Compositional Model Checking of Consensus Protocols Specified in TLA+ via Interaction-Preserving Abstraction

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

5th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

6th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—Consensus protocols are widely used in building reliable distributed software systems and its correctness is of vital importance. TLA+ is a lightweight formal specification language which enables precise specification of system design and exhaustive checking of the design without any human effort. The features of TLA+ make it widely used in the specification and model checking of consensus protocols, both in academia and industry. However, the application of TLA+ is limited by the state explosion problem in model checking. Though compositional model checking is essential to tame the state explosion problem, existing compositional checking techniques do not sufficiently consider the characteristics of TLA+.**

In this work, we propose the Interaction-Preserving Abstraction (IPA) framework, which leverages the features of TLA+ and enables practical and efficient compositional model checking of consensus protocols specified in TLA+. In the IPA framework, system specification is partitioned into multiple modules, and each module is divided to the internal part and the interaction part. The basic idea of the interaction-preserving abstraction is to omit the internal part of each module, such that another module cannot distinguish whether it is interacting with the original module or the coarsened abstract one.

We apply the IPA framework to the compositional checking of the TLA+ specifications of two consensus protocols Raft and ParallelRaft. Raft is a consensus protocol which is originally developed in the academia and then widely used in industry. ParallelRaft is the replication protocol in PolarFS, the distributed file system for the commercial database Alibaba PoloarDB. We demonstrate that the IPA framework is easy to use in realistic scenarios and at the same time significantly reduces the model checking cost.

***Index Terms*—Compositional model checking , Consensus , Interaction-Preserving Abstraction , TLA+**

I. Introduction

Consensus algorithms allow a collection of machines to work as a consistent group that can survive partial failures of its members [**?**], [**?**], [**?**]. They play a key role in building reliable large-scale distributed software systems. For example, consensus algorithms are used to build coordination services, e.g., Zookeeper [**?**] and etcd [**?**]. Consensus protocols are also used to achieve fault-tolerance for replicated databases, e.g., Chubby [**?**], [**?**], Spanner [**?**], CosmosDB [**?**], and PolarDB [**?**].

Since consensus protocols lie in the core of various mission-critical systems, its correctness is of vital importance. Traditional software validation techniques are intensively used to improve the reliability of mission-critical systems, e.g. intensive design reviews, code reviews, static code analysis, stress testing, and fault-injection testing [**?**]. However, deep and subtle bugs are still found to hide in complex concurrent fault-tolerant systems, and are manifested only in rare and extreme cases [**?**]. It is widely accepted that human intuition is poor at estimating the true probability of supposedly extremely rare combinations of events in systems operating at a scale of millions of requests per second [**?**].

TLA+ (Tempoal Logic of Actions) is a lightweight formal specification language, especially suitable for design of distributed and concurrent systems [**?**]. Leveraging simple math, TLA+ can express concepts much more elegantly and accurately than a programming language can. Specifying a system in TLA+ forces you to be precise in what you actually want. By unambiguously writing your specification, you understand it better. Problems become obvious even without further exploration. More importantly, unlike programming languages, e.g. Java and Go, which are designed to be run and are limited to what a computer can do, TLA+ is designed to be explored. We use a model checker TLC to execute every possible behavior of our specification without additional human efforts.

Supported by Grant Name (No. xxx). Author X is the corresponding author.

The features discussed above make TLA+ widely used in both academia and industry. For example, Paxos and Raft are formally specified and checked using TLA+ [**?**], [**?**], and TLA+ specifications for Zookeeper is under development [**?**]. TLA+ is extensively used by Amazon Web Services to help solve deeply-hidden design problems in critical systems [**?**]. PolarFS is using TLA+ to precisely document the design of its ParallelRaft protocol, in order to effectively guarantee the reliability and maintainability of the protocol design and implementation [**?**]. MongoDB further leverages the formally specified design, verified by model checking, to conduct model-based test case generation and model-based trace checking on large scale system implementations [**?**].

The programmer can view TLA+ specifications as "runnable designs", which can be machine checked without additional human effort. However, the model checking of TLA+ specifications is cursed by the notorious state explosion problem [**?**], which limits the scale of checking and restricts the usefulness of TLA+ specifications. Putting it in another way, increasing the scale of checking can greatly improve the confidence of the system developers that the system does not have bugs pertaining to the complexities and subtleties of fault-tolerant distributed protocol design.

Compositional model checking is essential to increasing the scale of model checking of large distributed systems. It addresses the state explosion problem by verifying the individual components without considering the whole system. Effectiveness of these methods depends on whether an coarse enough (to reduce the checking cost) yet accurate enough (to ensure the correctness of checking) context can be found for each component such that all the essential behavior of that component can be checked. However, existing compositional checking techniques do not sufficiently consider the characteristics of TLA+ specifications, and are thus not applicable or efficient in model checking of TLA+ specifications.

In TLA+, we model a distributed system in terms of a single global state. This is a simple but generally useful way to model distributed algorithms and systems, as backed by the wide use of TLA+ in both academia, open-source communities and industry. This salient feature of TLA+ specifications can be utilized to enable efficient compositional model checking. Moreover, TLA+ is a lightweight formal method. After the specification is given, its model checking is fully automatic. The compositional verification should also be automatic. Formal reasoning after the model checking of each component is not acceptable for the intended users of TLA+.

Toward the challenges above, we propose the Interaction-Preserving Abstraction (IPA) framework, which is aimed at practical and efficient compositional model checking of TLA+ specifications of realistic distributed consensus protocols. The framework addresses the challenges above in three steps:

1) We divide the system specification in TLA+ into function modules. Each module consists of some actions implementing a specific function. The division is mainly derived from the natural modularity in system design and implementation, which usually has high cohesion and low coupling. More importantly, toward the objective of efficient compositional model checking, each module can be divided into two parts: the *internal part* within the scope of one module and the *interaction part* handling interaction with other modules.
2) We abstract away all the internal logic of each module, while only preserve the interaction logic. To model check each module separately, we use the abstracted specification of all other modules as the execution context of the module being checked. We provide constraints on the abstraction to ensure that the abstraction preserves the interaction. The constraints are straightforward to check for the specification developers.
3) We provide correctness proof of the compositional checking based on our IPA framework.

We apply the IPA framework to reduce the model checking cost for the specifications of two consensus protocols: Raft and ParallelRaft (PRaft in short). Raft is a consensus protocol which is originally developed in the academia and then widely used in industry. PRaft is the replication protocol in PolarFS, the distributed file system for the commercial database Alibaba PoloarDB [**?**]. The design of PRaft is derived from Raft and Multi-Paxos [**?**], [**?**]. The case study shows that there are intuitive patterns to conduct the interaction-preserving abstraction, utilizing the characteristics of consensus protocols. The case study also shows that it is intuitive to guarantee interaction-preservation of the abstraction. Moreover, the constraints in the IPA framework can be conveniently employed to double check the interaction-preservation. Experimental evaluation shows that the cost for direct checking is up to about 300 times of the cost for compositional checking using our IPA framework.

The rest of this work is organized as follows. Section II overviews the IPA framework and Section III presents the formal definition. Section IV presents the case study. Section V reviews the related work. In Section VI, we summary this work and discuss the future work.

II. IPA Framework Overview

The Interaction-Preserving Abstraction (IPA) Framework is designed to enable efficient compositional model checking of TLA+ specifications. In this section, we first introduce the characteristics of TLA+ specifications. Then we present the workflow to use the IPA framework.

A. TLA+ Basics

In the TLA+ specification language, a system is specified as a state machine by describing the possible initial states and the allowed state transitions called $Next$. Specifically, the system specification contains a set of system variables V. A *state* is an assignment to the system variables. $Next$ is the disjunction of a set of actions $a_1 \vee a_2 \vee \cdots \vee a_p$, where an *action* is a conjunction of several clauses $c_1 \wedge c_2 \wedge \cdots \wedge c_q$. A *clause* is either an *enabling condition*, or a *next-state update*. An enabling condition is a state predicate which describes

the constraints the current state must satisfy, while the next-state update describes how variables can change in a step (i.e., successive states).

Whenever every enabling condition ϕ_a of an action a is satisfied in a given "current" state, the system can transfer to the "next" state by executing a, assigning to each variable the value specified by a. We use "$s_1 \xrightarrow{a} s_2$" to denote that the system state goes from s_1 to s_2 by executing action a, and a can be omitted if it is obvious from the context. Such execution keeps going and the sequence of system states forms a trace of system behavior.

TLA+ has a model checker named TLC which builds a finite state model of TLA+ specifications for checking invariance safety properties (in this work, we do not consider liveness properties). TLC first generates a set of initial states satisfying the specification, and then traverses all possible state transitions. If TLC discovers a state which violates an invariance property, it halts and provides the trace leading to the state of violation. Otherwise, the system passes the model checking and is verified to satisfy the invariance property.

In TLA+, correctness properties and system designs are just steps on a ladder of abstraction, with correctness properties occupying higher levels, systems designs and algorithms in the middle, and executable code and hardware at the lower levels [**?**]. This ladder of abstraction helps designers manage the complexity of real-world systems. Designers may choose to describe the system at several "middle" levels of abstraction, with each lower level serving a different purpose (such as to understand the consequences of finer-grain concurrency or more detailed behavior of a communication medium). The designer can then verify that each level is correct with respect to a higher level. The freedom to choose and adjust levels of abstraction makes TLA+ extremely flexible.

For example, a low-level specification for leader election mechanism of Raft may accurately describe how an eligible server is selected as leader through voting, while a high-level one may directly assign some eligible server to be leader and leave the details of voting unspecified.

B. IPA Workflow

The basic objective of the IPA framework is to divide the system specification to modules, and model check each module separately, as shown in Fig. 1. To achieve compositional checking, we abstract away the internal logic of each module and only preserve the interaction logic. The abstracted modules serve as the execution context for the module to be checked separately.

As shown in Fig. 1, we have three levels of specifications. In the lowest level, we have the original specification. In the middle level, we have the compositional specification. In the uppermost level, we have the abstract specification. We assume that the abstract specification has passed the model checking of the correctness property. The main task is to verify that the original specification refines the abstract specification, thus also satisfying the correctness property. Based on the IPA framework, we only need to model check that the compositional specifications refine the abstract specification. This will imply that the original specification refines the abstract specification, as we will prove in Section III-C. There are basically three steps when applying the IPA framework, as detailed below.

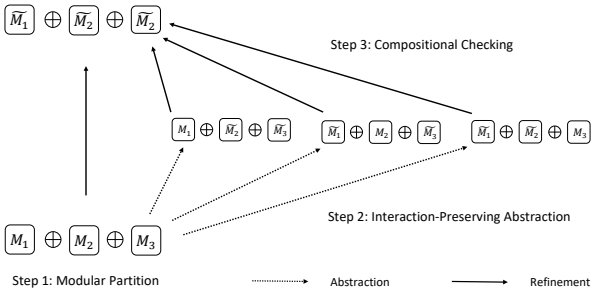

Fig. 1. The Interaction-Preserving Framework.

1) Modular Partition of Specification: Distributed systems are difficult to design and implement. To control the complexity of system development, it is common practice to design a distributed system as the composition of a collection of function modules. The modules are expected to have high cohesion and low coupling. For example, classic consensus protocol Raft can in high-level be divided into two modules: one for leader-based log replication and the other for recovery from failure of the leader.

The IPA framework leverages such modularity. The system specification is partitioned into multiple modules. Checking each module separately obviously can save the model checking cost significantly. However, the critical challenge in this partition process is to handle the unavoidable interaction among modules, and to construct an execution context for each module. The context should be sufficiently accurate, i.e., each module should be provided with the "illusion" that it interacts with other modules, not with the contexts minimized for compositional checking. The context should also be coarsened enough, otherwise the compositional checking cannot significantly reduce the checking cost.

We enforce the modular partition process by leveraging the characteristics of TLA+ specifications. In TLA+, we model a distributed system in terms of a single global state. This is a generally useful way to model distributed algorithms and systems, as backed by the wide use of TLA+ in the academia, the open-source community and the industry. As for TLA+ specifications, the key ingredients are the variables and the actions. To partition the system specification into modules is just to partition all the actions. Each module is just a subset of actions. Theoretically, this partition can be arbitrary. However, as discussed above, a distributed system is usually based on a modular design. This modularity should be and can naturally be preserved in the TLA+ specification. Module partition of the TLA+ specification should respect such modularity in system design because the high cohesion and low coupling nature of the design is expected to better reduce the model

checking cost.

The interaction among modules is based on read and write of system variables which are shared among modules, similar to global variables shared by multiple functions in C programming. For example in Raft, the log replication module and the leader election module access common variables (e.g. $term$ and log) and have subtle interdependencies, which makes it not feasible to check each module separately. We identify interaction among modules by identifying *interaction variables*, i.e., system variables which "convey" the interaction among modules. This enables further interaction-preserving abstraction. Based on the identification of the interaction variables, we can divide the logic of one module into two parts. One is the *internal* part, which just updates information within the module. The other is the *interaction* part, which involves interaction with other modules.

2) Interaction-Preserving Abstraction: To construct an execution context for each model as required by compositional model checking, we conduct interaction-preserving abstraction for each module. As indicated by its name, in this abstraction process, actions which do not interact with other modules are omitted. The coarsened abstract module is equivalent to the original detailed one, in the sense that other module cannot distinguish the abstract module from the detailed one during interaction.

When interaction among modules are simple, this abstraction process is often straightforward. For example, a distributed lock service has clear interfaces for other modules no matter how complicated the service is implemented. Therefore, the interaction-preserving abstraction for a lock service is simply the specification of the semantics of the lock service APIs. However, in many cases, the interaction among modules is much more complex and subtle. For example, the replicated log in Raft is accessed by multiple functionally different modules, e.g. the leader election module and the log replication module. Abstraction of actions manipulating the log is quite non-trivial. The key of the interaction-preserving abstraction process is to identify the system variables which "convey" the interaction among modules, as detailed in Section III-A.

3) Compositional Checking: The compositional checking based on the IPA framework is an indirect approach to verifying that the original specification satisfies the correctness property. Specifically, our main objective is to verify that the original specification refines the abstract specification, assuming that the abstract specification satisfies the correctness property, as shown in the left side of Fig. 1. In our indirect approach, we first conduct the interaction-preserving abstraction for each module and obtain the compositional specification for each module. Then we model check that each compositional specification refines the abstract specification. We will prove in Section III-C that the indirect checking implies the original direct checking. We will also show in Section IV-C that the indirect checking can significantly save model checking cost.

C. Maintaining the Integrity of the Specifications

The IEEEtran class file is used to format your paper and style the text. All margins, column widths, line spaces, and text fonts are prescribed; please do not alter them. You may note peculiarities. For example, the head margin measures proportionately more than is customary. This measurement and others are deliberate, using specifications that anticipate your paper as one part of the entire proceedings, and not as an independent document. Please do not revise any of the current designations.

III. Compositional Model Checking based on Interaction- Preserving Abstraction

In this section, we present formal description of the IPA framework. We first describe how to divide a system specification in TLA+ into modules and how to capture the interaction among modules. Then we describe how to conduct interaction-preserving abstraction for each module. Third, we prove that passing the compositional checking via the IPA framework implies passing the direct checking of the original specification.

A. Modules and Interactions among Modules

A system usually consists of several modules, each implementing some specific function. For example, the consensus protocol Raft may be divided into two modules: *log replication*, which describes how the nodes reach consensus as instructed by a leader and *leader election*, which specifies how a new leader is elected when the original leader fails. For the TLA+ specification of a distributed system design, we define:

Definition 1 (module): A module is a collection of actions. All the modules form a partition of all the actions in the specification.

Modules interact with each other through the system variables. To capture this, we first define the dependency variable of an action and that of a module:

Definition 2 (dependency variable): Suppose module $M = \{a_1, a_2, \cdots, a_m\}$, dependency variables of M, denoted as $\mathcal{D}_M$, is obtained recursively according to the following rules:

1) For any action $a_i \in M$, its dependency variables $\mathcal{D}_{a_i}$ are the variables which appear in some enabling condition ϕ_{a_i} of a_i.
2) $\bigcup_{1 \le i \le m} \mathcal{D}_{a_i} \subseteq \mathcal{D}_M$. That is, the dependency variables of each action in M belong to $\mathcal{D}_M$.
3) For any $v \in \mathcal{D}_M$ and any action $a_i \in M$, if the next-state update of a_i assigns to v a value calculated from multiple variables (denoted by variable set V_{dep}), then $V_{dep} \subseteq \mathcal{D}_M$. This is due to the fact that the dependency relation is transitive, i.e., if M depends on some variable v and v depends on another variable w, then M also depends on w.

Given the definitions above, we can now say that module M_i interacts with M_j by modifying $\mathcal{D}_{M_j}$.

The notion of the dependency variable alone is not sufficient to capture the interaction among modules, since even if D_{M_i}

are not modified by some action in M_j, M_i may still be affected indirectly. Suppose $x \in \mathcal{D}_{M_i}$, an action in another module M_j assigns to x the value of y (note that y will not be added to $\mathcal{D}_{M_i}$ by the Rule 3 in Definition 2, since x is assigned the value of y in module M_j, not in M_i). In this case, any assignment to y may also change the value of x in subsequent actions. To capture such indirect interactions between modules, we define the set of interaction variables $\mathcal{I}$:

Definition 3 (interaction variable): Suppose the specification contains k modules: $M_1, \cdots, M_k$. The set of interaction variables $\mathcal{I}$ is calculated recursively according to the follwing rules:

1) $\bigcup_{1 \le i < j \le k} (\mathcal{D}_{M_i} \cap \mathcal{D}_{M_j}) \subseteq \mathcal{I}$. That is, variables which are dependency variables of multiple modules belong to $\mathcal{I}$.
2) For any $v \in \mathcal{I}$ and any module M_i, if an action $a \in M_i$ assigns to v a value calculated from multiple variables (denoted by set V_{intr}), then add all variables in $V_{intr} \setminus \mathcal{D}_{M_i}$ to $\mathcal{I}$. That is, the value assigned to an interaction variable by any action in M_i should be calculated from values of variables in interaction variables or dependency variables of the module, i.e., $\mathcal{I} \cup \mathcal{D}_{M_i}$.
3) For any variable $v \in \mathcal{D}_{M_i} \setminus \mathcal{I}$ in any module M_i, if an action assigns to v a value calculated from multiple variables (denoted by set V'_{intr}), then add all variables in $V'_{intr} \setminus \mathcal{D}_{M_i}$ to $\mathcal{I}$. That is, the value assigned to a "internal" variable of M_i by any action should be calculated from values of interaction variables or from values of dependency variables of the module, i.e., $\mathcal{I} \cup \mathcal{D}_{M_i}$.

Note that in Rule 1 of this definition, we are a bit conservative. Some variable x in both $\mathcal{D}_{M_i}$ and $\mathcal{D}_{M_j}$ may not convey any interaction between M_i and M_j. However, in practice this case is rare (see details of our case study in Section IV and Appendix B and C) and we ignore this case to make our definition concise and easy to use.

Given the definition of the interaction variable, it is straightforward to verify that: for any two different modules M_i and M_j, $(\mathcal{D}_{M_i} \setminus \mathcal{I}) \cap \mathcal{D}_{M_j} = \emptyset$.

We define the internal variables of module M_i, denoted as $\mathcal{L}_{M_i}$, to be $\mathcal{D}_{M_i} \setminus \mathcal{I}$. Intuitively, if all variables but $\mathcal{L}_{M_i}$ stay unchanged in an action in M_i, then this action has no effect on other modules.

B. Interaction-Preserving Abstraction for Each Module

The main objective of our IPA framework is to enable separate model checking of each module, in order to reduce the cost for direct checking of the original specification. The critical challenge is to construct an execution context for each module, such that all the behaviors in the module can be checked separately.

To this end, we conduct interaction-preserving abstraction for each module. Suppose we have k modules $M_1, M_2, \cdots, M_k$. The abstraction of each module M_i is denoted by $\widetilde{M_i}$. When we check module M_i separately, the abstractions of all other modules, i.e. all $\widetilde{M_j}$ $(j \neq i)$, serve as the execution context of M_i

The key in the abstraction process is to omits internal details of every module as much as possible, and more importantly, the logic concerning interaction among modules must be preserved. We need to ensure that one module cannot distinguish whether it is interacting with the original specification of other modules or the abstracted specifications.

1) Formal Definition of Interaction-Preservation: We now present the formal definition of the interaction-preserving abstraction. The abstraction process obtaining each $\widetilde{M_i}$ may introduce new variables and actions. We can define the dependency variables of the abstracted module $\mathcal{D}_{\widetilde{M_i}}$ in the same way, according to Definition 2. The abstracted specification $\widetilde{M_i}$ should satisfy the following constraints:

1) As $\widetilde{M_i}$ is the abstraction of M_i, the dependency variables of $\widetilde{M_i}$ should not intersect with the local variables of other modules. Formally, $\mathcal{D}_{\widetilde{M_i}} \subseteq \mathcal{I} \cup \mathcal{D}_{M_i}$.
2) For updates of interaction variables in $\mathcal{I}$, the value assigned to any interaction variable by any action in $\widetilde{M_i}$ should be calculated from values of interaction variables or those of dependency variables of the module, not from values of internal variables of other modules. That is, for any variable $v \in \mathcal{I}$, the value assigned to v by any action of $\widetilde{M_i}$ is calculated from values of $\mathcal{D}_{\widetilde{M_i}} \cup \mathcal{I}$.
3) For updates of internal variables of each abstracted module, the value assigned to any internal variable of the module by any action should be calculated from values of interaction variables or those of dependency variables of that module, not from values of internal variables of other modules. That is, for any variable $v \in \mathcal{L}_{\widetilde{M_i}}$, the value assigned to v by any action is calculated from values of $\mathcal{I} \cup \mathcal{D}_{\widetilde{M_i}}$.
4) Abstraction of any module preserve all actions whose effect can be "perceived" by other modules. This requires that there is a mapping $f_i : M_i \to \widetilde{M_i}$, such that for any action $a \in M_i$ and any module $M_j (j \neq i)$, f and $f_i(a)$ modify the values of $\mathcal{D}_{\widetilde{M_i}} \cup \mathcal{D}_{M_j} \cup \mathcal{I}$ in the same way. Note that if action a only changes the values of $\mathcal{L}_{M_i}$ and leave all other variables unchanged, then $f_i(a)$ may be void. Specially, $f_i(a)$ preserves all assignment clauses to variables in $\mathcal{L}_{M_j}$ syntactically.

According to the constraints above, some internal variables as well as actions that only modifies these variables are omitted in the abstraction.

2) Three Layers of Specifications: Initially, we are given the original specification which is partitioned into modules: $S = \bigcup_{1 \le i \le k} M_i$. In order to define the compositional specification for each module, i.e., original specification for one module and abstracted specification for all other modules, we need to define the variables and actions of the compositional specification.

Define C_i to be the specification that combines M_i and every $\widetilde{M_j} (j \neq i)$, i.e. $C_i = (\bigcup_{j \neq i} \widetilde{M_j}) \cup M_i$. Let the system

variables of specification C_i be $V_{C_i} = \mathcal{I} \cup \mathcal{D}_{M_i} \cup (\bigcup_{j \neq i} \mathcal{D}_{\widetilde{M_j}})$. It is obvious that variables not in V_{C_i} are irrelevant to the execution of C_i because V_{C_i} contains all the dependency variables of modules in C_i and any assignment to variables of V_{C_i} is calculated from variables in V_{C_i}.

Define A to be the specification that combines all abstracted specifications for each module, i.e., $A = \bigcup_{1 \leq i \leq k} \widetilde{M_i}$. Variables $V_A = \mathcal{I} \cup \bigcup_{1 \leq i \leq k} \mathcal{D}_{\widetilde{M_i}}$ are all variables that are relevant to the execution of A.

3) Strong Refinement Relation between Specifications: By defining the compositional specifications and the abstract specification, we can now circumvent the direct checking (that S refines A) using the compositional checking (that every C_i refines A). The original definition of the refinement relation between two protocols only requires that there is a mapping between the traces of two protocols. Now in order to enable compositional checking in our IPA framework, we strengthen the definition of the refinement relation with additional requirement on the mapping between actions. Similar enhancement of the refinement relation is also used in the existing work [**?**]. First we present the formal definition of refinement between protocols:

Definition 4 (refinement): A refinement mapping from protocol B to A assigns to each variable v of A an expression $\bar{v}$, where $\bar{v}$ is defined in terms of variables of B. A refinement mapping defines for each state s of B a state s' of A in which the value of each variable v is mapped to the value of $\bar{v}$ in state s.

Protocol B refines A if and only if there is a refinement mapping from B to A such that for each valid trace of B: $s_1 \to s_2 \to \cdots \to s_t$, $s'_1 \to s'_2 \to \cdots \to s'_t$ is a valid trace of A.

The definition of refinement only requires a mapping from the state space of B to that of A. In order to get an abstract specification, it is common to omit some actions which are about low level details. In this case, there is an obvious correspondence between actions of specifications. That is, some actions are preserved in both the abstract and the detailed specifications, while some actions are directly omitted (mapping to a void action). Given a trace of the original detailed specification, we can use such correspondence between actions to construct a corresponding trace of the compositional specification, and then construct a corresponding trace of the abstract specification. This helps us prove the refinement from the original specification to the abstract specification. The detailed proof will be provided in Section III-C. Now we first define the strong refinement relation to capture the correspondence between actions:

Definition 5 (strong refinement): B strongly refines A, denoted by $B \Rightarrow A$, if and only if B refines A and there is a mapping $f(\cdot)$ from actions of B to those of A, such that for any valid trace of B : $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{t-1}} s_t$, $s'_1 \xrightarrow{f(a_1)} s'_2 \xrightarrow{f(a_2)} \cdots \xrightarrow{f(a_{t-1})} s'_t$ is a valid trace of A.
It is obvious to see that strong refinement is transitive.

Given the action mapping $f_i : M_i \to \widetilde{M_i}$ for each module M_i, we now establish strong refinement relations from S to $C_i (1 \leq i \leq k)$, C_i to A and S to A, as shown in Fig. 1. The strong refinement mapping has two parts: the mapping between variables, and the mapping between actions. According to the definitions of S, C_i and A, $V_A \subseteq V_{C_i} \subseteq V$ (here we assume that both variables in the original specification S and variables introduced in the abstract specifications $\widetilde{M_i}$ for $1 \leq i \leq k$ are in V), so refinement mappings concerning the variables are identity mappings.

Action mapping from S to C_i, denoted as g_i, is defined as follows. For any action $a \in S$, if $a \in M_i$, then $g_i(a) = a$. If $a \in M_j (j \neq i)$, then $g_i(a) = f_j(a)$. Action mapping from C_i to A, denoted as $\bar{g}_i$, is defined in a similar way: if $a \in M_i$, then $\bar{g}_i(a) = f_i(a)$ and if $a \in \widetilde{M_j} (j \neq i)$, then $\bar{g}_i(a) = a$. Action mapping from S to A, denoted as g, maps each action to its abstracted version: for any action $a \in M_i$, $g(a) = f_i(a)$. It is straightforward to see that for any i, g is the composite function of g_i and $\bar{g}_i$, i.e., $\bar{g}_i(g_i) = g$.

C. Correctness of Compositional Checking

We have presented the basic workflow using the IPA framework. The basic rationale behind the IPA framework is to use the compositional checking of each C_i to circumvent the direct checking of the original specification S. This circumvention is backed by the following theorem:

Theorem 1 (Correctness of compositional checking): $\forall\ 1 \leq i \leq k : C_i \Rightarrow A$ implies that $S \Rightarrow A$.

Proof sketch. Given the strong refinement mapping from each C_i to A, for each valid trace of S we construct a valid trace of A. Both states and actions in the trace of S are mapped to their counterparts in A, thus proving the strong refinement from S to A. There are four steps in construction, as is shown in Figure 2.

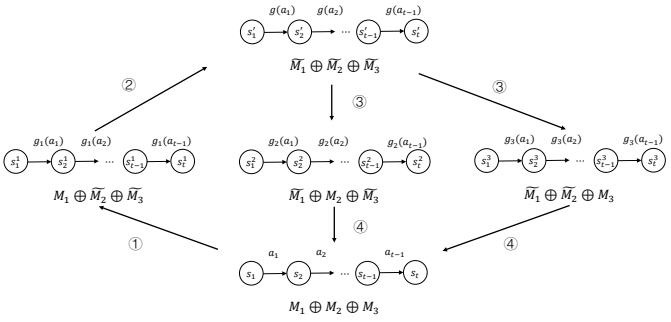

Fig. 2. Correctness of Compositional Checking

① Suppose S takes an action $a \in M_i$ in current state. Then C_i can also take a because all actions in M_i are preserved by C_i.
② Given that $C_i \Rightarrow A$, we have the mapping f of actions. Since M_i is abstracted in A, A can take the action $f(a)$.

Note that in the current step we are considering an action a from M_i. In the next step, S may take an action from any module other than M_i. Action a in the current step may affect

other modules which are taking an action in the next step. So we have to ensure state of any compositional specification $C_j (j \neq i)$ remains consistent. We therefore need step ③ and ④.

③ As module M_i is abstracted in C_j, C_j can take $f(a)$ just like A.

④ Comparing to A, module M_j is not abstracted in C_j and some internal variables of M_j may be modified by a. We prove that a and $f(a)$ modifies internal varianles of M_j in the same way, thus ensuring the states of S and C_j remain consistent.

We provide the detailed proof in Appendix A.

IV. Case Studies

In this section, we apply the IPA framework to reduce the model checking cost for the specifications of two consensus protocols: Raft and PRaft. Raft is a widely-used consensus protocol which is originally developed in the academia and then widely used in practice. PRaft is the replication protocol in PolarFS, the distributed file system for the commercial database Alibaba PoloarDB [**?**]. The design of PRaft is derived from Raft and Multi-Paxos [**?**], [**?**].

We first introduce the general pattern of interaction-preserving abstraction on realistic TLA+ specifications. Then we demonstrate how the IPA framework can be conveniently applied in practice. Finally, we show how much model checking cost can be saved through experiments.

Details on how each type of abstraction is conducted, including the TLA+ specifications before and after the abstraction, can be found in Appendix B and C. All the TLA+ specifications in the Raft case can be found in the anonymized GitHub repository[1]. Up till now, TLA+ specifications in the PRaft case cannot be open-sourced due to confidentiality reasons.

A. Patterns of Interaction-Preserving Abstraction

In Section III, we present the constraints the abstraction must conform to, in order to guarantee interaction-preservation. These constraints are necessary conditions and they do not tell the specification developer how to write the interaction-preserving abstractions in practice. In this section, we show via case studies that the interaction-preserving abstraction is quite intuitive. Moreover, useful patterns can greatly mitigate the burden of the developer.

1) The Polling Pattern: Consensus protocols usually involve some polling process in one way or another, in order to collect local information from distributed nodes/replicas and calculate certain global information. This type of polling process can generally be restricted within the scope of one function module. This means that other modules do not need to know the details of the polling process. They only care about the final result. For example, the leader election module often needs to poll multiple candidates to choose the most eligible one. However, when we model check other modules, we only need to know which node is the new leader.

[1]https://github.com/AnonymousAccountForReview/IPA

Thus, the details of the polling process can generally be abstracted away. In TLA+, since specification developers model a distributed system in terms of a single global state, the abstraction is quite straightforward. As shown in the illustrative example in Fig. 3, utilizing the global information in the specification, the specification developer can obtain required global information in one step, without the polling process. See more concrete examples of applying this "polling" pattern in Appendix B-A, B-B, B-C and C-A. This abstraction process is intuitively correct, and we can conveniently double check its correctness following the constraints in Section III.

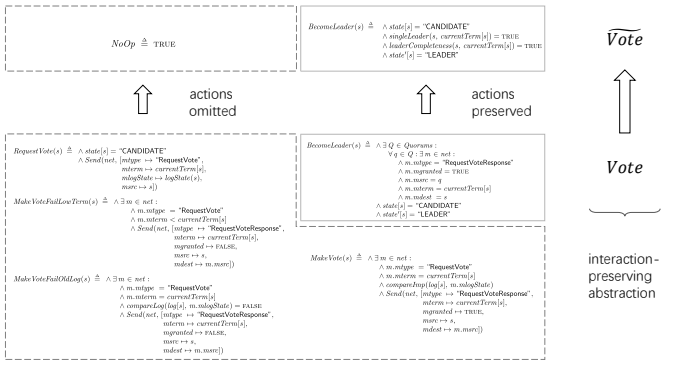

Fig. 3. Interaction-preserving abstraction of the Vote module.

2) Industry Implementation Patterns: In our case study, we intentionally choose the detailed specification of an industry-level consensus protocol PRaft. To improve maintainability, reduce implementation complexity and support dynamic upgrade, PRaft separates its control flow from the data flow by introducing a centralized coordinator. The coordinator is in charge of the control plane and the leader node and follower nodes passively receive commands from the coordinator. The coordinator regularly checks each server's state to see whether any error occurs. When errors such as follower reboot or network partition are detected, the coordinator starts the corresponding error handling process by sending servers commands they need to execute. Servers are passive followers and never make decisions on their own.

A typical control flow of PRaft is as follows: 1) the coordinator sends a command to a server; 2) the server executes the command received; 3) the coordinator sends a message to the server requesting its progress; 4) the server responds telling the coordinator whether it has finished execution; 5) when the server finishes its current job, the coordinator sends the next command. Similar communications between the coordinator and the followers exist in both module LeaderRecovery and module FollowerRecovery.

Such control flows are suitable for system implementations but add unnecessary complexities to model checking. The relative order between these control flow communication steps and other actions are undetermined. When performing model checking, the coordinator is often redundant because a system specification stands at a global point of view and the specification developer can specify that servers make decisions on their

own. Thus, step 1,3 and 5 can be omitted in abstraction. We apply this type of abstraction in multiple stages in the control flow of PRaft.

Besides the abstraction concerning the control flow, we also find that industry-level design often pays a lot of attention to performance optimization in realistic scenarios. From the perspective of interaction-preserving abstraction, the performance optimization protocol can often be replaced by a brute-force protocol.

For example, in PRaft, a centralized coordinator is introduced to instruct the operation of the leader node and the follower nodes. In the LeaderRecovery module, the coordinator has to calculate committed log entries from logs of a majority of servers. One way to implement this is that all nodes send their logs to the coordinator, taking one round of communication. But as the log is large, this may cause network congestion. To reduce the network load, PRaft uses two rounds of communications. In the first round, servers simply send the length of their logs to the coordinator, who selects a majority of servers whose log is more up-to-date. In the second round, only the selected server sends their logs to the coordinator. Thus network load is reduced using one more round of communication. This two-round log collection protocol is replaced by the brute-force one-round protocol in the abstraction.

B. Application of the IPA Framework

After presenting how the interaction-preserving abstraction is conducted in principle, we now discuss important details in applying the IPA framework.

1) Partitioning the TLA+ Specification into Modules: Basically, the Raft specification can be divided into two modules: the LogReplication module, which describes how the leader replicates log entries to the followers, and the Vote module, which describes how a new leader is elected when the former leader fails. In practice, the Raft protocol often includes the third module PreVote, which is used to prevent a disconnected follower from immoderately increasing the $term$ value.

The TLA+ specification for PRaft is developed to precisely document its design, find potential deep bugs, and improve the developer's confidence in its design and implementation. At the very beginning, the PRaft specification is divided into three modules: the Replication module, which describes how the leader replicates log entries to the followers, the LeaderRecovery module, which describes how a new leader is elected when the old leader fails, and the FollowerRecovery module, which describes how a lagged follower catches up. However, when we conduct interaction-preserving abstraction for each module, we find that the Replication module has little room for abstraction. It means that separating out this module will not reduce the overall compositional checking cost. Therefore, we merge the Replication module into the LeaderRecovery module.

2) Interaction-Preserving Abstraction in Practice: Given the partition of TLA+ specification into modules, the key is to identify the interaction variables $\mathcal{I}$, thus identifying the internal variables $\mathcal{L}$. In practical use of the IPA framework, the specification developer can easily classify the variables, since the developer is quite clear of the use of each variable when transforming the informal system design into TLA+ specifications. The high cohesion and low coupling of the modules also ease the burden of classifying the variables. Given the intuitive and tentative classification of the variables, the developer only needs to double check the classification according to the constraints defined in Section III-A.

The abstraction process basically follows intuitive patterns, as discussed in Section IV-A. Given that the developer has already transformed informal system design into detailed TLA+ specification, it is much easier to write the coarsened specification. During this process, the refinement mapping between the two levels of specifications is also intuitive. Note that, the IPA framework requires strong refinement, while in TLC, we can only check (the original) refinement mapping. Currently, the strong refinement mapping is manually checked and guaranteed by introducing auxiliary variables.

C. Experimental Evaluation

The main objective of the experimental evaluation is to explore how much model checking cost can be saved using our IPA framework. We model check the Raft and PRaft specifications, and compare the cost in time between direct checking and compositional checking. The model checking is conducted on one workstation with an Intel i9-9900X CPU (3.50GHz), with 10 cores and 20 threads, and 32GB RAM, running Ubuntu Desktop 16.04.6 LTS and TLC version 1.7.1.

We tune the scale of the system by tuning $term$ (the maximum number of phases the nodes can enter in the consensus process) and cmd (the number of commands the clients can send to the servers). The number of servers is set to 3. We record the checking time for each module and obtain the overall time for compositional checking. We also record the time for direct checking. The ratio of direct checking time to compositional checking time is calculated to illustrate the effect of compositional checking.

TABLE I
EXPERIMENT RESULTS

Raft $(term, cmd)$	T_{PreVote}	T_{Vote}	T_{Rep}	T_{comp}	T_{direct}	$\frac{T_{direct}}{T_{comp}}$
(1,1)	00:00:06	00:00:05	00:00:03	00:00:14	00:02:25	10.3
(1,2)	00:00:14	00:00:14	00:00:06	00:00:34	01:03:48	111.2
(1,3)	00:01:27	00:01:38	00:00:57	00:04:02	19:12:50	288.6
(2,1)	00:00:38	00:00:14	00:00:09	00:01:01	03:27:06	203.7
(2,2)	00:08:37	00:03:08	00:09:54	00:21:39	>100:00:00	>277.1
(2,3)	05:20:22	01:00:11	39:57:55	46:18:23	>200:00:00	>4.3

PRaft $(term, cmd)$	T_{RecL}	T_{RecF}	T_{comp}	T_{direct}	$\frac{T_{direct}}{T_{comp}}$
(1,1)	00:00:04	00:00:14	00:00:18	00:00:35	1.9
(1,2)	00:01:34	00:05:44	00:07:18	00:21:10	2.9
(1,3)	00:23:05	04:30:50	04:53:55	13:54:05	2.8
(2,1)	02:47:43	00:37:56	03:25:39	09:48:20	2.9
(2,2)	33:24:07	31:58:08	65:22:15	>200:00:00	>3.1

The experiment results are listed in Table I. T_{PreVote} denotes the compositional model checking of module PreVote and the checking time of other modules are named similarly. T_{comp} denotes the total compositional checking time for all modules

and T_{direct} denotes the time for direct checking of the original specification. In our analysis of the evaluation results, we mainly investigate the cost ratio, which is defined as $\frac{T_{direct}}{T_{comp}}$.

As for the Raft case, the cost ratio ranges from 10.3 to 288.6, showing that compositional checking based on the IPA framework can significantly reduce the model checking cost. Principally, the more complicated the model is, the larger the cost ratio. This is mainly because for complex modules, there will be more internal logic which can be abstracted away in the compositional checking. Note that in the case where $(term, cmd) = (2, 3)$, we stop the direct checking when the total checking time reaches 200 hours. So the result that $cost\ ratio > 4.3$ is a quite conservative estimation. It is reasonable to estimate that the actual cost ratio is much more than 4.3, probably also much more than 288.6.

As for the PRaft case, the cost ratio is around 3, relatively small compared to the ratio in the Raft case. It is mainly because, although the PRaft protocol is derived from Raft, it works much more like Multi-Paxos. Thus the abstractions in the Raft case are not applicable in the PRaft case. Moreover, in the PRaft case, we mainly abstract away the details of performance optimizations. Such details consist a smaller portion in the protocol design, compared to the Raft case. Although the cost ratio is smaller in the PRaft case, we argue that the IPA framework is practically effective in the PRaft case. It can save much time compared to the direct checking. Also note that, in the PRaft case, the TLA+ specifications are supplemented after the protocol design and implementation are principally finished, in order to precisely document the protocol design and find potential deep bugs in the implementation. Thus the abstraction process is intuitive and in some sense straightforward, for developers who are familiar with the PRaft design. This makes the application of the IPA framework highly worthwhile.

V. Related Work

Compositional model checking is essential to tackling the state explosion problem. It can be roughly classified as compositional minimization and compositional reasoning [**?**]. In compositional reasoning, verification of a system is broken into separate analyses for each component of the system. The result for the entire system is derived from the results of verifying individual components [**?**], [**?**], [**?**], [**?**]. In our approach, after the abstraction of each module is obtained, the following compositional checking is fully automatic.

The compositional reasoning imposes non-trivial burden on the developer, and it is not suitable for the intended users of our IPA framework.

Compositional minimization, in general, constructs the local model for each module in a system, minimizes it, and composes it with the minimized models of other modules to form a reduced global model for the entire system, on which verification is performed [**?**], [**?**]. Effectiveness of these methods depends on whether a coarse enough (to reduce the checking cost) yet accurate enough (to ensure the correctness of checking) context can be found for each component such that all the essential behavior of that component can be checked.

Existing compositional minimization techniques do not consider the characteristics of TLA+ specifications, and are thus not applicable or efficient in our target scenarios. Our IPA framework achieves compositional minimization based on the ladder of abstractions in TLA+ specifications.

The freedom to choose and adjust levels of abstraction is utilized to achieve the compositional minimization we need.

The interaction-preservation abstraction of this work is also inspired by the dynamic interface reduction technique in code-level model checking [**?**]. The dynamic interface reduction technique essentially identifies the interface interactions between running nodes of a distributed system and eliminates traces with the same interface behaviors so that the state space to be checked is reduced. Our compositional minimization is orthogonal to the reduction of model checking state space, but we borrow the basic idea of interface reduction.

VI. Conclusion and Future Work

In this work we present the IPA compositional model checking framework for TLA+ specifications of consensus protocols. We provide formal definition and correctness proof of our IPA framework. We also apply the IPA framework in model checking of two consensus protocols Raft and PRaft. The case study shows that the IPA framework is easy to use in practical model checking of realistic TLA+ specifications. It also shows that the compositional model checking based on IPA can significantly reduce the checking cost.

In our future work, we will apply the IPA framework to more scenarios, involving complex and subtle distributed protocols other than distributed consensus. We will also investigate whether the IPA framework can be used to reduce the cost of code-level model checking of distributed system implementations. Given sufficient application of the IPA framework in realistic scenarios, we will investigate how to integrate the IPA framework into the extreme modeling [**?**] paradigm of distributed system design and implementation.

References

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

APPENDIX A
CORRECTNESS PROOF OF THE COMPOSITIONAL CHECKING

We here present the detailed proof of Theorem 1 in Section III-C.

Proof. We prove by mathematical deduction that for all valid traces of S and $1 \leq i \leq k$, the traces deduced using mapping from actions of S to that of C_i are valid traces of C_i and that variables in V_{C_i} have the same values in corresponding states. First we assume in all specifications, the same variables are assigned the same values in the initial states. Suppose the proposition holds for all traces whose length are smaller than t.

Let $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots s_{t-1} \xrightarrow{a_{t-1}} s_t$ be any valid trace of S whose length is t. By induction hypothesis, we know that for any $1 \leq i \leq k$, $s_1^i \xrightarrow{g_i(a_1)} s_2^i \xrightarrow{g_i(a_2)} \cdots \xrightarrow{g_{i(a_{t-2})}} s_{t-1}^i$ is a valid trace of C_i and variables in V_{C_i} are assigned the same values in s_l and $s_l^i (1 \leq l \leq t-1)$. Because $C_i \Rightarrow A$, accroding to the definition of g and $\bar{g}_i$, we get that $s'_1 \xrightarrow{g(a_1)} s'_2 \xrightarrow{g(a_2)} \cdots \xrightarrow{g(a_{t-2})} s'_{t-1}$ is a valid trace of A and also variables in V_A are assigned the same values in s'_l as in $s_l (1 \leq l \leq t-1)$.

Suppose S executes an action $a_{t-1} \in M_i$ in state s_{t-1} and reach state s_t. As variables in V_{C_i} are assigned the same values in s_{t-1} as in s_{t-1}^i and $D_i \subseteq V_{C_i}$, every enabling condition of a_{t-1} must be satisfied in state s_{t-1}^i. Therefore, S_i can also execute a_{t-1} in state s_{t-1}^i to reach state s_t^i with variales in V_{C_i} still are assigned the same values in s_t as in s_t^i.

Since $C_i \Rightarrow A$, A can execute $g(a_{t-1})$ in state s'_{t-1} to reach state s'_t with variables in V_A are assigned the same values in s_t as in s'_t.

For any specificaion $S_j (j \neq i)$, as variables in V_A have the same values in s_{t-1}^j as in s'_{t-1}, S_j can also execute $g_j(a_{t-1}) = g(a_{t-1})$ to reach state s_t^j with V_A still assigned the same values in s'_t and s_t as in s_t^j.

For the internal variables $\mathcal{L}_{M_j} = \mathcal{D}_{M_j} \setminus \mathcal{I}$ of M_j, according to the definition of f_i, a_{t-1} has exactly the same assignment clauses to $\mathcal{L}_{M_j}$ as $g_j(a_{t-1})$ and variables in V_{C_j} have the same values in s_{t-1} as in s_{t-1}^j, so variables in $\mathcal{L}_{M_j}$ are assigned the same values by a_{t-1} and $g_j(a_{t-1})$. Thus, the values of variables in V_{C_j} are the same in s_t^j and s_t.

As the trace is chosen arbitrarily, by mathematical induction, we get that for any $1 \leq i \leq k$, $S \Rightarrow C_i (1 \leq i \leq k)$. As $C_i \Rightarrow A$, we get $S \Rightarrow A$.

APPENDIX B
CASE STUDY ON RAFT

We divide the specification of the Raft protocol into three modules: module PreVote describing the pre-vote mechanism, module Vote describing the election mechanism and module Replication describing the transmission of log entries from the leader to the followers. For each module, we mainly discuss how the interaction-preserving abstraction is conducted, as detailed below.

A. Abstraction for Module Replication

In Raft, log entries are replicated from leader to follower in sequence. A follower accepts a log entry from leader only if it has already accepted all previous log entries. If it receives a log entry when preceding entries are not fully accepted, it rejects the entry. When an entry is rejected, leader tries to send the previous one in its log. As leader does not know exactly which entry the follower would accept in an asynchronous distributed system, the process of sending and rejecting may take multiple rounds before the follower can accept its missing entry, in which the state variables of the leader and follower remain unchanged. Any newly elected leader has to find each follower's first unmatched log entry by such process. Therefore, a trace containing multiple elections can be very long due to such "invalid" communications between leader

$$
\begin{aligned}
&AppendEntries(s1,\, s2) \triangleq \text{LET } ind \triangleq nextIndex[s1][s2] \\
&\quad \text{IN} \\
&\quad\quad \wedge\, state[s1] = \text{"LEADER"} \\
&\quad\quad \wedge\, s1 \neq s2 \\
&\quad\quad \wedge\, Send(net, \\
&\quad\quad\quad\quad [mtype \mapsto \text{"AppendEntries"}, \\
&\quad\quad\quad\quad\ mterm \mapsto currentTerm[s1], \\
&\quad\quad\quad\quad\ mprevTerm \mapsto log[s1][ind-1].term, \\
&\quad\quad\quad\quad\ mprevIndex \mapsto ind-1, \\
&\quad\quad\quad\quad\ mentry \mapsto e, \\
&\quad\quad\quad\quad\ msrc \mapsto s1, \\
&\quad\quad\quad\quad\ mdest \mapsto s2])
\end{aligned}
$$

$$
\begin{aligned}
&ReplicateFailUnmatch(s1,\, s2) \triangleq \exists\, m \in net : \\
&\quad\quad \wedge\, m.mtype = \text{"AppendEntries"} \\
&\quad\quad \wedge\, m.msrc = s1 \\
&\quad\quad \wedge\, m.mdest = s2 \\
&\quad\quad \wedge\, \vee\, Len(log[s2]) < m.mprevIndex \\
&\quad\quad\quad\ \vee\, \wedge\, Len(log[s2]) \geq m.mprevIndex \\
&\quad\quad\quad\quad\ \wedge\, log[s2][m.mprevIndex].term \neq m.mprevTerm \\
&\quad\quad \wedge\, Send(net, \\
&\quad\quad\quad\quad [mtype \mapsto \text{"AppendEntriesResponse"}, \\
&\quad\quad\quad\quad\ mterm \mapsto currentTerm'[s2], \\
&\quad\quad\quad\quad\ msuccess \mapsto \text{FALSE}, \\
&\quad\quad\quad\quad\ mindex \mapsto m.mprevIndex+1, \\
&\quad\quad\quad\quad\ msrc \mapsto s2, \\
&\quad\quad\quad\quad\ mdest \mapsto m.msrc])
\end{aligned}
$$

$$
\begin{aligned}
&Replicate(s1,\, s2) \triangleq \exists\, m \in net : \\
&\quad \wedge\, m.mtype = \text{"AppendEntries"} \\
&\quad \wedge\, m.mdest = s2 \\
&\quad \wedge\, m.msrc = s1 \\
&\quad \wedge\, \vee\, m.mprevIndex = 0 \\
&\quad\quad \vee\, \wedge\, m.mprevIndex \neq 0 \\
&\quad\quad\quad \wedge\, Len(log[s2]) \quad \geq m.mprevIndex \\
&\quad\quad\quad \wedge\, log[s2][m.mprevIndex].term = m.mprevTerm \\
&\quad \wedge\, log'[s2] = Append(SubSeq(log[s2],\, 1,\, m.mprevIndex),\, m.mentry) \\
&\quad \wedge\, Send(net, \\
&\quad\quad\quad [mtype \mapsto \text{"AppendEntriesResponse"}, \\
&\quad\quad\quad\ mterm \mapsto currentTerm'[s2], \\
&\quad\quad\quad\ msuccess \mapsto \text{TRUE}, \\
&\quad\quad\quad\ mindex \mapsto m.mprevIndex+1, \\
&\quad\quad\quad\ msrc \mapsto s2, \\
&\quad\quad\quad\ mdest \mapsto m.msrc])
\end{aligned}
$$

Fig. 4. Specification for Replication

$$
\begin{aligned}
&HandleAppendEntriesResponseFail(s) \triangleq \exists\, m \in net : \\
&\quad\quad \wedge\, m.mtype = \text{"AppendEntriesResponse"} \\
&\quad\quad \wedge\, m.mdest = s \\
&\quad\quad \wedge\, m.mterm = currentTerm[s] \\
&\quad\quad \wedge\, state[s] = \text{"LEADER"} \\
&\quad\quad \wedge\, m.msuccess = \text{FALSE} \\
&\quad\quad \wedge\, nextIndex'[s][m.msrc] = max(1,\, nextIndex[s][m.msrc]-1)
\end{aligned}
$$

$$
\begin{aligned}
&HandleAppendEntriesResponseSuccess(s) \triangleq \exists\, m \in net : \\
&\quad\quad \wedge\, m.mtype = \text{"AppendEntriesResponse"} \\
&\quad\quad \wedge\, m.mdest = s \\
&\quad\quad \wedge\, state[s] = \text{"LEADER"} \\
&\quad\quad \wedge\, m.mterm = currentTerm[s] \\
&\quad\quad \wedge\, m.msuccess = \text{TRUE} \\
&\quad\quad \wedge\, m.mindex > matchIndex[s][m.msrc] \\
&\quad\quad \wedge\, nextIndex'[s][m.msrc] = m.mindex+1 \\
&\quad\quad \wedge\, matchIndex'[s][m.msrc] = m.mindex
\end{aligned}
$$

Fig. 5. Specification for Replication (continued)

and followers. Also many system states are generated due to the uncertain order between these invalid actions and other actions.

Figure 4 lists three actions of module Replication. Action $AppenEntries$ specifies the process when leader s_1 sends an $AppendEntries$ request to some follower s_2 to replicate log entries within the cluster. This action only modifies internal variable net which records all messages sent by servers. When a follower receives a log entry from leader, it performs the prefix check to ensure that it has already received all previous log entries. If prefix check fails, the follower simply responds to leader with the index of unmatched entry without modifying any other variable, as is specified by action $ReplicateFailUnmatch$. If a follower receives from leader exactly the log entry it misses, it adds the entry to its log and sends back an ack, as is specified by $Replicate$. Figure 5 lists actions specifying how leader handles responses from followers. When leader receives a response from a follower indicating that the entry is accepted, it records the match index and the index of next log entry to send to the follower, as is specified by $HandleAppendEntriesResponseSuccess$. Note that $matchIndex$ and $nextIndex$ are specific to replication mechanism implementation and thus internal variables which can be omitted in the abstracted specification. When a follower rejects the log entry from leader, leader learns that this log entry is not the first one the follower misses. So it reduces $nextIndex$ for the follower by 1 and tries to replicate the last log entry. Action $HandleAppendEntriesResponseFail$ specifies this process.

$$
\begin{array}{l}
Replicate(s1,\, s2) \;\triangleq\; \text{LET } t \;\triangleq\; currentTerm[s2] \text{ IN} \\
\qquad \land\, leader[t] = s1 \\
\qquad \land\, s1 \neq s2 \\
\qquad \land\, \exists j \in 1 \,..\, Len(leaderLog[t]) : \\
\qquad\quad \land\, consistent(leaderLog[t],\, log[s2],\, j-1) \\
\qquad\quad \land\, \lor\, Len(log[s2]) < j \\
\qquad\qquad \lor\, \land\, Len(log[s2]) \geq j \\
\qquad\qquad\quad \land\, leaderLog[t][j].term \neq log[s2][j].term \\
\qquad \land\, log'[s2] = Append(SubSeq(log[s2],\, 1,\, j-1),\, en)
\end{array}
$$

Fig. 6. Abstracted Specification for Replication

To conduct abstraction, a leader simply sends the exact log entry that followers miss without the process of trial and error because TLA+ allows users to model a distributed system in terms of a single global state and a leader can utilize global state of each server. Thus, the redundant steps of sending and receiving "invalid" messages in system traces are eliminated. Figure 6 shows the single action of the abstracted specification for this process. This action corresponds to action $Replicate$ in Figure 4. All other actions and internal variables such as $nextIndex$ and $matchIndex$ are reduced by abstraction.

B. Abstraction for Module PreVote

Raft relies on a leader election algorithm to elect a single leader for each term. If follower does not receive heart beat messages from leader for some time, it becomes candidate and starts an election by increasing its term and sending election messages concurrently to other system servers. Since network may be unreliable, a server partitioned from leader cannot receive messages from leader. Therefore, it tries to start election for multiple times and increases its term to a large value. When network condition becomes normal, its large term would be propagated within the cluster, forcing the leader to step down and the cluster has to elect a new leader unnecessarily.

To prevent such occasional network fluctuation from causing disruptions, Raft introduces the pre-vote mechanism. When a follower tries to start an election, it has to send pre-vote requests to other servers. Servers grant or refuse pre-vote requests based on their system states. Only if the server learns from a majority of the cluster that they would grant its pre-vote request can it increases its term and make election proposals. Pre-vote mechanism solves the issue of partitioned server disrupting the cluster when it rejoins since a partitioned server cannot increase its term unless a majority of the cluster agree to elect a new leader.

$$
\begin{array}{l}
PreVote(s) \;\triangleq\; \land\, state[s] = \text{"FOLLOWER"} \\
\qquad\qquad \land\, state'[s] = \text{"PRE\_CANDIDATE"} \\
\qquad\qquad \land\, Send(net,\, [mtype \;\mapsto\; \text{"PreVoteRequest"}, \\
\qquad\qquad\qquad\qquad mterm \mapsto currentTerm[s], \\
\qquad\qquad\qquad\qquad msrc \mapsto s, \\
\qquad\qquad\qquad\qquad mlog \mapsto log[s]]) \\
\\
HandlePreVote(s) \;\triangleq\; \land\, \exists\, m \in net : \\
\qquad\qquad\qquad \land\, m.mtype \;=\; \text{"PreVoteRequest"} \\
\qquad\qquad\qquad \land\, m.mterm = currentTerm[s] \\
\qquad\qquad\qquad \land\, \text{LET } granted \;\triangleq\; CompareLog(log[s],\, m.mlog) \\
\qquad\qquad\qquad\quad \text{IN} \\
\qquad\qquad\qquad\qquad Send(net,\, [mtype \;\mapsto\; \text{"PreVoteResponse"}, \\
\qquad\qquad\qquad\qquad\qquad mterm \mapsto currentTerm[s], \\
\qquad\qquad\qquad\qquad\qquad msrc \mapsto s, \\
\qquad\qquad\qquad\qquad\qquad mdest \mapsto m.msrc, \\
\qquad\qquad\qquad\qquad\qquad mgranted \mapsto granted]) \\
\\
BecomeCandidate(s) \;\triangleq\; \land\, state[s] = \text{"PRE\_CANDIDATE"} \\
\qquad\qquad\qquad \land\, \exists\, Q \in Quorums : \\
\qquad\qquad\qquad\quad \forall\, q \in Q : \exists\, m \in net : \\
\qquad\qquad\qquad\qquad\quad \land\, m.mtype \;=\; \text{"PreVoteResponse"} \\
\qquad\qquad\qquad\qquad\quad \land\, m.mterm = currentTerm[s] \\
\qquad\qquad\qquad\qquad\quad \land\, m.mdest \;= s \\
\qquad\qquad\qquad\qquad\quad \land\, m.msrc = q \\
\qquad\qquad\qquad\qquad\quad \land\, m.mgranted = \text{TRUE} \\
\qquad\qquad\qquad \land\, state'[s] = \text{"CANDIDATE"} \\
\qquad\qquad\qquad \land\, currentTerm'[s] = currentTerm[s] + 1
\end{array}
$$

Fig. 7. Specification for PreVote

$$
\begin{array}{l}
AbsPreVote \;\triangleq\; \land\, state[s] = \text{"FOLLOWER"} \\
\qquad\qquad\quad \land\, preVoteSet'[s] = \{a \in Server : canVote(s,\, a) = \text{TRUE}\} \\
\qquad\qquad\quad \land\, state'[s] = \text{"PRE\_CANDIDATE"} \\
\\
AbsBecomeCandidate(s) \;\triangleq\; \land\, state[s] = \text{"PRE\_CANDIDATE"} \\
\qquad\qquad\qquad\quad \land\, preVoteSet[s] \in Quorum \\
\qquad\qquad\qquad\quad \land\, state' = [state \text{ EXCEPT } ![s] = \text{"CANDIDATE"}] \\
\qquad\qquad\qquad\quad \land\, currentTerm'[s] = currentTerm[s] + 1
\end{array}
$$

Fig. 8. Abstracted Specification for PreVote

Figure 7 shows the three actions of module PreVote. The behavior that a follower starts an election by changing its

state to $PreCandidate$ and sending election requests to other servers for network reasons is modeled by action $PreVote$. Servers handle received election requests by executing action $HandlePreVote$. If the sender receives granted responses from a majority of servers, it changes state to $Candidate$ and updates its term, as is specified by action $BecomeCandidate$.

Note that according to action $HandlePreVote$, servers handle pre-vote requests by sending response messages without modifying their system states, which means that the action is transparent to other modules. We can thus omit this action in the abstraction of module PreVote. Figure 8 is the abstracted version containing 2 actions, each corresponding to $PreVote$ and $BecomeCandidate$ respectively. In action $AbsPreVote$, except for changing follower's state, a history variable $preVoteSet$ is used to record all possible servers that may grant the follower's pre-vote request. The follower trying to start an election can change state to $Candidate$ only if its $preVoteSet$ contains a quorum of servers, as is specified by action $AbsPreVote$. In this way, the action of handling election request is omitted while preserving the functionality of pre-vote mechanism.

C. Abstraction for Module Vote

Raft's leader election algorithm can also be abstracted similarly. To become a leader, a candidate sends election request currently to all servers requesting for votes. A server may grant or refuse a vote request according to its state and the information the message contains. Only when the candidate's vote is approved by a majority of servers can it become leader. Thus, a round of election takes multiple steps in a behavior trace as servers work asynchronously and each server's handling of the vote request takes one step. More over, as network is unreliable and messages can be delayed arbitrarily, the order each server handles the election request is undetermined, adding much more system states to be checked.

A natural abstraction for election is to choose a server and change its role to leader, taking only one step and avoiding possible permutations due to asynchrony. However, the election algorithm is delicately designed to ensure that every newly elected leader meets several essential properties such as single leader and leader completeness, which are critical to the correctness of Raft. Therefore, we figure out and specify these properties in our specification. With this, we can specify that an eligible server change its state to leader as abstraction for election, omitting details while perfectly matching original design.

Figure 9 shows the four actions of module Election. Action $RequestVote$ specifies a candidate sends election requests to all other servers when starting a new election. This action only changes variable net, which records all messages sent by servers. net is an internal variable, so action $RequestVote$ can be omitted by abstraction. Servers may grant or refuse election requests by comparing candidate's $term$ and log with their own. Action $MakeVoteFailLowTerm$ specifies the case when a server refuses an election request because the candidate's $term$ is smaller. Action $MakeVoteFailOldLog$

$$
\begin{aligned}
&RequestVote(s) \triangleq \ \wedge\, state[s] = \text{“CANDIDATE”} \\
&\qquad\qquad \wedge\, Send(net, [mtype \mapsto \text{“RequestVote”}, \\
&\qquad\qquad\qquad\qquad mterm \mapsto currentTerm[s], \\
&\qquad\qquad\qquad\qquad mlogState \mapsto logState(s), \\
&\qquad\qquad\qquad\qquad msrc \mapsto s]) \\
&MakeVoteFailLowTerm(s) \triangleq \ \wedge\, \exists\, m \in net : \\
&\qquad\qquad\qquad\qquad \wedge\, m.mtype = \text{“RequestVote”} \\
&\qquad\qquad\qquad\qquad \wedge\, m.mterm < currentTerm[s] \\
&\qquad\qquad\qquad\qquad \wedge\, Send(net, [mtype \mapsto \text{“RequestVoteResponse”}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad mterm \mapsto currentTerm[s], \\
&\qquad\qquad\qquad\qquad\qquad\qquad mgranted \mapsto \text{FALSE}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad msrc \mapsto s, \\
&\qquad\qquad\qquad\qquad\qquad\qquad mdest \mapsto m.msrc]) \\
&MakeVoteFailOldLog(s) \triangleq \ \wedge\, \exists\, m \in net : \\
&\qquad\qquad\qquad\qquad \wedge\, m.mtype = \text{“RequestVote”} \\
&\qquad\qquad\qquad\qquad \wedge\, m.mterm = currentTerm[s] \\
&\qquad\qquad\qquad\qquad \wedge\, compareLog(log[s], m.mlogState) = \text{FALSE} \\
&\qquad\qquad\qquad\qquad \wedge\, Send(net, [mtype \mapsto \text{“RequestVoteResponse”}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad mterm \mapsto currentTerm[s], \\
&\qquad\qquad\qquad\qquad\qquad\qquad mgranted \mapsto \text{FALSE}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad msrc \mapsto s, \\
&\qquad\qquad\qquad\qquad\qquad\qquad mdest \mapsto m.msrc]) \\
&MakeVote(s) \triangleq \ \wedge\, \exists\, m \in net : \\
&\qquad\qquad\qquad \wedge\, m.mtype = \text{“RequestVote”} \\
&\qquad\qquad\qquad \wedge\, m.mterm = currentTerm[s] \\
&\qquad\qquad\qquad \wedge\, compareImp(log[s], m.mlogState) \\
&\qquad\qquad\qquad \wedge\, Send(net, [mtype \mapsto \text{“RequestVoteResponse”}, \\
&\qquad\qquad\qquad\qquad\qquad mterm \mapsto currentTerm[s], \\
&\qquad\qquad\qquad\qquad\qquad mgranted \mapsto \text{TRUE}, \\
&\qquad\qquad\qquad\qquad\qquad msrc \mapsto s, \\
&\qquad\qquad\qquad\qquad\qquad mdest \mapsto m.msrc]) \\
&BecomeLeader(s) \triangleq \ \wedge\, \exists\, Q \in Quorums : \\
&\qquad\qquad\qquad\qquad \forall\, q \in Q : \exists\, m \in net : \\
&\qquad\qquad\qquad\qquad\qquad \wedge\, m.mtype = \text{“RequestVoteResponse”} \\
&\qquad\qquad\qquad\qquad\qquad \wedge\, m.mgranted = \text{TRUE} \\
&\qquad\qquad\qquad\qquad\qquad \wedge\, m.msrc = q \\
&\qquad\qquad\qquad\qquad\qquad \wedge\, m.mterm = currentTerm[s] \\
&\qquad\qquad\qquad\qquad\qquad \wedge\, m.mdest = s \\
&\qquad\qquad\qquad \wedge\, state[s] = \text{“CANDIDATE”} \\
&\qquad\qquad\qquad \wedge\, state'[s] = \text{“LEADER”}
\end{aligned}
$$

Fig. 9. Specification for Vote

specifies the case when the candidate's election request is refused because of outdated log. Both these two actions change no variables except for net and thus can also be omitted. If the candidate's $term$ and log are no older than the follower who received candidate's election reuqest, it grants the election request by sending an ack, as is specified by action $MakeVote$. This action also only modifies variable net. If a candidate receives ack from a quorum of servers, it changes its state to $LEADER$, as action $BecomeLeader$ specifies.

$$
\begin{aligned}
&BecomeLeader(s) \triangleq \ \wedge\, state[s] = \text{“CANDIDATE”} \\
&\qquad\qquad\qquad\quad \wedge\, singleLeader(s, currentTerm[s]) = \text{TRUE} \\
&\qquad\qquad\qquad\quad \wedge\, leaderCompleteness(s, currentTerm[s]) = \text{TRUE} \\
&\qquad\qquad\qquad\quad \wedge\, state'[s] = \text{“LEADER”}
\end{aligned}
$$

Fig. 10. Abstracted Specification for Vote

Figure 10 shows the abstracted specification of leader election. It contains only one action, $BecomeLeader$, which is enabled only if the two essential properties $singleLeader$ and $leaderCompleteness$, are true. Thus, any successful leader election in the abstracted specification guarantee these two properties. By such abstraction, a round of election takes only 2 steps no matter how many servers are in the cluster, greatly reducing the complexity of election algorithm, especially when the number of servers is big.

APPENDIX C
CASE STUDY ON PARALLELRAFT

PRaft specification is divided into two modules: module LeaderRecovery and module FollowerRecovery. We conduct interaction-preserving abstraction on PRaft mainly from three perspectives, as detailed below.

A. Asynchrony Elimination

The centralized coordinator learns the states of servers through polling. Nodes respond to polling messages by sending replies with their system states. In implementation level specification, such polling process takes multiple steps to finish as there are several nodes in the cluster and nodes handle coordinator message asynchronously, each node responding to the polling message takes one step. When performing leader election, coordinator has to learn the checkpoint of each server. We found that the checkpoint of each server stays unchanged during polling as no valid leader exists and client commands cannot be replicated among the cluster. This suggests that the polling process is transparent to other modules. Therefore, in abstracted specification, the coordinator learns the states of all nodes synchronously in abstraction, which is safe and takes only one step. By such abstraction, traces with different permutations of polling message handling actions are all mapped to a same trace of abstracted specification.

$$
\begin{aligned}
&GetLogLen \triangleq \ \wedge\, ctrlState = \text{“REC\_LEADER”} \\
&\qquad\qquad \wedge\, Send(net, [mtype \mapsto \text{“LogLenRequest”}]
\end{aligned}
$$

$$
\begin{aligned}
&HandleGetLogLen(s) \triangleq \ \wedge\, \exists\, m \in net : \\
&\qquad\qquad\qquad \wedge\, mmtype = \text{“LogLenRequest”} \\
&\qquad\qquad\qquad \wedge\, Send(net, [mtype \mapsto \text{“LogLenResponse”}, \\
&\qquad\qquad\qquad\qquad\qquad msrc \mapsto s, \\
&\qquad\qquad\qquad\qquad\qquad mlogLen \mapsto len(log[s])]
\end{aligned}
$$

$$
\begin{aligned}
&ProcessServerLog \triangleq \ \wedge\, ctrlState = \text{“REC\_LEADER”} \\
&\qquad\qquad\qquad \wedge\, \exists\, Q \in Quorums : \\
&\qquad\qquad\qquad\quad \wedge\, \forall\, s \in Q : \\
&\qquad\qquad\qquad\qquad \exists\, m \in net : \wedge\, m.mtype = \text{“LogLenResponse”} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \wedge\, m.msrc = s \\
&\qquad\qquad\qquad\quad \wedge\, DoSelection(Q)
\end{aligned}
$$

Fig. 11. Specification for Selection

Figure 11 shows the three actions of polling process. When performing leader recovery, coordinator sends to servers requesting checkpoint of each server by action $GetLogLen$. Server handle this request by action $HandleGeyLogLen$. When coordinator receives responses from a majority of servers, it does local calculation using replies, as is specified by Action $ProcessServerLog$. We found that when state of coordinator is REC_LEADER, checkpoints remain unchanged, which suggests that the order of servers handling “LogLenRequest” is irrelevant.

Figure 12 shows abstracted specification. It has a single action in which coordinator directly do local calculation using globally available checkpoints of servers. This is a typical example showing how we do abstraction for one module by eliminating asynchronous behaviors that are transparent to other modules.

$$
\begin{aligned}
&DirectProcessServerLog \triangleq \ \wedge\, ctrlState = \text{“REC\_LEADER”} \\
&\qquad\qquad\qquad\qquad \wedge\, \exists\, Q \in Quorum : \\
&\qquad\qquad\qquad\qquad\quad DoSelection(Q)
\end{aligned}
$$

Fig. 12. Abstracted Specification for Selection

B. Control Flow Simplification

$$
\begin{aligned}
&ProgressRequest(s) \triangleq \ \wedge\, ctrlState = \text{“WAIT\_TASK1”} \\
&\qquad\qquad\qquad\qquad \wedge\, Send([net, [mtype \mapsto \text{“ProgressRequest”}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad mdest \mapsto s]
\end{aligned}
$$

$$
\begin{aligned}
&ProgressResponse(s) \triangleq \ \wedge\, \exists\, m \in net : \\
&\qquad\qquad\qquad\qquad\quad \wedge\, m.mtype = \text{“ProgressRequest”} \\
&\qquad\qquad\qquad\qquad\quad \wedge\, m.mdest = s \\
&\qquad\qquad\qquad\qquad\quad \wedge\, Send([net, mtype \mapsto \text{“ProgressResponse”}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad m.msrc \mapsto s, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad mfinished \mapsto Task1Finished(s)]
\end{aligned}
$$

$$
\begin{aligned}
&Task1Finished \triangleq \ \wedge\, ctrlState = \text{“WAIT\_TASK1”} \\
&\qquad\qquad\qquad\quad \wedge\, \exists\, m \in net : \\
&\qquad\qquad\qquad\qquad \wedge\, m.mtype = \text{“ProgressResponse”} \\
&\qquad\qquad\qquad\qquad \wedge\, m.mfinished = \text{TRUE} \\
&\qquad\qquad\qquad\qquad \wedge\, Send(net, [mtype \mapsto \text{“DoTask2”}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad mdest \mapsto m.msrc])
\end{aligned}
$$

$$
\begin{aligned}
&BeginTask2(s) \triangleq \ \wedge\, \exists\, m \in net : \\
&\qquad\qquad\qquad\quad \wedge\, m.mtype = \text{“DoTask2”} \\
&\qquad\qquad\qquad\quad \wedge\, m.mdest = s \\
&\qquad\qquad\qquad\quad \wedge\, DoTask2(s)
\end{aligned}
$$

Fig. 13. Specification for a Typical Control Flow

Figure 13 shows a typical control flow of PRaft. Coorinator periodically checks whether a server has finished some task it assigns as is specified by action $ProgressRequest$. When the coordinator learns from the server that it has finished, coordinator sends a message requesting the server to begin doing subsequent task. Action $Task1Finished$ specifies this process. When receiving request from coordinator, server executes command as ordered.

$$
\begin{aligned}
&AbsBeginTask2(s) \triangleq \ \wedge\, Task1Finished(s) = \text{TRUE} \\
&\qquad\qquad\qquad\quad \wedge\, Task2Finished(s) = \text{FALSE} \\
&\qquad\qquad\qquad\quad \wedge\, DoTask2(s)
\end{aligned}
$$

Fig. 14. Abstracted Specification for Control Flow

Figure 14 shows the abstracted specification for this process. It omit the first three actions. When a server finishes one task, it starts doing subsequent task autonomously, as if it received an order from the coordinator. Also coordinator knows the progress of each server using global information. Therefore, the abstracted specification allows the same system behaviors as the implementation-level specification.

C. Omit Unnecessary Implementation Optimizations

Figure 15 shows the specification of coordinator collecting logs from a majority of servers using two rounds of com-

$$
\begin{array}{l}
\mathit{RequestLogFreshness} \triangleq \ \land\, \mathit{ctrlState} = \text{"REC\_LEADER"} \\
\qquad\qquad\qquad\qquad\quad \land\, \mathit{Send}(\mathit{net}, [\mathit{mtype} \mapsto \text{"LogFreshnessRequest"}])
\end{array}
$$

$$
\begin{array}{l}
\mathit{RequestLogResponse}(s) \triangleq \ \land\, \exists\, m \in \mathit{net} : \\
\qquad \land\, m.\mathit{mtype} = \text{"LogFreshnessRequest"} \\
\qquad \land\, \mathit{Send}(\mathit{net}, [\mathit{mtype} \mapsto \text{"LogFreshnessResponse"}, \\
\qquad\qquad \mathit{msrc} \mapsto s, \\
\qquad\qquad \mathit{mlogLen} \mapsto \mathit{Len}(\mathit{log}[s])])
\end{array}
$$

$$
\begin{array}{l}
\mathit{RequestLog} \triangleq \ \land\, \exists\, Q \in \mathit{Quorums} : \\
\qquad \land\, \forall\, q \in Q : \\
\qquad\qquad \exists\, m \in \mathit{net} : m.\mathit{mtype} = \text{"LogFreshnessResponse"} \\
\qquad \land\, \text{LET } \mathit{selected} \triangleq \mathit{SelectLog}(Q) \\
\qquad\quad \text{IN} \\
\qquad\qquad \mathit{Send}(\mathit{net}, [\mathit{mtype} \mapsto \text{"LogRequest"}, \\
\qquad\qquad\qquad \mathit{mdest} \mapsto \mathit{selected}])
\end{array}
$$

$$
\begin{array}{l}
\mathit{SendLog}(s) \triangleq \ \land\, \exists\, m \in \mathit{net} : \\
\qquad \land\, m.\mathit{mtype} = \text{"LogRequest"} \\
\qquad \land\, m.\mathit{mdest} = s \\
\quad \land\, \mathit{Send}(\mathit{net}, [\mathit{mtype} \mapsto \text{"LogResponse"}, \\
\qquad\qquad \mathit{msrc} \mapsto s, \\
\qquad\qquad \mathit{mlog} \mapsto \mathit{log}[s])
\end{array}
$$

Fig. 15. Specification Containing Two Rounds of Communication

munication. In the first round, it simply learns the length of each server's log. The first two actions specify this process. When coordinator receives replies from a majority of servers, it chooses the ones with longer logs and requests their logs, as is specified by action $RequestLog$. Note only the selected servers can receive this request. When receiving request from coordinator, server sends back its log, as is specified by action $SendLog$. Note that whether a server send coordinator the whole log or just its length makes no difference in model checking since network capacity is not considered.

Network capacity is not modeled in specification, so it has no effect on the cost of model checking. But more rounds of communication introduce more steps in behavior traces and more possible permutations of actions, which increase the cost of model checking. Thus, we choose the solution with one round of communication in the abstracted specification.

$$
\begin{array}{l}
\mathit{AbsRequestLogResponse}(s) \triangleq \mathit{Send}(\mathit{net}, [\mathit{mtype} \mapsto \text{"LogResponse"}, \\
\qquad\qquad \mathit{msrc} \mapsto \mathit{sm} \\
\qquad\qquad \mathit{mlog} \mapsto \mathit{log}[s]])
\end{array}
$$

Fig. 16. Abstracted Specification Containing One Round of Communication

Figure 16 is the abstracted specification. Each server simply sends the coordinator its whole log directly, taking only one round of communication. Note logs that coordinator receives from servers are internal variable of module leader recovery, so this difference has no influence on the other modules.