YaleNUSCollege

# Verifying Distributed Protocols: From Executable to Decidable

**Mark Yuen**

**Capstone Final Report for BSc (Honors) in Mathematical, Computational, and Statistical Sciences**

**Supervised by: Dr. Ilya Sergey**

**AY 2021/2022**

YALE-NUS COLLEGE

# *Abstract*

BSc (Hons)

**Verifying Distributed Protocols: From Executable to Decidable**

by Mark YUEN

Modern distributed protocols are notoriously complex, leading to bugs that are difficult to uncover through testing. Formal verification methods can *prove* distributed protocols correct, but have historically required mostly manual effort. TLA$^+$ and Ivy are two tools that have been developed for the purpose of *automated* verification of distributed protocols. TLA$^+$ is a popular tool that offers executable verification through model checking. It is quick to learn and has been successfully applied to real-world systems, but is unsound due to the finitude of model checking. Ivy offers sound and decidable verification through its use of first-order logic, but the tool is difficult to use due to its restrictive specification style.

In this report, we present a study towards systematic translation from TLA$^+$ to Ivy, enabling a shift from model checking to sound verification. We discuss why this process is difficult and offer rewriting rules to circumvent some common issues. Through this process we also present the *first* computer-aided verification efforts of the Suzuki-Kasami and NOPaxos distributed protocols using TLA$^+$ and Ivy.

**Keywords:** Distributed Systems, Automated Formal Methods, Safety and Liveness Verification, Model Checking, Decidable Logic, TLA$^+$, Ivy

# Contents

# Chapter 1

# Introduction

## 1.1 What are Distributed Systems?

Distributed systems are ubiquitous throughout the modern world, forming the backbone of nearly all online services. Distributed systems provide reliability and scalability to our services, but they are also notoriously difficult to design and implement properly [1]. With distributed systems utilizing physically distinct computers, or nodes, each running their own processors concurrently, there are a number of real-world constraints which introduce non-determinism in the execution of a system, making it difficult to comprehensively test them.

Typically, a distributed system comprises of a set of nodes working together on a shared network, where nodes communicate through message passing. Critically, real-world networks are characterized as being *asynchronous*, meaning that a shared conception of time and ordering is impossible. A given node's internal clock can (and will) differ from another node's internal clock, and message passing is unreliable—messages may be dropped, duplicated, and/or take infinitely long to be delivered. Furthermore, any mature system should also be able to handle unexpected network partitions and the inevitable hardware failure of nodes within the system. All these problems contribute to the difficulty of building systems that can effectively leverage the increased computing storage and power of a distributed system.

When we talk about using a distributed system, at a high level, we can think of ourselves as a client sending a request to a server (or group of servers) to instruct the server(s) to do some work and send back a response once completed. For instance, when we upload a file to Google Drive, we expect Google's servers to store and propagate that file through its data centers, then inform us once it is successfully uploaded. As with

any web service, we expect this service to effectively always be available to access. What this implies for distributed systems, however, is that most of the protocols which enable our systems to overcome the issues outlined above are running *forever*. This infinitude of distributed protocols leads us to evaluate them on different metrics than terminating programs. The two types of properties we use for evaluation are called safety and liveness—roughly, that nothing ever goes wrong and that something good occurs eventually, respectively.

Given the mission-critical nature of distributed systems—spanning fields such as healthcare, transportation, and banking, to name just a few—there are strong motivations to gain formal guarantees about our systems' underlying protocols. While testing is an important step towards building trust in our systems, for such important technology, we would like to go a step further and apply formal verification techniques to *prove* the safety and liveness properties we desire of our system.

## 1.2 Problem Statement

There have been a number of successful attempts at verifying distributed systems conducted in the past [4, 19, 24]. However, such attempts required mostly *manual* effort, with proof-to-code ratios exceeding one by a large margin and requiring many person-years of effort to prove *specific* protocols.[1] Such large-scale manual verification efforts are unamenable for industrial adoption. To move beyond manual verification, there is increasing effort being put towards *automated* verification techniques that shift the burden of proving from the human to the computer.

In this project we will look at two different tools that have been developed for the purpose of automated verification of distributed protocols. The first of these is TLA$^+$ (Temporal Logic of Actions), which offers a set-theoretic specification language to run *lightweight* verification

[1] The Verdi project [24] presented a Coq formalization of the Raft consensus protocol [14], taking over 50,000 lines of proof for just 500 lines of implementation code.

through model checking. Model checking conducts an executable search through the tree of states allowed by the specification, starting from an initial state. Although model checking is ultimately *unsound* because it is finite—and, thus, may admit specifications which are actually buggy—the expressive language makes it a tool that is easy for developers to pick up and use with immediate and impactful success [13]. The second tool we will explore is Ivy, which offers a much more restrictive specification language based on a decidable fragment of many-sorted first-order logic (FOL). While expressiveness is sacrificed—as is typically the case with automated proof techniques—the trade-off is a sound verification due to the decidability of the FOL formulae allowed by the language. Furthermore, many-sorted FOL allows us to specify infinite systems, since FOL quantification over sorts is agnostic to the size of a sort.

TLA$^+$ was first developed by Microsoft researchers more than twenty years ago and has proven itself valuable in both academic and industrial settings. As a result, there are ample existing TLA$^+$ specifications out there, and almost every notable distributed protocol has been formalized in it. On the contrary, Ivy is a product of the academic community that came out in 2016, and its adoption has almost exclusively stayed within the academic sphere due to the difficulty of its usage. In fact, a number of *individual* verification efforts of protocols in Ivy have led to publications in distinguished conferences [17, 18, 21].

The overarching problem this project seeks to address is the potential for *mechanized translation* from TLA$^+$ to Ivy, enabling a shift from model checking to a proper sound verification of protocols. If such a framework could be developed, it would allow us to verify existing accepted TLA$^+$ specifications—many which may serve as the basis for critical real-world systems. It would also provide an easier introduction to Ivy and formal methods as applied to distributed systems.

## 1.3 Contributions

The contributions of this paper can be summarized as follows:[2]

1. We present the first verification of the Suzuki-Kasami distributed mutual exclusion algorithm [20] in both TLA$^+$ and Ivy. The specifications for this effort are specifically tailored to each tool, and we use this as an example to outline differences in the formalization and verification styles of TLA$^+$ and Ivy.
2. We present the first sound verification of a significant portion of NOPaxos, a state-of-the-art distributed consensus protocol [8]. This effort was conducted by taking the TLA$^+$ specification created by the original paper's authors and *translating* it into Ivy.
3. We document key deviations between the TLA$^+$ and Ivy specifications of the Suzuki-Kasami and NOPaxos protocols, noting portions that had to be re-expressed to be suitable for Ivy's verification.
4. Finally, we build upon the previous contribution by taking a more structured approach to our core problem statement and sketching a framework for *mechanized* translation from TLA$^+$ to Ivy.

[2]All specifications referenced in this report are publicly available here: https://github.com/markyuen/tlaplus-to-ivy.

# Chapter 2

# Background

## 2.1 Specifying Distributed Protocols

Before we discuss our tools, we need to describe a method of *specifying* distributed protocols that is suitable for automated reasoning. Real-world implementations are often much too complex to be verified directly, since they involve networking, hardware architecture, and implementation language specifics that are not relevant to the core protocol we really care about. As such, we adopt a method of abstracting away the implementation details and focusing on the core state and actions of a protocol, using the generic language of FOL and some basic set theory.

A distributed protocol can be conceived of as a *discrete state-transition system*, where each node has an initial starting state and a set of possible actions, or transitions, it is allowed to take. An intuitive understanding of this design is that each type of message should correspond to an action in the system, and each action is enabled by conditions depending on the local recipient node's current state and incoming message's state. While nodes within this system may have distinct initial starting states, the actions and their constraints, called *enabling conditions*, are shared logic for all nodes. If an action can be taken, as determined by its enabling conditions, then we consider that action to be *enabled*. While this shared logic may sound restrictive in forcing uniformity, in practice it is simple to add local state and corresponding enabling conditions which restrict actions to specific node subtypes a system may like to define. By utilizing the language of FOL, the payoff for this uniform description of the protocol is that it will allow us to specify systems in a way that model checkers and satisfiability modulo theories (SMT) solvers can automatically verify.

With the protocol defined in terms of discrete states, the desired properties of safety and liveness can be fleshed out, giving us an idea of what

it means to verify a specification. For safety, a formula can be defined over the system's state which must hold in the initial state and repeatedly after all possible interleavings of enabled transitions. If this formula holds over all behaviors of the system and we can show that it implies the safety property of our protocol, then we will have verified that the specification describes a safe system. For liveness, we can define two different formulae. One will describe the *invocation* state of the liveness property and the other will describe the *fulfillment* state of the property. Then, we expect that whenever the former formula holds, then after some number of transitions, the latter formula will also become true. An example of liveness verification will be covered in Section 3.2.3, where we model-check liveness of the Suzuki-Kasami algorithm.

## 2.2 Verifying Protocols with TLA$^+$

TLA$^+$ is a tool created by Leslie Lamport for the lightweight verification of concurrent and distributed protocols [6]. A TLA$^+$ specification consists of an initial formula and a collection of state transitions. Each transition is constructed as a boolean formula, where a transition is enabled as long as the boolean formula evaluates to true. Formulae which define the post-state of a variable simply evaluate to true. On a more granular level, each transition can be split into two parts: the joining (by logical conjunction and/or disjunction) of the enabling condition formulae which determine whether the transition can be taken, and the assignment of the post-state of the system. Every transition in TLA$^+$ must describe the post-state completely, including if a variable remains unchanged.

A TLA$^+$ specification must consist of an `Init` formula along with a `Next` formula, which should be the disjunction of all transitions. Then, TLA$^+$'s model checker, TLC, will programmatically begin the exploration of all possible states stemming from the initial state, branching through all enabled actions of `Next`. This process corresponds to the temporal

formula `Init /\ []Next`,[3] where "`[]`" is the temporal operator for "always," which asserts that the initial state is enabled and we can always take a `Next` step. This specification does not constrain the system in any way, however, and as such, does not say anything interesting by itself.

To introduce safety and liveness properties we can add additional formulae to this specification formula (through conjunction) using temporal operators, in order to have the model checker assert properties about the state we expect of our system. For instance, a safety property, written as the formula `Inv`, may be checked by extending the formula to be `Init /\ []Next /\ []Inv`.[4] If we think of model checking as generating a tree of states with the initial state as the root node, our safety property can be understood as a formula evaluated at *every node* in the tree, and the liveness property as evaluating branches, or *behaviors*, of the tree.

### 2.2.1 Limitations of Model Checking

In a TLA$^+$ specification it is necessary to add a condition to make the search space of TLC tractable. Without a constraint, TLC will run forever on an infinite protocol. Since TLC explores *concrete* states and does not attempt to group together abstractly similar states or behaviors with previously-explored states/behaviors, there is never a reduction in the number of states. The exponential nature of the exhaustive search limits the usability of TLA$^+$ as the specification grows. In general, it is tough to say when the TLC model checking is complete, in the sense that it has

[3] More precisely, this should be written as `Init /\ [][Next]_vars`, where `vars` is a collection of all variables in the specification. The `[][Next]_vars` construct asserts that either a `Next` action is taken, *or* no action is taken and all variables remain unchanged. The latter phenomenon is called a stuttering step and TLA$^+$ considers such a step to be valid. This concept will reappear in Section 3.2.3 when we discuss liveness. One can refer to https://lamport.azurewebsites.net/tla/stuttering.html for a discussion on why this phenomenon is meaningful in the first place.

[4] Or, `Init /\ [][Next]_vars /\ []Inv`. Here, we do not have the same construct for `Inv` because that formula should still hold even if a `Next` step is not taken.

explored all potential "patterns" of interleaving actions, and TLC is only able to report errors on the concrete violations it encounters.

In addition to constraining the search space, it is also necessary to explicitly define the number of nodes within the system for an iteration of model checking. In reality, we may actually expect our protocol/system to be agnostic to the number of nodes. Using TLA$^+$, we would need to run multiple iterations of TLC with differing input sets to be sure that the specification is properly agnostic to a system's configuration (and again, we would need to somehow gauge when this checking is good enough).

## 2.3 Inductive Invariance for Safety Properties

For distributed protocols, in order to *prove* the correctness of a specification for preserving a safety property, we utilize the concept of seeking an *inductive invariant* for our specification. There are three conditions for a formula, `Inv`, to be an inductive invariant for a safety property `I`:

1. $\texttt{Init} \implies \texttt{Inv}$,
2. $\texttt{Inv} \land \texttt{Next} \implies \texttt{Inv}'$, and
3. $\texttt{Inv} \implies \texttt{I}$.

The first condition states that `Inv` must hold in the initial state. The second condition introduces the "inductiveness" of `Inv`. It states that if in some state `Inv` holds and it is possible to take a `Next` action, then `Inv` must also hold in the post-state. Finally, the third condition asserts that `Inv` captures the desired safety property. If an inductive invariant can be found for a specification, then we know that the specification is safe.

Bounded model checking is unable to verify if a formula is an inductive invariant, and TLA$^+$ does not provide support for checking inductive invariance. Note that the inductive invariant is almost always an *over-approximation* of the actual feasible state space while TLC only checks concrete states. Although TLA$^+$ falls short in this regard, TLA$^+$ *does* allow us to check liveness properties. When we talk about automated verification, we will focus solely on proving safety properties, precisely in

terms of inductive invariance. Liveness properties have yet to really be automatically proven, though there has been some recent work done on re-expressing liveness in terms of safety in [18].

## 2.4 Specifying Protocols with Ivy

In order to move beyond model checking and into proper verification, Padon et al. have refined a language called relational modeling language (RML) to describe infinite systems, and have created the tool, Ivy [16], to interactively search for an inductive invariant of RML specifications. RML describes infinite systems using the semantics of many-sorted FOL. Like TLA$^+$, RML specifications consist of an initial state and transitions, where a transition is enabled if it evaluates to true. To evaluate inductive invariance, Ivy converts RML specifications into their FOL representations for the Z3 SMT solver to process. In general, when working with SMT solvers, it is imperative to consider the *decidability* of our queries, since an undecidable query may not terminate. Thus, Ivy enforces the decidability of RML specifications, which ensures that the specification can be automatically verified. Henceforth, we will drop the RML name and refer to RML specifications/syntax as Ivy's specifications and syntax.

### 2.4.1 The Decidable Fragment of Many-Sorted FOL

Ivy is built upon the effectively propositional (EPR) class of FOL which is decidable [7]. EPR formulae are those which have a prenex normal form (PNF) quantifier structure of $\exists^* \forall^*$, where the "$*$" indicates any number of repetitions of the preceding quantifier. This EPR fragment is limited, but it can be extended to include *stratified* functions, meaning if a function used in an axiom or invariant has domain of sort $x$ and range $y$, there cannot also exist a function in an axiom or invariant with domain $y$ and range $x$. Furthermore, through a process called Skolemnization, $\forall^* \exists^*$ PNF quantifier alternations may also be present. Provided a PNF formula with quantifiers $\forall x \, \exists y$, Skolemnization can convert that into an *equisatisfiable* $\forall x$ formula at the cost of introducing a new function with domain

$x$ and range $y$. Ultimately, the functions of a specification—both used in axioms and invariants, and introduced through Skolemnization—can be synthesized into a *stratification graph*, where each sort is a vertex and edges are drawn from each function's domain to its range. If this stratification graph has no cycles then the specification will be decidable.

### 2.4.2 Verifying Protocols with Ivy

When provided with the initial state, transitions, and invariant formula, Ivy will check if the invariant is inductive. Looking at the three conditions from Section 2.3, it is trivial to ascertain condition 3, since the safety property formula(e) must be defined in conjunction with the invariant.[5] Conditions 1 and 2 are verified using SMT queries. Since SMT solvers will terminate once they find a satisfiable assignment to a query, Ivy constructs each respective formula, then *negates* them before sending them down to the SMT solver. These negated formulae are called the *verification conditions* (VC) of the specification. To prove that our conditions are *always* satisfiable, we verify that the VCs are *unsatisfiable*.

A critical realization of this process is that the invariant occurs both positively *and* negatively in the VC of condition 2. Examining it we have:

$$\neg(\texttt{Inv} \wedge \texttt{Next} \implies \texttt{Inv}') \iff \texttt{Inv} \wedge \texttt{Next} \wedge \neg\texttt{Inv}'$$

As such, although a formula with $\exists^* \forall^*$ quantifiers may be decidable when occurring positively (e.g., as an enabling condition for a transition in `Next`), if that formula appears in the invariant, then its negation will *swap* the quantifiers, pushing the specification's decidability into a question of whether or not the resulting Skolem function(s) introduces a cycle in the stratification graph.

[5] In fact, there is no syntactic difference when defining safety property invariants and other invariants in Ivy, and this is always a valid transformation of the safety property. Assume, for a contradiction, that a verifiable specification contains safety property formulae that are *inconsistent* with the other invariants defined. However, that means the constructed invariant will never be satisfiable, so the initial state will not satisfy the invariant and condition 1 will not hold. Therefore, the specification will not verify.

# Chapter 3

# Case Study: Suzuki-Kasami

In this section we will discuss a case study of specifying and verifying the Suzuki-Kasami algorithm in TLA$^+$ and Ivy. While the TLA$^+$ specification was developed first and referenced in the creation of the Ivy specification, the primary goal of this exercise was to accurately specify the Suzuki-Kasami mutex, as opposed to merely translating the TLA$^+$ version into Ivy. From the resulting specifications we will identify divergences in the encoding of certain portions of the algorithm which will help to inform our understanding of the limits of translation in later chapters.

## 3.1 The Suzuki-Kasami Mutex Algorithm

The Suzuki-Kasami distributed mutual exclusion algorithm is a token-based mutex algorithm [20]. The algorithm is deadlock- and starvation-free, with critical section (CS) invocations granted in a first-come-first-served (FIFO) manner. In a system with $N$ nodes, the algorithm takes at most $N$, and sometimes zero, message exchanges for each CS invocation. The main idea behind the algorithm is that there is a single token that will get passed around, and whoever has the token is allowed to enter the CS.

If a node wants to enter the CS and already has the token, it can immediately enter. If a node $n$ wants to enter the CS but does not possess the token, it can send a request message of the form `REQUEST(`$n$`,`$i$`)` to all other nodes, where $i$ is a sequence number indicating that node $n$ is requesting its $i$th instance of holding the token.

A node $n$ carries three fields of state: a boolean, `havePrivilege`, indicating whether or not the node currently possesses the token, another boolean, `requesting`, indicating whether or not the node is currently requesting to enter the CS, and an array, `RN`, of length $N$, where `RN[`$n$`]` is node $n$'s own sequence number and `RN[`$m$`]` is the latest sequence number

received from node $m$. When a node receives a request from node $m$ with sequence $i$, it updates `RN[`$m$`]` to be the max between `RN[`$m$`]` and $i$.

The token can be sent to another node through a privilege message of the form `PRIVILEGE(Q,LN)`, where `Q` is a queue of requesting nodes and `LN` is an array of length $N$, where `LN[`$n$`]` is the latest instance that a node $n$ has held the token. When a node $n$ finishes a CS invocation (and as such, holds the token), it will update `LN[`$n$`]` $:=$ `RN[`$n$`]`, then append to `Q` any node $m$ that is not already in `Q`, where `RN[`$m$`]` $=$ `LN[`$m$`]` $+ 1$. Then, the message `PRIVILEGE(`$tail$`(Q),LN)` can be sent to $head$`(Q)`, otherwise the token will be retained by $n$. If a `REQUEST(`$m$`,`$i$`)` is received when a node is not in the CS or requesting the CS, and the request is up to date (`RN[`$m$`]` $=$ `LN[`$m$`]` $+ 1$), then it can immediately send `PRIVILEGE(Q,LN)` to $m$.

Pseudo-code for the Suzuki-Kasami algorithm is provided below:

```
 1: procedure P1(n)                          ▷ Node n requests to enter the CS
 2:     requesting ← TRUE
 3:     if not havePrivilege then
 4:         RN[n] ← RN[n]+1
 5:         for all m ∈ {1, 2, ..., N} \ {n} do
 6:             send REQUEST(n, RN[n]) to m
 7:         end for
 8:         wait until PRIVILEGE(Q, LN) is received
 9:         havePrivilege ← TRUE
10:     end if
11:     Critical Section                    ▷ Node n can enter the CS here
12:     LN[n] ← RN[n]
13:     for all m ∈ {1, 2, ..., N} \ {n} do
14:         if not in(Q, m) and RN[m] = LN[m]+1 then
15:             Q ← append(Q, m)
16:         end if
17:         if Q ≠ empty then
18:             havePrivilege ← FALSE
19:             send PRIVILEGE(tail(Q), LN) to head(Q)
20:         end if
21:     end for
22:     requesting ← FALSE
23: end procedure
24:
```

25: **procedure** P2($m, i$) ▷ `REQUEST(`$m, i$`)` is received; P2 is indivisible
26: `RN[`$m$`]` $\leftarrow max($`RN[`$m$`]`$, i)$
27: **if** `havePrivilege` and not `requesting` and `RN[`$m$`]` $=$ `LN[`$m$`]` $+1$ **then**
28: `havePrivilege` $\leftarrow$ `FALSE`
29: send `PRIVILEGE(Q,LN)` to $m$
30: **end if**
31: **end procedure**

## 3.2 Modeling Suzuki-Kasami in TLA$^+$

### 3.2.1 Model State

In order to model the Suzuki-Kasami algorithm in TLA$^+$, two additional fields are added beyond what the paper proposes. To model distinct privilege messages, a token sequence number is added to the message, and each node contains an extra field to store information on the current (or latest) version of the token they hold. In the model state, which contains a set of all sent privilege messages, this allows for differentiation between token versions. The model utilizes four variables for state:

1. `nState`: where `nState[`$n$`]` is the local state of a node $n$, consisting of the four fields `havePrivilege`, `requesting`, `RN`, and `tokenSeq`,
2. `reqs`: the set of all request messages sent,
3. `tokens`: the set of all privilege messages sent, and
4. `crit`: the set of nodes in the critical section.

Each request message consists of three fields: `for` and `from` fields indicating the recipient and sender nodes, respectively, and `seq` containing the sender's sequence number. Each privilege message consists of four fields: `for` indicating the recipient (and hence, owner of this version of the token), `Q` and `LN` for the token state, and `seq` indicating the token sequence number. Since each privilege message contains the token, each node will directly access the *message's* token fields when interacting with their held token in the model. As such, the set of privilege messages is named `tokens` and its elements are considered to be versions of the token.

The model is initialized with two constant fields: a `Node` set containing identifiers for each node we want to include in an execution of the model, and a `MaxTokenSeq` value which is exclusively used to bound the search space to behaviors which do not proceed beyond a token sequence number of `MaxTokenSeq`. The paper uses numbers for node identifiers (ID), and describes `RN` and `LN` arrays indexed by node ID. In TLA$^+$, we can instead use a function to encode `RN` and `LN`, where the `Node` set is the domain, so its elements can be applied to access `RN` and `LN` values.

### 3.2.2 The `Init`-Formula and `Next`-State Transitions

The initial state is defined as follows:

```
Init ==
   /\ nState =
      [n \in Node |-> [havePrivilege |-> n = TokenInit,
                       requesting    |-> FALSE,
                       RN            |-> ArrInit,
                       tokenSeq      |->
                          IF n = TokenInit THEN 1 ELSE 0]]
   /\ reqs = {}
   /\ tokens = {CreateToken(TokenInit, <<>>, ArrInit, 1)}
   /\ crit = {}
```

In this definition, `TokenInit` is a random node from the `Node` set, and `ArrInit` is a function where for each node $n$, `ArrInit[`$n$`]` $:= 0$. The `tokens` set contains the initial token, which contains an empty queue (encoded as a sequence) and `ArrInit` for its `LN` array. The initial token has the sequence number of one, and the first token holder has privilege. The `reqs` and `crit` set are empty on initialization.

There are five `Next`-state transitions defined:

1. `Request(`$n$`)`: a node $n$ that has privilege can swap its `requesting` field to `TRUE` to request to enter the CS again. If $n$ does not have privilege, it will increment its sequence number, `nState[`$n$`].RN[`$n$`]`, by one, change its `requesting` field to `TRUE`, then broadcast request messages to all other nodes by adding them to the `reqs` set. This corresponds to lines 2–7 of the pseudo-code from Section 3.1.

2. `RcvRequest(`$n$`,`$r$`)`: for a given node $n$ and request message $r$ from node $m$, node $n$ will update `nState[`$n$`].RN[`$m$`]` $:=$ $r$`.seq` if $r$`.seq` $>$ `nState[`$n$`].RN[`$m$`]`. If $n$ has privilege but is not requesting the CS, it can generate the next privilege message for $m$, if `nState[`$n$`].RN[`$m$`]` is equal to $t$`.LN[`$m$`]` $+ 1$, where $t$ is the current token. This condition ensures that the received request has not already been fulfilled. This corresponds to procedure P2 of the pseudo-code.
3. `RcvPrivilege(`$n$`,`$t$`)`: for a given node $n$ and privilege message $t$ such that $t$`.seq` $>$ `nState[`$n$`].tokenSeq`, $n$ will update `nState` to indicate that it now has privilege and set its token sequence number to $t$`.seq`. The enabling condition ensures that only the latest token is received. This corresponds to lines 8–9 of the pseudo-code.
4. `Enter(`$n$`)`: a node $n$ that has privilege and is requesting the CS can enter the CS. This corresponds to line 11 of the pseudo-code.
5. `Exit(`$n$`)`: a node $n$ in the CS can exit the CS, switch its status to `nState[`$n$`].requesting` $:=$ `FALSE` and update the token. If the updated token queue is non-empty, it can send the next privilege message to *head*`(Q)`. This corresponds to lines 12–22 of the pseudo-code.

### 3.2.3 Safety, Liveness, and Tractability

The primary safety property we want to confirm is that only one node can be in the CS at any given moment. This is written as:

```
\A n, m \in crit : n = m
```

where "`\A`" is the universal quantifier and "`\in`" is set membership. This property can be added as an invariant when model checking. In addition to the mutex property, we define eight other invariants which help to assert certain properties we expect from the protocol.

The liveness property of the mutex we want to ensure is that any node that requests for the CS will eventually enter the CS. This can be expressed in TLA$^+$ using the "$\rightsquigarrow$" syntax, like so:

```
\A n \in Node : nState[n].requesting ~> n \in crit
```

This property states that if in any state `nState[`$n$`].requesting` is true, then in all behaviors stemming from that state, there will eventually be a future state such that node $n$ is in the CS.

In order for TLC to check this property, we need to impose weak fairness upon most of our actions. Since TLA$^+$ considers a behavior ending in infinite stuttering steps (i.e., no `Next` action is taken, no state is changed, and the invariants are still true) to be valid, the liveness property will be violated if the model stutters in a state before a requesting node can enter the CS, even if it is enabled to do so. By imposing weak fairness on all steps except `Request`, we ensure that the model will never allow these actions to remain repeatedly enabled forever, remedying this problem.

Finally, `MaxTokenSeq` is used to make the search space tractable by preventing TLC from searching beyond states where the constraint:

```
\A t \in tokens : t.seq < MaxTokenSeq
```

does not hold. Since each node can only have one outstanding request at any given moment, this constraint on the max token sequence number implicitly limits the number of request messages that can be generated (and explicitly limits the privilege messages). Without the ability to generate new messages, the model checker will run out of states to explore.

## 3.3 Modeling Suzuki-Kasami in Ivy

### 3.3.1 Model State and Transitions in Comparison to TLA$^+$

The specification in Ivy defines two sorts, `node` and `seq_t`, to represent nodes and natural numbers, respectively. The `seq_t` sort is instantiated as a total order with the key relation, `le` (less than or equal), being reflexive, transitive, anti-symmetric, and total. While Ivy enables the usage of some interpreted sorts (i.e., the background theory for the sort is built into the solver itself) that deal with numbers, in our experience, combining interpreted and uninterpreted sorts often lead outside of the decidable fragment and prevented us from stating certain formulae properly

(this is covered in greater detail in Section 6.2). As such, we define a custom sort and basic operations to compare and increment `seq_t`.

The state of each node is made up of two relations and two functions:

1. `relation n_have_privilege(N:node)`,
2. `relation n_requesting(N:node)`,
3. `function n_RN(N:node, M:node) : seq_t`, and
4. `function n_token_seq(N:node) : seq_t`.

The first element of each relation/function corresponds to the local node. This encoding of node state closely matches the TLA$^+$ encoding from Section 3.2.1, except that `nState` is destructured into its individual parts. In particular, the `RN` array would have been difficult to encode otherwise. To encode the request and CS sets, it is enough to use one relation for each.

The encoding for the `tokens` set is not so straightforward, however. Similar to how node state is destructured, `tokens` is also split into parts since the TLA$^+$ encoding uses a nested sequence for the queue and a function for the `LN` array. Since privilege messages can be uniquely identified by their token sequence number, we define two relations and a function with the first element corresponding to the token sequence:

1. `relation t_for(I:seq_t, N:node)`,
2. `function t_LN(I:seq_t, N:node) : seq_t`, and
3. `relation t_q(I:seq_t, N:node)`.

There are two important things to note about this encoding. Firstly, we have abandoned the FIFO property of the token queue. The queue is now encoded as a de facto set of nodes—for all $i$ and $n$ such that `t_q(`$i$`,`$n$`)` is true, we interpret this to mean the privilege message with sequence number $i$'s token's queue contains node $n$. In the `exit` transition where we need to send the token to the head of the queue, instead of having a FIFO ordering to determine the next recipient, we simply pick a random node from the "set." Crucially, without the FIFO properties of the queue, the algorithm does not maintain its starvation-freedom property and it is

possible for a requesting node to starve.[6] While this is not ideal, we do not believe this modification substantially detracts from the verification, where verifying the mutex property is of foremost importance.

Secondly, this encoding of privilege messages also *explicitly* exploits the fact that only one privilege message is generated per passing of the token. This can be seen through the `t_LN` function and `t_q` relation, which would not have the desired semantics if multiple privilege messages could share a token sequence number. This observation is made in comparison to the TLA$^+$ encoding, which does not force the uniqueness of token sequences through state, and instead verifies it through invariants. In TLA$^+$, the coupling of the entire token state as an element in `tokens` allows for differentiation between messages with the same sequence—this is *semantically* different to what is encoded in Ivy.

With regards to specifying transitions, TLA$^+$ utilizes a two-state vocabulary, in which the post-state must be completely defined for the transition to be accepted. In comparison, Ivy uses an imperative style of specification, where relation and function mappings only change if they are redefined within a transition. Actions in TLA$^+$ typically also do not use if-statements for control flow, and instead provide a *disjunction* of formulae with corresponding enabling conditions for each branch the action seeks to define. In Ivy, it is more natural to use if-statements. Nevertheless, after accounting for these minor difference in specification styles, the transitions in Ivy match those from TLA$^+$ closely, and there are no notable differences in transition logic.

[6] In a system with three nodes, $a$, $b$, and $c$, if $a$ and $b$ have higher priority when being selected from the queue, then $c$ can starve. Assume $c$ has requested access to the CS and has been added to the token queue. Assuming $a$ is currently in the CS, it can receive a request message from $b$ and add $b$ to the queue. Once $a$ exits, it will send the next token to $b$. Then, when $b$ is in the CS it can receive a request from $a$, causing it to add $a$ to the queue and send the token back to $a$ upon exiting. This can continue ad infinitum.

### 3.3.2 The Inductive Invariant

An inductive invariant for our specification consists of these five formulae; in Ivy, each free variable is implicitly universally quantified over its respective sort over the entire formula:

```
invariant [mutex]
    (crit(N) & crit(M)) -> N = M
invariant [allowed_in_crit]
    crit(N) -> (n_have_privilege(N) & n_requesting(N))
invariant [unique_tokens]
    (t_for(I, N) & t_for(I, M)) -> N = M
invariant [corresponding_tokens]
    n_token_seq(N) ~= init_seq -> t_for(n_token_seq(N), N)
invariant [current_privilege_latest_token]
    (n_have_privilege(N) & N ~= M) ->
        ~seq.le(n_token_seq(N), n_token_seq(M))
```

While this invariant is already inductive, we are free to extend the formula in order to assert other properties of the system, just as we did in TLA$^+$. Some version of each invariant defined in the TLA$^+$ specification has been ported to the Ivy version, bar `GoodTokenQueues`, which asserts that the token queue size is bounded by the number of nodes in the system. This trivially holds in the Ivy specification since our `t_q` relation implicitly limits the size of the queue to the cardinality of the `node` sort.

The only invariant that is not quite adequately ported over is the `CurrentPrivilegeLatestToken` invariant, despite its appearance in the inductive invariant. The Ivy form of `current_privilege_latest_token` states that if a node has privilege, then its token sequence number must be larger than all other nodes' token sequences. The other half of the TLA$^+$ version additionally asserts that if no node has privilege, then *there exists* some privilege message with sequence number greater than all nodes' token sequences. In a behavior, if this was not true, then the system would be in deadlock, since no node would have privilege and no privilege message would be in transit to grant a node privilege—so this invariant is critical to the liveness of our system. Furthermore, we would

like to ensure that there is *exactly one* privilege message in transit. Only one node should be able to receive the message to gain privilege.

An attempt to state this invariant looks like so:

```
(forall N:node. ~n_have_privilege(N)) ->
    (exists I:seq_t, M:node. t_for(I, M) &
        forall W:node. ~seq.le(I, n_token_seq(W)))
```

However, the consequent of this formula is not in the decidable fragment. Although positively it has the $\exists^* \forall^*$ EPR structure, as discussed in Section 2.4.2, through negation the VC will contain the quantifier structure of $\forall$`seq_t` $\forall$`node` $\exists$`node`, which introduces two edges on the stratification graph: one from `seq_t` to `node` and another from `node` to itself. Unfortunately, both edges introduce cycles in the graph. The latter is obviously a cycle, but the former also introduces a cycle since the function `n_token_seq`, which is used in this and other invariants, already introduces an edge from `node` to `seq_t`.

If we would like to retain this invariant, we will need to find a different way to state it. However, with only two sorts in our specification, we are limited in what we can express. Disregarding this desired invariant, the topological sort of our stratification graph is `node` $\rightarrow$ `seq_t`. Any other edge would make a topological sorting impossible. Even without this invariant, however, we are able to find an inductive invariant for the mutex safety property. This should not be too surprising, since the system that is deadlocked in a state where no node can gain privilege and thus, no node can enter the CS, trivially maintains the mutex property.

# Chapter 4

# Case Study: NOPaxos

In this section we will discuss another case study on the Network-Ordered Paxos (NOPaxos) protocol [8]. In the previous chapter we specified and verified the Suzuki-Kasami algorithm in TLA$^+$ and Ivy individually before comparing the specifications. In this chapter we will take a different approach that is more faithful to the goal of translation. We will take an existing TLA$^+$ specification of NOPaxos, provided by the paper's authors, and translate that into Ivy. We assume the TLA$^+$ specification to correctly specify the underlying protocol—we do not provide arguments for the correctness of the source TLA$^+$ specification. In comparison to the previous chapter's exercise, where certain encoding decisions were made in the Ivy specification that diverged from the TLA$^+$ specification but were justified for matching the protocol specification, here, we will focus primarily on translation without worrying about whether or not the specifications accurately describe the NOPaxos protocol.

## 4.1 The NOPaxos Consensus Protocol

NOPaxos is a distributed consensus protocol whose novelty comes from incorporating the network layer into distributed consensus, resulting in throughput and latency nearly matching *unreplicated* storage systems. Traditionally, when designing asynchronous state machine replication, there are two key problems to address: dropped messages and the arbitrary ordering of messages. When the network is treated as a black box of asynchrony, state machine replication relies on expensive consensus protocols between nodes of a system to provide an ordering over instructions to execute. However, the network does not need to be viewed in this manner. While dropped messages are currently unavoidable, something *can* be done to address message ordering. To this end, the authors of NOPaxos introduce the ordered unreliable multicast (OUM) network primitive that

can be implemented with programmable switches. This network-layer feature sends multicast messages with two important properties:

1. **Ordered Multicast:** If two messages, $m$ and $m'$, are multicast to a set of processes, $R$, then all processes in $R$ that receive $m$ and $m'$ receive them in the same order.
2. **Multicast Drop Detection:** If a message, $m$, is multicast to a set of processes, $R$, then either: (1) every process in $R$ receives $m$ or a drop notification for $m$ before receiving the next multicast message, or (2) no process in $R$ receives $m$ or a drop notification for $m$.

The OUM primitive is implemented in a switch called the *sequencer*. For a set of OUM recipients, the sequencer sits between the clients and recipients, and simply tags each multicast message with a sequence that increments by one for each message. In this way, recipients can uncover if they have missed a message from the sequencer.

The predetermined ordering granted by OUM allows NOPaxos to complete a round of consensus without the need for replica coordination. At any given moment, replicas in the system will keep track of the current *view*, which consists of a tuple of the OUM session ID, corresponding to the sequencer in charge for that session, and the leader sequence number, corresponding to the number of leader changes the system has undergone. The leader for each view is predetermined by this view ID.

In the "normal" case of the protocol, client requests are tagged with a sequence number then forwarded to the replicas. Replicas acknowledge client requests as long as the incoming sequence matches the expected sequence number (i.e., the replica has received all previous sequence numbers) and the sender is acknowledged as the current sequencer by the view ID. If a replica is the leader of the current view, it will execute the request and respond to the client with the result. All other replicas will respond to the client without including a result. Once the client receives a majority of responses with matching view IDs, one of which must be from the leader, the client will know that its request has been

persisted (and the result of its request). In this way, NOPaxos eliminates the need for replica coordination when all messages are delivered, which is an improvement over traditional distributed consensus protocols such as Multi-Paxos [22] and Raft [14].[7]

There are three "non-normal" subprotocols in NOPaxos that require coordination between the replicas:

1. **Gap Agreement:** Since the sequencer only sends each multicast out once and does not store a history of requests, the replicas need to coordinate amongst themselves to deal with dropped messages.
2. **View Change:** OUM session termination and leader failure require the functioning replicas to coordinate the move into the next view without violating consistency.
3. **Synchronization:** As an optimization, periodically, the leader will synchronize its logs with the other replicas, allowing the replicas to execute operations and update their local storage state. Recall that typically only the leader will execute the requests it receives.

For brevity, we omit further details on these subprotocols.

## 4.2 Overview of NOPaxos in Ivy

The specification in Ivy defines five sorts:

1. `seq_t`: totally ordered sort (same as the one defined in Section 3.3),
2. `replica`: nodes in the system that maintain a log of client requests—we expect this log to be consistent across replicas,
3. `r_state`: an enumerated type to represent a replica's status,
4. `quorum`: to represent quorums of `replicas`, and
5. `value`: values that clients can propose.

[7] In these protocols, only the leader for each round processes client requests. Once it picks a request for a round of consensus, it will forward the message to the other replicas and wait for a majority to acknowledge it. Once a majority responds, the leader applies the operation, responds to the client, and sends another message to the replicas informing them of the decision. This coordination increases the number of messages and time it takes for each round of consensus compared to NOPaxos, decreasing throughput.

The replica state consists of individual relations for each replica state field, and there is an individual relation for each message type the TLA$^+$ specification defines. To represent the sequencer, since the TLA$^+$ specification uses a function from numbers as sequencer IDs to numbers, we define a binary relation of two `seq_t` elements.

For the sake of decidability, our specification does not include any functions. In the rest of the specification there are two formulae that introduce edges in the stratification graph. The first formula is an axiom which states that every quorum intersects:

```
axiom [quorum_intersection]
    forall Q1:quorum, Q2:quorum.
        exists R:replica. member(R, Q1) & member(R, Q2)
```

This axiom adds an edge from `quorum` to `replica`. This sort of formula is the de facto method to capture the *core* property of set majorities in many-sorted FOL [10]. Traditionally, in protocol descriptions, set majority is verified by checking the *cardinality* of some $S' \subseteq S$ satisfies $|S'| \times 2 \geq |S|$. Since we cannot express set cardinality in FOL, we define a new sort, in this case, `quorum`, that captures the set majority property through an axiomatized relation, `member`, over the source sort, `replica`.

The second formula that adds edges to the stratification graph is our desired safety property. It states that for every two quorums of matching responses for a given log index the stored value is the same:[8]

```
invariant [consistency]
    forall V1:value, V2:value, I:seq_t. (
        (exists Q:quorum. member(lead, Q) &
            forall R:replica. member(R, Q) ->
                m_request_reply(R, V1, I)) &
        (exists Q:quorum. member(lead, Q) &
            forall R:replica. member(R, Q) ->
                m_request_reply(R, V2, I))
        ) -> V1 = V2
```

[8]This formula is more convoluted than consistency invariants for traditional consensus protocols since in NOPaxos requests are only confirmed to be persisted on the client-side—recall that the *client* waits for a quorum of matching responses.

This formula gives us edges from `value`, `seq_t`, and `quorum` to `replica`.[9] Thus, an acceptable topological sort for the sorts is `r_state` $\rightarrow$ `quorum` $\rightarrow$ `value` $\rightarrow$ `seq_t` $\rightarrow$ `replica`, and our specification is decidable.

In addition to the `consistency` invariant, there are sixteen other invariants specified which together form an inductive invariant for the specification. Nine of these invariants are added to constrain the relations of our model in a "sensible" manner—what this means will be discussed next. The other seven invariants help to assert expected behavior of the system. The TLA$^+$ specification did not come with any invariants besides the primary consistency property, so all these invariants were manually discovered. These invariants have also been "backported" to the TLA$^+$ specification to model-check their validity. We should always expect inductive Ivy invariants to hold in the source TLA$^+$ specification.

## 4.3 Deviations from TLA$^+$

In this section we will document notable differences between the TLA$^+$ and Ivy specifications. As was the case for the Suzuki-Kasami algorithm, plain old data (POD) types are more-or-less straightforward to define and modify in actions. On the other hand, it is particularly difficult to represent non-POD types, such as the log of NOPaxos.

### 4.3.1 Relational Fields

Certain POD replica state lends itself nicely to being expressed as a function (e.g., session message number and replica status). However, as briefly mentioned in Section 4.2, for the sake of decidability, we seek to avoid defining them as functions if possible. What we can do instead is to convert these fields into a binary relation where the first element is the local replica and the second is the POD field. Then, we can provide some basic bookkeeping to retain the semantics of a function.

[9] When converted to PNF, this results in a formula, $\varphi$, with the quantifier structure $\forall$`V1,V2:value` $\forall$`I:seq_t` $\forall$`Q1,Q2:quorum` $\exists$ `R1,R2:replica`. $\varphi$.

To capture the injective nature of a binary field relation, $R$, with elements of sort $S$ and $T$, respectively, we can define a "coherence" invariant:

$$\forall S\, \forall T_1, T_2.\ R(S, T_1) \land R(S, T_2) \implies T_1 = T_2$$

Since this formula is only universally quantified, we will always be able to specify it without sacrificing decidability. Ideally, we would also like to enforce the *existence* of this mapping in another invariant of the form:

$$\forall S\, \exists T.\ R(S, T)$$

However, this may lead the specification out of the decidable fragment since it will introduce an edge from $S$ to $T$. Even though we cannot always specify this "existence" invariant, this is not a real cause for concern due to the way that we access and modify these fields.

To access and/or modify such a field in an action, we can write a "guard" if-statement over the action which evaluates to true if a mapping is defined for the desired field's relation. For instance, provided a local first element, $S'$, using the Ivy syntax, we can wrap an action with:

$$\texttt{if some } T' : T.\ R(S', T')\ \texttt{\{ ... \}}$$

Then, the element $T' \in T$ will represent the current mapping of this field and will be usable in the body of the if-statement. When used in this manner, it turns out to not be imperative that we maintain the latter existence invariant since a mapping that is undefined in the pre-state will not satisfy the guard condition. If the entire transition logic is defined in the body of the if-statement, then the post-state will remain unchanged, so the inductive invariant will still hold. If a mapping *does* exist, by the first coherence invariant we know that the mapping is unique and there is only one choice for $T'$. Then, the action will be evaluated accordingly and we will still discover if the transition violates inductiveness.

### 4.3.2 Log Encoding

The trickiest part of the TLA$^+$ specification to translate is the replica's log. In the TLA$^+$ specification the log is defined as a sequence of values. In Ivy, we can use two relations to mimic the log:

1. `r_log_len(R:replica, I:seq_t)`, which specifies that the replica, `R`, has a log of length `I`, and
2. `r_log(R:replica, I:seq_t, V:value)`, which specifies that replica `R` has value `V` at index `I` of its log.

To constrain the log behavior we provide three basic invariants:

```
invariant [ll_coherence]
    (r_log_len(R, I1) & r_log_len(R, I2)) -> I1 = I2
invariant [log_coherence]
    (r_log(R, I, V1) & r_log(R, I, V2)) -> V1 = V2
invariant [log_valid]
    (r_log(R, I, V) & r_log_len(R, L)) -> seq.le(I, L)
```

The first two invariants are the same coherence invariants mentioned in Section 4.3.1, where we expect these relations to behave as a function. The last invariant, `log_valid`, asserts that the log length must be greater than or equal to every index of elements that exist in the log. Ideally, we would like to add at least one more invariant of the form:

```
(r_log_len(R, I) & seq.le(J, I)) ->
    exists V. r_log(R, J, V)
```

This would assert that every index less than or equal to the log length is filled. Unfortunately, this would introduce an edge from `replica` to `value`, which would create a cycle with our `consistency` property.

To access an element of the log, we need to again add a guard condition. However, this condition cannot always be checked before any logic of the transition is performed, since we may only know what index of the log to access after some prior computation, or only expect an index to be filled in a particular branch of an action. Therefore, there is a chance that parts of a transition are taken, but an expected log slot is undefined, leading to a state that violates the invariant. To resolve such an issue, we can *undo* any state change taken before the log access guard in the else-branch of the guard. This resolves such inconsistencies in a manner similar to the resolution of an undefined relational field, where the action does not modify any state so the invariant still holds.

### 4.3.3 View Change and Synchronization Subprotocols

Using this encoding of the log, we were unable to specify the view change and synchronization subprotocols of NOPaxos.[10] Both of these subprotocols require sending parts of the log in a message to another replica, and we were unsure of how to effectively model this. For the specification, since we do not model the view change subprotocol, we enforce that there is a single sequencer and single leader. This prevents view changes, which can be triggered by both new sequencers and other replicas.

To enforce a single sequencer, we use a pair of invariants which assert that only a defined individual `seq_t` element, `sequencer`, can be enabled:

```
invariant [single_sequencer_1]
    (S = sequencer & s_seq_msg_num(S, I)) -> seq.le(one, I)
invariant [single_sequencer_2]
    S ~= sequencer -> ~s_seq_msg_num(S, I)
```

To fix a single leader, we axiomatize a relation that is only true for an individual `replica` element, `lead`:

```
relation leader(R:replica)
axiom [single_leader] leader(R) <-> R = lead
```

Without view changes, the view ID remains static, so we drop the view ID field usage from the specification entirely.

Like many of the consensus protocols, the *failure* cases of the protocol are the most complex, and thus, some of the most important to verify. Unfortunately, we were unable to come up with a way to encode sending portions of the log, at least in the current way that the log is encoded. In the final chapter, Chapter 6, we will cover some additional portions of Ivy that we did not utilize that may have allowed us to fully specify NOPaxos, such as defining a modularized log *type* that could be used as elements of relations and functions.

[10] Naive attempts to encode log combination during view changes led us out of the decidable fragment when querying for the max index of multiple log relations. This called for quantifying over replicas, indices, and values for the existence of an individual replica, index, and value (i.e., $\forall^* \exists^*$ quantifier alternations).

# Chapter 5

# Towards Mechanized Translation

In this chapter we will begin laying down the groundwork for some basic translation rules from TLA$^+$ to Ivy. One of the preliminary concerns to keep in mind is that TLA$^+$ is *dynamically-typed*. While it is possible to "strongly-type" a specification by adding what is customarily named a `TypeOK` invariant which ensures that every transition maintains type coherence (see the Suzuki-Kasami specification for an example of this), such a formula is ultimately optional in TLA$^+$. This allows for certain shortcuts, such as defining a single set to hold all required message types (this design is used in the TLA$^+$ specification of NOPaxos). On the other hand, Ivy is strongly-typed—every FOL term and formula must be assigned a static sort. As with any dynamically-typed language, there is always a concern that the source file is ill-typed in some fashion. Before we attempt to translate, we would like to ensure that a source TLA$^+$ specification is well-typed. We will assume that sufficiently powerful type-inference can be conducted on a TLA$^+$ specification which will add type annotations to any well-typed specification and reject any ill-typed specification [12].

Another difference to reiterate here lies in the specification design choice. Ivy builds upon the syntax of many-sorted FOL. On the other hand, TLA$^+$ is based upon Zermelo-Fraenkel set theory with the axiom of choice (ZFC), which *is* a single-sorted FOL theory where the only sort is the set.[11] However, TLA$^+$ includes a number of extensions and idiosyncrasies that make translation difficult, such as using sequences, records, or even other functions as the domain of a function. Previous work has already attempted to define certain rewriting rules for the syntax of TLA$^+$ into FOL [11]. Here, we will expand on concerns specific to Ivy.

[11] This provides a justification for why TLA$^+$ is untyped. Sets, as conceived in ZFC, can contain anything except themselves.

## 5.1 Numbers

The least troublesome way to encode numbers in Ivy is through a totally ordered sort, as we did in both case studies. This encoding captures the key property of total ordering over numbers but fails to specify arithmetic operations such as addition and subtraction. This will be acceptable for certain protocols (e.g., Suzuki-Kasami), but will ultimately be inadequate for more complex protocols.[12] In the final chapter, in Section 6.2, we will discuss some options to use arithmetic theories, as well as the complexity they introduce and why we did not utilize them.

## 5.2 Sets and Functions

Each constant set in TLA$^+$ should be translated into a sort in Ivy (e.g., `Node` from Suzuki-Kasami). To encode variable POD-element sets in Ivy, the set can be conceived of as a mapping from its elements to a boolean indicating whether or not an element is in the set. If a TLA$^+$ set consists of records of $k$ POD fields (e.g., request messages in Suzuki-Kasami with three fields), then a $k$-ary relation can be constructed in Ivy to model the set. Then, to model adding to the set we can define the specified relation mapping to true, and to remove, we define the mapping to be false. Even if a TLA$^+$ set is not well-typed, we can attempt to extract each type into its own relation, like how we dealt with messages in NOPaxos.

For top-level functions defined in TLA$^+$ with POD domain and range, we can follow the translation detailed in Section 4.3.1 to ensure decidability. For nested functions (e.g., `nState` in Suzuki-Kasami), we can destructure them into their separate parts then convert any functions into

[12] This would have eventually been a problem for the NOPaxos protocol. When a sequencer invokes a view change, meaning a new sequencer will take over, the new sequencer will begin its session message number count from one. Then, for a replica to query into its log based off a session message number, it needs to compute an *offset* from the tail of the log, requiring subtraction. See the `HandleSlotLookup` transition in the TLA$^+$ specification for this logic.

relations as before. By the translations outlined above, we can convert all POD sets and functions from TLA$^+$ into relations in Ivy. This converts set membership and function application queries in TLA$^+$ into relational mapping queries in Ivy. Further work still needs to be done to translate non-POD sets and functions.

## 5.3 Actions and Modifying State

The use of destructured sets and functions in actions is convenient to translate due to the imperative style of programming in Ivy. We only need to concern ourselves with updating the modified relations/functions in a transition, so our specification does not get cluttered by needing to specify the post-state of unmodified state, as would be the case in TLA$^+$. Since actions are essentially always performed over individual nodes (as expected of a distributed system), state change in TLA$^+$ typically does not involve quantification over sets or functions. This makes translation more or less straightforward since we only need to adjust individual relation mappings at these steps in Ivy.

There are, however, instances in TLA$^+$ where we need to perform an update contingent on an entire set, such as when we want to broadcast a message to all nodes in the system. In Ivy, we would not be able to do this in a definite number of individual update steps since we do not have a concept of the size of sorts. For such "multi-updates" over a relation, we can use the Ivy syntax that allows us to dispatch an update over the entire domain of a sort. This is best displayed through an example:

```
m_marked_client_request(R, m_value, slot) := true;
```

This line is from the `handle_client_request` action of the NOPaxos specification in which the sequencer tags a client request then multicasts it to the replicas. By using the *capitalized* variable,[13] `R`, Ivy understands that

[13]Capitalized variables always reference the entire domain of a sort. Variables that do not begin with a capital letter can only refer to individual elements. The pseudo-exception to this rule is under the existential quantifier—since all quantified variables must also be capitalized—even though an existential variable refers to a single element.

this update should be applied for each element of the sort of `R`, which is deduced from the interface of `handle_client_request`.

All control flow in TLA$^+$ can be translated to Ivy, though sometimes TLA$^+$ specifications can use disjunctive formulas instead of explicit if-statements for control flow.[14] However, as long as the enabling conditions for each branch of a disjunction are translated, there is no semantic difference in converting this TLA$^+$ logic into if-statements. TLA$^+$ does not offer any other forms of control flow. Finally, only actions of TLA$^+$ occurring in the `Next` formula should be prefaced with `export` in Ivy, which will inform Ivy to verify the action for inductive invariance. Other actions that are only called within other actions should not be expected to maintain inductive invariance (e.g., `replace_item` in NOPaxos).

## 5.4 Sequences

Linear data structures are crucial to many distributed protocols. We saw how the Suzuki-Kasami mutex used a queue of nodes and NOPaxos used a log of values. The default linear data structure in TLA$^+$ is the sequence, which provides an interface powerful enough to act as an array from a general-purpose programming language. One possibility for encoding a basic sequence can be seen in Section 4.3.2. For some protocols this may be enough,[15] but as witnessed in Section 4.3.3, there are clear limitations. To support full translation we would like to encode an Ivy data type that can match the sequence interface of TLA$^+$. In the final chapter we will discuss some built-in library collections Ivy provides—including an array module—that might assist in this goal, but for now we note this pronounced gap in translation capability.

[14] For instance, our Suzuki-Kasami TLA$^+$ specification does *not* use if-statements for control flow in any of its actions of `Next`.

[15] For instance, this encoding would be acceptable for Raft's log operations, since Raft's view change protocol only requires sending and updating individual log slots.

# Chapter 6

# Conclusion and Future Work

Distributed systems and protocols are foundational to modern computing, and we want to ensure that they are always safe. The central motivation for this project lies in the capability of expressing distributed protocols in a decidable logic, allowing Ivy to discharge the proofs of inductive invariance to SMT solvers. In this paper we began to build a framework for mechanized translation from TLA$^+$ to Ivy despite the significant differences in specification languages. Ultimately, if this translation can be realized, this will go a long way not just towards building safer systems *now*, but also towards normalizing and making formal methods accessible for the future. Admittedly, there is still a lot more work to be done here, specifically in terms of translating complex types and actions into FOL. Fortunately, there are some key portions of Ivy that we did not utilize in our case studies. Here, we will discuss them briefly to explain how they can contribute to this task. Finally, as an extension of translation, we will also discuss recent work on automatic inductive invariant inference.

## 6.1 Modules and Modular Decidability

In our specifications, and in the vast majority of public Ivy specifications, the module system that Ivy provides goes unused. Most specifications consist of a single top-level specification to verify. However, one of the benefits of the Ivy module system is that it can assist in breaking cycles in the stratification graph. Ivy adopts an *assume-guarantee* reasoning where an individually verified module can expose an interface that another module, which is also individually verified, can use and assume to be correct. In fact, both modules can rely on one another. While this sounds cyclic on the surface, when we individually verify a module, we are assuming that the provided interface from the other module has always held *in the past*. In this way, we are proving that neither module is

the first to violate an assumption. Thus, by modularizing specifications, there is the potential to break undecidable quantifier alternations into separate modules for individual verification. While this sounds powerful, it takes a level of ingenuity to figure out how to split a specification into complimentary modules. The authors of the Ivy tool discuss examples of using a modular design for decidability in [21].

Ivy also provides some built-in library modules for types such as arrays, maps, queues, and iterators, as well as object interfaces modeling TCP, UDP, and timer functionality. While we have known of these files since the start of using Ivy, there is no existing documentation displaying their usage. We only uncovered a set of specifications using these modules late into the project timeline, and our attempts at emulating their usage of these library types and objects were unsuccessful. If we can figure out how to understand and properly utilize these modules—which will rely on a strong comprehension of the assume-guarantee reasoning of multi-module specifications—this has the potential to make translation much more natural by exposing interfaces to common data structures.

## 6.2 Interpreted Theories

The Z3 SMT solver provides some built-in interpreted theories for Ivy to use, including natural numbers, integers, and bit vectors. While we attempted to use these built-in number theories in previous versions of our verification attempts, they constantly led us out of the decidable fragment. Specifically, there are the constraints that universally quantified variables may only occur as arguments to uninterpreted symbols or as arithmetic literals.[16] For instance, we are unable to state the following (from a version of Suzuki-Kasami specified with built-in numbers):

```
invariant [no_consecutive_privilege]
    (t_for(I, N) & J = I + 1 & t_for(J, M)) -> N ~= M
```

[16] See the section on the finite almost interpreted fragment from: http://microsoft.github.io/ivy/decidability.html.

Since `I` is a universally quantified interpreted variable, it cannot be used as an argument to "+," which is an interpreted symbol. Naturally, basic arithmetic is imperative to many protocols, and using interpreted theories is the easiest way to gain access to this functionality. Interpreted theories, perhaps in conjunction with the module system, can be used in a manner to circumvent their constraints. For an extended discussion on our experiences using interpreted theories, see Appendix A.1.4.

## 6.3 Code Extraction

The final feature of Ivy we did not utilize was the C++ code extraction. While we can believe that our *specification* is properly verified, we should still be wary, firstly, of whether our specification correctly specifies the desired protocol, and secondly, whether the specification can produce a correct implementation [2]. In our case, we were unable to execute Ivy's code extraction command successfully on either the Suzuki-Kasami mutex or the NOPaxos protocol in their current state. We will need to explore this feature in greater detail to understand under what conditions code extraction is possible and what assumptions the Ivy developers take on to produce implementations from specifications.

## 6.4 Automatic Invariant Inference

Even after translating a TLA$^+$ specification into Ivy, it can take a lot of manual effort to come up with an inductive invariant for the system (see Appendix A.1.2 for insights into this process). Recent work has attempted to ease this burden by automating the discovery of inductive invariants. The ability to automatically search for an inductive invariant would greatly augment a tool for translation from TLA$^+$ to Ivy, since we do not expect that invariants specified in a TLA$^+$ specification will be inductive on their own. Basic invariants may not even be defined, like in the NOPaxos TLA$^+$ specification.

Such tools for automatic inductive invariant inference that take in Ivy specifications as input include I4 [9], SWISS [3], and DistAI [25]. I4 and

SWISS adopt a similar approach of attempting to generalize invariants that hold over finite instances of a system, whereas DistAI constructs an enumerated set of candidate invariants before filtering out those that fail to be inductive. Notably, only SWISS is able to generate invariants containing existential quantifiers, though these formulae are generated from templates, so the tool is not fully automated. I4 and DistAI only consider universally quantified candidate invariants.

Another more comprehensive tool for automatic inductive invariant inference that is closely related to Ivy—and was heavily inspired by Ivy—is mypyvy, developed by James R. Wilcox [23]. It has its own specification language—which matches the two-state vocabulary of TLA$^+$ quite closely—that can be used to verify infinite-state systems nearly identically to how Ivy verifies a system. For invariant inference, it employs different techniques than the other tools mentioned, applying some combination of universal property-directed reachability [5] and the primal-dual Houdini algorithm [15].[17] Wilcox writes in his doctoral thesis that there are plans to provide translation from Ivy to mypyvy and back again, allowing one to seamlessly search for inductive invariants.

While we initially sought to use mypyvy for this project, we found the imperative language of Ivy much more convenient for describing transitions. In mypyvy, it is not possible to modify variables more than once in a transition.[18] Nevertheless, if translation between Ivy and mypyvy can be realized, then mypyvy appears to be the most promising tool for the task of automatic inductive invariant inference, being the most actively-developed tool in this area.

[17] Since we did not end up using mypyvy seriously, we did not read very carefully into these techniques.

[18] For instance, the `exit` transition from the Suzuki-Kasami specification calls for two modifications of the token queue, and we struggled to express this in mypyvy.

# References


[1] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. "Paxos Made Live: An Engineering Perspective." In: *PODC*. ACM, 2007.

[2] Pedro Fonseca et al. "An Empirical Study on the Correctness of Formally Verified Distributed Systems." In: *EuroSys*. ACM, 2017.

[3] Travis Hance et al. "Finding Invariants of Distributed Systems: It's a Small (Enough) World After All." In: *NSDI*. USENIX, 2021.

[4] Chris Hawblitzel et al. "IronFleet: Proving Practical Distributed Systems Correct." In: *SOSP*. ACM, 2015.

[5] Aleksandr Karbyshev et al. "Property-Directed Inference of Universal Invariants or Proving Their Absence." In: *J. ACM* 64.1 (2017).

[6] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[7] Harry R. Lewis. "Complexity results for classes of quantificational formulas." In: *Journal of Computer and System Sciences* 21.3 (1980).

[8] Jialin Li et al. "Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering." In: *OSDI*. USENIX, 2016.

[9] Haojun Ma et al. "I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols." In: *SOSP*. ACM, 2019.

[10] Kenneth McMillan and Oded Padon. "Deductive Verification in Decidable Fragments with Ivy." In: Springer, Cham, 2018.

[11] Stephan Merz and Hernán Vanzetto. "Encoding TLA+ into unsorted and many-sorted first-order logic." In: *Science of Computer Programming* 158 (2018).

[12] Stephan Merz and Hernán Vanzetto. "Refinement Types for TLA+." In: *NASA Formal Methods*. Springer, 2014.

[13] Chris Newcombe et al. "How Amazon web services uses formal methods." In: *Commun. ACM* 58.4 (2015).

[14] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm." In: *ATC*. USENIX, 2014.

[15] Oded Padon et al. "Induction Duality: Primal-Dual Search for Invariants." In: *Proc. ACM Program. Lang.* 6.POPL (2022).

[16] Oded Padon et al. "Ivy: safety verification by interactive generalization." In: *SIGPLAN Not.* 51.6 (2016).

[17] Oded Padon et al. "Paxos Made EPR: Decidable Reasoning about Distributed Protocols." In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017).

[18] Oded Padon et al. "Reducing liveness to safety in first-order logic." In: *Proc. ACM Program. Lang.* 2.26 (2018).

[19] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. "Programming and Proving with Distributed Protocols." In: *Proc. ACM Program. Lang.* 2.POPL (2017).

[20] Ichiro Suzuki and Tadao Kasami. "A Distributed Mutual Exclusion Algorithm." In: *ACM Trans. Comput. Syst.* 3.4 (1985).

[21] Marcelo Taube et al. "Modularity for Decidability of Deductive Verification with Applications to Distributed Systems." In: *PLDI*. ACM, 2018.

[22] Robbert Van Renesse and Deniz Altinbuken. "Paxos Made Moderately Complex." In: *ACM Comput. Surv.* 47.3 (2015).

[23] James R. Wilcox. "Compositional and Automated Verification of Distributed Systems." PhD thesis. University of Washington, 2021.

[24] James R. Wilcox et al. "Verdi: A Framework for Implementing and Formally Verifying Distributed Systems." In: *PLDI*. ACM, 2015.

[25] Jianan Yao et al. "DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols." In: *OSDI*. USENIX, 2021.

# Appendix

## A.1 Experiences Using Ivy

In this section we will discuss, informally, some of our experiences with Ivy. Ivy comes with documentation,[19] but it is non-exhaustive in its coverage of the tool's functionality. Perhaps most crucially, it does not cover Ivy's library modules or examples using them. While the Ivy repository contains a variety of specifications of ranging complexities, they are not typically well documented, and thus, difficult to reason carefully about.

### A.1.1 Revisiting Decidability

Our initial understanding of decidability came from one of the first papers on Ivy [17], which constructs the stratification graph with these rules:

1. **Function Edges:** Let $f$ be a function from sorts $s_1, \ldots, s_k$ to sort $s$. Then, there is an edge from $s_i$ to $s$ for every $1 \leq i \leq k$.
2. **Quantifier Edges:** Let $\exists x : s$ be an existential quantifier that resides in the scope of the universal quantifiers $\forall x_1 : s_1, \ldots, \forall x_k : s_k$. Then, there is an edge from $s_i$ to $s$ for every $1 \leq i \leq k$.

However, this definition is not the most precise. From experience, we know that *not all* defined functions in a specification contribute to the stratification graph.[20] This led us to the definition of Section 2.4.1 where we only account for functions used/introduced in axioms and invariants.

As it turns out, this also is not as precise as it can be. In fact, we *can* have functions that induce cycles—or are cycles by themselves—occur in invariants. For a simpler example of this phenomenon we can consider an older version of the NOPaxos specification where we expressed a sequencer's session message number as a function, like so:

```
function s_seq_msg_num(S:seq_t) : seq_t
```

[19] View the Ivy documentation here: http://microsoft.github.io/ivy/.

[20] By the above definition, `function t_LN(I:seq_t, N:node) : seq_t` from the Suzuki-Kasami specification should induce a loop on `seq_t`.

Using this definition, we were still able to define the invariant:

```
invariant [single_sequencer_1]
    S = sequencer -> seq.le(one, s_seq_msg_num(S))
```

From our understanding, this function should have introduced a loop on `seq_t` in the stratification graph, causing Ivy to reject the specification for verification. While a better understanding of decidability is desired, in the framework described by our work it is possible to entirely avoid this complication by re-expressing functions as relations.

### A.1.2 Building Inductive Invariance

In general, the process of building an inductive invariant comes from identifying and eliminating a pre-state that is not actually reachable. Recall that an inductive invariant typically specifies an over-approximation of the states that the system will actually enter, given the initial state. When considering a counterexample to induction, once the offending portion of the pre-state is identified, then a new invariant can be added to prevent such a state from satisfying the prospective inductive invariant.

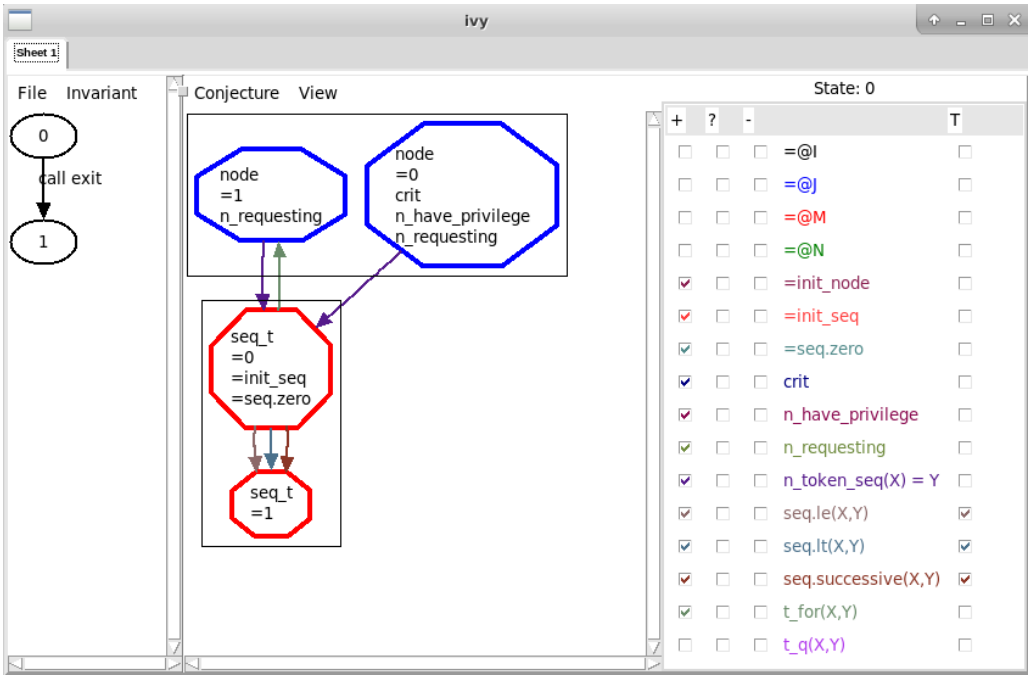
When verifying a specification, to view a counterexample to induction, the `ivy_check` command can be executed with the `trace=true` flag. Ivy will identify a minimal counterexample and print the offending transition's pre-state to the console, followed by any state change triggered by the transition. Unfortunately, this command does not actually print out the entire post-state of the transition, and one needs to carefully sift through the action's state modifications to identify the cause for failure. This process quickly becomes tedious as a specification's state increases.

As a note of caution to future users of Ivy, it can be disorienting when Ivy uses numbers to represent elements of a total order, but the ordering is the inverse of what we expect. For instance, a counterexample involving two totally-ordered elements may very well be axiomatized as:

```
seq.le(0,0) = true        seq.le(1,0) = true
seq.le(0,1) = false       seq.le(1,1) = true
```

### A.1.3 Using the GUI for Counterexamples

Ivy also provides a graphical user interface (GUI) to view counterexamples if the `ivy_check` command is passed the `diagnose=true` flag. Ivy draws functions and relations as arrows between individual elements. Below is an example of the GUI display from Suzuki-Kasami:

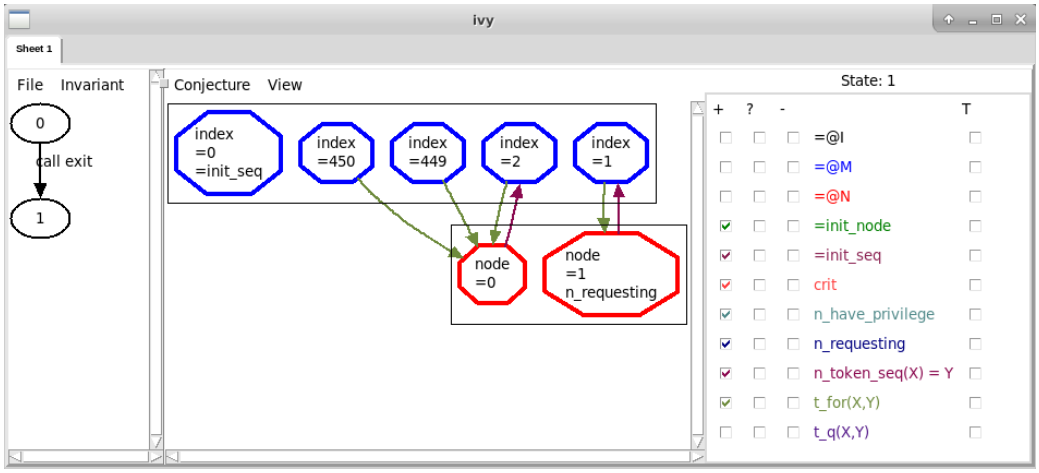



Using the GUI, it is possible to view the entire pre- and post-state of the counterexample transition. In our experience, the GUI was helpful when searching for the inductive invariant of the Suzuki-Kasami specification. However, a severe limitation of the tool is that it can only display relations up to an arity of two, and functions with one or fewer arguments. This made the GUI unusable for the NOPaxos specification since most of the message relations have an arity greater than two.

### A.1.4 Quirks of Interpreted Theories

When writing our Ivy specifications, some of our first attempts involved trying the built-in natural number and integer theories for numbers. However, in addition to the undecidability mentioned in Section 6.2, we also commonly encountered counterexamples that we could not understand.

For instance, one may notice that the inductive invariant specified for the Suzuki-Kasami specification in Section 3.3.2 does not consist of any formulae that would prevent decidability by the constraints outlined in Section 6.2. However, when such an invariant is maintained in an equivalent version of the Suzuki-Kasami specification where the only difference is interpreting the sequence sort as a built-in natural number (or integer), we encounter such a counterexample:




This *post-state* allegedly violates this invariant:

```
invariant [unique_tokens]
    (t_for(I, N) & t_for(I, M)) -> N = M
```

However, as displayed by the GUI, there do not appear to be any `t_for` relations that violate the invariant (look for the green arrows from `index` to `node` elements, they all appear to be injective). The output produced when verifying with the `trace=true` flag is also not helpful, since it prints the portion of the action modifying `t_for` like so:

```
suzuki_kasami_int.ivy: line 117:
    t_for(n_token_seq(fml:n) + 1,loc:m) := true
```

We are unsure of how to proceed given the provided counterexample information. This example is available in the code repository.[21]

[21] Link to the Ivy file: https://github.com/markyuen/tlaplus-to-ivy/blob/main/ivy/suzuki_kasami_int.ivy.

# *Acknowledgements*

This project would not have been possible without the support of my friends, family, and faculty. I would like to extend my sincerest thanks to Prof. Ilya, for piquing my interest in this topic two years ago which I have come to deeply appreciate, and for their invaluable guidance throughout this project. I would also like to thank George for their insights, contributions, and feedback through this process.

I am also grateful to my suitemates, old and new, for their rapport through these years; to a Capstone support group, including Dani, Linda, Yanhua, and Ziting for tiding me through the final weeks of the project, and to Zhang Liu for their feedback on the manuscript; and finally to LX, for their unwavering companionship.

**Yale-NUS College Capstone Project**

**DECLARATION & CONSENT**

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.

2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

**ACCESS LEVEL**

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

☑ Unrestricted access
Make the Thesis immediately available for worldwide access.

o Access restricted to Yale-NUS College for a limited period
Make the Thesis immediately available for Yale-NUS College access only from _____ (mm/yyyy) to _____ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
_____.

After this period, the Thesis will be made available for worldwide access.

o Other restrictions: (please specify if any part of your thesis should be restricted)
_____
_____

Mark Yuen, Elm College
Name & Residential College of Student

Mark Yuen
Signature of Student

April 1, 2022
Date

Name & Signature of Supervisor

April 1, 2022
Date