff $s@i = \mathsf{return}\ v$ for any node i. For an Async system instance compiled from a Sync program, given any completed state s_f, we can turn $s_f{\downarrow}$ into a record value of type $[\![\tau]\!]$, where the different values are broken out by role. We define $(\!| p |\!){\downarrow}$ to be the set of all such output values of the completed states reachable from the initial state. This is the set of possible outputs of the program p in the Async semantics.

Properties

A critical aspect of this construction is the relation between global behaviors (traces) and local views of behaviors. In particular, we make use of two key lemmas: a decomposition lemma that shows all global behaviors are necessarily arrangements of local behaviors and a composition lemma that shows we can

[4] Note that this may not be a terminal state in the transition system, as Byzantine sends could happen on channels.

always assemble traces from local systems into a global trace when there are suitable global labels.

We first define the projection from a sequence of global labels to sequences of local labels.

Definition 2.4.9 (Global Label Sequence Projections)**.** Given a sequence of global labels $L \in (\Lambda_G)^*$ and a local system identifier i, the subsequence of global labels related to i is defined as follows:

$$\mathsf{filter}_i(L) \coloneqq \begin{cases} [] & L = [] \\ \mathsf{filter}_i(L') & L = l :: L' \wedge l@i = \mathtt{tt} \\ l :: \mathsf{filter}_i(L') & L = l :: L' \wedge l@i \neq \mathtt{tt} \end{cases}$$

The projection of L to a sequence of i-local labels is defined as follows:

$$\mathsf{proj}_i(L) \coloneqq \mathsf{map}\ (@i)\ \mathsf{filter}_i(L)$$

Assume a system $\mathcal{T} = \mathcal{T}_1 \bowtie_\Lambda \mathcal{T}_2$, where $\mathcal{T}_i = \langle S_i, \Lambda_i, \rightarrow_i \rangle$.

Lemma 2.4.1 (Decomposition)**.** Given a sequence of labels $L \in \Lambda^*$ and compound states s and t such that $s \xrightarrow{L} t$, then $s@i \xrightarrow{\mathsf{proj}_i(L)}_i t@i$.

Lemma 2.4.2 (Composition)**.** Given component label sequences $L_i \in \Lambda_i^*$ and states s_i, t_i such that $s_i \xrightarrow{L_i}_i t_i$, and a compound label sequence L such that $\mathsf{proj}_i(L) = L_i$, then $(s_1, s_2) \xrightarrow{L} (t_1, t_2)$.

Lemma 2.4.1 implies that any property proven for all possible behaviors of local systems is preserved by the global system. Intuitively, a local system is most "free" when the environment it is interacting with is arbitrary. Composing

it with other systems provides more information about the environment and restricts its possible behaviors.

Lemma 2.4.2 shows that any combination of possible local system behaviors can happen in the global system if there exist suitable global labels. This can be used to prove a sequence of global labels are *permissible* (*i.e.,* lead from one global state to another) by establishing all its projections are permissible local label sequences.

2.5 Adequacy Proof

In this section, we sketch the proof that our denotational semantics is adequate for the compiled Async operational semantics.

For any closed Sync program $\Delta; \emptyset \vdash_{\mathcal{R}} p : \tau$, $[\![p]\!]$ produces a set of possible outputs of type $[\![\tau]\!]$. The program also compiles to a unique Async labeled transition system and yields a set of possible outputs of that system $(\!| p |\!)\!\downarrow$.

Theorem 2.5.1 (Adequacy)**.** $(\!| p |\!)\!\downarrow \subseteq [\![p]\!]$.

We prove the above result in two steps:

Step 1: By definition, any possible output of Async is witnessed by a trace L such that there exists a completed state s_f which gives the output and $s_0 \xrightarrow{L} s_f$. We show that we can reduce the set of traces to be considered in the operational semantics to an "aligned" subset that contains the same outputs and whose execution order puts all of the operations on a given channel together.

Step 2: We prove $s_f\!\downarrow \in [\![p]\!]$ assuming L is an aligned trace.

2.5.1 Reduction to Aligned Traces

We begin by defining the notion of the *alignment* of a global trace with respect to a channel context. Intuitively, this captures the idea that the outputs of a given trace are the same as a trace that forces all of the interactions to happen in an order specified by Δ.

Definition 2.5.1 (Alignment of a Global Trace)**.** Given a channel context Δ and a sequence of global Async labels L, the alignment of L with respect to Δ is given by:

$$\mathsf{align}_\Delta(L) \coloneqq \mathsf{flatmap}\ (\lambda c.\ \mathsf{filter}_c(L))\ \mathsf{Dom}(\Delta)$$

That is, $\mathsf{align}_\Delta(L)$ takes a list of labels, and for each channel c, projects out the sub-list of labels corresponding to non-empty transitions for c, and then concatenates those lists back together in the order that the channels are listed in Δ. Note that for every global label, there is a unique channel that transitions. Thus, $\mathsf{align}_\Delta(L)$ produces a permutation of the labels in L.

For example, consider a system with two good nodes n_1 and n_2 with n_1 sending messages to n_2 through channel c_1 followed by c_2, and a Byzantine node b sending a message on c_1. One possible sequence of actions we could see is the following

$$[n_1(\mathsf{send}\ c_1\ v_1); n_1(\mathsf{send}\ c_2\ v_2); n_2(\mathsf{receive}\ c_1\ \vec{v_1}); n_2(\mathsf{receive}\ c_2\ \vec{v_2}); b(\mathsf{byz_send}\ c_1\ w)]$$

where we only show the relevant node-portions of the labels. After aligning the trace, we will have:

$$[n_1(\mathsf{send}\ c_1\ v_1); n_2(\mathsf{receive}\ c_1\ \vec{v_1}); b(\mathsf{byz_send}\ c_1\ w); n_1(\mathsf{send}\ c_2\ v_2); n_2(\mathsf{receive}\ c_2\ \vec{v_2})]$$

So that all of the c_1 actions are performed before the c_2 actions, but otherwise the relative order of transitions is preserved.

Theorem 2.5.2 (Aligned Trace Permissibility)**.** Let s_0 be the initial state of an Async system built from a well-typed Sync program p with the channel context Δ and suppose $s_0 \xrightarrow{L} s$. Then $s_0 \xrightarrow{\mathsf{align}_\Delta(L)} s$.

Proof. We use the composition lemma (Lemma 2.4.2) to prove $\mathsf{align}_\Delta(L)$ is a permissible trace of the system. This requires us to prove that any local projection of $\mathsf{align}_\Delta(L)$ to a local system is a permissible trace of that local system. By the decomposition lemma (Lemma 2.4.1), we know any projections of the given trace L are permissible. So it suffices to prove that any local projection of $\mathsf{align}_\Delta(L)$ is equivalent to that of L.

In Async, we have two kinds of local systems: channels and nodes. Recall that $\mathsf{align}_\Delta(L)$ reorders the labels in the channel order captured by Δ, so it preserves the projections to the channels naturally. More formally, for any channel c in the system, we have:

$$
\begin{aligned}
\mathsf{proj}_c(\mathsf{align}_\Delta(L)) &= \mathsf{map}\ (@c)\ (\mathsf{filter}_c(\mathsf{flatmap}\ (\lambda c'.\mathsf{filter}_{c'}(L))\ \mathsf{Dom}(\Delta))) \\
&\qquad \backslash\backslash \text{ unfold definitions} \\
&= \mathsf{map}\ (@c)\ (\mathsf{flatmap}\ (\lambda c'.\mathsf{filter}_c(\mathsf{filter}_{c'}(L)))\ \mathsf{Dom}(\Delta)) \\
&\qquad \backslash\backslash \text{ list properties} \\
&= \mathsf{map}\ (@c)\ (\mathsf{filter}_c(L)) \quad \backslash\backslash \text{ other channels give []} \\
&= \mathsf{proj}_c(L) \quad \backslash\backslash \text{ by definition}
\end{aligned}
$$

For nodes, we begin by defining a relation ($\sim_R$) that holds when a sequence

of transition labels respects the order in Δ according to a given role R:

$$[] \sim_R \Delta$$

$$(\mathsf{send}\ c\ v) :: (\mathsf{receive}\ c\ \vec{v}) :: L \sim_R (c : (R, R, \tau)) :: \Delta \text{ when } L \sim_R \Delta$$

$$(\mathsf{send}\ c\ v) :: L \sim_R (c : (R, S, \tau)) :: \Delta \text{ when } L \sim_R \Delta$$

$$(\mathsf{receive}\ c\ \vec{v}) :: L \sim_R (c : (S, R, \tau)) :: \Delta \text{ when } L \sim_R \Delta \wedge R \neq S$$

$$L \sim_R (c' : (S_1, S_2, \tau)) :: \Delta$$

$$\text{when } L \sim_R \Delta \wedge R \neq S_1 \wedge R \neq S_2$$

We then extend this relation to describe when an Async respects Δ. Our intuition is that all of the possible traces the program can generate should respect the ordering in Δ:

$$\mathsf{return}\ v \sim_R \Delta \text{ when } R \notin \Delta$$

$$\mathsf{sendThen}\ c\ v\ (\mathsf{rcvThen}\ c\ k) \sim_R (c : (R, R, \tau)) :: \Delta \text{ when } \forall \vec{v}, k(\vec{v}) \sim_R \Delta$$

$$\mathsf{sendThen}\ c\ v\ k \sim_R (c : (R, S, \tau)) :: \Delta \text{ when } k \sim_R \Delta \wedge R \neq S$$

$$\mathsf{rcvThen}\ c\ k \sim_R (c : (S, R, \tau)) :: \Delta \text{ when } \forall \vec{v}, k(\vec{v}) \sim_R \Delta$$

$$t \sim_R (c' : (S_1, S_2, \tau)) :: \Delta$$

$$\text{when } t \sim_R \Delta \wedge R \neq S_1 \wedge R \neq S_2$$

Lemma 2.5.3 (Compilation Respects Channel Ordering)**.** Suppose $\Delta; \emptyset \vdash_{\mathcal{R}} p : \tau$, and let $R \in \mathcal{R}, i \in R$. Let L be a sequence of Async labels and t an Async program such that $(\!| P |\!)_{R,i} \xrightarrow{L} t$. Then $L \sim_R \Delta$.

Proof. It is easy to see by induction on the typing derivation for p that $(\!| p |\!)_{R,i} \sim_R \Delta$. We argue that for any Async program t such that $t \sim_R \Delta$, that if $t \xrightarrow{L} t'$, then $L \sim_R \Delta$, and furthermore, there exists a suffix of Δ, Δ' such that $t' \sim_R \Delta'$. The argument proceeds by induction on the length of L and then via case analysis on the structure of t. □

Now for any role R and non-Byzantine node i in that role, let $L_i = \mathsf{proj}_i(L)$. By Lemma 2.5.3, we know that $L_i \sim_R \Delta$ and we need to show $L_i = \mathsf{proj}_i(\mathsf{align}_\Delta(L))$. The proof proceeds by induction on the derivation of $L_i \sim_R \Delta$. □

2.5.2 Adequacy of Aligned Traces

We now prove Theorem 2.5.1 by showing that the output of any completed and aligned trace belongs to the set of outputs given by the denotational semantics.

Formally, we show for any closed Sync program P, well-typed with $\Delta; \Gamma \vdash_{\mathcal{R}} P : \tau$,

$$(\!| P |\!)\!\downarrow \subseteq [\![P]\!]$$

By definition of $\downarrow$, for any output $o \in (\!| P |\!)\!\downarrow$, there exists a trace l and a state F such that F is a complete state, $F\!\downarrow = o$, and $(\!| P |\!) \xrightarrow{l} F$.

By the alignment theorem, we can assume $l = \mathsf{align}_\Delta(l)$ without loss of generality.

Thus, assume $\Delta = [c_1, c_2, \ldots, c_n]$, $l = l_1 +\!\!+ \; l_2 +\!\!+ \; \ldots l_n$, where l_i contains only symbols associated with channel c_i.

Additionally, we can assume P is in the let-normal form:

$$\begin{aligned} P = \texttt{let}\ x_1 &\coloneqq \texttt{comm}\ c_1\ m_1\ d_1\ f_1\ \texttt{in} \\ \texttt{let}\ x_2 &\coloneqq \texttt{comm}\ c_2\ m_2\ d_2\ f_2\ \texttt{in} \\ &\ldots \\ \texttt{let}\ x_n &\coloneqq \texttt{comm}\ c_n\ m_n\ d_n\ f_n\ \texttt{in} \\ \texttt{ret}\ &\{R_i \mapsto e_i\} \end{aligned}$$

We now define a big-step operational semantics for any closed, well-typed, and normalized Sync program P with configuration $\mathcal{C}$.

$$\frac{\Delta(c) = (S, R, \tau_m) \quad L = [l_1, l_2, \ldots, l_{n_R - b_R}] \qquad \forall i, l_i \in \mathsf{Netwk}_S(\llbracket m \rrbracket) \quad y = \{R \mapsto \mathsf{map3\ foldl}\ \llbracket f \rrbracket\ \llbracket d \rrbracket\ L\}}{\texttt{let}\ x \coloneqq \texttt{comm}\ c\ m\ d\ f\ \texttt{in}\ P' \xrightarrow{L} P'[y/x]}$$

We use $\langle P \rangle\!\downarrow$ to denote the set of possible outputs in this big-step semantics, defined as follows:

$$\langle \texttt{ret}\ \{R_i \mapsto e_i\} \rangle\!\downarrow = \mathsf{singleton}\{R_i \mapsto \llbracket e_i \rrbracket\}$$
$$\langle \texttt{let}\ x \coloneqq \texttt{comm}\ c\ m\ d\ f\ \texttt{in}\ P'_x \rangle\!\downarrow = \bigcup_{P \xrightarrow{L} P'} \langle P' \rangle\!\downarrow$$

This splits the proof of the adequacy theorem into two parts $(\!| P |\!)\!\downarrow \subseteq \langle P \rangle\!\downarrow$ and $\langle P \rangle\!\downarrow \subseteq \llbracket P \rrbracket$.

Big-step to Denotational

To prove $\langle P \rangle\!\downarrow \subseteq \llbracket P \rrbracket$, we perform an induction on the structure of P.

Proof. Base case: when $P = \mathsf{ret}\{R_i \mapsto e_i\}$, $\langle P \rangle\!\downarrow = \mathsf{singleton}\{R_i \mapsto [\![e_i]\!]\} = [\![P]\!]$.

Inductive case: when $P = \texttt{let}\ x\ \texttt{:=}\ \texttt{comm}\ c\ m\ d\ f\ \texttt{in}\ P'_x$. By definition of the big-step semantics, $\langle P \rangle\!\downarrow = \bigcup_{P \xrightarrow{L} P'} \langle P' \rangle\!\downarrow$. It suffices to prove for any L and P' such that $P \xrightarrow{L} P'$, $\langle P' \rangle\!\downarrow \subseteq [\![P]\!]$.

By the induction hypothesis, we know $\langle P' \rangle\!\downarrow \subseteq [\![P']\!]$. It is enough to prove $[\![P']\!] \subseteq [\![P]\!]$.

This is straightforward by the definition of the big-step semantics and the denotational semantics. □

Async to Big-step

To move from Async traces to Big-step traces, we first need a well-formedness property of Async traces.

Definition 2.5.2 (Finished channel)**.** Recall that a state of a Async channel c is a tuple $\langle F_s, F_r, F_b, M \rangle$, where F_s is the set of non-Byzantine senders and F_r is the set of non-Byzantine receivers.

A state is *finished*, if and only if $F_s = \mathsf{sender}(c)$ and $F_r = \mathsf{receiver}(c)$. This means each non-Byzantine sender has performed their send operation, and each non-Byzantine receiver has performed their receive operation.

The first lemma we need here is that all channels are finished in a complete Async trace.

Proof Sketch: In a complete trace, each node must be complete. Because the compilation guarantees each node respects the channel context, Δ, for each

channel, each of the non-Byzantine senders and receivers must have performed their send and receive actions on this channel. Thus, every channel is finished.

For convenience, we extend the definition of Async channel state with two extra bookkeeping states. $M_s : \mathsf{list}\ \tau_m$ records each message sent by each non-Byzantine sender node. $M_r : F_r \rightarrow \mathsf{list}\ \tau_m$ records the list of messages received by each non-Byzantine receiver node. We modify the transitions accordingly to put information into those two pieces of the states.

Importantly, with the extra information, we can extract a big-step label $L = [l_1, l_2, \ldots, l_{n_R - b_R}]$ from a finished channel state by having $l_i = M_r^f(r_i)$, where M_r^f is the M_r of the finished state and r_i is a specific node of the receiver role.

So the second lemma we need is that this always gives a valid L such that $\forall i, l_i \in \mathsf{Netwk}_S(M_s^f)$. M_s^f is the M_s of the finished state and contains the messages sent by all non-Byzantine sender nodes.

Proof Sketch: We do an induction over the Async channel trace. By definition, each received list of messages satisfies Netwk_{Async} with the list of messages that have been sent to the receiver at the time of the receive action. We need to prove that each received list of messages satisfies $\mathsf{Netwk}_S(M_s^f)$. This can be proven with the invariant that $M(r)$ is always a subset of $\mathsf{add_any}\ M_s^f\ b_R$ and thus $\mathsf{Netwk}_{Async}(M(r)) \subseteq \mathsf{Netwk}_S(M_s^f)$.

A third lemma is needed to tie the changes in the node state to the actions that happened in the channel. More specifically, a node that sends a message will always send the message described by its program and continue with a unit; a node that receives a list of messages will continue with the result of folding its handler over the list of received messages. This lemma can be proven following

the definitions of the Async transitions.

The last lemma allows us to erase finished channels from the Async state. For any Async trace $S \xrightarrow{l} F$ and S contains a finished channel state $S@c$, then $S' \xrightarrow{l'} F'$, where S' is S except $S@c$, F' is F except $F@c$, and l' is l without labels associated with c. This lemma holds because the only transitions in l associated with c are Byzantine send operations on channel c due to $S@c$ being in a finished state and these operations can only affect the state of c.

To prove $(\!| P |\!)\downarrow \subseteq \langle P \rangle\downarrow$, we perform an induction on the structure of P.

Proof. Base case: when $P = \mathsf{ret}\{R_i \mapsto e_i\}$, by the definition of the compilation rules, both sides are the singleton set of $\{R_i \mapsto [\![e_i]\!]\}$.

Inductive case: when $P = \texttt{let } x \coloneqq \texttt{comm } c\ m\ d\ f \texttt{ in } P'_x$. It suffices to show, for any Δ-aligned Async trace $(\!| P |\!) \xrightarrow{l} F$, $F\downarrow \in \langle P \rangle\downarrow$.

We consider a block of labels of the same channel at a time. Because l is Δ-aligned, it must start with a block of labels associated with c, so $l = l_c +\!\!+\ l'$ and $(\!| P |\!) \xrightarrow{l_c} S \xrightarrow{l'} F$. By the first lemma above, because F is a completed state, it contains a finished channel c state $F@c$. By the definition of alignment, l' contains no labels associated with c, so $S@c = F@c$ and $S@c$ is a finished channel state. By the second lemma above, we can extract a valid big-step label L from $S@c$. This gives us a new Sync program P' through $P \xrightarrow{L} P'$. By the third lemma above, the node states in S match those of $(\!| P' |\!)$. They only differ because P' does not have channel c. And by the fourth lemma, there exists F', such that $(\!| P' |\!) \xrightarrow{l'} F'$. We know $F'\downarrow = F\downarrow$ because the only difference between F and F' is that F includes channel c's state and F' does not, and the extracted output does not depend on the channel states. By the induction hypothesis, $F'\downarrow \in \langle P' \rangle\downarrow$. By the definition of

the big-step semantics, $\langle P' \rangle\downarrow \subseteq \langle P \rangle\downarrow$. So we conclude $F\downarrow \in \langle P \rangle\downarrow$. $\square$

2.6 Related Work

Ironwood is one of many formal verification frameworks that are built to formally describe, specify, and verify distributed systems. We categorize existing frameworks by their features and compare them to Ironwood below, followed by a close-up discussion on the state-of-the-art on reasoning about fault-tolerant distributed systems.

Reducing Asynchrony to Synchrony. Ironwood utilizes the insight that certain asynchronous protocols can be verified by reasoning about their synchronous counterparts. This observation is shared by the PSync [41] framework and [34]. PSync is embedded in Scala and provides a high-level, synchronous language for describing crash fault-tolerant systems, which can be executed on partially asynchronous networks with a runtime. [34] proposes and formalizes the concept of *communication-closed* protocols, in which communication happens in rounds and messages are either received in that round or not at all. It then generalizes the asynchrony to synchrony reduction to such protocols. Both are based on the Heard-Of model [24]. Different from those two, Ironwood's fault model covers Byzantine faults. As a framework, Ironwood offers a functional, denotational semantics for compositional theorem-proving in Coq, while PSync and [34] provide monolithic automated verification of manually annotated imperative programs through custom verifiers. Furthermore, well-typed Sync programs are adequate for reasoning by construction, while in [34], the communication-closed restrictions are enforced by manually annotating C programs and then checked

by a stand-alone checker tool.

Modeling and Reasoning about Byzantine Failures. Ironwood models both crash failures and Byzantine failures through the unified Netwk abstraction. Velisarios [104] and its successor Asphalion [128] are verification frameworks built in Coq that also supports reasoning about Byzantine behaviors. Using a state machine model and the logic of events [14], the protocols need to be reasoned about at a relatively low level similar to that of Async. However, the frameworks provide epistemic models for the Byzantine adversary and built-in primitives for cryptographic signatures. The authors used the framework to verify and produce an executable artifact of the PBFT protocol [22].

Layered Refinements. While Ironwood provides a functional semantics, many existing works use trace-based semantics. Layered refinements have been shown to be effective at reducing the complexity of those proofs. Adore [60, 61] is a Coq-based verification framework for distributed systems that provides several state machine-based trace semantics at different abstraction levels. For instance, the top-level trace models the runtime behavior operationally as a non-deterministically growing tree, while at lower levels, the system state contains more details, such as program counters for each node and network messages. The authors verified a consensus protocol that supports dynamic reconfiguration using their layered semantics by modeling it at those different levels and proving refinement relations between them. Verdi [130] also uses a state machine-based semantics for distributed systems and proposes *Verified System Transformer* (VST) as a new abstraction. VSTs can ease some of the network assumptions, such as reordering, dropping, or duplicating messages for verification by adding layers of message handlers. They are compositional in the sense that they can be

applied to protocols and help the user focus on verifying the core properties of the system. The Raft protocol [99] is verified monolithically and provided as a VST.

Separation Logic. Ironwood focuses on system-level reasoning and uses a simple model for node-level behaviors because the two are largely orthogonal. Alternatively, distributed systems that assume more benign execution environments can adopt off-the-shelf techniques such as separation logic for reasoning and provide richer node-local semantics. Aneris [71] is a Coq framework that utilizes the concurrent separation logic framework Iris [65] to provide local reasoning for the nodes involved. It supports higher-order store and network sockets and has been used to verify a load balancer with concurrent local programs. Disel [112] is also a separation logic-based framework in Coq and supports compositional reasoning about interactions between an abstract core distributed system and multiple clients. Grove [113] is another concurrent separation logic library in Coq. It supports many system features, including time-based leases, reconfiguration, crash recovery, and thread-level concurrency.

Model checkers and Solver-based [68, 122, 126, 17] use model checking and solvers to trade off trusted computing base size for better automation. Using Ironwood's functional semantics for model checking or generating queries to solvers is also possible.

Logics. Besides frameworks, specialized logics have been proposed as tools to reason about distributed systems formally. Logic of Events [14] is a constructive logic that supports reasoning about discrete distributed events with a well-founded partial ordering that dictates their causal relationships, significantly reducing the number of possible behaviors to cover. Temporal Logic of Actions

(TLA) [77] is a modal logic with temporal quantifiers, which can be used to specify and prove both the safety and liveness properties of distributed systems but require the users to describe their systems as logical formulas. Different from frameworks such as Ironwood, those logics only provide means to formally describe and reason about abstract models of systems.

Artifacts. While both frameworks and specialized logics target the verification problem in general, there are also case studies that focus on verifying an artifact while developing techniques suited for the particular verification target. Ironfleet [52] is a verified implementation of a state machine replication protocol based on Paxos, which uses a mixture of state machine refinement at the high level and Hoare-logic verification at the low level. ShadowDB [109] is a formally verified replicated key-value store. The distributed system is described in EventML [103] and is extracted separately to executable code and a logic formula used for formal reasoning in the Nuprl proof assistant [31]. Chapar [80] is a verified replicated key-value store that reasons about the database and the clients separately and composes them through an interface designed to verify causal consistency. Noise* [56] formally verifies the Noise family of secure channel protocols in the F* proof assistant [119]. This work includes reasoning about both the cryptographic primitives used and their usage in the distributed protocol.

Other Programming Language Techniques. Besides aiming for full functional correctness, many programming languages designed for distributed systems provide some guarantee on the behavior of the systems. Session types [58] use static types to guarantee the communication behaviors of a program. Further developments in this line of work include multi-party session types [59] and multi-role session types [38] that generalize the type system to handle communi-

cation between more than two parties and dependent session types [123] that enable the communication behaviors to be dependent on runtime values. Choreographies [90, 33, 55] is a programming paradigm that uses a single program to describe a distributed system instead of having separate programs written for each component of the system. Choreographies make it easier to coordinate and restrict the behaviors of individual components globally and provide new ways of composing distributed systems. While those techniques do not guarantee full functional correctness on their own, they only require little overhead in engineering efforts and may provide a good foundation for further formal reasoning.

2.6.1 Detailed Discussions

The state-of-the-art today on reasoning about distributed systems either deploys informal, theoretical analysis for assurance, which is the case for most new protocols proposed, or applies more heavy-weight formal verification for existing protocols. We now look at some of those efforts in more detail.

Informal Reasoning of Consensus Protocols

The theoretical analysis of consensus protocols in the literature follows a relatively fixed structure. It begins by introducing the setting: assumptions on the network, faults, and sometimes cryptography primitives used. Then, the actual protocol is described in pseudo-code. Although the system is an abstract entity that includes all the nodes and the network, the pseudo-code is intended for a single node. It is often the case that all the nodes in the system, if not faulty, all

perform the operations described by the same piece of pseudo-code, potentially with different parameters. Informal proof usually starts after the description of the protocol to show that the system satisfies desired properties.

This pattern has been proven to be effective in practice. System builders follow the pseudo-code to implement the described protocol in their production systems. Despite the proof being informal, it does communicate the most non-trivial parts of why the protocol is correct, or rather, why the protocol would work when implemented. So it does add some degree of assurance to the protocol and to the systems being built using the protocol.

However, there are many aspects being omitted in the theoretical analysis, which could compromise the correctness of the system. Firstly, the semantics of pseudo-code languages for a single node is not formally defined. The gap here is that pseudo-code languages often assume rich primitives, such as broadcast or verify signatures. The analysis does not deal with the complexity of those primitives themselves and makes implicit simplifying assumptions about the effects of those primitives that may cause inconsistencies. Secondly, the semantics of the network or of a faulty node is also lacking. The problem here is that another set of simplifying assumptions must be made to hide away the complexity of real networks and the infinite possibilities of a Byzantine node. Thirdly, given how individual components should operate, how those components' behaviors compose and form a system is under-defined. This part is essential because our final goal is to prove the properties of the whole system. Last but not least, on top of all those missing parts, the informal proof often uses unstated reasoning principles that are lemmas about the implicit semantics. It is not always clear what those principles are and whether and why they are sound. For instance,

the Zyzzyva consensus protocol [69] proposed a novel speculative Byzantine fault-tolerant consensus protocol, but a counterexample that breaks the safety property of the protocol was later found [1]. Michael et al. [85] also found several counterexamples that break safety properties proven informally by previously published protocols through a more rigid formal semantics.

To showcase the informal reasoning in the distributed system literature, we quote two example protocols and part of their theoretical analysis from their original papers: Bosco and HotStuff.

Bosco

Bosco [114] is a Byzantine fault-tolerant consensus protocol. In the paper, the protocol is described in pseudo-code as in Algorithm 1.

The pseudo-code is expected to be executed by all correct nodes in the system. As a one-step algorithm, it assumes an underlying consensus algorithm and invokes it when a decision cannot be made after a single round of communication.

Algorithm 1 Bosco: a one-step asynchronous Byzantine consensus algorithm

Input: v_p
1: broadcast $\langle \texttt{VOTE}, v_p \rangle$ to all processors
2: wait until $n - t$ `VOTE` messages have been received
3: **if** more than $\frac{n+3t}{2}$ `VOTE` messages contain the same value v **then**
4: DECIDE(v)
5: **if** more than $\frac{n-t}{2}$ `VOTE` messages contain the same value v,
6: and there is only one such value v **then**
7: $v_p \leftarrow v$
8: `Underlying-Consensus`(v_p)

The authors defined the agreement property as follows:

Definition 1. *Agreement. If two correct processors decide, then they decide*

the same value. Also, if a correct processor decides more than once, it decides the same value each time.

They argued that the Bosco protocol satisfies the agreement property:

Theorem 3. *Bosco satisfies Agreement.*

Proof. There are two cases to consider. In the first case, no processor collects sufficient votes containing the same value to decide in line 4. This means that all decisions occur in `Underlying-Consensus`. Since `Underlying-Consensus` satisfies Agreement, Bosco satisfies Agreement. In the second case, some correct processor p decides some value v in line 4. By Lemma 3, any other processor that decides in line 4 must decide the same value. By Lemma 4, all correct processors must change their local estimates to v in line 6. Therefore, all correct processors will invoke `Underlying-Consensus` with the value v. Since `Underlying-Consensus` satisfies Unanimity, all correct processors that decide in `Underlying-Consensus` must also decide v. □

As an example, the referenced Lemma 4 is:

Lemma 4. *If a correct processor p decides a value v in line 4, then any correct processor q must set its local estimate to v in line 6.*

Proof. Assume otherwise, that a correct processor p decides a value v in line 4 and a correct processor q does not set its local estimate to v in line 6. Since processor p decides in line 4, it must have collected

more than $\frac{n+3t}{2}$ votes for v in line 2. Since processor q does not set its local estimate to v in line 6, it must have collected no more than $\frac{n-t}{2}$ votes for v, or collected more than $\frac{n-t}{2}$ votes for some value v', $v' \neq v$. For the first case, consider that since there are only n processors in the system, processor q must have collected votes from at least $n - 2t$ of the senders that processor p collected from. Among these, more than $\frac{n+t}{2}$ sent a vote for v to q. Since at most t of these processors can be Byzantine, processor q must have received more than $\frac{n-t}{2}$ votes for v. This is a contradiction. For the second case, if q collects more than $\frac{n-t}{2}$ votes for some value v', $v' \neq v$, then more than t of these senders must be among those that sent a vote for v to processor q. This is a contradiction, since, no more than t of the processors in the system can be Byzantine. □

HotStuff

HotStuff [134] is a more sophisticated Byzantine-fault tolerant consensus protocol. To save space, we only quote part of the basic version of the protocol as described in the original paper:

We also quote part of the safety proof below:

Safety. We first define a quorum certificate qc to be valid if $tverify(\langle qc.type, qc.viewNumber, qc.node\rangle, qc.sig)$ is true.

Lemma 1. *For any valid qc_1, qc_2 in which $qc_1.type = qc_2.type$ and $qc_1.node$ conflicts with $qc_2.node$, we have $qc_1.viewNumber \neq qc_2.viewNumber$.*

Algorithm 2 Basic HotStuff protocol (for replica r)

```
1: for curView ← 1, 2, 3, ... do
   ▷ PREPARE phase
2:    as a leader // r = LEADER(curView)
      // we assume special NEW-VIEW messages from view 0
3:       wait for (n − f) NEW-VIEW messages: M ←
         {m | MATCHINGMSG(m, NEW-VIEW, curView − 1)}
4:       highQC ← (arg max_{m∈M} {m.justify.viewNumber}).justify
5:       curProposal ← CREATELEAF(highQC.node, client's command)
6:       broadcast MSG(PREPARE, curProposal, highQC)
7:    as a replica
8:       wait for message m : MATHCINGMSG(m, PREPARE, curView) from
         LEADER(curView)
9:       if m.node extends from m.justify.node ∧ SAFEN-
         ODE(m.node, m.justify) then
10:         send VOTEMSG(PREPARE, m.node, ⊥) to LEADER(curView)
   ▷ PRE-COMMIT phase
   ...
```

Proof. To show a contradiction, suppose $qc_1.viewNumber = qc_2.viewNumber = v$. Because a valid QC can be formed only with $n - f = 2f + 1$ votes (i.e., partial signatures) for it, there must be a correct replica who voted twice in the same phase of v. This is impossible because the pseudocode allows voting only once for each phase in each view. □

Theorem 2. *If w and b are conflicting nodes, then they cannot be both committed, each by a correct replica.*

Proof. We prove this important theorem by contradiction. Let qc_1 denote a valid $commitQC$ (i.e., $qc_1.type =$ COMMIT) such that $qc_1.node = w$, and qc_2 denote a valid $commitQC$ such that $qc_2.node = b$. Denote $v_1 = qc_1.viewNumber$ and $v_2 = qc_2.viewNumber$. By Lemma 1, $v_1 \neq v_2$. W.l.o.g. assume $v_1 < v_2$.

We will now denote by v_s the lowest view higher than v_1 for which there is a valid $\mathit{prepareQC}$, qc_s (i.e., $qc_s.\mathit{type} = \text{PREPARE}$) where $qc_s.\mathit{viewNumber} = v_s$ and $qc_s.\mathit{node}$ conflicts with w. Formally, we define the following predicate for any $\mathit{prepareQC}$:

$$E(\mathit{prepareQC}) \coloneqq (v_1 < \mathit{prepareQC}.\mathit{viewNumber} \leq v_2) \\ \wedge (\mathit{prepareQC}.\mathit{node} \text{ conflicts with } w).$$

We can now set the first switching point qc_s:

$$qc_s \coloneqq \underset{\mathit{prepareQC}}{\arg\min}\{\mathit{prepareQC}.\mathit{viewNumber} \\ \mid \mathit{prepareQC} \text{ is valid } \wedge E(\mathit{prepareQC})\}.$$

Note that, by assumption such a qc_s must exist; for example, qc_s could be the $\mathit{prepareQC}$ formed in view v_2.

Of the correct replicas that sent a partial result $\mathit{tsign}_r(\langle qc_1.\mathit{type}, qc_1.\mathit{viewNumber}, qc_1.\mathit{node}\rangle)$, let r be the first that contributed $\mathit{tsign}_r(\langle qc_s.\mathit{type}, qc_s.\mathit{viewNumber}, qc_s.\mathit{node}\rangle)$; such an r must exist since otherwise, one of $qc_1.\mathit{sig}$ and $qc_s.\mathit{sig}$ could not have been created. ...Moreover, $m.\mathit{justify}.\mathit{viewNumber} > v_1$ would violate the minimality of v_s, and so the disjunct in Line 27 of Algorithm 1 is also false. Thus, SAFENODE must return false and r cannot cast a PREPARE vote on the conflicting branch in view v_s, a contradiction. □

From those two examples, we can catch a glimpse of the virtues and vices of informal reasoning practices. On one hand, they concisely convey the essence of a correctness argument, which the reader is convinced of its existence. On the other hand, they appeal to the reader's intuition, as the semantics of a single

node's behavior and the network are underspecified. Therefore, it is not difficult to see that formally proving the correctness of the described systems requires much more than "translating the pen-and-paper proofs."

Formal Reasoning of Consensus Protocols

Verification of distributed systems using formal, deductive theorem proving is a more recent research trend. How everything can be expressed formally remains a challenging research problem. More specifically, three ingredients are necessary for the formal verification: a model, which is a formal description of the system to be verified; a specification, which expresses the intended behaviors of the system; and finally, a proof that the model meets the specification. Despite the whole development being done eventually in an interactive theorem prover such as Coq, it is not just a matter of proof engineering. A series of design questions have to be answered to make the conceptual task of verification concrete as a collection of formal constructs.

The first question is how to formally describe a distributed system with all its behavioral quirks. The classic programming language approach is to define some formal semantics. A common choice in the case of distributed systems is the *global state semantics* [23], which describes the whole system as a single state machine. The state of the state machine reflects the global state of the system, which contains a piece of local state for each process and the network state. A transition of the state machine either delivers a message from the network to some process or is an internal transition of a process and may create new messages. This semantics models the asynchronous concurrency of distributed systems where the delay of the network is unbounded, and the execution of

different processes can be arbitrarily interleaved.

The second question is how to specify ideal system behavior. Because the global state semantics produces a set of traces, ideal behavior is often expressed as predicates on traces, called properties in systems research, or as predicates on a set of traces, called hyperproperties. Systems research also distinguishes between safety and liveness properties/hyperproperties. Informally, safety means bad things never happen, and liveness means good things eventually happen. Safety properties can usually be formalized in first-order logic, while liveness often requires modal quantifiers to express temporal concepts such as eventually or fairness. For the remainder, unless mentioned otherwise, we will be discussing the verification of safety properties, as it is considered a simpler goal than liveness and is more studied in the literature at the time of writing.

The third question is how to prove a system satisfies desired properties. A prominent issue is the combinatorial explosion of the number of possible behaviors of a distributed system modeled this way. Because of asynchronous concurrency, the number of possible global behaviors often grows exponentially with the number of processors and the size of the individual programs. So simply enumerating every possible behavior and checking whether they satisfy the desired property does not scale. The first-line remedy is to use inductive invariants, which are predicates on a global state with two additional properties to be proven: 1. Any initial state of the system satisfies the predicates; 2. Any state that satisfies the predicates can only transit to another state that satisfies those predicates. Inductive invariants can be used to prove safety properties by overestimating the set of all possible system behaviors and excluding any possible behavior not allowed by the specification. They are effective against the

combinatorial explosion problem. To prove a predicate is an inductive invariant, one only needs to check it for the initial state and prove the inductive case for any possible transition out of a state that satisfies the predicate. Conceptually, this reduces the number of cases to consider from the number of possible traces to the number of single-step transitions, which enables further reductions utilizing structures in the transitions, such as symmetry between different processors or representing a group of states by a single symbolic state.

Moreover, the inductive invariants are complete in the sense that any true safety property can be proven by finding the right invariant that entails the desired property. It is shown in [3] that such an invariant always exists in an "appropriate" setup that may require adding additional history information to the original global states.

However, there is tension between a predicate being an inductive invariant of the system and the complexity of proving the predicate implies the desired safety property. On one hand, while a precise invariant that captures the exact set of reachable states can be constructed, which is an inductive invariant by construction and, in theory, entails any true safety property about the system, such an invariant is not helpful in practice. This is because this invariant would be an overlong logical predicate that covers every detail of the system and makes the later proof step that the invariant implies the desired property practically impossible for both automatic solvers and manual proofs, especially when the system model contains many low-level implementation details. On the other hand, the desired safety property as a predicate would trivially imply itself, but it is often too imprecise to be an inductive invariant of the system. Finding the "right" inductive invariant that keeps the whole complexity manageable is a

challenging task and has been identified and investigated as a research problem on its own [82].

A proposed remedy to tame the complexity of the invariant approach is layered refinement [75], which adds structures to the formal reasoning of invariants. Instead of building a single model of the system to be verified, two or more models that describe the same system at different abstraction levels are built. At the most abstract level, the model is presumably relatively simple, and the invariant approach is applied directly. For each level below, instead of proving the safety property directly, the invariant approach is used to prove a refinement relationship between the behaviors of this level and the more abstract level above, which implies the safety property of the current level when the refinement relationship is properly chosen.

Adore

Adore [61] introduces a verification framework that pushes the limit of invariants and refinements to formally verify the implementation of distributed systems with reconfiguration. The authors used four different models, which are all state machine semantics, to describe the same target system From the highest to the lowest abstraction level, they are:

1. State machine replication(SMR) [110]. In the more general distributed system context, SMR often refers to the technique that provides fault tolerance and implements decentralized control. In this case, it refers to a formal state machine semantics whose state is an append-only log that contains all commands that have been committed, and the only transition is to append an item, called a command, to the end of the log. This is a high-level

abstraction because any implementation detail of the log is not modeled. The log in the global state is only conceptual and may not exist physically on any machine in the system, and a single transition may correspond to many instructions and interactions in the system.

2. Atomic distributed object(ADO) [60]. The ADO model is another state machine semantics that exposes more operational details of the distributed system. The state of the ADO model contains a list of committed commands, a tree of uncommitted commands, called the cache, an identifier of the currently active process, and for each process, a pointer pointing to *null* or somewhere in the cache as the "active" local state. This state presents a global view of the system: enough processes agree on the committed prefix to make it persistent, and different branches of the cache represent temporary disagreement between different processes' local states. The local states themselves are omitted except when they could potentially advance the global state, in which case, they are mentioned by the pointers to the cache. Only the process that is identified as currently active has the exclusive privilege of appending to the cache.

   The ADO model has three classes of transition: *pull*, *method*, *push*. Each transition is associated with a specific process.

   - *pull* corresponds to a process synchronizing its local state with the global state and setting the currently active process to itself and its local pointer to a certain command. There are three variations of *pull*: a completely failed *pull* does not change the ADO state; a partially failed *pull* would reset the currently active process and the local pointer to *null*, blocking other processes as a result; a successful *pull* would set both.

- *method* represents any call that may update the cache tree. Only the process that is currently active may append new commands after the command pointed by its local pointer. There are also two variations of *method*: a failed one has no effect, and a successful one adds new commands to the cache tree.
- *push* finalizes part of the cache, adds those commands to the persistent log, and may delete part of the cache that conflicts with the finalized part. Only the process that is currently active may perform a *push* step. *push* may fail and do nothing. When it is successful, it may finalize any prefix from the cache root to the command pointed by the local pointer of the currently active process.

3. The ADORE model. The ADORE model can be seen as a variation of the ADO with more details to support the reasoning about reconfiguration, an operation that changes the set of processes participating in the consensus protocol. The difference in the state is that a description of system *configuration*, i.e., the set of nodes that are considered actively participating at the time, is added to all commands. There is also a new kind of transition, *reconfiguration*, which is treated as a special *method* command that may change the set of active participants.
4. Network-based model. The network-based model is the low-level model described above, where the state contains local computational states and a network state, each transition corresponds to the delivery of a single message, and the execution of different processes can interleave. This is a standard model used by many verification projects [52, 112, 130]. However, the authors of [61] further argue that verification of the protocol's correctness directly in this model is challenging.

The authors then used invariants and refinements to prove the safety property. The property is rather trivial in the SMR model because it has a single consistent view of the history. The refinement between the ADO model and the SMR model is also straightforward because the persistent log in the ADO model is exactly the SMR log. The refinement between the ADORE model and the ADO model requires some amount of work. One problem here is how to reason about separate branches of the cache tree created by reconfiguration, as two leaf nodes may end up with no overlap in the processes participating. In [61], the authors introduced the concept of *rdist*, the number of reconfiguration events between two nodes of the cache tree. One could relate two separate branches by performing an induction on *rdist*, walking through the reconfiguration history. The refinement between the network model and the ADORE model is more complicated and is dependent on the verification target. The authors used the Raft [99] consensus protocol as an illustrative example. They introduced another layer of state machine semantics called SRaft, which is a simplified version of Raft where some operations are reordered. Then, the authors were able to prove the refinement between the SRaft and ADORE and that between the actual Raft and SRaft with invariants.

Verdi As another example, even using an existing verification framework does not ensure an easy victory. Doug et al. [131] used the Verdi [130] verification framework to formally verify the Raft protocol. Their proof required iteratively discovering and proving 90 system invariants, including all the invariants described by the original Raft paper. The main content of the paper is various engineering methodologies the authors applied to wrestle with the complexity of the proof burden.

Contrasting those cases with the informal proofs above, while the theoretical distributed system literature can often justify its claims within a few pages, a full paper on formal verification of distributed systems barely has enough space to show the tip of the iceberg. Of course, as we have analyzed above, many parts of the reasoning are omitted in the informal world. But even then, why is it possible for the informal proofs to omit them and still deliver the most critical aspects of the proof?

Relations to Ironwood

We argue that informal reasoning uses high-level abstractions that offer different kinds of compositionality. As a result, the correctness proof can be naturally decomposed into smaller goals, and the informal papers cover the most difficult ones. In contrast, formal reasoning using state machines and invariants is monolithic. The system has to be described as a whole, and the reasoning must cover all parts of the system at the same time.

In Ironwood, we show how this gap can be bridged by formalizing those abstractions that enable compositional reasoning. The high-level language, Sync, is inspired by the informal reasoning of distributed systems with abstract communication primitives. The low-level language, Async, uses ideas from the existing formal reasoning of distributed systems that model distributed systems as monolithic state machines. By explicitly defining the semantics of distributed systems formally and restricting the class of systems that can be modeled through syntax and types, we are able to connect those two worlds and offer the high assurance of formal reasoning with the relatively low overhead of informal reasoning.

2.7 Future Work

Part of our ongoing work is mechanizing the adequacy proof described in Section 2.5 to improve confidence in our results.

A limitation of our framework is the lack of support for reasoning about protocols that use cryptographic schemes to restrain the power of the Byzantine adversary, such as [134, 22, 79]. The challenge is two-fold. On the one hand, a richer adversary model is necessary to formally reason about those protocols whose correctness relies on some ideal semantics of cryptographic primitives. On the other, how a protocol-level proof can be composed with proofs for those primitives is still an open problem. For instance, many protocols assume signatures are free of side effects. However, in practice, signatures might be compromised by the adversary through interactions specified by the protocols. A future direction of our work is to close this gap by introducing foundational cryptographic models such as [40, 21] in a modular way.

Currently, we only support sequential control flow in our high-level language but it is common for distributed systems to have more complex control flow structures. For example, in Paxos [76], an acceptor can "skip" rounds of communication if a message from a later round is received, which currently cannot be modeled by our framework.

Besides safety properties, liveness results are also crucial for reasoning about distributed system protocols. Reasoning about liveness properties often needs a different network semantics that provides stronger guarantees, such as bounded delivery time. Ideally, we would parameterize our framework to easily swap in different network semantics to support different assumptions flexibly.

CHAPTER 3

HARMONY

Concurrency is indispensable for modern computer systems in their pursuit of performance. However, building efficient concurrent systems is notoriously error-prone, even for the most experienced system builders. Developers in the industry have recently experimented with *prototyping* as a methodology for incorporating verification into system building [8, 17]. In prototyping, there is a split between a *prototype*, usually treated with verification, and a *production system* built alongside the prototype. The system programmers are responsible for transferring lessons learned from one to another. This offers some of the benefits of verification at an affordable development overhead.

Model checking is one of the most widely adopted verification techniques for prototyping. State-of-the-art model-checking tools come in roughly two classes. In one class, the input language of the model checker is a popular implementation language such as C or Java [97, 92, 91, 10, 36, 51, 86, 32, 66]. These model checkers sometimes require writing code in a particular (usually callback-style or state machine) format, but familiarity with the language lowers the learning curve for using them. However, limited by the compatibility design, the scope of which programs can be checked tends to be limited. In the other class, model checkers come with their own programming language, usually based on a formal logic and closely matching the abstractions for which the model checker has been designed [64, 29, 78, 73, 20, 57, 47, 4]. Such model checkers scale relatively well in terms of the complexity of prototypes to be checked, but the models tend to be far removed from the actual production code. As a result, using those model checkers requires more expertise in both system and verification and

incurs significant overhead in transferring lessons between the prototype and the production system.

We propose Harmony, a general concurrent algorithm prototyping tool that combines the usability of the former class and the scalability of the latter. Harmony has its own surface language with syntax close to Python and a virtual machine semantics that models a shared memory and captures the non-determinism caused by concurrency as the interleaving of machine instructions. This results in a system development language that supports dynamic memory allocation, pointers, and interrupts and is familiar to system programmers and students. By virtue of this design, specifying desired properties in Harmony can be done without expertise in specialized logic. Harmony provides built-in specifications and checks for common system properties such as deadlock freedom and data race freedom. Users can write custom specifications in the form of program assertions or employ behavior checking to specify the desired behaviors as a program. Harmony's design also reduces the overhead of transferring lessons learned between the prototype and the production system. When an error is found in the model-checking process, Harmony presents a counterexample as a concrete program trace explained in comprehensible natural language. The counterexample is picked and optimized through heuristics for a shorter length to reduce user burden.

To provide the usability mentioned above without sacrificing scalability, Harmony's custom model checker is specialized for its virtual machine semantics. The model checker achieves competitive performance via optimizations such as partial order reduction [127]. It also integrates various novel optimizations that utilize the properties of our semantics, including anonymous threads, dead

state elimination, and caching intermediate results. To scale to large systems, Harmony supports modular model checking by separating specification and implementation.

Harmony has been used to model check everything from lock implementations to concurrent file systems to Byzantine consensus protocols. Our case studies show that its performance is competitive with TLA+ [77] and SPIN [57], two popular and highly-optimized general-purpose model checkers, with significantly less user effort. Our preliminary user study suggests that system programmers without verification expertise can more easily find concurrency errors in programs written in Harmony using our tool than in programs written in Python using the standard Python execution environment.

Harmony and an accompanying textbook are publicly available at `https://harmony.cs.cornell.edu/`.

3.1 Using Harmony

This section introduces Harmony by walking through prototyping a spinlock. In concurrent system development, locks are ubiquitous and usually considered a primitive of synchronization. But in Harmony, they are implemented by more basic abstractions instead of being given as a magical black box. The user can import the lock as a primitive from one of the supplied modules written in Harmony.

```
1  def Lock() returns lock:
2      lock = False

4  def acquire(lk):
5      atomically when not !lk:
6          !lk = True

8  def release(lk):
9      atomically !lk = False
```

```
1  def tas(s) returns old:
2      atomically:
3          old = !s
4          !s = True

6  def Lock() returns lock:
7      lock = False

9  def acquire(lk):
10     while tas(lk):
11         pass

13 def release(lk):
14     atomically !lk = False
```

Figure 3.1: Specification of a lock and a spinlock based on atomic test-and-set

3.1.1 Describing Lock Algorithms

Describing the prototype is the first step to prototyping. Harmony currently supports a single front-end syntax that is similar to Python. This syntax is imperative, does not have type annotations, and uses significant indentation. We chose this syntax to reduce the learning curve, as programming language research shows that syntax plays a large role in the initial learning process [117].

Figure 3.1 (left) shows an abstract specification in Harmony of the ideal functionality of a lock and defines the basic interface: `Lock`, the constructor, and `acquire` and `release`, the common operations of a lock. `Lock` returns the initial value of a lock "object". In this description, a lock is a single memory cell that holds either **True** or **False**. `acquire` blocks the execution until it successfully obtains the lock, which is accomplished by the **atomically when** primitive. Here *lk* points to the memory cell that contains the value of the lock. It repeatedly checks !*lk*, the value of the memory cell, until it is **False**, and then sets it to **True**. **atomically** ensures that the check and setting the value are done atomically, that is, the machine instructions compiled from the code block will be considered a single unit when interleaved with other concurrent instructions. Also, two

conflicting accesses to the same shared memory blocks are not considered a data race if both accesses are inside an **atomically** block. `release` releases the lock by atomically setting its value to **False**. The **atomically** here avoids a data race with a concurrent `acquire`.

Real-world processors do not have an **atomically** primitive. Thus this specification does not directly help programmers build such functionality. But it can work as a specification that defines the intended behavior of a lock in a natural way.

Figure 3.1 (right) shows a Harmony implementation of the classic spinlock that uses test-and-set (TAS). This specification defines 4 procedures: `tas`, `Lock`, `acquire`, and `release`. `tas` models an instruction available in many modern CPUs. A test-and-set instruction modifies a memory cell and returns its old value. Line 3 stores !*s*, the current value stored at location *s*, into a temporary memory cell *old* that is returned by the method. Line 4 sets the value at address *s* to **True**. These two lines are wrapped in an **atomically** block. `Lock`, `acquire`, and `release` in this description implement the same interface as the specification. `Lock` also returns a single memory cell that holds either **True** or **False**. `acquire` blocks the execution until it successfully obtains the lock, accomplished by calling `tas` in a while loop until it returns **False**. `release` releases the lock by atomically setting its value to **False**. The **atomically** here is necessary to avoid a data race with a concurrent `tas`. Again, modern CPUs support instructions that atomically store values, hence this code is fully implementable.

There are many ways to implement the lock specification. This includes implementations that suspend threads and keep suspended threads in a priority queue, which can be implemented through Harmony's support of continuations.

3.1.2 Testing and Debugging Lock Algorithms

When the description is more complex than can be assumed to be correct, further investigation becomes necessary. To this end, some specification of what is considered correct or incorrect is required. Harmony not only offers built-in specifications, such as data race freedom, which are checked by default, but also specifications through assertions and through another Harmony program. We will describe the built-in specifications in detail in Section 3.3 and continue our example of locks to explain the usage of assertions and the debugging support provided by Harmony.

Assertions are familiar to programmers because they are broadly used in non-concurrent scenarios. In Harmony, an assertion atomically computes a Boolean expression and throws an error if the result is **False**. Because our spinlock description only provides the definition, to leverage assertions one needs to create a scenario where a lock is used. Figure 3.2(a) shows a test program where multiple concurrent threads use the lock to enter a critical section zero or more times.

Line 1 imports the lock constructor, *acquire* and *release* from one of our earlier descriptions. In Harmony, it is easy to specify which lock module to be imported through the `-m` flag to the command line tool. One can also use the `-c` flag to overwrite constants, such as `NTHREADS` on line 3, defined in the program. Line 5 constructs a new lock. Lines 8 to 14 describe the test thread, which acquires and releases the lock zero or more times. The **choose** expression on line 8 non-deterministically computes to an element from the set of **True** and **False**. The expression ?*thelock* is a *thunk* [62]. If p is a thunk of expression e, then !p evaluates to e. In this context, one can think of ?*thelock* as the address of variable *thelock*

```
1   from lock import *
3   const NTHREADS = 5
5   thelock = Lock()
6   count = 0
8   def thread():
9       while choose { False, True }:
10          acquire(?thelock)
11          atomically count += 1
12          assert count == 1
13          atomically count −= 1
14          release(?thelock)
16  for i in {1..NTHREADS}:
17      spawn thread()
```
(a)

```
from lock import *
const NTHREADS = 5
thelock = Lock()
def thread(self):
    while choose { False, True }:
        print(self, "acquiring")
        acquire(?thelock)
        print(self, "acquired")
        print(self, "releasing")
        release(?thelock)
        print(self, "released")
for i in {1..NTHREADS}:
    spawn thread(i)
```
(b)

Figure 3.2: Test programs for locks. (a) is based on invariants, (b) on external behavior checking

- Schedule thread T0: __**init**__()
  - Line 5: Initialize thelock to False
  - Line 6: Initialize count to 0
- Schedule thread T1: thread()
  - Line 9: Choose True
  - Preempted in thread() –> acquire(?thelock) –> tas(?thelock) about to store True into thelock in line lock/4
- Schedule thread T2: thread()
  - Line 9: Choose True
  - Line lock/4: Set thelock to True (was False)
  - Line 11: Set count to 1 (was 0)
  - Preempted in thread() about to execute atomic section in line 12
- Schedule thread T1: thread() –> acquire(?thelock) –> tas(?thelock)
  - Line lock/4: Set thelock to True (unchanged)
  - Line 11: Set count to 2 (was 1)
  - Line 12: Harmony assertion failed

Figure 3.3: A Harmony counter-example

while $!p$ dereferences the address stored in p.

Lines 16 and 17 start separate threads of execution whose entry point is the `thread` function. The most important statement is line 12, an assertion that checks the number of threads currently executing line 12, kept track of in variable *count*, is exactly one.

Running Harmony on the lock specifications will test both mutual exclusion and progress. Progress is violated if a thread can never enter the critical section. If we remove `atomically` in Line 2 from Figure 3.1(b), Harmony describes a counterexample in natural language (Figure 3.3). Harmony also supports a graphical user interface that allows single stepping through the code forwards and backwards while displaying the values of the global variables and the stack traces of the threads.

Besides assertions, Harmony also supports invariants, which are checked at all possible program states.

3.1.3 Checking Refinement Relations

While assertions avoid using specialized logic, their expressive power is limited. For instance, a ticket lock is FIFO and provides fairness for the waiting threads. This property is difficult or even impossible to express using only assertions [74]. To solve this, Harmony supports comparing the possible observable behaviors of one program to another. A program P refines program Q if the set of possible behaviors of P is a subset of that of Q. This is called *refinement*, also known as *weak bisimulation* [9]. Despite the abstract notion of FIFO and fairness being

hard to grasp, the idea that any observable behaviors of one program are also observable behaviors of another program only involves comparing concrete traces.

Checking behavioral equivalence allows verifying an implementation without having to think about specific properties the implementation must have. Figure 3.2(b) shows a Harmony test program to enable behavioral equivalence checking for locks. Essentially, the programmer just needs to put descriptive messages before and after invoking an API. To use the test program, the programmer can first run it against the lock specification, capturing all possible behaviors, then run it again against a lock implementation to be checked. For instance, running the test program against the broken spinlock implementation (making `tas` non-atomic) in Harmony produces a counterexample, illustrating a violation of mutual exclusion but without explicit assertions or invariants.

Similar test programs can check more complicated properties such as fairness and linearizability without having to specify those properties in formal logic. For example, a reader/writer lock could be implemented with a plain lock, never allowing more than one reader in a critical section. An invariant-based test program would not find an issue, but a behavioral test program would find that some behaviors of a reader-writer lock are missing. Such properties are challenging to express and test for otherwise.

3.2 Design

The core of Harmony design is a virtual machine semantics bundled with model checking. Our semantics is a model of Interleaving Instructions with a Strictly

Consistent shared memory (IISC). We believe IISC simplifies prototyping through similarity and familiarity. Harmony model checks IISC to provide the benefit of precision and rigor without incurring too much of a burden on the users. Based on those two cornerstones, Harmony supports contextual specification, such as assertions and refinements, instead of a stand-alone specification in formal logic.

3.2.1 Modeling Concurrency and Memory

The behavior of a program is the effect of executing the program. Modeling the behavior of a program, or the set of possible behaviors of a program in the case of concurrency, is the first step toward further analysis. Verification tools based on formal logic are often very expressive and can be adapted for modeling different aspects of program behaviors in various manners. In contrast, Harmony only offers a fixed way of modeling the state-changing behaviors of concurrent programs, which is Interleaving Instructions with a Strictly Consistent shared memory (IISC). This semantics can be understood as a virtual machine that has an unbounded number of cores and can execute a fixed basic instructions set. Each core has its own private state, but all cores share the same memory. Furthermore, the memory is strictly consistent, which means the effect of a memory read or write instruction is reflected immediately. Besides the arithmetic primitives, this model only captures two aspects of concurrent programs' behaviors: non-determinism and shared memory access.

Interleaving One specific behavior of a concurrent program is modeled as some combination of behaviors of multiple concurrent threads; each executes its own sequence of instructions in program order. The whole program is assumed

to execute in discrete steps, and the behaviors of those multiple threads are combined together by repeatedly picking one of the threads non-deterministically to execute a single machine instruction while all other threads do nothing.

This is far from how current physical computer architectures execute concurrent programs, as concurrency literally means instructions executing at the same time. The justification for assuming step-wise instruction execution is that if multiple concurrent machine instructions are independent (e.g., they do not access the same memory), then the effect of their execution should be equivalent to the effect of executing those instructions sequentially in any particular order. In the case of concurrent execution of dependent instructions, also known as a *data race* (e.g., a load and a store operation accessing the same memory location), the effect is either the same as executing those instructions sequentially in some specific order or (typically) undefined, depending on the memory consistency model. In Harmony, a warning will be raised if some execution of the program leads to a data race.

Shared Memory We choose to directly model a shared memory because concurrent threads often need to communicate and interact, and shared memory is the most basic interface provided by the hardware to the programmers.

There are different models of shared memory [42], among which the strictly consistent memory is the most restrictive. For weaker memory models, including those that would match the memory models defined for system programming languages such as recent versions of C++, Rust, and Java, prior research [2] shows that classes of important properties are preserved for many of them under the additional assumption that the programs have no data races.

Also, modern CPUs incorporate many micro-architectural optimizations that change the effect of executing an instruction, for instance, write buffering, read caching, and out-of-order and speculative execution. IISC does not cover any of these features as given. As a result, exploring program behaviors with micro-architectural features may require extra user effort. But IISC and our tool can still be helpful in those settings because any error reported is a true positive. This is because the set of program behaviors IISC allow would still be a subset of all possible program behaviors.

State, Contexts, and Behavior Based on the two basic ideas above, IISC models concurrent behavior rigorously as mathematical objects. In IISC, the state of the whole program contains at least the code (a static array of instructions), the shared memory, and the *context bag*, which is a multiset of local states for separate instances of execution, i.e., threads. At a minimum, a *context* contains a program counter pointing to an instruction in the code. (It may contain additional private variable values, as explained later.)

The context bag represents the multiset of concurrent activities. It is a bag because Harmony threads are anonymous, and thus multiple threads can be in the same state. Doing so can significantly increase scalability of model checking as, when multiple threads are in the same state, there are fewer interleavings to consider.

A single *behavior* of a program is a sequence of states produced by executing a sequence of instructions and the additional information of which thread is picked at each state. A *step* in the behavior involves the execution of exactly one instruction pointed to by the program counter of the picked context. Executing

the instruction will typically update the program counter, replacing the context in the state with a new one. (An exception is when the instruction is a “jump” to the instruction itself).

An instruction may have other effects on the state, including changing the value of a memory location or adding or removing a context from the context bag. Adding a context to the context bag represents creating a new concurrent activity. A behavior terminates when its last instruction removes the last context from the context bag.

3.2.2 Harmony Virtual Machine

Following the abstract idea of IISC, we built the Harmony Virtual Machine (SVM), a virtual machine with a concrete set of machine instructions and serving as the compilation target of Harmony programs. We introduce SVM by introducing its state and transitions.

A state of the SVM consists of three parts: the code, which is a list of SVM instructions; the shared memory, which is a dictionary that maps shared variable names to values; and the context bag, which is a multiset of contexts (thread local states), one for each thread.

Optionally, the state may also include the state of a deterministic finite automaton (DFA) to check the legality of the external behavior of the SVM, which is used for checking refinements, as shown in the lock example.

A thread is essentially a stack machine. Its state is an SVM context, which basically includes a program counter, an atomic region counter, and a private

stack of SVM values. An SVM context does not include a thread identifier, and therefore multiple threads can be in the same local state. The atomic counter is used to describe whether the thread is in an *atomic region*. Together with the `AtomicInc/AtomicDec` instructions, SVM uses them to support **atomically** in the lock examples. The thread is in an atomic region if and only if the counter is non-zero, which means that the thread has exclusive access to the shared memory. Because atomic regions can nest, a counter is used instead of a flag.

The initial state of the SVM consists of an empty dictionary and a single context with program counter 0 and atomic counter 1. The thread whose state is defined by the context will initialize the shared variables and spawn additional threads by adding contexts to the context bag. Newly spawned threads have a stack consisting of a single argument, a program counter pointing to the thread's method, and an atomic counter of zero.

The SVM makes a transition by executing an SVM instruction using one of the contexts in the context bag. There are about 30 SVM instructions, with the most important ones classified as follows:

- *local operations*: a deterministic operation that only updates the context of the executing thread, such as pushing a value onto its stack or incrementing its program counter.
- `Load`: dereferences Harmony addresses, which can be used to read shared state but also to make method invocations.
- `Store`: an operation that updates the shared state.
- `Print`: producing outputs, which are considered "observable" for refinement checks.
- `Choose`: nondetermistic selection from a set.

- `Frame/Return`: method start/end.
- `Spawn/Save/Stop/Go`: continuation operations.
- `AtomicInc/AtomicDec`: atomic region enter/leave.
- `Assert`: value check.

Each transition modifies at least the current context, if only by changing its program counter. The `Store` and `Stop` instructions also modify the shared variables, while `Spawn` and `Go` add new contexts to the context bag. The `Return` instruction terminates the current thread if its stack is empty. The SVM stops when the context bag becomes empty. If a DFA was specified, it is a failure if the DFA is not also in a final state. The SVM also stops when a thread executes an illegal instruction, such as trying to pop a value of an empty stack or trying to decrement a zero atomic counter.

Different from modern ISAs, SVM does not provide any primitive synchronization instructions but one can model the effect of those instructions with atomic regions. Below we describe in more detail some of the more unusual instructions.

Choose. `Choose` pops a set value off the stack and pushes a selected element back onto the stack. The model checker exhaustively enumerates each selection.

Load. An *SVM address* consists of a function and a list of arguments; `Load` dereferences an address. A function can be a method call (program counter), but also a Harmony value that maps values to other values (dictionary, list, or string).

Stop/Go. `Stop` saves the current context (with its program counter incremented) in a specified shared location (such as the tail of a scheduling queue) and

removes the context from the context bag. `Go` pops a context value and a return value off the stack, pushes the return value onto the stack of the popped context, and adds the context to the context bag. A stopped context can be re-instantiated multiple times, allowing the implementation of thread forking.

Print. The only way for the SVM to produce output, `Print` pops a value off the stack and increments the program counter. In case a behavior automaton has been specified, the value is applied to the DFA and checked for validity.

As the Harmony compiler aims to help the programmer reason about the behavior of the generated code, it only performs those standard compiler optimizations that do not re-order expression evaluations.

3.2.3 Model checking

Charm, Harmony's model checker back-end, is an exhaustive and stateful model checker. Given a program in SVM instructions, it enumerates all possible behaviors following the definitions of SVM and generates a Kripke structure [70], which is a directed graph whose vertices are states of the SVM and edges are transitions of the SVM. As such, Harmony can only handle a finite number of states.

The algorithm for building the Kripke Structure is straightforward at a high level. Charm performs a breadth-first search (BFS) and tries all possible non-deterministic choices at each state. However, a naive implementation can lead to too many reachable states. One important optimization we perform to reduce the number of states without sacrificing exhaustiveness is partial order reduc-

tion [127]. Given the concrete definition of SVM above, Charm can recognize more such opportunities than a more general model checker. For instance, the effects of local operations of different threads that can be interleaved in either order do not depend on their relative ordering.

Another important optimization is to cache the effects of executing a thread starting in a particular context and state of global variables. This is very effective as, due to interleaving, many states in the Kripke structure may have the same context and global variables. Yet another optimization is the compression of identical contexts in states. Charm treats two contexts as interchangeable if they contain the same values and only explores one transition for a group of identical contexts. This is only possible by our SVM design, where no unique identifier is assigned to a thread by default. A concrete case is a readers-writer lock when multiple readers have succeeded in acquiring the lock. Finally, Harmony uses *live variable analysis*, a standard part of register allocation in compilers, for eliminating part of a state that no longer influences future execution to reduce state explosion further.

Harmony also features finding a short counter-example. Harmony orders counter-examples first by the number of context switches, then by the number of states encountered. Finding the shortest counter-example in the Kripke structure is unfortunately an expensive operation, not scaling well with the size of the Kripke structure. Instead, Harmony uses a cheap approximation. To this end, Harmony maintains some information per state and per edge in the Kripke structure. For each state S, there is a backpointer to the state from which S first was computed. Therefore, Harmony provides a trivial path back to the initial state. Given that the Kripke structure is computed breadth-first, this path is

minimum in the number of states, but not necessarily in the number of context switches.

To optimize the length, Harmony reorders the steps to minimize the number of context switches. There are constraints on this reordering. First, it is not possible to reorder the steps of a thread. Second, steps by different threads conflict if they access the same variable and at least one of these accesses is a store operation. They also conflict if both steps print something. Conflicting steps cannot be reordered. For this optimization, the Kripke structure maintains for each edge the accesses that were done and the values that were printed.

3.3 Implementation

The performance of model checking can benefit from concurrent execution. A typical model checker, including Charm, maintains a queue of states that must be evaluated. It is initialized with the initial state. Then, iteratively, for each state of the queue, the neighboring states are computed. A neighboring state thus computed may or may not already be in the Kripke structure. If not, it has to be added to the Kripke structure and also placed on the queue.

These evaluations are done in parallel through a set of worker threads in Charm. Each worker, one per core, executes in a tight loop of dequeuing a state and computing the resulting neighboring states. Each worker thread uses its private memory as much as possible, but the worker threads share the queue and the Kripke structure. Charm maintains the Kripke structure as a hash table, mapping states to metadata that includes the list of outgoing edges.

In order to maintain its O(1) performance, the number of buckets in the hash table must be increased dynamically. The queue must also grow dynamically, but here, we focus on the trickier issue of growing the hash table. The set of worker threads run in a loop of three phases separated by synchronization barriers:

1. evaluate states on the queue;
2. allocate additional memory for the hash table as needed;
3. if so, rehash the hash table.

The number of buckets is always a power of 2 for efficient rehashing. Phase (1) executes concurrently. After a number of states have been evaluated (which usually leads to a collection of new states), phase (2) decides whether the hash table must grow. Currently, this is done when fewer than half of the buckets are left empty. The number of buckets is then increased to 8 times the size of the old number of buckets. In the last phase, the workers split the hash table amongst themselves. They can concurrently rehash their section of the old hash table without interfering with the other workers. After completing this, they return to phase (1) to evaluate the remaining states on the queue.

After running the Charm model checker, there are two cases to consider. If the SVM aborted due to some problematic transition, which includes data races, then Charm reports the shortest counterexample in terms of path length in the Kripke structure. If not, the resulting graph is analyzed for various other problems, including deadlock and busy waiting. We detail below how some of these problems are found.

Deadlock and Livelock. Harmony transforms the Kripke structure into a DAG of its strongly connected components. The states in the sink components of

this DAG can be considered the terminal states of the Harmony program. States in which the context bag is empty represent states in which all threads have terminated. In the case the context bag is not empty, we check if there are any non-*eternal* contexts. If so, Harmony reports either a *deadlock* (if all non-eternal threads are blocked) or a *livelock* otherwise.

Active Busy Waiting. A thread is considered blocked if it can only modify the shared state if another thread updates the shared state first. This is not usually a problem: a thread waiting on a lock in a spinloop is a common way of synchronizing threads, and Harmony does not report such issues. We call this *passive busy waiting* as the thread does not modify the state.

A thread is considered *actively busy waiting* if it is in a loop updating the shared state, but cannot escape its strongly connected component without another thread also updating the shared state. A common example of this is a synchronization algorithm that checks for a condition in a loop, acquiring and releasing the lock each time. Harmony checks each strongly connected component for the existence of such situations.

Incomplete Behavior. The Kripke structure can be considered as a non-deterministic automaton, with on each edge zero or more *symbols* (values that were printed). If a behavior automaton was specified, then there are now two automata: the specified automaton $\mathcal{S}$ that was presented to Charm as an input as part of the *specification*, and the generated automaton $\mathcal{I}$ that was generated by the given Harmony *implementation*. Harmony already checks on-the-fly whether $\mathcal{I} \subseteq \mathcal{S}$, that is, that every behavior (sequence of symbols) of the implementation is also a behavior of the specification. Harmony also checks if $\mathcal{I} \neq \mathcal{S}$ and, if so, produces a warning.

Missing behaviors are not always a problem, but they can be. For example, a reader/writer lock implemented as a plain lock meets the specification of a reader/writer lock, but it does not allow multiple readers in the critical section. By running Harmony one more time, essentially reversing the implementation and the specification, Harmony can produce a shortest counter-example of such a behavior.

Data Races. If Harmony does not find other problems, it scans the graph for data races, that is, states in which multiple threads access the same shared variable and at least one those accesses is a store operation.

3.4 Evaluation

In this section, we show the applicability of Harmony for concurrent programming through three more use cases: dining philosophers, a concurrent queue, and a concurrent journaling file system. We also evaluate the performance of Harmony in those cases. We did experiments on two platforms, referred to below as *laptop* and *server*. The laptop is a 2019 Macbook Pro with a 2.8 GHz Intel Core i7 processor (4 cores, 2 hyperthreads per core) and 16 GB 2133 MHz LPDDR3 memory running MacOS Ventura 13.2.1. The server has a 2.8 GHz Intel Xeon(R) Gold 6242 CPU (16 cores, 2 hyperthreads per core) with 196 GB 2933 MHz DDR4 memory running Linux 5.15. (Harmony also runs on Windows—performance should be comparable as systems calls are rare.)

```
--algorithm diningRound0{
  variable fork = [k ∈ Procs ↦ N] ;

  define {
  forkAvailable(i) ≜ fork[i] = N
  LeftF(i) ≜ i
  RightF(i) ≜ IF (i = 0) THEN (N − 1) ELSE (i − 1)
   }

   fair process ( j ∈ Procs )
   variable state = "Thinking" ;
    {
    J0: while ( TRUE ) {
      H: either {
                if ( state = "Thinking" ) state := "Hungry" ;
                 }
       or P: {
          if ( state = "Hungry" ) {
                await (forkAvailable(RightF(self))) ;
                        fork[RightF(self)] := self ;
            E: await (forkAvailable(LeftF(self))) ;
                fork[LeftF(self)] := self ;
                state := "Eating" ;
              }
                }
      or T: {
               if ( state = "Eating" ) {
                    state := "Thinking" ;
                    fork[LeftF(self)] := N ;
                  R: fork[RightF(self)] := N ;
                  }
                 }
       }
     }
}
```

Figure 3.4: PlusCal/TLA+ implementation of Dining Philosophers (by Murat Demirbas). *fork*$[i] = N$ means fork i is available; otherwise, *fork*$[i]$ contains the identifier of the philosopher holding the fork.

```
byte fork[N];
byte nr_eat;

init {
    atomic {
        byte i = 0;
        do
        :: i < N -> run Philosopher(i); i++;
        :: else -> break;
        od;
    }
}

proctype Philosopher(byte id) {
Think:
    if
    :: atomic { id == 0 -> fork[1] == 0 -> fork[1] = 1; };
    :: atomic { id != 0 -> fork[id] == 0 -> fork[id] = id + 1; };
    fi;
One :
    if
    :: atomic {
        id == 0 -> fork[0] == 0 -> fork[0] = 1;
        nr_eat++;
    }
    :: atomic {
        id != 0 -> fork[(id + 1)%N] == 0 -> fork[(id + 1)%N] = id + 1;
        nr_eat++;
    }
    fi;
Eat:
    d_step { nr_eat--; fork[(id + 1)%N] = 0; }
    fork[id] = 0;
    goto Think;
}

#define SomeOneEats (nr_eat > 0)
never {
T0_init:
    if
    :: (!((SomeOneEats))) -> goto accept_S4
    :: (1) -> goto T0_init
    fi;
accept_S4:
    if
    :: (!((SomeOneEats))) -> goto accept_S4
    fi;
}
```

Figure 3.5: Promela/SPIN implementation of Dining Philosophers. *fork*$[i] = 0$ means fork i is available; otherwise, *fork*$[i]$ contains the identifier of the philosopher holding the fork.

```
1   from synch import Lock, acquire, release
2
3   const N = 5
4
5   fork = [Lock(),] * N
6
7   def diner(which):
8       let first = min(which, (which + 1) % N)
9       let second = max(which, (which + 1) % N):
10          while choose({ False, True }):
11              # think
12              acquire(?fork[first])
13              acquire(?fork[second])
14              # eat
15              release(?fork[first])
16              release(?fork[second])
17
18  for i in {0..N–1}:
19      spawn diner(i)
```

Figure 3.6: Harmony implementation of Dining Philosophers where forks are represented as locks. Because Harmony programs must have terminating executions, a diner eats 0 or more times before leaving.

3.4.1 Dining Philosophers

This section intends to demonstrate that using an input language with abstractions familiar to developers does not need to result in a reduction in scalability for the model checker.

The Dining Philosophers problem features a set of N threads acquiring a set of N locks. We evaluated this problem in two popular model checkers: TLA+/TLC (using the PlusCal surface language) and SPIN (using the Promela surface language), and compared with Harmony. Figure 3.4 shows a PlusCal implementation (due to Murat Demirbas); Figure 3.5 shows a Promela/SPIN implementation (author unknown); Figure 3.6 shows a Harmony implementation. PlusCal is transpiled to TLA+ and then model-checked by TLC, the TLA+ model

checker. TLA+ is in popular use by companies including Amazon, Microsoft, and Oracle. The Promela and Harmony solutions use a mix of left-handed and right-handed philosophers to remove deadlock, so that the entire state space is searched—this does not affect the search space size. The Harmony model leverages the lock specification (Figure 3.1).

Figure 3.7 shows the number of states generated by the models and the time it takes to compute those states by the respective model checkers as a function of the number of philosophers. The compiled code for the method `diner()` consists of 55 SVM instructions. This is the code that is model-checked by Charm. Without partial order reduction, the number of possible interleavings is astronomical ($(N \cdot M)!/(M!)^N$, where $M = 55$ and N is the number of philosophers). However, after partial order reduction, the number of generated states is similar to that of the SPIN model, which only has a few transitions. Note that the Harmony transitions are much closer to those of conventional hardware, and the Harmony program is trivial to translate to executable code. Harmony outperforms TLA+ by about three orders of magnitude.

The figure also demonstrates the efficacy of Harmony's memoization of computation steps. Harmony $--$ are the same experiments removing memoization from the model checker and runs out of memory for 10 diners. Section 3.4.5 provides additional evaluation of memoization.

3.4.2 Concurrent Queue

In this section, we show that Harmony is capable of modeling and checking implementations of concurrent algorithms, including those that use memory allo-

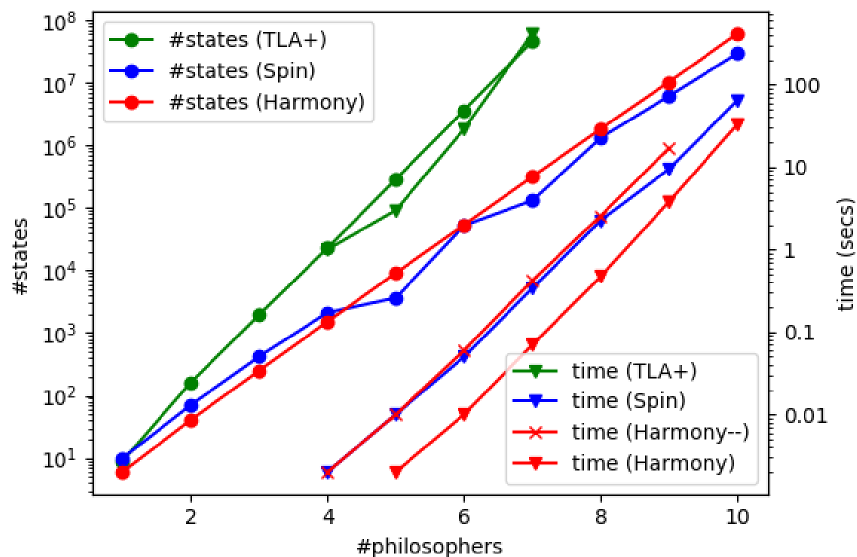

Figure 3.7: The number of states and times used to detect deadlock as function of the number of philosophers using the TLA+ model checker, SPIN, and Harmony. Harmony-- removes the memoization of computation. Times are measured on the laptop. Very short times are omitted.

```
1   def initialize() returns queue:
2       queue = [ ]
3
4   def enqueue(q, v):
5       atomically !q += [v,]
6
7   def dequeue(q) returns next:
8       atomically:
9           if !q == [ ]:
10              next = None
11          else:
12              next = (!q)[0]
13              del (!q)[0]
```

Figure 3.8: Harmony specification of a concurrent queue

```
structure node_t  {value: data type, next: pointer to node_t}
structure queue_t {Head: pointer to node_t, Tail: pointer to node_t, H_lock: lock type, T_lock: lock type}

initialize(Q: pointer to queue_t)
    node = new_node()
    node–>next.ptr = NULL
    Q–>Head = Q–>Tail = node
    Q–>H_lock = Q–>T_lock = FREE

enqueue(Q: pointer to queue_t, value: data type)
    node = new_node()
    node–>value = value
    node–>next.ptr = NULL
    lock(&Q–>T_lock)
        Q–>Tail–>next = node
        Q–>Tail = node
    unlock(&Q–>T_lock)
```

```
from synch import Lock, acquire, release
from alloc import malloc, free

def initialize() returns queue:
  let node = malloc({ .next: None }):
    queue = { .Head: node, .Tail: node,
              .H_lock: Lock(), .T_lock: Lock() }

def enqueue(Q, value):
  let node = malloc({ .value: value, .next: None }):
    acquire(?Q→T_lock)
    Q→Tail→next = node
    Q→Tail = node
    release(?Q→T_lock)
```

```
1  import queue

3  const NOPS = 3
4  tq = queue.initialize()

6  def enqueue_test(self):
7      print("call enq", self)
8      queue.enqueue(?tq, self)
9      print("done enq", self)

11 def dequeue_test(self):
12     print("call deq", self)
13     let v = queue.dequeue(?tq):
14         print("done deq", self, v)

16 for i in {1 .. NOPS}:
17     spawn enqueue_test(i)
18     spawn dequeue_test(i)
```

Figure 3.9: (a) A reproduction of the pseudo-code for the 1996 Michael&Scott two-lock concurrent queue implementation [87] and its equivalent in Harmony (within the box); (b) a behavioral test program

cation and pointers. Figure 3.8 presents a Harmony specification of a concurrent queue. The queue is represented by a list. In enqueue(q, v), variable q *points* to a variable containing a queue and the operation atomically adds v to the end of the list.

Figure 3.9(a) shows part of a reproduction of the well-known Michael&Scott two-lock concurrent queue [87] along with a Harmony version of the same algorithm (omitting the dequeue operation). Harmony complains, correctly,

	NOPS	#states	space	server	laptop
spec	1	30	0.000	0.00	0.00
	2	1,134	0.000	0.01	0.01
	3	54,882	0.002	0.02	0.03
impl	1	105	0.000	0.01	0.01
	2	14,772	0.000	0.01	0.01
	3	3,330,373	0.178	0.54	1.16
refi	1	113	0.000	0.01	0.01
	2	26,174	0.002	0.01	0.01
	3	22,991,203	3.697	2.81	10.69

Table 3.1: The number of states, memory usage (GBytes), and running time (seconds) of model checking the concurrent queue test program. The server used all 64 hyperthreads (32 cores), while the laptop used all 8 hyperthreads (4 cores).

about a data race in Figure 3.9(a) when there is a concurrent `enqueue` and `dequeue` operation on an empty queue. Harmony finds a similar issue with the lock-free algorithm in the same paper [87]. This issue is fixed in the following experiments by making load and store accesses to the *next* field atomic. We check if every behavior of the (corrected) concurrent queue implementation is also a behavior of the specification using Harmony's refinement check. This refinement ensures the implementation is *linearizable* [54].

Figure 3.9(b) shows a Harmony program that spawns 6 threads total, 3 that perform an `enqueue` operation and 3 that perform a `dequeue` operation. Each thread performs a `print` operation before and after the queue operation. The `print` operation before outputs the thread's intention to execute a particular operation, while the `print` operation after outputs the result of the operation.

We ran the test program in three configurations:

- `spec`: uses the queue specification (Figure 3.8);
- `impl`: uses the Michael/Scott implementation in Figure 3.9 (fixed as described above);
- `refi`: also runs the implementation, but checks refinement using the

specification-generated DFA.

We did this for `NOPS` = 1, 2, and 3 (Table 3.1). The refinement check leads to significant state explosion because the state of the specification-generated DFA becomes part of the implementation state. Still, even on the laptop this can be done under a minute for three `enqueue` and three `dequeue` operations executing concurrently.

3.4.3 Concurrent Journaling File Server

This section demonstrates that it is possible to scale model checking with Harmony to larger concurrent systems. While model checkers have been used to check I/O-concurrent file systems and crash consistency [132, 18, 118], to the best of our knowledge, the system described below is the first model-checked implementation of a file system that also involves concurrent processing.

To this end, we have implemented a file service in Harmony. The specification models the file system as a list of files, with each file being a list of blocks. Operations on the files are atomic and include `read`, `write`, and `getsize` operations. The implementation implements the same interface but uses a Unix-like file system data structure with inodes and indirect blocks stored on a disk. (We did not model double indirect blocks.) The free blocks are maintained using a bitmap. In order to support crash recovery, updates to multiple blocks are logged in a Write-Ahead Log (WAL) [88].

The disk itself is modeled as a list of blocks, where a block can be an arbitrary Harmony value. Updates to the disk are not atomic. The WAL supports an

operation that writes a set of blocks and updates the bitmap. The operation is not atomic either, but its operations are guaranteed to run to completion or not at all. In practice, this would be accomplished by storing the operation in a log and replaying the log after a crash.

The rest of the file system is implemented in about 200 lines of Harmony. It currently only supports operations to read and write files as well as get their sizes. The file system is implemented as a a collection of worker threads. Client threads communicate with the worker threads over blocking queues. There is one queue that clients use to send requests to the workers. Moreover, there is one queue per client that workers use to send responses to the client.

The workers synchronize with one another using locks. There is a single lock for the bitmap through which workers allocate and release blocks. Each inode block (which typically contain multiple inodes) has a reader/writer lock. Query operations on a file acquire a read lock on the inode of the file, while update operations acquire a write lock.

The test program is like the one in Figure 3.9(b). Each client non-deterministically chooses a file and an offset in the file to access using the **choose** operation. The program is first run against the file system specification and then against the file system implementation to check behavioral equivalence.

Running the test program demonstrates the efficacy of modular model checking. If we model a single system with just two files, each having at most two blocks, two worker threads, and two processes each selecting and requesting a read or write operation, then model checking using the specifications of locks, reader-writer locks, and blocking queues results in 35103 states. This took 2

Algorithm	#lines	#states	time
Reader/writer lock ($n = 8$)	58	441,929	1.97
Parallel bucketsort ($n = 5$)	41	2,340,909	3.60
Barrier synch ($n = 8$)	37	16,611,117	9.81
Non-blocking queue	31	23,864	0.57
Alternating Bit Protocol	51	2,778	0.68
Ring Leader Election ($n = 5$)	34	33,005	0.66
Two Phase Commit	71	666,316	2.32
ABD protocol ($f = 1$)	64	7,449,569	2.95
Chain Replication ($f = 2$)	75	201,408	0.91
Crash-tolerant Bosco ($f = 2$)	45	584,056	7.14
Byzantine Bosco ($f = 1$)	45	8,917	1.60
Paxos ($f = 1$)	65	22,059,648	36.56
Needham-Schroeder	42	558	1.34

Figure 3.10: Selection of other models evaluated in Harmony. Time is in seconds.

Megabytes of memory and 60 milliseconds. Next we replaced the modules by implementations. We used a futex implementation for locks and condition variables, a reader/writer lock based on condition variables, and a blocking queue by extending the concurrent queue implementation above with a condition variable. Model checking this fully refined file system took 7.7 seconds, resulting in 5,613,341 states and taking up 2.3 Gigabytes of memory. That is, modular model checking results in a roughly 3 orders of magnitude decrease in resources.

Model checking using specifications of the underlying modules allows us to check larger models with, for example, more operations and larger files.

3.4.4 Other Models

This section intends to demonstrate the generality of the Harmony tool. Figure 3.10 presents a selection of other models evaluated in Harmony. The reader/writer lock implementation is based on condition variables—all interleavings with up to 8 threads entering the critical section zero or more times are checked.

The parallel bubblesort checks sorting all sequences of up to 5 elements. The barrier synchronization is checked with 3 processes entering the barrier 4 times. The non-blocking queue implementation is from [53].

The remaining algorithms are distributed algorithms, where the network is modeled as a set of messages. The leader election algorithm uses 5 processes. For 2PC, the model checks up to 2 transfer transactions across three banks.

Crashes are modeled as interrupts. Harmony will try all possible crash points. The ABD protocol implements a replicated atomic read/write register using $2f + 1$ processes [7]. The chain replication is checked with chains up to three servers and two operations [106]. Bosco is a consensus protocol [114]. Paxos [76] is checked with up to one acceptor failure and up to one leader failure for up to three ballots. Finally, Harmony has also been used to find the well-known bug [81] in the Needham-Schroeder authentication protocol [95].

Model checking has been used before to verify certain properties of Paxos [124, 16, 37]. Usually Paxos is initialized in a certain state, and then, using model checking, it is verified that certain bad states are not reachable. In contrast, the Harmony Paxos model mentioned here starts in the initial state and runs through all possible reachable states with up to three ballots. Using a refinement check from an abstract consensus specification, we show that all behaviors of Paxos are also behaviors of consensus.

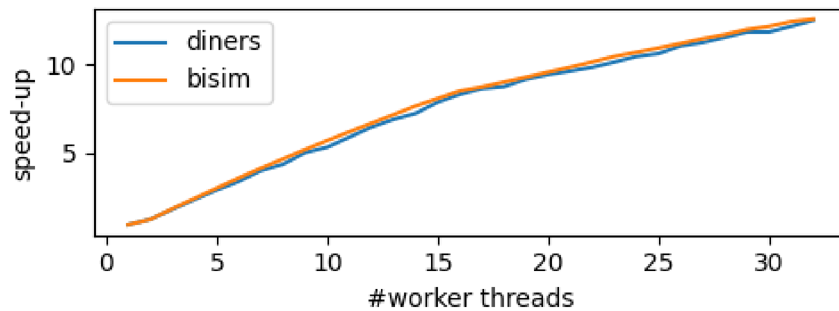

Figure 3.11: Speed-up of concurrent model checking as a function of the number of threads on the server. There are two models: Dining Philosophers (using 9 philosophers) and Weak Bisimulation of the Michael/Scott concurrent queue implementation using NOPS=3.

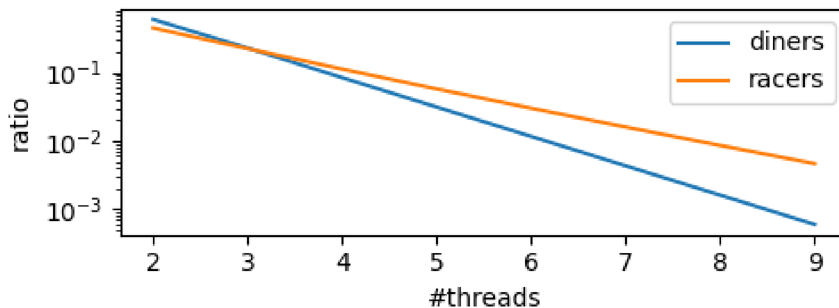

Figure 3.12: Number of SVM evaluations divided by the number of edges in the Kripke structure for different numbers of Harmony threads. Shown for both dining philosophers (diners) and barrier synchronization (racers)

3.4.5 Evaluation of Model Checking

The Harmony model checker is a concurrent C program. Figure 3.11 shows the speed-up obtained as a function of the number of worker threads. This is done for the Dining Philosophers and refinement of the Michael/Scott two-lock concurrent queue.

The server CPU has 16 cores and 2 hyperthreads per core. When running with up to 16 hyperthreads, Harmony pins the threads to one hyperthread per core. In so doing, the model checker achieves a speed-up of 8.3. With two hyperthreads

N	identified			anonymous			ratio
	states	time	size	states	time	size	
2	412	0.01	0	210	0.00	0	2.0
3	5k	0.01	0	990	0.00	0	5.3
4	64k	0.02	2	3771	0.00	0	17.0
5	768k	0.23	21	13k	0.01	0	59.3
6	9m	2.75	1221	43k	0.02	2	213.9

Table 3.2: Comparing identified and anonymous threads. Time is in seconds and size is in megabytes

per core and thus 32 threads, the model checker achieves a speedup of 12.7.

The reason for the sublinear speed-up is contention on the concurrent hash tables used by Charm. Charm uses three such hash tables: a hashmap that maps states to information about those states (such as incoming and outgoing edges), a hashset of all the Harmony values in use, and a hashmap that caches steps taken by the virtual machine. Experiments with different implementations of the hashmap, including lock-free ones, have not yet resulted in further performance improvements.

Figure 3.12 demonstrates the efficacy of caching SVM computations for the dining philosophers (diners) and for barrier synchronization (racers). As the number of Harmony threads increases, there is an exponentially increasing number of interleavings to explore, but the number of distinct computations grows much slower. In the limit, the model checker spends very little time evaluating SVM code compared to building the Kripke structure.

Table 3.2 demonstrates the efficacy of anonymous threads. The example used is a test program for reader/writer locks, with the number of threads (N) ranging from 2 to 6. Reported are the number of states, the time to model check, and the memory size used in megabytes. The ratio between the number of states is also given. When possible, anonymous threads can significantly reduce the

state space to explore without loss of completeness. We have used anonymous threads in various of the models, including the acceptors in Paxos.

Dead state elimination, where thread variables are removed from the state when no longer in use, can be effective at times. For example, the 2PC and ABD models in Figure 3.10 would not have been possible without it, even for the server. Scaled down experiments with those two models yielded a factor of 155 and 21 respectively of reduction in the number of states.

3.5 Preliminary User Study

This section describes our design and results of a preliminary user study that compares the usability of Harmony to Python in finding bugs in concurrent programs. Our hypothesis is as follows:

Hypothesis: System programmers without verification expertise can more easily find concurrency errors in programs written in Harmony using the Harmony tool than in programs written in Python using the standard Python execution environment.

Because Python is not a prototyping tool, this hypothesis does not directly address whether Harmony is a better prototyping tool than the state-of-the-art. Instead, testing this hypothesis provides empirical evidence for discussing a major aspect of the usability of verification tools: does our tool brings significant benefit without incurring too much burden to be counter-productive to regular system programmers?

As we want to analyze how our tool affects debugging concurrent programs,

this study requires human subjects. We recruited 8 undergraduate and 2 first-to-second-year graduate students majoring in computer science for a total of 10 volunteers for this preliminary user study. All students we recruited reported they had some familiarity with Python and concurrent programming, but little to no exposure to Harmony before the study. This population was split randomly into a Harmony group and a Python group, each containing 4 undergraduate students and 1 graduate student.

The study is designed to be autonomous: the students are given self-study material that contains a short tutorial followed by 3 tasks. The tutorial explains the format of the tasks and shows how to use their assigned tool to debug concurrent programs by walking through an example task. Each task first presents the students with one or more programs that involve basic concepts of concurrency, then asks multiple questions of the following form: is it possible for [program] to [exhibit certain behavior]? The programs and the questions in the materials for Harmony and Python are identical except for minor syntactic differences. The students are required to answer yes or no and provide an explanation. The whole study took a single session of approximately 90 minutes. We collected students' feedback through a survey after the study.

Table 3.3(a) shows the total score of the Harmony group compared to the Python group, with each question correctly answered worth one point. We discouraged students from guessing the answer and provided an "I don't know" option for them to skip the questions. Those answers are treated as incorrect in the table. Table 3.3(b) shows the average subjective difficulty of all students from a scale of 1 (very easy) to 5 (very difficult). It also reports the average subjective confidence of their answers for the two groups on a scale from 1 (very uncertain)

	Harmony	Python
0	10/10	7/10
1	12/15	12/15
2	32/50	16/50
Total	54/75	35/75

(a)

	0	1	2
Difficulty	2.4	2.8	4.2
Confidence (H)	4.2	3.8	2.2
Confidence (P)	4.2	4.2	1.5

(b)

Table 3.3: Summary of (a) study results; and (b) student feedback for the three tasks (0, 1, 2)

to 5 (very certain).

By our design, task 0 is the simplest, and task 2 is the hardest. This is also reflected in students' evaluation of the difficulty and their confidence. Although Table 3.3(a) shows the Harmony group has a higher score (54 out of possible 75) than the Python group (35 out of possible 75), and an even larger gap for task 2, we cannot draw any conclusion with statistical significance from this preliminary study due to our small sample size. One of our future goals is to conduct a larger-scale study with a wider range of participants.

3.6 Related Work

Harmony is preceded by many other model checking tools and languages for concurrent systems, including Alloy [64], CDSchecker [97], CHESS [91]. CMC [92], Coyote [36], Cosmo [10], Java Pathfinder [51], DSLabs [86], JBMC [32], MaceMC [66], NuSMV [29], PlusCal/TLA+ [78], Prism [73], SLMC [20], SPIN/Promela [57], Verisoft [47], and Zing [4]. Harmony differs from these systems in a variety of ways.

First, Harmony reduces the learning curve for non-experts: a Python-like programming language, a conventional system model (including a call stack,

memory allocation, and interrupts), behavioral specifications, and easy-to-read counter-examples. In particular, Harmony has built-in support for verifying weak bisimulation relations between specification and implementation, eschewing the need for complicated temporal formulas to capture properties such as fairness and linearizability.

Second, Harmony uses a plurality of techniques, some novel, to reduce state explosion. These include first-class support for modular model checking, partial order reduction, memoization of execution steps and effects, anonymous threads, and dead state elimination. The SVM is designed to support effective partial order reduction and memoization.

Model checkers have been used to check file systems [132, 18, 118]. These mostly focus on correct sequential execution and crash recovery. The example used in this chapter to demonstrate modular model checking verifies correctness of a concurrent implementation of a file system.

We are also strongly influenced by empirical studies on programming language usability. As pointed out in [116], questions regarding the usability aspect of programming languages and related tools can and should be discussed with empirical evidence. The study most related to ours is [94], which compared the concurrency primitives in Java and SCOOP in the educational setting.

3.7 Future Work

Part of the ongoing work on Harmony is to support differential behavior testing of native C code with a specification and a test program written in Harmony. We

treat native code as black boxes with a callable API interface and use behaviors specified by the Harmony program to guide the exploration of the behaviors of the native code. This differential testing of native code trades off exhaustiveness for testing production code directly. The behaviors of the native code cannot be fully explored without controlling the implicit states of the operating system and the hardware, which would require considerable task-specific effort. In exchange for exhaustiveness, this methodology does not require (re-)implementing the system in Harmony and closes the gap between the verified prototype and the actual code, thus a more flexible and practical option for system builders.

Another direction we are actively exploring is to further improve the performance of Harmony through finer-grain static and dynamic analysis of Harmony programs, which is only possible thanks to the design of our semantics. For instance, when model checking, Harmony currently employs the optimization that it runs a local thread continuously until a breakpoint happens, such as reading or writing a piece of the shared global state. It then reselects a local thread from the available pool to continue execution. An insight is that not every piece of the shared global state influences every part of the execution of the local threads. In fact, the influence relation is sparse, leading to redundant exploration of the program behaviors. This redundancy can be reduced by a finer-grain analysis of the program and the runtime state.

We would also like to incorporate more communal effort in refining Harmony. Harmony is already open source, but we plan to further improve its code quality and test coverage and introduce Harmony to the general public through community events.

3.8 Conclusion

We introduce Harmony, a prototyping tool for concurrent programs that provides both usability and scalability for students and system programmers. Harmony has a Python-like syntax and is built upon the Harmony Virtual Machine, a virtual machine semantics that captures the vital aspects of concurrent program behaviors. Harmony uses the SVM design and model checking to provide better performance and gives easy-to-interpret feedback to the users. Its support for behavioral equivalence simplifies the specification of many important properties, while its support for modular model checking increases its scalability. Harmony also introduces novel optimizations, including anonymous threads, dead state elimination, and effect memoization, further increasing its scalability.

CHAPTER 4

ASN1⋆

Abstract Syntax Notation One (ASN.1) is a data type declaration language standardized by both ITU-T and ISO/IEC since 1984.[1] It is used for exchanging structured data between platforms in a variety of settings, notably in the X.509 [15] standard for public-key certificates. The latter forms the cornerstone of digital identities and secure communication on the Internet and, as such, the ASN.1 and X.509 standards and their implementations are security critical components of societal infrastructure.

The ASN.1 language supports describing structured data of many varieties, including a wide collection of base types, products, sums, sequences, and sets. For example, we give below an ASN.1 *declaration* for two-dimensional points, where the base type `INTEGER` denotes integers of arbitrary size.

```
1 Point2D ::= SEQUENCE { x INTEGER, y INTEGER }
```

ASN.1 declarations can be grouped into ASN.1 modules. For example, the format of X.509 certificates is one such ASN.1 module. We give below its top-level declaration, a triple of fields:

```
1 Certificate ::= SEQUENCE {
2       tbsCertificate TBSCertificate,
3       signatureAlgorithm AlgorithmIdentifier,
4       signature BIT STRING }
```

where `tbsCertificate` is the certificate contents 'to be signed' using `signatureAlgorithm`, and `signature` is the resulting signature value.

[1] `https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx`

ASN.1 decouples data type declarations from their formats. It provides several classes of *encoding rules* that govern the wire format of data types, one of which known as the *distinguished encoding rules* or DER, following the general tag-length-contents encoding pattern. For example, the point $(0, 0)$ is encoded into the 8-byte string ”30 06 02 01 00 02 01 00” where 30 is the tag locally assigned to points in their ASN.1 module, 02 is the primitive tag of integers, and 06, 01, 01 encode their content lengths.

DER are designed to ensure that every value of a given ASN.1 type has a distinct, canonical wire format representation. That is, DER formats are intended to be *unambiguous* and *non-malleable*, in the sense that given a bit string b that encodes a value v, every parser will yield back v, whereas changing any bit in b either produces an invalid representation or yields a distinct value $v' \neq v$. These properties are particularly important in security applications, inasmuch as they depend on values v but apply cryptographic protection only on binary formats b. In particular, the X.509 standard[2] requires that certificates be formatted using DER, to prevent any ambiguity between the claims signed by the issuer in `tbsCertificate` and their interpretation by the relying party after verifying the signature.

Despite the maturity of the standard and the presence of libraries in several languages that support their use, ASN.1 and DER have a reputation for being difficult to master. Implementations have suffered from parsing bugs that have led to critical vulnerabilities. For example, [84] discovered that Microsoft’s CryptoAPI component would incorrectly parse a string containing a null character in a domain name in the subject’s Common Name (CN) field of an X.509 certificate, e.g., parsing the string `"a.com\0b.com"` as `"a.com"` thereby misinterpreting

[2] `https://www.rfc-editor.org/rfc/rfc5280`

the certificate issuer's intent and enabling an attacker to spoof a certificate to carry out a man-in-the-middle attack. This is a classic example of security vulnerability due to the use of a malleable parser—the parser simply ignores the content of the string after the null character. We discuss other security vulnerabilities related to X.509 parsing in §4.4. Of course, many vulnerabilities discovered in implementations of X.509 and related standards involve software flaws beyond parsing (e.g., in certificate chain validation [19])—however, ensuring that parsing is correct and non-malleable is a necessary basic requirement.

ASN1⋆: A Formalization of ASN.1 DER

Our long-term ambition is to provide high-assurance implementations of tools to parse and serialize data to and from ASN.1 DER, and to build provably correct cryptographic applications upon such tools. We present a first milestone towards that long-term goal, namely ASN1⋆, a mathematical formalization of ASN.1 DER, deeply embedding its syntax and providing several related denotational semantics within the $F^\star$ proof assistant [119]. It provides a precise, mathematical basis on which to understand and further study a widely used Internet standard that has, to date, only been specified in several voluminous natural-language documents.

We formalize the syntax of ASN.1 DER as a family of mutually inductive indexed types, the primary one being $\mathsf{declaration} : \mathsf{set}\ \mathsf{id_t} \rightarrow \mathsf{Type}$, the type of a single ASN.1 declaration. For example, `Point2D` and `Certificate` are represented in $F^\star$ as instances of declaration. The index on declaration enforces a certain well-formedness property on ASN.1 DER specifications, a form of static discipline discussed in §4.2.

We provide two related denotational semantics. First, a *type denotation* asn1_as_type : declaration s → Type that interprets every well-formed ASN.1 DER declaration as a type in the meta-language, i.e., F*. For example, the type denotation of `Point2D` is an F* pair of mathematical integers, int & int. Second, a *parser denotation* that interprets every declaration as a pure function from a sequence of bytes (a DER wire format) to either a value of its type denotation or an error. Our main theorem, outlined below

```
1 val asn1_as_parser : (d:declaration s) → parser (asn1_as_type d)
```

establishes that our parser denotation can be typed as a `parser`, the type of correct, non-malleable parsers defined in the EverParse framework [105], applied to our type denotation. (§4.1 provides background on F* and EverParse.) That is, we show that every well-formed ASN.1 DER declaration can be interpreted both as an F* type and a non-malleable parser from a sequence of bytes to that type.

A key technical contribution of our development is that it yields a *compositional* semantics of ASN.1 DER where, despite complications of the standard such as optional elements, default elements, and local retagging, (which require careful custom treatment) our top-level theorem still offers a clear, canonical correctness and non-malleability result in terms of EverParse's parser abstraction. To this end, we also contribute new parser combinators, notably for sequence, choice, and state-machine-based parsers, together with their proofs of correctness and non-malleability.

Validating ASN1$\star$

To validate that our formalization corresponds to the practice of ASN.1 DER in existing standards and interfaces, we use F*'s extraction mechanism to produce, for selected ASN.1 declarations expressed as instances of $\mathsf{v : declaration\ s}$, functions in OCaml that parses a sequence of bytes. We wrote ASN1$\star$ format declarations for X.509 version 3 certificates, covering its most popular extensions, and tested our extracted OCaml parser on a corpus of more than 10,000 certificates, including both positive and negative test cases, confirming that we correctly handle them all. We also tested on a further $\sim$2,000 (mostly ill-formed) certificates dataset produced by fuzzing, and again confirmed that we correctly handle them all. We also wrote a ASN1$\star$ format declarations for Certificate Revocation Lists (CRLs) and evaluated our parsers on $\sim$4,000 CRLs found in the wild.

Extensions and Limitations

Our formalization aims to cover a practical version of ASN.1 DER, sufficient to express many formats used in the wild. We support features that are not core to ASN.1 but are commonly used in informal side conditions. For example, many specifications prescribe additional formatting constraints in natural language, e.g., X.509 has a notion of *expansion lists*, which our formalization does cover. On the other hand, we do not support a form of set that is seldom used with DER and does not occur in our case studies (see §4.3.1).

Although our formalization offers executable OCaml code for parsing, we have not attempted to optimize this code at all, and make no claims about its efficiency. Indeed, as mentioned earlier, we see our work as "merely" the

formal foundation towards producing in the future high-performance, provably correct, low-level implementations of ASN.1 DER parsers and serializers, and cryptographic applications to be built using them, including certificate chain and policy validation.

In summary, our contributions include:

1. The first formalization of ASN.1 DER, providing a basis on which to understand long-standing, widely used natural language standards. Our main theorem proves that all well-formed ASN.1 DER specifications induce non-malleable parsers.
2. New correct- and non-malleable-by-construction parser combinators for sequences, choice, and state-machine-based parsers.
3. An experimental validation of our formalization by evaluating the parsers from our semantics on a corpus of ASN.1 DER formatted data in the wild, including for X.509 and CRL, confirming that our semantics is faithful to the intent of the official standard.

ASN1⋆ is publicly available as a pull request into EverParse: `https://github.com/project-everest/everparse/pull/66`.

4.1 Background on F⋆ and EverParse

F⋆ is a programming language and proof assistant based on a dependent type theory (like Coq, Agda, or Lean). F⋆ also offers an effect system, extensible with user-defined effects, and makes use of SMT solving to automate some proofs.

F⋆ syntax is roughly modeled on OCaml (val, let, match etc.) with differences to account for the additional typing features. Binding occurrences b of variables take the form x:t, declaring a variable x at type t; or #x:t indicating that the binding is for an implicit argument. The syntax $\lambda(b_1) \ldots (b_n) \rightarrow t$ introduces a lambda abstraction, whereas $b_1 \rightarrow \ldots \rightarrow b_n \rightarrow c$ is the shape of a curried function type. Refinement types are written b{t}, e.g., x:int{x≥ 0} is the type of non-negative integers (i.e., nat). As usual, a bound variable is in scope to the right of its binding; we omit the type in a binding when it can be inferred; and for non-dependent function types, we omit the variable name. The c to the right of an arrow is a *computation type*. An example of a computation type is Tot bool, the type of total computations returning a boolean. By default, function arrows have Tot co-domains, so, rather than decorating the right-hand side of every arrow with a Tot, the type of, say, the pure append function on vectors can be written #a:Type → #m:nat → #n:nat → vec a m → vec a n → vec a (m+n), with the two explicit arguments and the return type depending on the three implicit arguments marked with '#'. We often omit implicit binders and treat all unbound names as implicitly bound at the top, e.g., vec a m → vec a n → vec a (m + n)

F⋆ programs are not executable per se. Instead, F⋆ extracts OCaml code from F⋆ code. To this end, F⋆ distinguishes between pure computations, which extract to OCaml, and *ghost* computations for proof purposes only (where use of axioms such as excluded middle or indefinite description is allowed), erased at extraction. (F⋆ also supports effectful code, and extraction to C via Low*, a fragment of F⋆ shallowly embedding a subset of C, but this is out of the scope of our treatment.)

EverParse

EverParse is a formally verified library and toolchain to build verified parsers and serializers for binary data formats such as TLS or network virtualization protocols. Formal guarantees supported by EverParse include proofs of unique binary representation, a.k.a. non-malleability, for the purpose of secure authentication and hashing; proofs that serializer and parser are (partial) inverse of each other; bounds on the size of the byte representation. (EverParse also allows generating executable C code for such parsers, via the Low* fragment of F⋆, allowing some performance optimizations, for which EverParse proves memory safety, arithmetic safety, functional correctness with respect to the original parser specification.) To establish such guarantees, EverParse builds on its core component, called LowParse, a library of monadic parser and serializer combinators formally verified in F⋆. Such parser combinators supported by LowParse include dependent pairs (a.k.a. tagged unions), filter refinements, rewriters, lists, and data prefixed with its size in bytes. Such combinators were initially tailored to support formats such as TLS handshake messages. On top of LowParse, EverParse provides several front-ends: QuackyDucky [105] targeting TLS handshake messages, and 3D [120] targeting network virtualization packets. With those front-ends, EverParse allows users to define their data formats in a high-level descriptive language, and to push a button to automatically generate formally verified parser and serializer code for their formats, by assembling LowParse combinators, with zero user proof effort. Thus, EverParse as a toolchain is similar in spirit to recent efforts in automatic parser generation for binary data formats such as Protocol Buffers or Cap'n Proto, except that, contrary to EverParse, those two toolchains come with their own classes of supported data formats, excluding existing network protocol formats (one cannot, say, define the TLS handshake

message formats in Protocol Buffers.) Moreover, EverParse distinguishes itself by generating formally verified code.

4.2 A Brief Primer on ASN.1 and DER

Figure 4.1 presents an informal summary of the concrete syntax of ASN.1, distilled from the ITU's X.680 standard [63]. Figure 4.2 shows an actual snippet of ASN.1 declaring the type of X.509 to-be-signed certificate contents introduced in §4. We use them to establish some basic concepts and intuitions, and to convey some of the challenges involved in their formalization, presented next in §4.3.

An ASN.1 module declares a collection of data types, including finite sums, dependent and non-dependent products, variable-length sets and lists over a collection of base types. Each module is a list of declarations; each declaration associates a name with either a constant value (such as an object identifier) or a data type, and may refer to prior declarations by name. In Figure 4.2, for example, Version and AlgorithmIdentifier refer to prior declarations in scope.

A data type is either a *terminal*, such as an integer, or a type constructed from more basic types: a SEQUENCE is the product of a given list of field names f_i and *decorated* declarations, where the decorations can marks a field as optional, provide a default value when the field is omitted, and modify its tag—we discuss this in detail shortly; a SEQUENCE OF is a list of an arbitrary number of t-typed elements; the CHOICE constructor is the sum of a given list of data types. ASN.1 also offers SET and SET OF constructors that are unordered analogs of SEQUENCE and SEQUENCE OF.

A design goal of ASN.1 is to decouple type declarations from their binary formats. To this end, ASN.1 settles on an encoding scheme where all data type values are encoded in binary as identifier-length-content (ILC) tuples—the precise form of these tuples varies between the different encoding rules that ASN.1 provides, DER, our focus, being among them. The identifier, or tag, mainly serves as an indicator for the type of the value, for example, to distinguish between different cases of sum. The length specifies the length of the content field in bytes and eliminates ambiguity when a binary string can be fragmented in different ways. Although the identifier and the length fields are not always necessary, they usually do not cause much overhead, and they enable applications to skip over contents in binaries.

Primitive ASN.1 types have their own built-in identifiers. For example, the type INTEGER has identifier 02 (in hex), so 0 is in ASN.1 DER as 02 01 00, where the first byte is the identifier for integers, the second is the length of the content (1 byte); and 00 is the content itself.

ASN.1 allows users to *override* the (otherwise decoupled) binary encoding of identifiers for their declarations with the IMPLICIT decoration. For example, one can declare MYINT ::= [1] IMPLICIT INTEGER, and the encoding of 0 as a MYINT becomes 81 01 00. The identifier byte 81 expanded in binary digits is 10 0 00001, where the first two bits indicate that this is a context-specific user-defined identifier, the next bit indicates that the data type is primitive, and the last 5 bits encode the user-chosen constant 1.

Identifier formats are actually variable-length. For example, a long identifier such as [128] IMPLICIT takes 3 bytes: the first byte is 10 0 11111, where the first 3 bits are as before, but the last five signal a long-form identifier. The next two

bytes are 1 0000001 and 0 0000000, where the leading bit of the first byte signals that more bytes are to follow, and the leading bit of the third byte signals that this is the final byte of the identifier, overall representing 8 bits spread across the last two bytes. Note that a correct parser must reject unnecessary long forms, as they would break non-malleability.

ASN.1 also allows to *wrap* an encoding within a custom ILC tuple with the EXPLICIT decoration. For example, the encoding of 0 as a WRAPPED_INT ::= [1] EXPLICIT INTEGER is A1 03 02 01 00, where the leading A1 in binary is 10 1 00001, representing a constructed user-defined short identifier; the length of the wrapped contents is 3; and the content itself is the built-in encoding of 0.

$$
\begin{array}{lllr}
c & ::= & \mathsf{INTEGER} \mid \mathsf{BITSTRING} \mid \ldots & \textit{Terminals} \\
t & ::= & c \mid \mathsf{SEQUENCE}\ \{f_1\ \tau_1, \ldots, f_n\ \tau_n\} & \textit{Declarations} \\
 & \mid & \mathsf{CHOICE}\ \{f_1\ \tau_1, \ldots, f_n\ \tau_n\} & \\
 & \mid & \mathsf{SEQUENCE\ OF}\ t \mid \mathsf{SET\ OF}\ t \mid \ldots & \\
\tau & ::= & t \mid \tau\ \mathsf{OPTIONAL} \mid \tau\ \mathsf{DEFAULT}\ v & \textit{Decorated decls} \\
 & \mid & [n]\ \mathsf{EXPLICIT}\ \tau \mid [n]\ \mathsf{IMPLICIT}\ \tau &
\end{array}
$$

Figure 4.1: Informal syntax of ASN.1

```
1  TBSCertificate ::= SEQUENCE {
2    version [0] EXPLICIT Version DEFAULT v1,
3    serialNumber CertificateSerialNumber,
4    signature AlgorithmIdentifier,
5    issuer Name,
6    validity Validity,
7    subject Name,
8    subjectPublicKeyInfo SubjectPublicKeyInfo,
9    issuerUniqueID [1] IMPLICIT Uid OPTIONAL,
10   subjectUniqueID [2] IMPLICIT Uid OPTIONAL,
11   extensions [3] EXPLICIT Extensions OPTIONAL }
```

Figure 4.2: An ASN.1 declaration from X.509

ASN.1 has further decorations to mark certain fields in sequence as optional, or optional with default values. For example, in a TBSCertificate the Version field

may be omitted in binary format, which must be interpreted as the constant v1 (a value in scope), and any of the last three fields may also be omitted. This complicates parsing, and motivates the use of IMPLICIT and EXPLICIT identifiers to prevent any ambiguity. For example, when parsing the optional field Uid, if the next byte encodes the identifier for [1] IMPLICIT, then the content must be a Uid, but if it encodes the identifier for [3] EXPLICIT, then both Uid fields are absent, and one should start parsing the extensions. (Binary encodings of Extensions may start with any identifier, hence the need to wrap them.)

To ensure that a declaration can be unambiguously parsed there are various well-formedness conditions, e.g. all the fields in a consecutive block of OPTIONAL and DEFAULT fields, and the plain field that immediately follows them (if any) must have distinct identifiers. As such, not every syntactic instance of an ASN.1 declaration is admissible.

4.3 ASN1⋆

Figure 4.3 summarizes our formalization of ASN.1. In §4.3.1, we present an intrinsically typed syntax for ASN.1, whose typing constraints ensure the well-formedness of ASN.1 declarations. We offer some syntactic conveniences to help transcribe ASN.1 concrete syntax into our formal ASN1⋆ notation, though the correspondence is only established empirically. In §4.3.2, we show that every well-formed ASN1⋆ term can be denoted as an F* type. This part of our semantics is independent of the binary format, in keeping with the ASN.1 view that the type declarations and binary representations are to be decoupled. §4.3.3 contains our main formal result, namely that every ASN1⋆ term has a denotation as a non-

malleable parser for values of the type denotation. Our parser semantics yields OCaml code for parsing ASN.1 DER formatted data, and in §4.4 we test our code against concrete ASN.1 DER binary formatted data to confirm empirically that our semantics is faithful to the ASN.1 DER standard.

4.3.1 Syntax and Well-Formedness of ASN.1

Figure 4.4 shows the formal syntax and well-formedness rules of ASN1⋆. We omit the definition of terminal_k, the language of terminal types, and their interpretation as F* types, terminal_t : terminal_k → Type. The content type is the core syntax of taggable content, while the declaration type associates an identifier with a content term—we leave the length out of the specification, since it is a dynamically computed value. The d_declaration type associates a decoration with an declaration value, and decorated and decorateds are just abbreviations. For compactness, we adopt a convention where free names are universally bound as implicit parameters at the top of the type of each constructor.

Identifiers

The type id_t below models identifiers, as explained in §4.2. For example, the identifier [2] IMPLICIT encoded as byte 10 0 00010 has class CONTEXT_SPECIFIC, flag PRIMITIVE, and value 2. We bound identifier values to 32 bits, though we could have also chosen to use unbounded integers in F*—identifiers longer 32 bits are very uncommon.

```
1 type id_class_t = | UNIVERSAL | APPLICATION | PRIVATE
2                   | CONTEXT_SPECIFIC
```

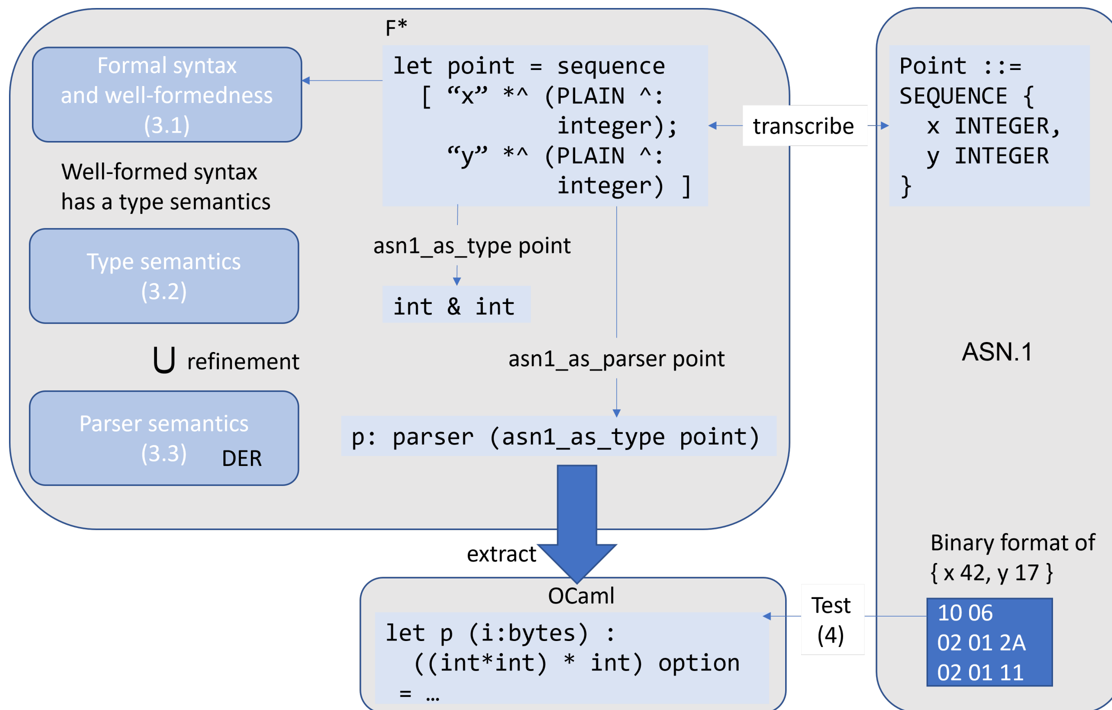

Figure 4.3: Architecture of our development

```
3 type id_flag_t = | PRIMITIVE | CONSTRUCTED
4 type id_t = {class:id_class_t; flag:id_flag_t; value:U32.t}
```

The content Type

The TERMINAL constructor supports a form of decidable refinement. For example, to represent the type of natural numbers less than 4, one can write TERMINAL INTEGER ($\lambda v \to 0 \leq v$ && $v < 4$). Similar side conditions expressed in natural languages are not strictly a part of ASN.1 as a data format language. But they are very common in the specifications that use ASN.1. While complex semantic properties, for instance, the validity of a signature, are out of scope for a parser, simple cases such as a non-empty list or an integer with bounds are easy to check. So we include them in our formal language as refinement types.

SEQUENCE_OF, and SET_OF are just like their informal analogs in Figure 4.1. SEQUENCE is almost the case with an extra proof obligation in decorateds. This proof term ensures that the identifiers and the decorators of the sequence do not lead to ambiguity. For instance, the valid set of identifiers of an option field cannot intersect with the set of the following field.

PREFIXED models the wrapping of data types using EXPLICIT, e.g., ILC id (PREFIXED t) require that the inner type be wrapped with identifier id.

ANY_DEFINED_BY is the most complex content type. For example, the X.509 specification has a type for (mathematical) fields of characteristic two for some elliptic curves, given below in ASN.1 concrete syntax.

```
1 Characteristic−two ::= SEQUENCE {
2     m INTEGER, - Field size 2^m
3     basis OBJECT IDENTIFIER,
4     parameters ANY DEFINED BY basis }
```

This declares a record of an integer m, followed by an object identifier basis, and then some parameters whose legal values are determined by the value of basis. The specification also includes (in natural language text) the basis/parameters pairs that are supported. In the constructor ANY_DEFINED_BY, the prefix represents fields (such as m) that precede the keys and values. The fields id and key are the identifier and type of the keys, which must be a terminal type (such as OBJECT IDENTIFIER). The field kvs represents the supported key-value pairs. The field def is an optional default value, which some specifications use to represent a default case not included in kvs. The final field is a proof obligation. (squash p is the F* type of proof-irrelevant proofs of p.) It confirms that the fields in prefix and id are well-formed, similar to the case of `SEQUENCE`. It also excludes

repeats in the kvs list and its encodings.

Although ASN.1 includes a SET constructor, ASN1⋆ does not support it. Much like SEQUENCE, SET is used to declare a record, but with the intent that the ordering of its fields is unimportant. This is at odds with DER, which requires that binary representations of elements of SET and SET_OF be strictly sorted. We decided to fully support SET_OF but to ignore SET, since it does not occur in any of our case studies; it can usually be replaced with a SEQUENCE with the same fields and a simpler format; and it would require parsers for corner cases such as

```
1 SET { [2] IMPLICIT INTEGER,
2       CHOICE { [1] IMPLICIT INTEGER,
3                [3] IMPLICIT INTEGER }}
```

which declares a pair of integers, but insists their binary format order them by tags: either 1,2 or 2,3. (By contrast, SET OF declares sets where all elements have the same type, so we check their representations are strictly ordered but need not consider re-orderings.)

The declaration Type

The declaration s type associates an identifier with a content type, where the index s represents the set of valid first identifiers that may be encountered in the binary format of the type—this is used below in the well-formedness of decorated types. The CHOICE_ILC is for a sum and associates a distinct identifier with every content type in the sum. Finally, the ANY_ILC is used to represent any identifier-length-content tuple.

The decorated Type

The type d_declaration associates a decoration with an declaration type. The DEFAULT case supports refined terminals and requires a proof that the default value satisfies the refinement. Rather than using d_declaration, we use its packaged variants decorated and decorateds. The latter type enforces that all the fields in a consecutive block of OPTION and DEFAULT fields, and the PLAIN field that immediately follows them (if any) have distinct identifiers.

Smart Constructors

Writing a value of type declaration directly from its constructors can be tedious, especially due to the proof obligations on several of the constructors. To assist with this, we introduce a layer of smart constructors that internalize some of the proof obligations and provide tactics for them. These constructors enable writing specifications in our embedded declaration language in a style relatively close to the concrete ASN.1 syntax, while also formally capturing constraints that are typically left to natural language in concrete specifications. For example, we give below the specification in ASN1⋆ of the `Characteristic-two` declaration presented earlier, with an asn1_integer m as prefix, followed by the key name basis, and a choice between the three legal key-value pairs—the proof obligations are dispatched by seq_tac and choice_tac, tactics we developed for ASN1⋆.

```
1 let characteristic_two = asn1_any_oid_prefix
2      ["m" *^ (PLAIN ^: asn1_integer)]
3      "basis"
4      [(gnBasis_oid, gnBasis_parameters);
5       (tpBasis_oid, tpBasis_parameters);
```

```
6         (ppBasis_oid, ppBasis_parameters)]
7       (_ by (seq_tac())) (_ by (choice_tac()))
```

In the future, we may leverage user-defined syntax extensions proposed for F* to streamline this further.

4.3.2 Denoting ASN1⋆ Declarations as F* Types

Figure 4.5 shows our interpretation of ASN1⋆ syntax as F* types, following the structure of the inductive type definitions in Figure 4.4. In the spirit of ASN.1, this first denotational semantics is independent of the binary representation.

Denoting content

TERMINAL t v is interpreted as an F* refinement of the denotation of t. SEQUENCE ds is interpreted as an n-ary tuple, where n is the length of ds, followed by a trailing unit (left here for simplicity, but optimized away in our implementation). SEQUENCE_OF and SET_OF are both denoted as lists. In principle, the latter could be quotiented by a relation that equates lists up to permutation, though F* lacks native support for quotient types. PREFIXED only affects the binary format and has no effect on the type denotation. ANY_DEFINED_BY is represented as a tuple beginning with prefix followed by a sum defined by the kv association-list, with an optional default case.

Denoting declaration

In an ILC id k, the identifier id concerns only the binary format. The CHOICE_ILC lc case maps the content_t interpretation over the list of cases, and then forms a (strong) sum type, *aka* a dependent pair, where the type is uninhabited in the case of an unexpected identifier. Finally, ANY_ILC is just a pair of an identifier and a string of bytes. Although we could have written helper functions like any_t and cases_t using combinators like map, F*'s termination checking rules make it much easier to write explicit, mutually recursive definitions in place. Additionally, the termination checker needs a couple of hints in the form of decreases annotations to accept this definition.

Denoting Decorated Types

The denotation of decorated types is straightforward, with PLAIN having no impact; OPTION denoted as an option; and DEFAULT_TERMINAL denoted as default_tv defaultv, a refined form of option with constructors Default and Nondefault of (v:_ { v ≠ defaultv }).

Terminals

Our semantics of terminals formally capture the properties of many ASN.1 types previously described only in natural language. We omit the details, and only discuss the UTF8String terminal, loosely defined in the standard as any byte string tagged with a special identifier, followed by 13 pages of English text for the actual specification. The intended usage is to first parse its contents as a byte sequence, then to separately check that it is a valid UTF8String. Instead, we

encode those constraints directly with F* propositions and inductive types, and we prove that our parser, described next, only accepts values of this more precise type.

4.3.3 A Constructive Formalization of DER

Our main formal result, summarized in this section, is that every ASN1⋆ type definition t : declaration s can be interpreted as a parser asn1_as_parser t of a byte sequence representation of asn1_as_type t. The specific format accepted by our parsers is intended to represent ASN.1 DER. We prove that the parser asn1_as_parser t is *injective*, i.e., for every v:asn1_as_type t there exists at most one valid binary representation.[3] Thus, ASN.1 DER is a non-malleable format.

Injectivity of parsers is a *relational property* or a *hyperproperty* [30]. Proofs of hyperproperties are known to be challenging, with many special- and general-purpose logics proposed for various classes of hyperproperties [13, 12, 11, 50]. For the specific scenario of proving injectivity properties of parsers, the EverParse [105] library offers a family of injective-by-construction parser combinators. The library is structured around a type called parser k t, outlined below.

```
1 let parser (k:parser_kind) (t:Type) =
2   p:(b:bytes → option (t & n:nat { n ≤ length b })) {
3     has_kind k p ∧
4     (∀ b0 b1. match p b0, p b1 with
5          | Some (v0, l0), Some (v1, l1) →
```

[3]The converse property, that every v:asn1_as_type t has at least one valid binary representation is not guaranteed by our proofs, though we test the non-triviality of the generated parsers empirically. Furthermore, EverParse takes parsers as the primary building block, and defines serializers correct with respect to parsers. This choice is arbitrary, the converse design is also plausible.

```
6         v0 == v1 ⟹ slice b0 l0 = slice b1 l1
7       | _ → ⊤) }
```

In addition to injectivity, EverParse provides a language of *parser kinds* that characterize various other properties. For our purposes, we are interested in only two parser kinds, strong and weak, where strong parsers are insensitive to input extension. That is, appending any bytes to the input does not change the return value of a strong parser. We write weak_parser and strong_parser instead of parser weak and parser strong. Kinds are combined according to a small algebra, but we refer the reader to prior work on EverParse for the details.

EverParse provides several basic parsers and combinators to compose parsers, e.g., parse_u8 to parse a single byte, or nondep_then to parse two values in sequence while returning them as a pair. The type of combinators like nondep_then encodes a proof rule which ensures that the sequential composition of injective parsers is injective.

```
1 val parse_u8 : parser u8_kind U8.t
2 val nondep_then (p0:parser k0 t0) (p1:parser k0 t1)
3   : parser (and_then_kind k0 k1) (t0 & t1)
```

In giving a parser denotation to ASN.1, the main challenge was to define a compositional semantics so that both their type-correctness (that they parse well-typed values according to the type denotation) and their injectivity follow structurally. In the process, we also extended EverParse with new general-purpose, injective-by-construction parser combinators, notably a combinator parameterized by a state machine, which should be of interest and applicability beyond the context of ASN.1 and DER.

Main Theorem

Figure 4.6 shows a few selected pieces from the parser denotation of ASN1⋆. The type of asn1_as_parser (reproduced below for clarity) is our main theorem: every ASN1⋆ declaration k:declaration s can be interpreted as a strong injective-by-construction parser returning a value of type asn1_as_type k, the type denotation of ASN1⋆. Since a parser is a total function, this proof is also constructive in the sense that it yields executable code for a parser for any ASN1⋆ type definition.

```
1 val asn1_as_parser (#s:set id_t) (k : declaration s) :
2   parser strong (asn1_as_type k)
```

The proof of this theorem is the bulk of our development, comprising about 6,000 lines of F⋆ code. Next, we summarize a few of the main ideas behind the proof.

Content, LC, and ILC Parsers

At the top-level of our semantics (Figure 4.6 line 6) content_as_parser interprets a k:content as a weak_parser (content_t k). A bare content parser is not a strong parser—for example, a sequence parser would accept additional elements appended at the end of its input—but it can be strengthened by first parsing a length and then requiring that the content consume exactly the specified number of bytes. We thus define strong length-content (LC) parsers, using length field parsers and a combinator that invokes the content parser on the input byte sequence truncated to a specific length. We obtain an ILC parser (line 17) by first parsing a leading identifier. The identifier parser itself involves a non-trivial, automata-like logic; it is based on a combinator described in §4.3.4.

For a given ASN.1 declaration, some identifiers may be fully determined by the context, and may thus be omitted. Some more compact ASN.1 encodings, e.g., the Packed Encoding Rules, include optimizations to eliminate redundant identifiers, but they are not widely adopted due to their increased complexity and marginal benefits. The DER does not include such optimizations.

Sequence Parsers

Sequences would be simple to parse if all their fields were always present, but this is not the case with fields decorated with OPTION or DEFAULT. More generally, the well-formedness constraints on SEQUENCE ensure that any consecutive block of omittable fields and the plain field (if any) that immediately follows must have distinct identifiers, so one can use the next identifier value to tell which field comes next and which ones should take their default value. However, this breaks the one-to-one correspondence between identifier and ILC tuple, hence a first challenge for parsing sequences is handling *dangling identifiers*, that is, identifiers that determine the values of multiple fields. A second challenge is to handle omittable suffixes, since, for example, an empty string is a valid encoding of a sequence whose fields are all optional or default.

Dependent LC and Twin Parsers

To tackle the resolution of dangling identifiers, we introduce an alternate form of LC-parsers that depend on a previously-parsed identifier. That is, a p:dlc_parser t (Figure 4.6 line 1) expects an identifier i and ensures that p i is a strong_parser t, while guaranteeing that p i is injective in i—different values of i must return

parsers that accept different values. By decoupling the parsing of identifiers and the length-content, we can construct sequence parsers while accounting for optional and default fields. When a block of omittable fields is encountered, our sequence combinator first parses an identifier and tries to match it against the set of identifiers for each field. If the identifier matches, the dlc_parser for the (undecorated) field is invoked, using the identifier that was just parsed. If the identifier does not match, the omittable field is filled with the default value and the dangling identifier is passed to the next field. In some cases it is useful to interpret a decorated type (line 24) both as a standard ILC parser as well as a dlc_parser for its underlying undecorated form—we call these *twin* parsers (line 3).

Defaultable Parsers

We solve the problem of omittable suffixes with a new parser combinator called defaultable, which overrides the behavior of an existing parser when an empty string is encountered by returning a pre-determined value. To maintain injectivity, it requires the underlying parser to never return the default value.

Choice Parsers

As we've seen, the type denotation of a CHOICE_ILC is a dependent pair. As such, if two different cases have the same underlying type, they are still distinguishable, since the identifier of the cases differs. The well-formedness condition on ASN1$\star$ definitions ensures that the identifiers for all the cases must be distinct. We implemented the ASN.1 choice combinator with a generic tagged union com-

binator provided by EverParse, which first reads the identifier value, then looks it up in the list of cases. Once a match is found, the parser for the corresponding element is invoked to handle the rest of the input.

Any-defined-by Parsers

ANY_DEFINED_BY also roughly assembles a tagged union. However, it differs from CHOICE in that it uses an explicit field (usually an object identifier) in the context of a sequence instead of a tag. Furthermore, its payload is a list of decorated sequence fields, instead of a single piece of content. We implemented a generic parser for ANY_DEFINED_BY by combining the techniques we used for choice and sequence parsers. First, a potential prefix of decorated fields is parsed (which may leave a dangling identifier), then the key field is parsed, its value is compared to the list of known values and, if a match is found, the corresponding continuation is invoked, otherwise the fallback parser is invoked.

4.3.4 Automata-Based Parser Combinator

While EverParse offers a variety of generic parser combinators, building multi-step parsers with branches and loops can be burdensome because relational proofs of parser kinds and injectivity must be provided for the continuation of each step before the combinators can be assembled. We developed a new parser combinator for generic, automata-based parsers that simplifies the construction of such proofs, and used this combinator to build parsers for several terminal types, including, notably, UTF-8 code points, which we use to illustrate the design of our automata parser combinator.

Range	Byte 1	Byte 2	Byte 3	Byte 4
$\leq$ U+007F	0xxxxxxx			
$\leq$ U+07FF	110xxxxx	10xxxxxx		
$\leq$ U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
$\geq$ U+10000	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Table 4.1: Encoding Unicode points to UTF-8

Bit Pattern	Action
0xxxxxxx	Accept, return the byte value
10xxxxxx	Reject: invalid first byte
1100000x	Reject: not using minimum number of bytes
110xxxxx	Transit to S_1 with buffer xxxxx
11100000	Transit to S_2' for extra checks
1110xxxx	Transit to S_2 with buffer xxxx
11110000	Transit to S_3' for extra checks
11110xxx	Transit to S_3 with buffer xxx
11111xxx	Reject: invalid first byte

Table 4.2: Transitions for the initial state and the first byte

The ASN.1 specification requires handling the UTF8STRING terminal type, which is a sequence of valid Unicode code points, up to 21-bit values, each encoded in UTF-8, which takes between one and four bytes (see Table 4.1). A code point may have more than one representation, by using more bytes than necessary and filling the highest bits with 0s. To maintain non-malleability, the standard thus requires that each code point be encoded with the minimal number of bytes.

It is natural to structure a parser for UTF-8 code points as an automaton that reads one byte at a time and, whenever it accepts a code point, emits its value as an integer in $0..2^{21} - 1$. For this, it is convenient to maintain auxiliary state that keeps track of the bit prefix of the code point parsed, rather than encoding this memory in the states of the automaton itself—we refer to this auxiliary state as a "buffer".

For example, Table 4.2 gives the transitions from the initial state, depending

on the value of the first byte. Similarly, the other states have different transitions depending on the byte they read. They all check that their input is of form 10xxxxxx, then S_1 adds bits to the buffer and returns the content; S_2 and S_3 add bits to the buffer; S'_2 and S'_3 check the encoding is minimal and initialize the buffer with the correct bits. The transitions to $S2'$ and $S3'$ mention extra checks needed to ensure the uniqueness of representations, e.g., the 2 byte encoding allows representing code points encoded in 8–11 bits, while 3 bytes must only be used to encode values that require 12–16 bits.

Our automata combinator supports defining parsers with a "control plane" and a "data plane." The control plane contains the states, the alphabet (a single byte in this case), and the conditions for rejecting, accepting, and transitioning for each state. The data plane describes the behavior of the buffer. For example, the control plane of Table 4.2 is captured by three functions below, whereas the data plane for UTF-8 uses bitwise operations to reassemble the code points.

```
1  let reject_init (ch : byte) : bool
2  = (0b10000000 ≤ ch && ch ≤ 0b11000001) || 0b11111000 ≤ ch
3
4  let accept_init (ch : byte {reject_init ch = false}) : bool
5  = ch ≤ 0b01111111
6
7  let transit_init
8    (ch:byte {reject_init ch = false && accept_init ch = false})
9  : state
10 = if (ch < 0b11100000) then S1
11   else if (ch = 0b11100000) then S2′
12   else if (ch < 0b11110000) then S2
13   else if (ch = 0b11110000) then S3′
```

```
14    else (* ch < 0b11111000 *) S3
```

Given the description of the automata and a parser for the alphabet (just a byte parser for UTF-8), the automata combinator assembles a parser that follows the specification of the state machine.

The main novelty is the way in which our combinator structures relational proofs of strong parser kinds and injectivity. The strong parser kind property directly follows from the byte parser having this property. Injectivity is proven by structural induction on the transitions of the automata. This induction is performed automatically by the automata combinator and reduces the goal to proving, for each state of the automata, the injectivity of the suffix that it parses. For UTF-8 code points, the initial state has three cases:

(1) *If the initial state accepts both bytes* b_1 *and* b_2*, and returns the same value, then* $b_1 = b_2$. This is trivial because the initial state returns the byte value.

(2) *If the initial state accepts* b_1 *but transits to another state on* b_2*, the final output will be different.* This holds because the initial state's return value is less than 2^7 while all other states eventually returns larger values (since they correctly reject over-long forms).

(3) *If the initial state transits to other states that return the same output values, then* $b_1 = b_2$. If the next states differ, then the return values differ because they have different number of bits. If the next states are the same, then the control bits in b_1 and b_2 are the same. The induction hypothesis that the suffix parser is injective shows the buffer contents must be the same, and thus $b_1 = b_2$.

Importantly, these proof goals are only propositions about the control and

data plane, and are separated from the low-level parsing actions. In our implementation, all cases are automatically verified by the SMT solver backend of F*.

Our key insight of the automata combinator is the monotonicity innate to multi-step parsers. Each step parses some prefix of the input and "consumes" it such that the later steps can no longer depend on those bytes directly, but only through the control state and the partial output buffer. A necessary condition for injectivity is that each step must preserve enough information about the prefix parsed so far which also implies the information encoded in the control state and the output buffer must grow monotonically. This is what enables the use of structural induction and to decompose the goal into smaller goals about individual states. The manual proofs for each state verifies that the amount of information added in each step is equivalent to that in the prefix consumed.

4.4 Experimental Evaluation

We experimentally evaluate the precision and completeness of our model by writing in ASN1⋆ some of the most commonly used ASN.1 formats, and by executing our formally-verified parsers on large corpuses of inputs collected from real world internet usage, as well as synthetic invalid inputs created for security testing via systematic fuzzing.

The parsers we use in the experiments are extracted from the specification-level parsers derived from ASN.1 declarations (with `as_parser`) using the OCaml backend of F*, and thus, they are much less efficient than low-level in-place C validators available for some other combinators in the EverParse library.

We leave the extraction of optimized C code to future work. All experiments are conducted on an Apple Macbook laptop from 2021.

4.4.1 X.509 Certificates

A major use case of ASN.1 from its conception is to represent cryptographic identities and credentials for internet communication. Like ASN.1, X.509 is a standard created by the International Telecommunication Union (ITU) in 1988 and used to this day to encode digital certificates, which associate entities to public keys and capture trust relations. X.509 certificates are critical to internet security: most websites, and many individuals, are issued certificates to authenticate themselves, for instance when creating a secure HTTP connection (indicated by a padlock icon in many browsers). There are certificate transparency logs that record the issuance of new certificates; at the time of writing (2022), they collect an average of 5 million new entries every day. Moreover, there is a long history of vulnerabilities in ASN.1 parsers causing major exploits in X.509 validation libraries. Surprisingly, although the format of certificates has not significantly evolved in the past 30 years, new vulnerabilities are routinely found in well-established ASN.1 parsers. For instance, looking at the history of documented attacks against OpenSSL, the most popular secure channel and cryptography library commonly used to validate certificates, new ASN.1 exploits[4] were found in 2003 (4 occurences), 2006, 2012, 2015 (6 occurences), 2016 (4 occurences), 2018 and 2021. Interestingly, the ASN.1 vulnerabilities are diverse: CVE-2021-3712 is a buffer overrun caused by functions wrongly assuming ASN.1-encoded strings are NULL-terminated (a problem similar to a famous exploit by Eliot Phillips at

[4] `https://www.openssl.org/news/vulnerabilities.html`

Black Hat 2009 that allows an attacker to impersonate any website using NULL bytes in the middle of domain names); CVE-2018-0739 results from recursive parsers causing stack overflows; CVE-2016-2108 is an interesting combination of vulnerabilities in the INTEGER parser (which can overflow when dealing with the incorrect negative encoding of 0) and the ASN.1 tag parser (which could misinterpret a large universal tag as a negative zero); CVE-2006-4339 is a famous attack by Bleichenbacher that relies on the ASN.1 parser accepting non-canonical serializations to forge RSA signatures. The same trend is observed when looking at MITRE's Common Vulnerabilities and Exposures (CVE) database, which lists ASN.1 vulnerabilities in the past 5 years in most operating systems (Linux, iOS, tvOS, macOS) and cryptographic libraries (OpenSSL, NSS, MatrixSSL, worlfSSL, RSA BSAFE, axTLS). Most are memory safety and functional correctness issues that could be prevented by formally verified parsers.

Format Declaration

Figure 4.7 shows the top-level ASN1$\star$ declaration for X.509 certificates, translated from the ASN.1 declaration in RFC 5280 shown in Figure 4.2. We make a few adaptations compared to the reference declaration; most notably, we try to capture data dependencies in a more precise way. The format of extensions and public keys depend on tags (typically object identifiers) whose possible values are not fully specified in the declaration (to leave the ability to define new ones in future revisions). For instance, extensions use an identifier to indicate their type, a boolean flag to indicate if the extension is critical, and an OCTET STRING that will contain the ASN.1 serialization of the extension payload, which depends on the extension type. An application is supposed to go over the list of extension,

and further parse the payload using the right parser for this extension's type. If it encounters an extension with an unrecognized identifier, and the extension is marked critical, it must reject the certificate. It is useful to perform some of these application-level checks in the parser itself, thus limiting the chance that the checks are mishandled or omitted in the application. For example, we extend ANY DEFINED BY with a default definition, in case the identifier's value is not one of the specified ones. In this case, the fallback representation is the same as the generic definition, but requires the critical flag to be false: The altered definition parses all supported extensions in a single pass and guarantees critical unknown extensions are rejected during parsing. Overall, our X.509 module consists of 143 intermediate declarations in 608 lines of F* code, and can be found in `ASN1.X509.fst`.

Datasets

To evaluate our X.509 module, we use one public dataset from the Electronic Frontier Foundation (EFF) [45] consisting of certificates collected from the wild by scanning the IPv4 address space, and a second synthetic dataset of certificates that have been systematically altered to introduce DER and ASN.1 violations, and is used as part of the OpenSSL build tests to check for regressions.

The EFF dataset was created as part of the SSL Observatory effort in August 2010 by trying to initiate a TLS handshake with all reachable IPv4 addresses on port 443 (typically used for HTTPS), and capturing the collected certificate chains. The scan only captures objects that are at least recognized by OpenSSL at the time of processing as a certificate, which doesn't mean that it is valid or well-formed. Indeed, many of these certificates use undefined X.509 version

Dataset	Total	Accept	Reject	Fail	Time
EFF	11451105	10689353	761750	1	
EFF (subset)	10138	9131	1007	0	198s
OpenSSL	2242	61	2181	0	30s
EFF CRL	4109	3388	703	18	68s
OpenSSL CRL	2063	15	2048	0	30s

Table 4.3: Results of running extracted X.509 and CRL parsers

numbers. The dataset is not labelled so we must manually inspect the rejected certificate to understand the cause of failure.

The OpenSSL dataset is used to check for regressions using `libfuzzer` each time the library is built. It contains a corpus that captures all the known ASN.1 vulnerabilities found in previous versions, and many variants produced by fuzzing. By construction, all certificates in this dataset are invalid; however, in some cases the error doesn't appear during parsing but during signature validation instead. Since we only implement parsing, we do not detect errors introduced after RSA encryption, e.g. in the payload of signatures.

Analysis of Results

The top part of Table 4.3 shows the results of running the X.509 module on the EFF and OpenSSL datasets. Due to the large number of certificates in the EFF X.509 dataset, we select a subset of 10,138 certificates by arbitrarily taking the range of IP addresses from 108.0.100.238 to 109.95.49.5 to manually inspect each of the 1007 rejected certificates from that subset to determine what is the first error. We manage to attribute all such failures to one of the following classes:

Default Field	Identifier	Terminal Type	Empty Sequence
710	196	65	36

Default field means that an optional field contains its default value, which is prohibited under DER. This error appears either in the basic constraints extension, which is used to indicate if a certificate can sign other certificates or not, or in the parameters of RSA public key algorithm, which must be NULL. **Identifier** means the identifier doesn't match those stated in the standard. Again, these cases are often found inside an `ANY` structure. Issuer/subject fields of the certificate are prone to this kind of error. A typical case is that the standard requires a more restrictive string type, for instance the printable string, but the certificate uses a general one, for instance an ASCII string. **Terminal Type** is a class that includes all cases where a certain terminal type, such as boolean and integer, is not encoded correctly. A representative case is that of UTCTime, which require the letter Z to be used at the end of the representation to denote the Greenwich time instead of +/-0000 for non-malleability. For another example, a peculiar certificate encoded a very large integer but did not use the least number of bytes for it. These kinds of errors are hard to detect for conventional parsers because they are niche cases for the implementation of a particular terminal parser while the tests are usually for the whole datatype. **Empty Sequence** occurs in certain sequence of structures that cannot be empty. We found this kind of error frequently shows up in extensions as well.

In summary, all 1007 rejected certificates are indeed invalid. Conversely, we cannot manually confirm the 9,131 accepted certificates are indeed valid. Instead, we rely on our results from the OpenSSL regression test. 97% of their certificates are indeed correctly rejected; we manually inspect each of the accepted certificates and confirmed that either the error only appears in the signature (which we cannot detect) or in an extension that we do not implement.

4.4.2 Certificate Revocation Lists

Format Declaration

The standard definition of the CRL format can be found in the Section 5 of [15]. It is similar to X.509 in the style of definition with its own extensions. Our CRL module consists of 8 declarations in 69 lines of F* and can be found in `ASN1.CRL.fst`.

Datasets

We did not find any large public corpus of revocations lists, so we wrote a script that extracts the URLs where the certification authority publishes their CRL from the "CRL distribution endpoint" certificate extension. We managed to collect 4,109 samples with this method.

The OpenSSL regression tests also includes tests for CRLs, which we use for negative testing. It contains 2,063 samples.

Analysis of Results

The bottom part of Table 4.3 shows the results of running the CRL module on the 2 datasets. It is worth noting that CRLs can be much larger than certificates if a CA has revoked many certificates. This triggers a limitation in our OCaml extraction: since our byte buffers are modelled using F* sequences, they extract to non-flat OCaml lists, which means some linear operations on flat buffers may be extracted to quadratic algorithms. Hence, in 18 cases we fail to execute our

parsers. This can be fixed by using a flat memory representation for buffers, however extracting efficient OCaml parsers is not our goal and we would rather invest effort on extraction to C. The findings are very much aligned with the X.509 dataset: failures align with the 4 classes of errors in the EFF dataset. Similarly, the only OpenSSL samples that we do accept have errors in their signature or in extensions that we did not specify.

4.5 Related Work and Conclusions

Formally Verified Parsers

While our work focuses on non-malleability of ASN.1 DER, formally verified parsers have covered various binary data formats and provided various properties on those formats and their implementations. Existing work [108] addresses the problem of parsing and serializing regular grammars using a Brzozowski derivative-based matching algorithm implemented and verified in Coq. Narcissus [35] is a library of parsing and serialization combinators verified in Coq and extracted to OCaml focused on the correctness of encoders with respect to decoders; it has been used to harden the network stack (TCP, UDP, IPv4, ARPv4, Ethernet) of the Mirage OS kernel [83]. Narcissus has also been used for Protocol Buffers [133]. EverParse [105] provides not only encoder correctness proofs, but also non-malleability, and extracts to C instead of OCaml, giving rise to efficient zero-copy C implementations proven memory safe and functionally correct with respect to the data format specifications. While EverParse was initially designed to support TLS handshake messages, our work is based on EverParse and extends it with ASN.1 parsing combinators with non-malleability proofs. Other

extensions of EverParse such as EverParse3D for network virtualization packet formats [120] prove additional properties such as absence of double fetches to ensure secure efficient parsing on volatile input buffers where two reads from a given byte cannot be guaranteed to return the same value.

Formal Studies of ASN.1

While ASN.1 predates many modern verification tools, there have been some early attempts to gain confidence in its security properties. Rinderknecht [107] proved properties of ASN.1 on paper such as non-malleability of a subset of "well-labeled" ASN.1 format descriptions, but without clearly relating this subset to DER. Conversely, Steckler [115] wrote an executable semantics of ASN.1 in Haskell but no associated formal proofs. In a case study of Quviq QuickCheck[6], the authors partially specified ASN.1 for the declarations involved. They used property-based testing to compare the ASN.1 specification against specifications generated from other format descriptions and uncovered several inconsistencies among them. DICE* [121] is an implementation of secure measured boot for IoT formally verified in F* and extracted to C code to be run as part of the boot firmware of micro-controllers. As one of its main components, it includes a formal semantics of a small subset of ASN.1 used to create the unique certificate of a device. This subset cannot capture general purpose certificate as it lacks several important constructors such as CHOICE or ANY DEFINED BY. Tullsen et al. [125] formally verify C implementations of ASN.1 decoders and encoders for a vehicle-to-vehicle (V2V) messaging system, using the annotation-based SAW verification framework [39] turning annotated C programs into first-order formulae to be checked by SMT solvers. While their work provides both non-

malleability and encoder correctness, their proofs focus on the C implementations for the purpose of the security of the enclosing V2V system, rather than a full formal specification of ASN.1 per se. In other words, they have not proven the functional correctness of their C encoders or decoders against any formal data format specification. Moreover, they do not support `CHOICE`. [102] describe verification methodology challenges to verify an existing ASN.1 description compiler for C, ASN1C [129], by first formalizing the corresponding subset of ASN.1 in Coq, and then separately proving the functional correctness of ASN1C with respect to their specification using Appel's Verified Software Toolchain [5]. However, we are not aware of any completed results from their effort yet.

Security of ASN.1 Parsers

Because of the security-critical nature of the remaining applications of ASN.1 such as X.509 and the PKCS standards for encryption, signature, and wrapping, many techniques have been applied to find vulnerabilities in ASN.1 applications. Frankencert [19], Mucert [27] and Coveringcerts [67] are three domain-specific fuzzing tools to evaluate the security of real-world parsers and use various techniques to guarantee coverage and ensure that alterations pass through cryptographic integrity checks; general-purpose tools such as Nezha [101] and SAGE [48] have also been specialized for this purpose. Other papers such as Chen et al. [27] and Symcerts [26] attempt to detect non-compliance by discovering discrepancies between implementations, either by testing or by symbolic execution. Attacks that exploit the malleability of ASN.1 parsers to forge signatures have also been found in PGP [43], NSS [25], GnuTLS [98], Bouncy Castle [72], or even in the Nintendo 3DS boot ROM [111].

Conclusion

We have presented the first formalization of the semantics of ASN.1 and its Distinguished Encoding Rules, yielding parsers for binary formatted ASN.1 data that are type correct and non-malleable. Through testing, we have confidence that our formalized semantics matches the usage of ASN.1 in the wild, notably on X.509 certificates and certificate revocation lists. We aim to continue testing our semantics on more applications to further increase trust in our formalization. Additionally, we plan to use our semantics as a basis on which to build high-assurance cryptographic applications such as X.509 certificate chain validation.

```
1  type decorator = | PLAIN | OPTION | DEFAULT
2  type content : Type =
3  | TERMINAL :
4            k:terminal_k →
5            is_valid:(terminal_t k → bool) →
6            content
7  | SEQUENCE : decorateds → content
8  | SEQUENCE_OF : declaration s → content
9  | SET_OF : declaration s → content
10 | PREFIXED : declaration s → content
11 | ANY_DEFINED_BY :
12           prefix:list decorated →
13           id:id_t → key:terminal_k →
14           kvs:list (terminal_t key & decorateds) →
15           def:option decorateds →
16           squash (wf_any prefix id kvs) →
17           content
18
19 and declaration : set id_t → Type =
20 | ILC : id:id_t → content → declaration (singleton id)
21 | CHOICE_ILC :
22     choices:list (id_t & content) →
23     squash (no_repeats (map fst choices)) →
24     declaration (as_set (map fst choices))
25 | ANY_ILC : declaration (complement empty)
26
27 and d_declaration : set id_t → decorator → Type =
28 | PLAIN_ILC : k:declaration s → d_declaration s PLAIN
29 | OPTION_ILC : k:declaration s → d_declaration s OPTION
30 | DEFAULT_TERMINAL :
31     id:id_t →
32     is_valid:(terminal_t k → bool) →
33     defaultv:terminal_t k →
34     squash (is_valid defaultv) →
35     d_declaration (singleton id) DEFAULT
36
37 and decorated = s:set id_t & d:decorator & d_declaration s d
38 and decorateds = items : list decorated &
39     squash (sequence_k_wf (map proj12 items))
```

Figure 4.4: Formal syntax and well-formedness

```
1  let rec content_t (k:content) : Type = match k with
2    | TERMINAL t is_valid → x:terminal_t t { is_valid x }
3    | SEQUENCE ds → decorateds_t ds
4    | SEQUENCE_OF k → list (asn1_as_type k)
5    | SET_OF k → list (asn1_as_type k)
6    | PREFIXED k → asn1_as_type k
7    | ANY_DEFINED_BY prefix _ _ kv def _ →
8      sequence_t prefix (choice_t (any_t kv) (def_t def))
9
10 and asn1_as_type (k:declaration s) : Tot Type (decreases k) =
11   match k with
12   | ILC id k → content_t k
13   | CHOICE_ILC lc _ → choice_t (cases_t lc) ⊥
14   | ANY_ILC → id_t & octetstring_t
15
16 and decorated_t (d:decorated) : Type =
17   let (| _, _, dk |) = d in
18   match dk with
19   | PLAIN_ILC k → asn1_as_type k
20   | OPTION_ILC k → option (asn1_as_type k)
21   | DEFAULT_TERMINAL id is_valid defv → default_tv defv
22
23 and decorateds_t (| l, _|) = sequence_t l unit
24
25 and def_t d = match d with
26   | None → ⊥
27   | Some ds → decorateds_t ds
28
29 and any_t (ls:list (t & decorateds)) : Tot _ (decreases ls) =
30   match ls with
31   | [] → []
32   | (x, ds) :: tl → (x, decorateds_t ds) :: any_t tl
33
34 and choice_t (lc:list (key & Type)) (def:Type) =
35   k:key & assoc k lc def
36
37 and cases_t (lc:list (id_t & content)) : list (id_t & Type) =
38   match lc with
39   | [] → []
40   | (x,y) :: t → (x, content_t y) :: cases_t t
41
42 and sequence_t (items:list decorated) (suffix_t:Type) : Type =
43   match items with
44   | [] → suffix_t
45   | hd :: tl → decorated_t hd & sequence_t tl suffix_t
```

Figure 4.5: Denoting ASN.1 definitions as F* types

```
1  let dlc_parser t = lc:(id_t → strong_parser t) {cases_injective lc}
2  let twin_t t = strong_parser t & dlc_parser t
3  type twin = { d: decorated; ps:twin_t (undec_d_t d) }
4  let twins ds = lp : list twin_d{map (λ x → x.d) lp == ds}
5
6  let rec content_as_parser (k:content)
7  : weak_parser (content_t k) =
8    match k with
9    | TERMINAL k v →
10     weaken ((terminal_as_parser k) ‘filter‘ v)
11   | SEQUENCE (| ds, _ |) →
12     mk_seq_parser (seq_as_twins ds)
13   . . .
14 and asn1_as_parser (k : declaration s)
15 : strong_parser (asn1_as_type k) =
16   match k with
17   | ILC id k′ → parse_ILC id (content_as_parser k′)
18   . . .
19 and seq_as_twins (ds : decorateds) : twins ds
20   match ds with
21   | [] → []
22   | hd :: tl → decorated_as_twin hd :: seq_as_twin tl
23   . . .
24 and decorated_as_twin (d:decorated)
25 : (tw:twin {tw.d == d}) =
26   let (| _, _, dk |) = d in
27   match dk with
28   | PLAIN_ILC k | OPTION_ILC k →
29     { d; ps=asn1_as_twin k }
30   | . . .
31 and asn1_as_twin (k : declaration s)
32 : twin_t (asn1_as_type k) =
33   match k with
34   | ILC id k′ →
35     let p = content_as_parser k′ in
36     ilc_twin_case_injective id p; (* lemma *)
37     parse_ILC id p, parse_ILC_twin id p
38   | CHOICE_ILC lc pf →
39     let lp = cases_as_parser lc in
40     choice_twin_cases_injective lc pf k lp; (* lemma *)
41     make_choice_parser lc pf k lp,
42     make_choice_parser_twin lc pf k lp
43   . . .
```

Figure 4.6: The parser denotation of ASN1⋆ (fragments)

```
1 let x509_TBSCertificate= asn1_sequence [
2   "version" *^ (PLAIN ^: (mk_prefixed (mk_custom_id
3      CONTEXT_SPECIFIC CONSTRUCTED 0) version));
4   "serialNumber" *^ (PLAIN ^: certificateSerialNumber);
5   "signature" *^ (PLAIN ^: algorithmIdentifier);
6   "issuer" *^ (PLAIN ^: name);
7   "validity" *^ (PLAIN ^: validity);
8   "subject" *^ (PLAIN ^: name);
9   "subjectPublicKeyInfo" *^ (PLAIN ^: subjectPublicKeyInfo);
10  "issuerUniqueID" *^ (OPTION ^: (mk_retagged
11    (mk_custom_id CONTEXT_SPECIFIC PRIMITIVE 1) uId));
12  "subjectUniqueID" *^ (OPTION ^: (mk_retagged
13    (mk_custom_id CONTEXT_SPECIFIC PRIMITIVE 2) uId));
14  "extensions" *^ (PLAIN ^: (mk_prefixed
15    (mk_custom_id CONTEXT_SPECIFIC CONSTRUCTED 3)
16    extensions))]
17  (_ by (seq_tac ()))
18
19 let x509_certificate = asn1_sequence [
20  "tbsCertificate" *^ (PLAIN ^: tBSCertificate);
21  "signatureAlgorithm" *^ (PLAIN ^: algorithmIdentifier);
22  "signatureValue" *^ (PLAIN ^: bitString)]
23  (_ by (seq_tac ()))
24
25 (* Extension ::= SEQUENCE {
26         extnID OBJECT IDENTIFIER,
27         critical BOOLEAN DEFAULT FALSE,
28         extnValue OCTET STRING *)
29
30 let extension_fallback = mk_gen_items [
31  "critical" *^ (DEFAULT ^: critical_field_MUST_false);
32  "extnValue" *^ (PLAIN ^: asn1_octetstring)]
33  (_ by (seq_tac ()))
34
35 let extension = asn1_any_oid_with_fallback
36  "extnId" supported_extensions extension_fallback
37  (_ by (seq_tac ())) (_ by (choice_tac ()))
```

Figure 4.7: Representing X.509 in ASN1⋆

CHAPTER 5

CONCLUSION

In this dissertation, I have demonstrated how to provide high assurance for domain-specific systems of fault-tolerant distributed protocols, concurrent programs, and parsers, systematically and foundationally, through building and analyzing models in formal modeling languages. The key ingredient of this approach is the design of those languages that capture the essence of the specific domain and alleviate the effort required for building and analyzing those models. The generality of having those formal modeling languages instead of building specific models for individual systems is crucial as the adequacy of all the models described in the language can be established through meta-theoretic analysis once and for all, significantly reducing the overhead.

5.1 Future Work

The exploratory research presented in this dissertation is an important step towards solving the more general problem of building high-assurance production systems, which are magnitudes larger in scale than the systems investigated and may involve many components from various domains. In this section, I discuss some of the potential future research directions that may further approach this goal.

5.1.1 A Formal Foundation For New Systems And Optimizations

Besides the domains discussed in this dissertation, new classes of systems, like permissionless blockchains and systems utilizing specialized hardware, are being posed by systems researchers frequently. There are also new unintended behaviors, such as side-channel attacks and supply chain attacks. For those new challenges, new domain-specific treatments are needed. Through a solid, formal foundation, I believe a higher level of assurance can be provided at a lower cost. A vivid example is the IPDL project [46], where the authors verified a cryptographic scheme with a 2-page machine-checked proof when it used to be a 100-page pen-and-paper proof. What makes this direction particularly challenging is the fact that new systems are often highly optimized for performance. It is still an open question how those optimizations can be reasoned about effectively. Ideally, those optimizations and their reasoning can be done separately and compositionally from the base system.

Furthermore, I believe that a formal foundation will also open new opportunities for systems research as well. Being able to validate designs can bring forth more performance and functionality in the form of optimizations and extensions. For instance, the Paxos consensus protocol has been studied for decades, and yet we discovered novel optimizations in the Ironwood project through more precise formal analysis.

5.1.2 Cross-domain Compositionality

As mentioned above, realistic production systems are seldom restricted to a single domain. For instance, many consensus protocols, such as Nakamoto [93] and Hotstuff [134], leverage cryptographic primitives or economic mechanisms. For another example, parsers, like the ones we formalize in ASN1⋆, are also always components of a more extensive system. How do we ensure the correctness of the whole system, given components verified in different domains? We should aim to minimize the overhead and reuse the existing domain-specific reasoning as much as possible.

5.1.3 Large-scale Development of High-assurance Systems

An obstacle to applying the aforementioned approach at scale is its relatively high requirement for expertise. If only a few experts could use those languages and tools, it would be impractical to develop large-scale systems like the production systems we have today. Fully addressing the challenge of assuring our systems requires us researchers to revolutionize the practice of software engineering outside of academia. The Rust programming language, whose principles were developed by decades of computer science research, demonstrated how secure memory management can be supported in large-scale development. Going beyond memory safety for large-scale developments is no doubt a multifaceted challenge: not only do we need better automation and abstractions, but also efforts into user experience and ecosystem. I believe this needs to be tackled with cross-domain collaborations beyond programming language and systems research communities.

5.2 Conclusions

For system builders today, programming languages mostly provide the necessary means to implement their designs but aid them little in analyzing or validating their designs. The three formal modeling languages proposed in this dissertation – Ironwood, ASN1⋆, and Harmony– illustrated the potential of programming languages to provide a solid foundation and significant help in conducting the latter while still fulfilling the responsibilities of the former. Different from general-purpose system programming languages, these languages are deeply specialized for the corresponding domain in their design. In return, they are able to assume strong restrictions on the structures of the programs, which enables those programs to be analyzed at a more affordable cost. Different from efforts that target specific systems, the overhead of connecting abstract formal models closer to reality is paid once and for all through meta-theoretic analysis of these languages instead of on individual models being built.

As our society grows more and more reliant on the services provided by software systems, assuring those systems will behave as intended in deployment is becoming more and more of a necessity. The bad news is that this impending crisis cannot be resolved just by pouring more resources into the conventional means of improving assurance due to the nature of software systems. But there is still hope that we can regain our intellectual grasp on software systems' behaviors by building and analyzing formal models that are grounded in those systems' reality. To realize this dream of generations of computer scientists, the models and analysis must fit a tight budget – a task best suited for programming languages as it is arguably the one discipline that started the prevalence of software systems through affordable development costs from the very beginning.

BIBLIOGRAPHY

[1] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting Fast Practical Byzantine Fault Tolerance, December 2017.

[2] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. *ACM SIGARCH Computer Architecture News*, 18(2SI):2–14, May 1990.

[3] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, September 1987.

[4] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A Model Checker for Concurrent Software. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 484–487, Berlin, Heidelberg, 2004. Springer.

[5] Andrew W. Appel. Verified Software Toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer.

[6] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, September 2006. Association for Computing Machinery.

[7] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.

[8] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotić, Carsten Varming, and Blake Whaley. Reachability Analysis for AWS-Based Networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 231–241, Cham, 2019. Springer International Publishing.

[9] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, Cambridge, MA, April 2008.

[10] Paolo Ballarini and Benoît Barbot. Cosmos: Evolution of a Statistical Model Checking Platform. *ACM SIGMETRICS Performance Evaluation Review*, 49(4):65–69, June 2022.

[11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Product programs and relational program logics. *Journal of Logical and Algebraic Methods in Programming*, 85(5, Part 2):847–859, August 2016.

[12] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Probabilistic Relational Hoare Logics for Computer-Aided Security Proofs. In Jeremy Gibbons and Pablo Nogueira, editors, *Mathematics of Program Construction*, pages 1–6, Berlin, Heidelberg, 2012. Springer.

[13] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 14–25, New York, NY, USA, January 2004. Association for Computing Machinery.

[14] Mark Bickford, Vincent Rahli, and Robert L. Constable. Logic of events, a framework to reason about distributed systems, 2012.

[15] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. RFC 5280, May 2008.

[16] Péter Bokor, Marco Serafini, and Neeraj Suri. On Efficient Models for Model Checking Message-Passing Distributed Protocols. In John Hatcliff and Elena Zucca, editors, *Formal Techniques for Distributed Systems*, pages 216–223, Berlin, Heidelberg, 2010. Springer.

[17] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 836–850, New York, NY, USA, October 2021. Association for Computing Machinery.

[18] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating*

Systems, ASPLOS '16, pages 83–98, New York, NY, USA, March 2016. Association for Computing Machinery.

[19] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *2014 IEEE Symposium on Security and Privacy*, pages 114–129, USA, May 2014. IEEE Computer Society.

[20] Luís Caires and Hugo Torres Vieira. SLMC: A Tool for Model Checking Concurrent Systems against Dynamical Spatial Logic Specifications. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 485–491, Berlin, Heidelberg, 2012. Springer.

[21] Ran Canetti. Universally Composable Security. *Journal of the ACM*, 67(5):28:1–28:94, September 2020.

[22] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*, pages 173–186, New Orleans, LA, February 1999. USENIX Association.

[23] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[24] Bernadette Charron-Bost and André Schiper. The Heard-Of model: Computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, April 2009.

[25] Sze Yiu Chau. A Decade After Bleichenbacher '06, RSA Signature Forgery Still Works. https://i.blackhat.com/USA-19/Wednesday/us-19-Chau-A-Decade-After-Bleichenbacher-06-RSA-Signature-Forgery-Still-Works-wp.pdf, 2019.

[26] Sze Yiu Chau, Omar Chowdhury, Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 503–520, USA, May 2017. IEEE Computer Society.

[27] Yuting Chen and Zhendong Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th*

Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pages 793–804, New York, NY, USA, August 2015. Association for Computing Machinery.

[28] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, New York, NY, USA, September 2008. Association for Computing Machinery.

[29] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, pages 495–499, Berlin, Heidelberg, 1999. Springer.

[30] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 51–65, USA, June 2008. IEEE Computer Society.

[31] Robert L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., USA, 1986.

[32] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 183–190, Cham, 2018. Springer International Publishing.

[33] Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. Functional Choreographic Programming. In Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu, editors, *Theoretical Aspects of Computing – ICTAC 2022*, Lecture Notes in Computer Science, pages 212–237, Cham, 2022. Springer International Publishing.

[34] Andrei Damian, Cezara Drăgoi, Alexandru Militaru, and Josef Widder. Communication-Closed Asynchronous Protocols. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 344–363, Cham, 2019. Springer International Publishing.

[35] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: Correct-by-construction derivation of

decoders and encoders from binary formats. *Proceedings of the ACM on Programming Languages*, 3(ICFP):82:1–82:29, July 2019.

[36] Pantazis Deligiannis, Aditya Senthilnathan, Fahad Nayyar, Chris Lovett, and Akash Lal. Industrial-Strength Controlled Concurrency Testing for C{\#} Programs with Coyote. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 433–452, Cham, 2023. Springer Nature Switzerland.

[37] Giorgio Delzanno, Michele Tatarek, and Riccardo Traverso. Model Checking Paxos in Spin. *Electronic Proceedings in Theoretical Computer Science*, 161:131–146, August 2014.

[38] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. *ACM SIGPLAN Notices*, 46(1):435–446, January 2011.

[39] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. Constructing Semantic Models of Programs with the Software Analysis Workbench. In Sandrine Blazy and Marsha Chechik, editors, *Verified Software. Theories, Tools, and Experiments*, pages 56–72, Cham, 2016. Springer International Publishing.

[40] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

[41] Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. PSync: A partially synchronous language for fault-tolerant distributed algorithms. *ACM SIGPLAN Notices*, 51(1):400–415, January 2016.

[42] Hesham El-Rewini El-Rewini and Mostafa Abd-El-Barr. Shared Memory Architecture. In *Advanced Computer Architecture and Parallel Processing*, chapter 4, pages 77–102. John Wiley & Sons, Ltd, Hoboken, NJ, 2004.

[43] Hal Finney. Bleichenbacher's RSA signature forgery based on implementation error. https://mailarchive.ietf.org/arch/msg/openpgp/5rnE9ZRN1AokBVj3VqblGlP63QE/, 2006.

[44] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[45] Electronic Frontier Foundation. The EFF SSL observatory. https://www.eff.org/observatory, 2010.

[46] Joshua Gancher, Kristina Sojakova, Xiong Fan, Elaine Shi, and Greg Morrisett. A Core Calculus for Equational Proofs of Cryptographic Protocols. *Proceedings of the ACM on Programming Languages*, 7(POPL):30:866–30:892, January 2023.

[47] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, January 1997. Association for Computing Machinery.

[48] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, March 2012.

[49] Eli Goldweber, Nuda Zhang, and Manos Kapritsos. Brief Announcement: On the Significance of Consecutive Ballots in Paxos. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 172–174, New York, NY, USA, July 2020. Association for Computing Machinery.

[50] Niklas Grimm, Kenji Maillard, Cédric Fournet, Cătălin Hriţcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella-Béguelin. A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 130–145, New York, NY, USA, January 2018. Association for Computing Machinery.

[51] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, March 2000.

[52] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, October 2015. Association for Computing Machinery.

[53] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming*

Languages, POPL '87, pages 13–26, New York, NY, USA, October 1987. Association for Computing Machinery.

[54] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[55] Andrew K. Hirsch and Deepak Garg. Pirouette: Higher-order typed functional choreographies. *Proceedings of the ACM on Programming Languages*, 6(POPL):23:1–23:27, January 2022.

[56] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise*: A library of verified high-performance secure channel protocol implementations. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 107–124, USA, 2022. IEEE Computer Society.

[57] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, Boston, MA, 1st edition, April 2011.

[58] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 122–138, Berlin, Heidelberg, 1998. Springer.

[59] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *Journal of the ACM*, 63(1):9:1–9:67, March 2016.

[60] Wolf Honoré, Jieung Kim, Ji-Yong Shin, and Zhong Shao. Much ADO about failures: A fault-aware model for compositional verification of strongly consistent distributed systems. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):97:1–97:31, October 2021.

[61] Wolf Honoré, Ji-Yong Shin, Jieung Kim, and Zhong Shao. Adore: Atomic distributed objects with certified reconfiguration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 379–394, New York, NY, USA, June 2022. Association for Computing Machinery.

[62] P. Z. Ingerman. Thunks: A way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, January 1961.

[63] International Telecommunication Union. Abstract syntax notation one ASN.1: Specification of basic notation, 2021.

[64] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, MA, January 2012.

[65] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. *ACM SIGPLAN Notices*, 50(1):637–650, January 2015.

[66] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, page 18, Cambridge, MA, 2007. USENIX Association.

[67] Kristoffer Kleine and Dimitris E. Simos. Coveringcerts: Combinatorial Methods for X.509 Certificate Testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 69–79, USA, March 2017. IEEE Computer Society.

[68] Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. *ACM SIGPLAN Notices*, 52(1):719–734, January 2017.

[69] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39, January 2010.

[70] Saul Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[71] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In Peter Müller, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 336–365, Cham, 2020. Springer International Publishing.

[72] Ulrich Kühn, Andrei Pyshkin, Erik Tews, and Ralf-Philipp Weinmann. Variants of bleichenbacher's low-exponent attack on PKCS#1 RSA signatures. In *SICHERHEIT 2008 – Sicherheit, Schutz Und Zuverlässigkeit. Beiträge*

Der 4. Jahrestagung Des Fachbereichs Sicherheit Der Gesellschaft Für Informatik e.V. (GI), pages 97–109. Gesellschaft für Informatik e. V., Bonn, 2008.

[73] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic Symbolic Model Checker. In Tony Field, Peter G. Harrison, Jeremy Bradley, and Uli Harder, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, pages 200–204, Berlin, Heidelberg, 2002. Springer.

[74] Leslie Lamport. What it means for a concurrent program to satisfy a specification: Why no one has specified priority. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 78–83, New York, NY, USA, January 1985. Association for Computing Machinery.

[75] Leslie Lamport. Refinement in state-based formalisms. Technical Report 1996-001, Microsoft Coporation, December 1996.

[76] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[77] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA, July 2002.

[78] Leslie Lamport. The PlusCal Algorithm Language. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009*, pages 36–60, Berlin, Heidelberg, 2009. Springer.

[79] Leslie Lamport. Byzantizing Paxos by Refinement. In David Peleg, editor, *Distributed Computing*, pages 211–224, Berlin, Heidelberg, 2011. Springer.

[80] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified causally consistent distributed key-value stores. *ACM SIGPLAN Notices*, 51(1):357–370, January 2016.

[81] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, November 1995.

[82] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental inference of inductive

invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 370–384, New York, NY, USA, October 2019. Association for Computing Machinery.

[83] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, March 2013. Association for Computing Machinery.

[84] Moxie Marlinspike. Null prefix attack against TLS server certificates. https://nvd.nist.gov/vuln/detail/CVE-2009-2510, 2009.

[85] Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. Recovering shared objects without stable storage (extended version). Technical Report UW-CSE-TR-17-10-01, University of Washington, October 2017.

[86] Ellis Michael, Doug Woos, Thomas Anderson, Michael D. Ernst, and Zachary Tatlock. Teaching Rigorous Distributed Systems With Efficient Model Checking. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–15, New York, NY, USA, March 2019. Association for Computing Machinery.

[87] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, May 1996. Association for Computing Machinery.

[88] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[89] Atsuki Momose and Jason Paul Cruz. Force-Locking Attack on Sync Hotstuff, 2019.

[90] Fabrizio Montesti. *Choreographic Programming*. PhD thesis, IT University of Copenhagen, 2013.

[91] Madan Musuvathi, Shaz Qadeer, and Thomas Ball. CHESS: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Coporation, November 2007.

[92] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, 36(SI):75–88, December 2003.

[93] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, May 2009.

[94] Sebastian Nanz, Faraz Torshizi, Michela Pedroni, and Bertrand Meyer. Design of an Empirical Study for Comparing the Usability of Concurrent Programming Languages. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 325–334, USA, September 2011. IEEE Computer Society.

[95] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[96] Haobin Ni, Antoine Delignat-Lavaud, Cédric Fournet, Tahina Ramananandro, and Nikhil Swamy. ASN1*: Provably Correct, Non-malleable Parsing for ASN.1 DER. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, pages 275–289, New York, NY, USA, January 2023. Association for Computing Machinery.

[97] Brian Norris and Brian Demsky. CDSchecker: Checking concurrent data structures written with C/C++ atomics. *ACM SIGPLAN Notices*, 48(10):131–150, October 2013.

[98] Yutaka Oiwa, Kazukuni Kobara, and Hajime Watanabe. A new variant for an attack against RSA signature verification using parameter field. In *Proceedings of the 4th European Conference on Public Key Infrastructure: Theory and Practice*, EuroPKI'07, pages 143–153, Berlin, Heidelberg, June 2007. Springer-Verlag.

[99] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, USA, June 2014. USENIX Association.

[100] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12:115–116, 1981.

[101] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632, USA, May 2017. IEEE Computer Society.

[102] Nika Pona and Vadim Zaliva. Research Report: Formally-Verified ASN.1 Protocol C-language Stack. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 308–317, USA, May 2020. IEEE Computer Society.

[103] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. EventML: Specification, verification, and implementation of crash-tolerant state machine replication systems. *Science of Computer Programming*, 148:26–48, November 2017.

[104] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In Amal Ahmed, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 619–650, Cham, 2018. Springer International Publishing.

[105] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1465–1482, Cambridge, MA, 2019. USENIX Association.

[106] Robbert Van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, page 7, San Francisco, CA, December 2004. USENIX Association.

[107] Christian Rinderknecht. *Une Formalisation d'ASN.1 - Application d'une Méthode Formelle à Un Langage de Spécification Télécom*. PhD thesis, Université Pierre et Marie Curie, December 1998.

[108] John Sarracino, Gang Tan, and Greg Morrisett. Certified Parsing of Dependent Regular Grammars. In *2022 IEEE Security and Privacy Workshops (SPW)*, pages 113–123, May 2022.

[109] Nicolas Schiper, Vincent Rahli, Robbert Van Renesse, Marck Bickford, and Robert L. Constable. Developing Correctly Replicated Databases Using Formal Tools. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 395–406, USA, June 2014. IEEE Computer Society.

[110] Fred B. Schneider. The state machine approach: A tutorial. In Barbara Simons and Alfred Spector, editors, *Fault-Tolerant Distributed Computing*, Lecture Notes in Computer Science, pages 18–41, New York, NY, 1990. Springer.

[111] Michael Scire, Melissa Mears, Devon Maloney, Matthew Norman, Shaun Tux, and Phoebe Monroe. Attacking the Nintendo 3DS Boot ROMs, February 2018.

[112] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages*, 2(POPL):28:1–28:30, December 2017.

[113] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. Grove: A Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 113–129, New York, NY, USA, October 2023. Association for Computing Machinery.

[114] Yee Jiun Song and Robbert van Renesse. Bosco: One-Step Byzantine Asynchronous Consensus. In Gadi Taubenfeld, editor, *Distributed Computing*, Lecture Notes in Computer Science, pages 438–450, Berlin, Heidelberg, 2008. Springer.

[115] Paul Steckler. A formal semantics for ASN.1, 2007.

[116] Andreas Stefik and Stefan Hanenberg. The Programming Language Wars: Questions and Responsibilities for the Programming Language Community. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 283–299, New York, NY, USA, October 2014. Association for Computing Machinery.

[117] Andreas Stefik and Susanna Siebert. An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education*, 13(4):19:1–19:40, November 2013.

[118] Wei Su, Yifei Liu, Gomathi Ganesan, Gerard Holzmann, Scott Smolka, Erez Zadok, and Geoff Kuenning. Model-Checking Support for File System Development. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '21, pages 103–110, New York, NY, USA, July 2021. Association for Computing Machinery.

[119] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 256–270, New York, NY, USA, January 2016. Association for Computing Machinery.

[120] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. Hardening attack surfaces with formally proven binary format parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 31–45, New York, NY, USA, June 2022. Association for Computing Machinery.

[121] Zhe Tao, Aseem Rastogi, Naman Gupta, Kapil Vaswani, and Aditya V. Thakur. {DICE*}: A Formally Verified Implementation of {DICE} Measured Boot. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1091–1107, Cambridge, MA, 2021. USENIX Association.

[122] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 662–677, New York, NY, USA, June 2018. Association for Computing Machinery.

[123] Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, PPDP '11, pages 161–172, New York, NY, USA, July 2011. Association for Computing Machinery.

[124] Tatsuhiro Tsuchiya and André Schiper. Using Bounded Model Checking to Verify Consensus Algorithms. In Gadi Taubenfeld, editor, *Distributed Computing*, pages 466–480, Berlin, Heidelberg, 2008. Springer.

[125] Mark Tullsen, Lee Pike, Nathan Collins, and Aaron Tomb. Formal Verification of a Vehicle-to-Vehicle (V2V) Messaging System. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 413–429, Cham, 2018. Springer International Publishing.

[126] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: Synchronous verification of asynchronous distributed programs. *Proceedings of the ACM on Programming Languages*, 3(POPL):59:1–59:30, January 2019.

[127] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, pages 491–515, Berlin, Heidelberg, 1991. Springer.

[128] Ivana Vukotic, Vincent Rahli, and Paulo Esteves-Veríssimo. Asphalion: Trustworthy shielding against Byzantine faults. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):138:1–138:32, October 2019.

[129] Lev Walkin. Vlm/asn1c, March 2024.

[130] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’15, pages 357–368, New York, NY, USA, June 2015. Association for Computing Machinery.

[131] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 154–165, New York, NY, USA, January 2016. Association for Computing Machinery.

[132] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423, November 2006.

[133] Qianchuan Ye and Benjamin Delaware. A verified protocol buffer compiler. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 222–233, New York, NY, USA, January 2019. Association for Computing Machinery.

[134] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 347–356, New York, NY, USA, July 2019. Association for Computing Machinery.

[135] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.