TU WIEN Informatics

Comparative evaluation of TLA+ and R-CHECK specifications

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Alexander Tepaev, BSc.,
Matrikelnummer 01327880

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Ezio Bartocci
Mitwirkung: Univ.Ass. Luca Di Stefano, PhD

Wien, 29. April 2026

Alexander Tepaev	Ezio Bartocci

TU WIEN Informatics

Comparative evaluation of TLA+ and R-CHECK specifications

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Alexander Tepaev, BSc.,
Registration Number 01327880

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Ezio Bartocci
Assistance: Univ.Ass. Luca Di Stefano, PhD

Vienna, April 29, 2026

Alexander Tepaev	Ezio Bartocci

Declaration of Authorship

Alexander Tepaev, BSc.,

I hereby declare that I have written this thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work – including tables, maps and figures – which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.
I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix "Overview of Generative AI Tools Used" I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Vienna, April 29, 2026

Alexander Tepaev

Acknowledgements

I would like to sincerely thank my family for making this path possible and for supporting me throughout my studies. I am also grateful to my supervisors for the guidance and feedback during the work on this thesis.

Kurzfassung

Diese Arbeit präsentiert eine vergleichende Evaluierung zweier formaler Modellierungsframeworks — TLA+ und R-CHECK — mit besonderem Fokus auf kommunikationszentrierte und rekonfigurierbare Multi-Agenten-Systeme. Der Vergleich erfolgt in zwei Richtungen. Erstens untersuchen wir aktuelle TLA+-Fallstudien sowie öffentlich verfügbare Beispiele, um die Verfügbarkeit kommunikationszentrierter MAS zu bewerten, und übertragen mehrere repräsentative TLA+-Modelle nach R-CHECK, um die Ausdrucksstärke von R-CHECK zu evaluieren und die wesentlichen Modellierungsentscheidungen, den erforderlichen Aufwand sowie die semantischen Verschiebungen zu dokumentieren, die bei der Übersetzung auftreten. Zweitens rekonstruieren wir ausgewählte R-CHECK-Funktionalitäten in TLA+, um den zusätzlichen Modellierungsaufwand auf Seiten von TLA+ zu bewerten.

Die Ergebnisse zeigen, dass beide Frameworks hinreichend ausdrucksstark sind, um die untersuchten Systeme zu modellieren, dass sie sich jedoch deutlich im Modellierungsstil unterscheiden. TLA+ folgt einer Globalzustandsperspektive auf das System, während R-CHECK den Zustand innerhalb der Agenten kapselt und native Unterstützung für die Modellierung von Kommunikation und dynamischer Konnektivität bietet. Dadurch ermöglicht R-CHECK einen direkteren Modellierungsstil für kommunikationszentrierte und rekonfigurierbare Multi-Agenten-Systeme, während TLA+ in der Regel eine explizitere Kodierung von Konzepten auf der Interaktionsebene erfordert.

Abstract

This thesis presents a comparative evaluation of two formal modeling frameworks — TLA+ and R-CHECK — with a focus on communication-centric and reconfigurable multi-agent systems. The comparison is performed in two directions. First, we survey recent TLA+ case studies and publicly available examples to assess the availability of communication-centric MAS, and we port several representative TLA+ models to R-CHECK to evaluate the expressiveness of R-CHECK and to document the main modeling decisions, required effort, and semantic shifts that arise during translation. Second, we reconstruct selected R-CHECK features in TLA+ to evaluate the additional modeling machinery required on the TLA+ side.

The results show that both frameworks are expressive enough to model the studied systems, but that they differ significantly in modeling style. TLA+ follows a global-state view of the system, whereas R-CHECK encapsulates state inside agents and provides native support for expressing communication and dynamic connectivity. As a result, R-CHECK offers a more direct modeling style for communication-centric and reconfigurable multi-agent systems, while TLA+ typically requires more explicit encoding of interaction-level concepts.

Contents

CHAPTER 1

Introduction

Whenever dealing with complex systems with many concurrent components and non-trivial execution scenarios, formal modeling and verification are useful approaches for reasoning about such systems. This is especially true for multi-agent systems (MAS), where the overall behavior also depends on the coordination and communication between the agents — in such systems, communication becomes a central part of the problem, which means that the modeling language must be able to capture not only the state and structure of the underlying system, but also the structure and semantics of the communication between the agents. This becomes particularly important and challenging in systems with dynamic communication topologies, where communication channels can change, and the connectivity conditions of different agents change depending on the system's state.

In this thesis, we address this problem by studying and comparing two formal modeling languages: TLA+ and R-CHECK. TLA+ is a specification language designed as a general-purpose tool for specifying concurrent and distributed systems. Its modeling view relies on global shared variables and actions (i.e., state transition relations) over the global state. R-CHECK, on the other hand, is designed around agents that communicate with each other over channels — communication is a first-class concept in R-CHECK. For communication-centric multi-agent systems, the difference in modeling perspective between TLA+ and R-CHECK is exactly what makes the comparison interesting — the question is not only whether both frameworks are expressive enough to model the same underlying MAS, but also how natural the resulting modeling process is, what must be encoded explicitly, and where differences arise.

We set up the following research questions for this thesis:

1. *To what extent can representative TLA+ models be ported to R-CHECK? How hard is it to do so?* The purpose of this question is to evaluate whether R-CHECK

is expressive enough to capture behaviors commonly specified in TLA+ and how difficult it is to perform such translations in practice.

2. *What modeling trade-offs and semantic differences arise during the translation between the two frameworks?* This question aims to identify where the two modeling approaches differ, even when the same high-level system behavior is preserved.

3. *When reconstructing R-CHECK-style MAS features in TLA+, what additional modeling machinery must be introduced explicitly?* The motivation for this question is to make explicit which communication-centric concepts are native in R-CHECK and must be encoded indirectly in TLA+.

At the beginning, the scope for this thesis was broader — we wanted to carry out a more general comparison of the two frameworks, which would also include the verification performance and computational resources together with the modeling effort across a much wider class of models. However, during the study of existing TLA+ case studies and the features of R-CHECK, it became clear that the focus had to shift to the most meaningful comparison points. The main distinction between R-CHECK and TLA+ is not that one or the other is a universally better framework, but that they both can be useful in different scenarios. In that sense, R-CHECK offers a modeling view that is better suited for communication-centric multi-agent systems with dynamic topology. The survey also showed that many existing TLA+ case studies rely on enumeration-based verification tools, whereas R-CHECK was designed to exploit symbolic model checking procedures. Due to this difference in the verification setting, a direct comparison of verification performance would only have limited significance with respect to the main goals of this thesis. Thus, the final focus of the thesis is more concrete: we compare the two frameworks based on their expressiveness and modeling overhead with respect to systems consisting of communicating entities with some degree of communication reconfigurability.

In order to address the defined research questions, we first carried out a survey of TLA+ case studies in the available literature and public repositories with TLA+ specifications. We came to the conclusion that while TLA+ is widely used in the industry and offers a large collection of examples, we could not find examples that fit the target setting of this thesis, namely communication-centric MAS with communication reconfigurability being the main point of the model. Thus, we refined our approach into two directions that complement each other. First, we translate several existing TLA+ models into R-CHECK with the purpose of showing that R-CHECK is expressive enough to capture the behaviors specified in TLA+. During the translation, we document the modeling decisions and semantic shifts to show the differences between the two approaches. Second, we developed a running MAS example using R-CHECK to highlight the features of R-CHECK and reconstructed the same example in TLA+. This allowed us to study the comparison from the other direction — whether native R-CHECK functionality can be easily ported and maintained in TLA+.

The main outcome of the evaluation is therefore not quantitative (i.e., verification runtime, memory consumption), but qualitative: how directly communication and dynamic topology can be expressed and how much additional state and bookkeeping logic is required.

The contributions of the thesis can be summarized as follows. First, we provide a survey of existing TLA+ case studies in the academic literature. Second, we present and discuss several ports of existing TLA+ models to R-CHECK, with emphasis on what is preserved, what changes, and why. Third, we construct a feature-oriented comparison between TLA+ and R-CHECK by reconstructing selected R-CHECK features in TLA+ with the help of a running example, which makes explicit the additional modeling effort required on the TLA+ side.

The remainder of this thesis is structured as follows. In Chapter 2, we summarize the background knowledge on model checking, multi-agent systems, R-CHECK, and TLA+/PlusCal, and we briefly survey relevant TLA+ case studies from the literature. In Chapter 3, we present the translation of selected TLA+ models to R-CHECK and discuss the modeling decisions and differences. In Chapter 4, we perform the comparison from the opposite direction by studying how selected R-CHECK features can be implemented in TLA+. Finally, in Chapter 5 we summarize and discuss the findings of the thesis and outline possible directions for future work. The models, translations, and other artifacts developed during this thesis are available in an accompanying public repository.[1]

[1] `https://github.com/alexandertepaev/rcheck-vs-tla`

CHAPTER 2

Background and Related Work

2.1 Model Checking Multi-Agent Systems

Multi-Agent Systems (MAS) A *multi-agent system* (MAS) [Woo92] is a distributed system which is composed of entities (*agents*) that collaborate on solving a specific task. An agent is typically characterized by (i) the agent's environment, (ii) parameters within this environment, and (iii) actions that are carried out by the agent based on the sensed environment; such actions result in changes in the environment [DKJ18].

The MAS concept proved to be a practical approach to solving complex problems by decomposing them into smaller sub-problems and distributing these tasks among agents. Just like computing entities in distributed systems, agents collaboratively solve tasks, but in addition to that they provide more flexibility because they can learn and make autonomous decisions. Agents communicate with their environment, which allows them to learn new contexts and possible actions. Also, based on the environment (local and global), they decide what to do next and then act in order to progress on their assigned task. This kind of flexibility is one of the main reasons why multi-agent systems are used across a broad range of disciplines, including computer science, civil engineering, and electrical engineering [DKJ18].

But these advantages do not come for free. Once we introduce multiple agents, an evolving environment, and communication between agents, the overall system complexity grows quickly, and the algorithms/protocols that coordinate the agents start to become non-trivial. In particular, the authors of [DKJ18] point out that one of the central challenges in MAS is the *coordination control* problem, which includes several following closely related sub-problems: (1) *consensus* (agents must reach agreement on some value of interest), (2) *controllability* (can we direct the system toward desired behaviors), (3) *synchronization* (aligning agent dynamics in time), (4) *connectivity* (ensuring the communication graph is always up-to-date), and (5) *formation* (reaching and maintaining

structural configurations). These coordination aspects all stack up, they make informal reasoning about functionality of underlying systems hard and non-intuitive.

This is exactly why formal verification and model checking are often brought in when studying MAS [LQR17, KLPP19].

Model Checking Model checking is a technique that performs an automated verification of a system's model against its formal specification using algorithmic means. In simple words, model checking answers the following question: given a formal model of a system, does the model satisfy or not a certain property?

Naturally, the following two questions arise: how to obtain the formal model of a system and how to specify properties we would like to verify. In model checking, the system under analysis is represented as a model in terms of its states and transitions. Properties of the system are expressed in a formalized way using temporal logics, such as Linear Temporal Logic (LTL) [Pnu77] or Computation Tree Logic (CTL) [CE82]. A model checker usually takes these formalized properties, checks them against the model and, in case the property is violated, produces a counterexample execution trace. This allows us to prove correctness of a system or find a bug within the system with respect to the provided properties.

Model Checking originated in the early eighties [CE81] and the first algorithms were based on enumerating reachable states of the system and checking if all reachable states satisfy the given specifications of the system. However, this approach has a well-known drawback: as the complexity of the system model grows, the size of the set of reachable states can grow exponentially, resulting in longer verification times. This phenomenon is known as the *state-space explosion problem*. This, of course, restricts the capacity of state-enumeration-based model checking techniques, limiting their use on complex systems in the industry.

To cope with the problem of state-space explosion, researchers developed symbolic representations of state spaces [BCM$^+$92]. For instance, Boolean formulas or binary decision diagrams (BDDs) are used to represent sets of states and transitions between states, allowing verification without the explicit enumeration of system states.

Another popular symbolic technique is *Bounded Model Checking* (BMC) [BCCZ99], where the executions of a system up to a given bound are encoded as a Boolean satisfiability (SAT) problem. This is effective but provides only bounded verification, i.e., it is only able to find counterexamples that do not exceed the given bound. By moving from SAT to satisfiability modulo theories (SMT), symbolic techniques can handle *infinite-state* systems with unbounded data domains: for example, agents that can manipulate integers or reals of any size (i.e., not limited to a fixed number of bits) [AMP06]. Symbolic verification can also extend to full-system verification by using induction-based techniques such as IC3 [Bra11].

Overall, symbolic model checking works with symbolic representations of states and

transitions of a system. This mitigates the state-space explosion problem and makes model checking techniques particularly interesting for the MAS research field.

2.2 R-CHECK Overview

We saw that in MAS coordination and communication between the agents become a central point of the system besides the computation itself. Additionally, reconfigurability of the underlying network topology plays a significant role: agents can join/leave, channels can change, links can be re-wired, and so on. From the model checking point of view, this means that the choice of modeling language matters a lot: we need a language where (i) communication is a first-class concept that is easy to observe and specify, (ii) system composition is natural (since MAS literally involve many agents in parallel), and topology changes can be expressed without extensive changes to the model.

R-CHECK is a model-checking toolkit specifically designed for reconfigurable multi-agent systems (MAS) [AAP22]. It is designed with exactly this kind of setting in mind. Instead of forcing the entire system into one single global view, it provides explicit broadcast/multicast communication primitives and constructs to describe agents and their local state. In addition to that, it has a simple system composition and label-based event observability that makes it easy to specify interaction-level LTL properties (e.g., "a request is eventually acknowledged", "a valid announcement leads to subscription", "a switch triggers a channel update"). In this section, we give an overview of R-CHECK.

The ReCiPe Formalism

The underlying theoretical base of R-CHECK is the `ReCiPe` formalism [AP21], which was developed to provide researchers and developers with MAS models that are close to real-world distributed implementations. In this framework, a system is composed of a set of concurrent agents, where each agent contains its own local state and communicates with other agents, which results in the state change.

The main highlight of `ReCiPe` lies in how it handles communication and connectivity. The following core features define the communication:

- **Communication Modes (Broadcast vs. Multicast):** There are two communication modes in the formalism. First, the *Broadcast* mode, which follows a non-blocking send semantics: the sender proceeds regardless of whether receivers are present and ready to receive a broadcast message. As a consequence, the sender does not know whether the message was actually received by anyone until it receives some kind of follow-up message which makes it observable. Second, the *Multicast* mode, which, on the other hand, blocks the sender until all receiving agents are ready to perform a receive transition.
- **Channels (CH):** All communication occurs over a set of agreed links - *channels*. This includes a unique, broadcast channel ($\star$), which is always available for all

agents, and number of user-defined multicast channels. The channels represent the medium for the communication.

- **Message Data Variables (D):** Messages can carry a payload in the form of *data variables*. Data variables allow agents to exchange actual values, which can be used by the receivers to update their local state.
- **Transitions ($\mathcal{T}$):** The logic of each agent is defined by send and receive transition relations. These transitions are defined by guards that must hold before a transition can occur.
- **Receive-Guards (g^r):** Connectivity to a specific channel is a property that is expressed via *receive-guards*. These guards are parameterized by the agent's local state and the channel name, allowing agents to dynamically decide which multicast channels they are connected to.
- **Send-Guards (g^s) and Common Variables (CV):** To specify the target of a message, `ReCiPe` uses *send-guards* defined over a set of *common variables*. Each agent maps its local state to common variables, allowing the sender to target agents based on these variables.

This structure of agents and interactions between them is what enables the symbolical representation of the `ReCiPe` formalism. Because each agent's behavior is defined by discrete send and receive transition relations, the entire state space of an agent is expressed using a first-order predicate. Furthermore, by composing multiple predicates of different agents, we can derive a global transition relation, ρ. This relation encodes the entire MAS into a single Boolean formula. This formula-based representation is what handles the state-space explosion problem: instead of enumerating every possible state of the system, the correctness can be verified by checking the satisfiability of the resulting formula using state-of-the-art SAT or SMT solvers.

The R-CHECK Model Checker

R-CHECK [AAP22] is a model checker and a small input language that implements the `ReCiPe` formalism. The language allows us to model the multi-agent systems in a form that strongly resembles programming. We use so-called *commands* together with sequencing and/or non-deterministic choices to describe the behavior of communicating agents, and then compile them into the underlying symbolic `ReCiPe` transition relation for model checking. The actual model checking is performed by `nuXmv` [CCD+14], which provides support for symbolic techniques such as BMC and IC3.

In R-CHECK, a system is represented as a set of agent instances composed to run in parallel. Listing 2.1 shows the basic skeleton of such a model: it declares two agent types, `Sender` and `Receiver`. Both agents have an empty local state (`local`), trivial initialization (`init: true`), and a trivial connectivity condition (`receive-guard: true`). Their behavior is placed inside the `repeat: (...)` process, which defines the

```
1  agent Sender
2    local:
3    init: true
4    receive-guard: true
5    repeat:
6    (
7    )
8
9  agent Receiver
10   local:
11   init: true
12   receive-guard: true
13   repeat:
14   (
15   )
16
17 system = Sender(s0, true) ||
18          Receiver(r0, true) ||
19          Receiver(r1, true)
```

Listing 2.1: R-CHECK model skeleton

agent's loop (in this minimal example the loop body is empty). The last line composes the system by instantiating one `Sender` agent (`s1`) and two `Receiver` agents (`r1`, `r2`) using parallel composition `||`. Each instance is given an additional initial constraint (here just a simple `true`), which can be used to specialize initial values per instance.

As we have already mentioned, communication is the main part of the underlying formalism, and thus, of the R-CHECK language itself. An agent can execute *commands* of two kinds:

$$\text{Send Command: } \{\Phi\}\ ch!\ (\pi)\ (d)\ \{U\}$$

and

$$\text{Receive Command: } \{\Phi\}\ ch?\ \{U\}$$

Here, Φ is the precondition over the current local state (for receives it may also refer to received message fields), ch is either the broadcast channel ($\star$) or the name of a variable storing a channel name, π is the sender predicate over common variables (`true` can be used if no targeting is needed), d denotes assignments to message fields, and U is the local state update (assignments to local variables).

Inside the agent's repeat block, commands can be combined into a process using the following rules:

$$\text{Process P} ::= P; P \mid P + P \mid \text{rep } P \mid C$$

Here, C denotes a single command (send or receive), which forms the smallest behavioral building block. The term $P; P$ denotes sequencing: the left part executes first, and then, once it finishes, the right part continues. The term $P + P$ denotes non-deterministic choice: one of the two branches is selected non-deterministically for execution. The term

rep P denotes looping: the process P is executed repeatedly, which is the typical way to model cyclic behavior of an agent.

To illustrate the behavior and semantics of R-CHECK, we use a simple example presented in Listing 2.2. The example is constructed in such a way, that most of the R-CHECK syntax and the key semantics are touched: agents and system composition, broadcast vs. multicast communication commands and their composition, channel connectivity, sender's predicates and relabeling, and, finally, LTL properties over command labels. In the remaining part of this section, we discuss the example line by line.

The first block declares enumeration types using the `enum` keyword (lines 1–2). So, `msgType` contains two message kinds (`announce`, `update`) and `channel` contains channel (`c1`) that we will use for multicast communication. The definition of enums is there in order to make message content and channels explicit. Next, `message-structure` defines the fields that can be transmitted in a message (lines 4–5). In this example we only carry one field (`MSG`), using which we distinguish between the "announcement" and the "update" messages.

Lines 7–8 introduce a common variable. Common variables are not part of the local state of any particular agent. Instead, they are used on the sender side to specify constraints over the receivers without referencing receiver-local variables directly. In our case, the `isReady` common variable tells the sender if a receiver is ready to receive an update message. We will later explain how `isReady` is interpreted locally via the `relabel` block of the sender agent.

The `agent Sender` definition (lines 10–20) is the first example of agent definition syntax. It contains an `init` predicate and a `receive-guard`, which are both set to (`true`), meaning the sender has no local state and connectivity constraints. The actual behavior is placed in the `repeat: (...)` block (lines 13–20). The sender process is written as a sequence of two send operations (hence the `!`): `sAnnounce ; rep(sUpdate)` (lines 15, 18). The operator `;` is the sequencing operator: the left-hand command runs once, and then the right-hand part is executed. The operator `rep` is the repetition operator: `rep(sUpdate)` means that `sUpdate` is executed repeatedly forever. So, in our example, the sender has a clear behavior which consists of two phases: send broadcast `announce` first, then keep sending `update` forever over a multicast channel.

Now for the commands themselves. In `sAnnounce` (line 15), the sender performs a *broadcast send* on channel `*`. The payload sets `MSG := announce`. The important part here is the sender's predicate `(@isReady == false)`. This formula predicates over the common variable `isReady` and tells which receivers are "targeted". In R-CHECK, broadcast is non-blocking, meaning that the sender can perform this step regardless of how many receivers exist or whether they are ready. Receivers that do not satisfy the predicate `(@isReady == false)` simply ignore the message. The local update `[]` is empty, so `sAnnounce` does not change the sender's local state. The command `sUpdate` (line 18) is a *multicast send* on channel `c1` with payload `MSG := update`. In

```
enum msgType {announce, update}
enum channel {c1}

message-structure:
  MSG: msgType

property-variables:
  isReady: bool

agent Sender
  init: true
  receive-guard: true
  repeat:
  (
    sAnnounce: {true} *! (@isReady == false)(MSG := announce)[]
    ;
    rep (
      sUpdate: {true} c1! (true)(MSG := update)[]
    )
  )

agent Receiver
  local:
    ready: bool,
    state: bool
  init: state == false
  relabel:
    isReady <- ready
  receive-guard: (chan == c1) & ready
  repeat:
  (
    rAnnounce: {MSG == announce} *? [ready := true]
    +
    rUpdate: {MSG == update} c1? [state := !state]
  )

system = Sender(s0, true) ||
         Receiver(r0, ready == true) ||
         Receiver(r1, ready == false)

// only the not-ready receiver reacts to the broadcast announce
SPEC G (s0-sAnnounce -> F (!r0-rAnnounce & r1-rAnnounce))
SPEC G (s0-sAnnounce -> F r1-ready)

// readiness results in progress
SPEC G ((r0-ready) -> F r0-rUpdate)
SPEC G ((r1-ready) -> F r1-rUpdate)

// multicast toggles r1.state when r1 participates
SPEC G ((s0-sUpdate & r1-state == true) -> X r1-state == false)
SPEC G ((s0-sUpdate & r1-state == false) -> X r1-state == true)
```

Listing 2.2: Minimal R-CHECK model

contrast to broadcast, multicast is blocking, meaning that the send step is only enabled if all receivers may execute a matching receive command. This is where connectivity and synchronization become explicit: multicast requires all connected receivers to have a corresponding receive step and models a behavior where the sender and all connected receivers change their state together.

The `agent Receiver` definition block (lines 22–35) introduces the remaining piece of the example. First, it declares local variables `ready` and `state` (lines 23-26). The `init` predicate initializes only `state == false` (line 26). The `ready` variable is left out so that it can be initialized per instance in the `system` composition below. Next, the `relabel` block (lines 27–28) defines how the receiver maps its local state to common variables: `isReady <- ready`. This is the main point that provides a link between sender's predicate and the receiver state in the sender's `sAnnounce`. The `receive-guard` line (line 29) is used to model multicast connectivity `chan == c1 & ready`. Intuitively, it says: "this receiver is connected to channel `c1` only when `ready` is true". This allows us to model the dynamic topology change: initially, a receiver with `ready == false` is disconnected from multicast `c1` and thus cannot participate in `sUpdate/rUpdate`. After it becomes ready, it becomes connected to `c1` and multicast update can take place. Receiver behavior is defined inside the `repeat` block as a non-deterministic choice (lines 32–34) using the + operator, which means that in each iteration of the block any enabled receive can be executed. The first receive `rAnnounce` (line 32) is a receive on the broadcast channel `*`. Its precondition checks that the received message is of type `announce`, and, if this is the case, its update sets `ready := true`, making the receiver ready for the subsequent multicast communication. The second branch `rUpdate` (line 34) is a multicast receive on `c1`. It again checks the type of the received message `MSG == update`, and, if this is the case, the update toggles the agent's internal state `state := !state`.

The `system` block (lines 37–39) composes one sender instance `s0` and two receiver instances `r0` and `r1` and allows to add extra state initialization as the additional instantiation parameter. In our case, `r0` starts with `ready == true` and `r1` starts with `ready == false`. This is done so for the purpose of demonstrating the common-variables, relabeling and connectivity functionality of R-CHECK: we want `sAnnounce` to be sent to the receivers with `ready == false` to allow such receivers to dynamically connect to the `c1` channel.

Finally, the `SPEC` lines (lines 41–51) show how labeled R-CHECK commands can be useful for defining properties to model check. Each labeled command (e.g., `sAnnounce`, `rAnnounce`, `sUpdate`, `rUpdate`) becomes a boolean predicate for an agent, (e.g., `s0-sAnnounce` or `r1-rUpdate`) and can be used directly in the properties. The first two properties verify the effect of the broadcast announcement: after `s0-sAnnounce`, eventually `r1` performs `rAnnounce` (while `r0` does not), and `r1-ready` becomes true. The next two properties verify the progress of the system: once a receiver is ready, it should eventually receive an update via `rUpdate`. The last two properties check the local effect of the update: whenever `s0-sUpdate` happens, the receiver's `state` toggles

in the next step.

Overall, this example provides a compact demonstration of the R-CHECK syntax and core semantics. Also, this illustrates which class of systems R-CHECK is well suited for: interaction-centric multi-agent systems where communication and dynamic reconfiguration are part of the problem, and where properties to be checked are event-based.

2.3 TLA+/PlusCal Overview

In this section, we give a brief introduction to TLA+ (*Temporal Logic of Actions*) [Lam03] and its companion PlusCal [Lam09]. For simplicity, we will use the term TLA+ throughout this section and explicitly mention PlusCal only when talking about specific features only relevant to this language. The goal of this section is not to provide a full tutorial on the basics of TLA+, but to give the reader enough background to be able to understand TLA+'s modeling style and paradigm and how it differs from R-CHECK. In particular, we would like to address TLA+'s notion of *actions* and its global view of the system's state.

TLA+

TLA+ is a language that was invented by Leslie Lamport [Lam03] for specification of concurrent and distributed systems. The main modeling idea behind TLA+ is to represent a system as a set of states the system can be in, where each state is an assignment of values to a fixed set of global variables. An execution of such a system is then simply a sequence of such states. The theoretical background for TLA+ strongly relies on standard mathematics: sets, functions, and relations are the main building blocks that are used to describe the data and structure of the system. From this point of view, a TLA+ model is not a program, but a mathematical description of the allowed states and the allowed transitions between these states.

Transitions within a TLA+ model are defined by so-called *actions*. These are the TLA+ language constructs that describe a transition from the current state to the next: constraints over the current variables (unprimed `state`) are related to the values of the next state (primed `state'`). Usually a TLA+ specification contains an initial state and the next-state actions: `Init` and `Next`.

TLA+ comes with its own TLC model checker [YML99]. It works by exploring all states reachable by the specified system and checking that the defined invariants and temporal properties are satisfied by these states: TLC simply enumerates all reachable states applying the defined `Next` relation starting from `Init`, and, whenever a property is violated, it produces a counterexample. This brings up the problem of state space explosion: in complex systems and their models, the number of variables and possible interleavings grow, which, in turn, can lead to TLC's slow down or resource exhaustion. This limitation of TLC is addressed by symbolic-based approaches. For example, tools

such as APALACHE [KKT19] encode the system's behavior as SMT problems for further analysis using an SMT solver.

As in the case with R-CHECK, we show the basics of TLA+'s modeling style by looking at a small example and explaining how each line contributes to the overall specification of the underlying system. Listing 2.3 contains a minimal client/server request–ack protocol with an explicit global message buffer for modeling the communication. The configuration file in Listing 2.4 is needed to tell TLC how to instantiate constants and what to check. We begin by defining constants and variables that we are going to use in our model. In TLA+, `CONSTANTS` are model parameters that will be set to a constant value by the TLC configuration, while `VARIABLES` are changed by the model. In our examples, we define two constants: `REQ` representing a "request" message type and `ACK` representing an "acknowledge" message type. The variables have the following meaning: `msgs` represents the network as a set of in-flight messages, and `clientState` / `serverState` encode the states of both endpoints.

Next, we define a record type `Message == [type: {REQ, ACK}]` (line 7). In TLA+ all structured data are represented using records, functions, and sets. So, the initial condition `Init` (lines 9–12) defines the system's initial state: in the beginning there are no messages (`msgs = {}`), and both client and server start in `"Idle"`.

The actual transition logic of a model is written using TLA+'s *actions*. An action is simply a logical formula on the current state and the next state: it relates the current state to the next state by asserting what needs to hold now and what changes need to be applied to the next state. In our case, each step is written as a separate action (lines 14–30). For example, `ClientSendReq` says that when `clientState = "Idle"`, the model can send by updating the message buffer as `msgs' = msgs ∪ {[type |-> REQ]}`. The action also updates `clientState'` to `"Waiting"` and uses `UNCHANGED serverState` to state explicitly that the server state does not change in this step. This `UNCHANGED` pattern is standard in TLA+: every action must define the next-state value of every variable, and `UNCHANGED` is the idiomatic way to say "this variable remains unchanged". Receiving a message is then modeled by consuming it from the buffer. For example, `ServerRecvReq` says that if there exists a request message in `msgs` (`\E m \in msgs: m.type = REQ`), then we can consume it by removing it from the buffer using TLA+'s set difference: `msgs' = msgs \{[type |-> REQ]}`. The server then moves to `"GotReq"`. The two remaining actions, `ServerSendAck` and `ClientRecvAck`, follow the same idea: add/consume a message, then advance the control state machine.

In TLA+, actions can also be used to express concurrency. This is done by combining multiple actions using disjunction. So, our global next-state relation `Next` (lines 36–37) is the disjunction of all protocol steps: `ClientSendReq ServerRecvReq ServerSendAck ClientRecvAck`. In each step, any enabled action can be executed, which results in nondeterminism and interleavings for the TLC model checker.

Finally, to combine `Init` and `Next` and finalize the specification (lines 40–42), TLA+

```
1  -------------------------- MODULE MsgBuffer --------------------------
2  EXTENDS TLC
3
4  CONSTANTS REQ, ACK
5  VARIABLES msgs, clientState, serverState
6
7  Message == [type: {REQ, ACK}]
8
9  Init ==
10   /\ msgs = {}
11   /\ clientState = "Idle"        \* "Idle" | "Waiting" | "Done"
12   /\ serverState = "Idle"        \* "Idle" | "GotReq"  | "SentAck"
13
14 ClientSendReq == /\ clientState = "Idle"
15   /\ msgs' = msgs \cup { [type |-> REQ] }
16   /\ clientState' = "Waiting"
17   /\ UNCHANGED serverState
18
19 ServerRecvReq == /\ serverState = "Idle"
20   /\ \E m \in msgs: m.type = REQ
21   /\ msgs' = msgs \ { [type |-> REQ] }
22   /\ serverState' = "GotReq"
23   /\ UNCHANGED clientState
24
25 ServerSendAck == /\ serverState = "GotReq"
26   /\ msgs' = msgs \cup { [type |-> ACK] }
27   /\ serverState' = "Idle"
28   /\ UNCHANGED clientState
29
30 ClientRecvAck == /\ clientState = "Waiting"
31   /\ \E m \in msgs: m.type = ACK
32   /\ msgs' = msgs \ { [type |-> ACK] }
33   /\ clientState' = "Idle"
34   /\ UNCHANGED serverState
35
36 Next ==
37   ClientSendReq \/ ServerRecvReq \/ ServerSendAck \/ ClientRecvAck
38
39 vars == <<msgs, clientState, serverState>>
40 Spec == Init /\ [][Next]_vars /\ WF_vars(ClientSendReq)
41   /\ WF_vars(ServerRecvReq) /\ WF_vars(ServerSendAck)
42   /\ WF_vars(ClientRecvAck)
43
44 ReqImpliesWaiting == [](REQ \in msgs) => (clientState = "Waiting"))
45 AckImpliesWaiting == [](ACK \in msgs) => (clientState = "Waiting"))
46
47 WaitingEventuallyIdle ==
48     [](clientState = "Waiting" => <>(clientState = "Idle"))
49
50 ReqEventuallyConsumed ==
51   [](REQ \in msgs) => <>~(REQ \in msgs))
52 ============================================================================
```

Listing 2.3: Minimal TLA+ model

```
1  CONSTANTS
2      REQ = REQ
3      ACK = ACK
4
5  SPECIFICATION
6      Spec
7
8  PROPERTIES
9      ReqImpliesWaiting
10     AckImpliesWaiting
11     WaitingEventuallyIdle
12     ReqEventuallyConsumed
```

Listing 2.4: TLC configuration the TLA+ Model in 2.3

uses a specific pattern: `Spec == Init / [][Next]_vars` (line 40) with `vars` being the tuple of global variables (line 30). The `[][Next]_vars` construct says that the next possible steps are described by `Next` action, but we also allow stuttering for the variables. The outer part (`[]`) means "always" and forces this condition to be satisfied at all times. Since we allow stuttering steps, TLC can produce trivial counter examples (e.g, "system remains in initial state forever"). For this reason, `Spec` includes weak fairness for each next-state action: `WF_vars(Action)`. Intuitively, weak fairness ensures that in case an action is enabled for execution, then it must eventually be executed. For our examples this means that the "do nothing forever" trace is ignored.

In order to verify the correctness of the model, we define a set of temporal properties that we expect to hold (lines 44–51). In TLA+, we usually use the `[]` (always) and `<>` (eventually) operators to express the temporal properties. In our example, `ReqImpliesWaiting` and `AckImpliesWaiting` say that whenever a request/ack is in-flight, the client must be in the "waiting" state. The properties `WaitingEventuallyIdle` and `ReqEventuallyConsumed` express system progress: if the client is waiting, it eventually returns to idle, and if a request is in the buffer, it eventually disappears.

In summary, this example clearly shows the basic TLA+ workflow: define the state of the system via `VARIABLES`, specify initial states with `Init`, specify allowed transitions with actions combined into `Next`, combine initialization and transition under `Spec`, and finally write properties for TLC to check.

PlusCal overview

PlusCal can be seen as a syntactic sugar added on top of the TLA+ language, which allows writing a more algorithm-like description of a model, i.e., with variables, control flow, and multiple concurrent processes. In essence, the PlusCal translator compiles this algorithmic description to an equivalent TLA+ specification. The key point of PlusCal that is relevant for our discussion is that it does not change the modeling approach of TLA+: when using PlusCal, we still write transition relations over a set of global variables that represent the state of the model. PlusCal makes it more convenient to express

parallelism within the system: in PlusCal, concurrency is expressed using `process` blocks, and TLC evaluates the model by interleaving these. Under the hood, during the compilation, an explicit program counter (`pc`) is introduced to the TLA+ code, which tracks the state of each process. Each PlusCal step is compiled into a TLA+ action that is assigned to the correspondingpc value.

What is also relevant for this thesis is the notion of *macros* in PlusCal. These are a structuring mechanism, which allows grouping a block of algorithmic code under one name and reusing it across the specification. From the TLA+/TLC point of view, a macro call is executed as an atomic step since the translator assigns it a dedicated `pc` value. In Section 3.4, we will see a protocol specification written in PlusCal, so it is important to keep in mind that macro bodies are atomic operations.

Concluding remarks

In general, we see that TLA+ and PlusCal are both designed to be general-purpose and expressive — with these tools we can model various concurrent and distributed systems. At the same time, when the system is communications-centric (as multi-agent systems usually are), some things have to be encoded manually: so, for example, agent populations must be represented indirectly (e.g., using indexing over agent arrays), communication must be modeled via shared global buffers such as `msgs`, and properties about "events" usually require additional bookkeeping. This does not make TLA+ weaker in any terms, it simply shows that TLA+ follows a different modeling perspective. In the following chapters, we evaluate this difference on communication-centric MAS models written in TLA+ and R-CHECK.

2.4 TLA+ in practice: case studies and examples

In this section, we give a brief overview of the application of TLA+ in practice. For this, we used two sources: (i) a DBLP [dT26] search for papers mentioning "TLA+" in the years 2015–2025, and (ii) the public TLA+ examples repository [LAKM+]. The purpose of this short survey is to show that TLA+ is widely applied both in academia and in the industry, and to see if there are case studies that explicitly focus on the niche targeted by R-CHECK — communication-centric, reconfigurable multi-agent systems.

A very good entry point for this research is the systematic literature review of industrial TLA+ practice [BLTK24]. To extend our research even further, we collected a sample of recent papers from DBLP and for each paper recorded the following information: (i) chosen modeling language, (ii) chosen model checker, and (iii) whether the model is publicly available. These paper-level details are listed in Table 2.1.

The case studies we collected address the following set of application areas: distributed databases and storage systems, consensus/coordination services, cloud infrastructure components, and security-critical protocols. The resulting high-level classification is shown in Table 2.2.

Paper	Language	Checker	Model available
A Tendermint Light Client [BBK+20]	TLA+	APALACHE	in paper
Formal Verification of Consensus in the Taurus Distributed Database [GZL+21]	TLA+	TLC	public repo
Understanding Inconsistency in Azure Cosmos DB with TLA+ [HRK23]	TLA+	TLC	public repo
Following the White Rabbit: Integrity Verification Based on Risk Analysis Results [JWSH21]	TLA+/PlusCal	TLC	on request
Designing and Proving Properties of the Abaco Autoscaler Using TLA+ [PS21]	TLA+	TLC	public repo
LogPlayer: Fault-tolerant Exactly-once Delivery using gRPC Asynchronous Streaming [RRZB19]	PlusCal	TLC	in paper
Design and Analysis of a Logless Dynamic Reconfiguration Protocol [SZDT21]	TLA+	TLC	public repo
The Development of a TLA+ Verified Correctness Raft Consensus Protocol [GJZ24]	TLA+	TLC	public repo
Leveraging TLA+ Specifications to Improve the Reliability of the ZooKeeper Coordination Service [OHH+23]	TLA+	TLC	public repo
Verifying Payment Channels with TLA+ [GH22]	TLA+	TLC	public repo
Specifying and Verifying SDP Protocol Based Zero Trust Architecture Using TLA+ [DNZZ22]	TLA+	TLC	public repo
Verifying Hyperproperties With TLA [LS21]	TLA+	TLC	public repo
Improving Security in SCADA Systems through Model-checking with TLA+ [OP21]	TLA+	TLC	in paper
Formal Verification of SDN-Based Firewalls by Using TLA+ [KK20]	TLA+	TLC	in paper
Specification and Verification of the Zab Protocol with TLA+ [YZF20]	TLA+	TLC	in paper
Modeling a Smart School Building System Using UML and TLA+ [OP20]	TLA+	TLC	in paper
Automated Validation of State-Based Client-Centric Isolation with TLA+ [SvdSV20]	TLA+	TLC	public repo
Specification and Verification of a Multi-agent Coordination Protocol with TLA+ [PSB18]	TLA+	TLC	in paper
Formal Verification of Usage Control Models: A Case Study of UseCON Using TLA+ [GGM18]	TLA+	TLC	in paper

Table 2.1: Survey details: language, checker, model availability

Category	Papers	Modeling view
Distributed protocols / coordination	[BBK+20, SZDT21, GJZ24, OHH+23, GH22, YZF20]	Distributed protocol behavior specified via global state and actions; communication modeled through shared global structures (e.g., message sets) with nondeterministic asynchronous delivery.
Data systems / transactions / logs	[GZL+21, HRK23, RRZB19, SvdSV20]	Database/transaction behavior and recovery logic; concurrency expressed via ID-indexed collections.
Explicit MAS coordination protocols	[Niy25, PSB18]	MAS-flavored models typically encode agent populations as collections over agent IDs and express interaction through global state updates and/or asynchronous message pools.
Other (security/CPS/methodology)	[JWSH21, PS21, DNZZ22, LS21, OP21, KK20, OP20, GGM18]	Case studies outside distributed protocol/database scope (security architectures, CPS architectures, and one methodological paper), but following the same global-state modelling style.

Table 2.2: Recent TLA+ case studies.

In addition to the presented case studies, the public TLA+ examples repository [LAKM+] is a good source for specifications, varying from models that can be used for learning the TLA+, to classic puzzles and well-known protocol-style examples.

Overall, TLA+ is widely used for modeling and verification of distributed protocols and concurrency-heavy systems. At the same time, only a couple of found usage examples explicitly address multi-agent systems. Our survey also shows that plain TLA+ with TLC is the dominant combination used in practice, whereas PlusCal and APALACHE appear less frequently in the surveyed case studies.

2.5 Summary

In this chapter, we introduced the background knowledge that is required to understand the rest of the thesis. We started with the general definition of multi-agent systems and the central aspects of such systems — communication and coordination between agents. We saw that, in systems with a lot of concurrency and non-deterministic interleaving, the use of formal modeling and model checking provides a systematic way to reason about their correctness.

Next, we introduced two modeling frameworks: R-CHECK and TLA+. The main takeaway is that R-CHECK is designed around an explicit agent view of the system and provides dedicated communication primitives (broadcast, blocking multicast, channels, connectivity guards, sender predicates, and relabeling), which makes communication and dynamic reconfiguration main concepts of the modeling language. TLA+, on the other

hand, provides a very general way of modeling a system, making it a good all-round language.

Finally, we briefly looked at TLA+ case studies and examples available in the literature. The papers that we covered in the survey confirms that even though TLA+ is widely for distributed protocols, data systems, and security/architecture verification, still only a small number of the found examples targets multi-agent systems.

CHAPTER 3

Porting TLA+ systems to R-CHECK

This chapter provides the first part of the thesis evaluation by implementing and discussing R-CHECK ports of existing TLA+ models from the official TLA+ Examples Repository [LAKM+]. The goal is to show that R-CHECK is expressive enough to model representative problems from the TLA+ framework. Our examples include a non-agentic puzzle, a concurrent data structure, and a non-trivial distributed consensus protocol, with two of the source models written directly in TLA+ and one in PlusCal. Because PlusCal compiles to TLA+ code, both modeling frameworks share the same underlying system view (global shared state, atomic actions), so our discussion of TLA+ also applies to the PlusCal model.

As we will see, when performing a translation, our goal is not to come up with a translation that fully preserves all aspects of the original model. Instead, we focus on expressing the behavior of the underlying system in a way that is natural for R-CHECK's communication- and agent-centric view. This results in the fact that ports do not exactly replicate the original models, but are more fine-grained because the interaction structure is made explicit. Therefore, we document design decisions alongside with "where and why" semantic differences arise.

3.1 Porting Idea

Due to the nature of TLA+, it is not an easy task to port TLA+ specifications to R-CHECK. A system model in TLA+ is defined through a set of global variables and a transition relation over these global variables. In R-CHECK, on the other hand, models are structured around agents with local states that communicate with each other. This leads to the fact that two models describing the same system using the TLA+ and

R-CHECK approaches may still have notable semantic differences. These semantic differences appear as the result of different modeling perspectives: how system's state is defined (global vs. local state variables), how communication is represented (a global buffered message set vs. explicit communication steps with non-blocking broadcast and blocking multicast), and the granularity of atomicity (single global action that changes multiple parts of the system vs. multiple steps).

In order to present the example translations in a systematic way, we follow a simple porting strategy:

1. **Identify agents and roles**
   We begin with identifying the actors in the TLA+ model and converting them to agent types in R-CHECK.
2. **Define agent states**
   We distribute the global state from TLA+ model to the R-CHECK agents presented in the previous step.
3. **Introduce communication**
   We identify the communication (can also be implicit) between the actors in the TLA+ and introduce explicit message exchange in R-CHECK.
4. **Validate the ported model against properties**
   Next, we validate the ported model using the set of properties that capture the expected behavior of the system, ideally reusing the same properties used in the original TLA+ model whenever they can be expressed naturally in R-CHECK
5. **Document semantic differences**
   Since ports are not always 1:1, we state and document the semantic differences that were introduced during the translation.

3.2 Tower of Hanoi

The Tower of Hanoi port serves as a demonstration of converting a non-communication-centric model to R-CHECK. The main goal here is to show that a classic non-agentic TLA+ specification, which is written over a single globally defined state, can still be expressed in R-CHECK, even though R-CHECK is not designed for this kind of modeling in the first place. In particular, we will see that while this example is easy to model in TLA+ due to its transition relation over globally available data, the R-CHECK version is more structured and expressive — the same behavior is reproduced by introducing agents and communication between them.

The Tower of Hanoi puzzle consists of three towers and a set of disks of different sizes, which are initially stacked on one tower in ascending order (smallest disk on top of the stack, largest disk at the bottom of the stack). The goal is to move the stack to another

tower by moving one disk at a time, maintaining the constraint that a larger disk may never be placed on top of a smaller disk.

3.2.1 Original TLA+ specification

The TLA+ model of Tower of Hanoi [Nie20] is presented in Listing 5.1. It models each tower as a sequence of natural numbers, where the numbers represent disk sizes (the larger the number, the larger the disk). The three towers are then placed into a single global sequence `towers` (Line 23). This "sequence of sequences" is the only state variable of the model and all transitions are defined through updates to this variable. The initial state is given by `Init == towers = «A, B, C»` (Lines 22–23), where `A`, `B`, and `C` are constant initial configurations (e.g., `« «1,2,3», «», «» »`).

The next-state transitions of the model are defined by the `Next` action (Lines 53–56). Inside this action, we non-deterministically pick two towers `from` and `to` form the range `1..Len(towers)` and check whether a move of the top disk from `towers[from]` to `towers[to]` is allowed. This check is captured by `CanMove(from,to)` (Lines 32–40) — a move is valid if following conditions hold:

- `from` and `to` are different towers;
- the source tower is non-empty;
- either the destination tower is empty or the top disk of the source tower is smaller than the top disk of the destination tower.

The actual state update logic is implemented in the `Move(from, to)` action (Lines 46–51). It updates the `towers` by removing the head disk from the source tower and assigning it to the destination tower. Last but not least, the specification defines `NotSolved` predicate (Lines 108–113), which is the negated configuration where all disks are on the third tower. This forces TLC to produce a counterexample trace with a valid solution.

The specification relies on TLA+ operators that come as a standard library (e.g., `Head`, `Tail`), sequence concatenation `\o`, and `Len` and from `Sequences`. This keeps the model compact. From the structural point of view, the model is centered around a single global state object `towers`, rather than being distributed across multiple interacting components as it would typically be in a real implementation.

The TLA+ module and the configuration file used for running the TLC on the given problem are shown in Listing 5.2 and Listing 5.3, respectively. The module instantiates the initial configuration for three disks. The configuration file instructs TLC to check that `TypeOK`, `NoNewElements`, `TotalConstant`, and `NotSolved`, defined in Listing 5.2, are invariants of the system. The `TypeOK` invariant (Lines 72–75) checks that the state variable preserves the expected structure and value domains throughout the run. `NoNewElements` (Lines 81–88) ensures that no new disk values are introduced during execution. In order to exclude the possibility for disks to disappear, the `TotalConstant`

invariant (Lines 93–100) asserts that the total number of disks remains unchanged throughout the run. Finally, `NotSolved` (Lines 108–113) is the typical property used in models of puzzles: it says that the state of the system we want to end up in can never be reached. TLC attempts to maintain it as an invariant, and once it is violated, the resulting counterexample trace corresponds to a valid solution of the puzzle.

3.2.2 Applying the porting recipe

The ported R-CHECK model is presented in Listing 5.4.

Identify agents and roles. Since agents are the main building blocks of R-CHECK, the first thing we need to do is to identify the agent types that we will use in our R-CHECK model. Obvious candidates are therefore `Tower` and `Disk` agent types (Line 8–39 and 42–60). In our concrete instance we model three towers and three disks, so the system spawns three `Tower` agents and three `Disk` agents (Lines 62–70).

Define agent states. In the TLA+ models, the state is fully encapsulated in the single variable `towers`, which is represented by a sequence of sequences. In our R-CHECK port, we need to distribute this global state across agents. Concretely, each tower must know which disks it currently holds and in which order. Also, each disk must know on which tower it is currently located. This is exactly what we model in R-CHECK: each `Tower` stores a bounded representation of its disk stack using three slots `s1/s2/s3` and a pointer to the next occupied slot `head` (Lines 12–15). Each `Disk` stores its current location `loc` (Line 45). This, of course, results in some duplication of information (a tower knows which disks it stores, and a disk knows where it is), but structurally it makes the model agent-centric and keeps updates local: a tower updates its own stack, and a disk updates its own location.

Introduce communication. All interaction is modeled via a single multicast channel `ch`. A move is performed when a `Disk` agent sends a message on `ch` with payload `(DISK_ID, FROM, TO)` (Lines 50–57). In words, this models a disk' proposal to move from its current tower `FROM` to a target tower `TO`. Each `Tower` agent receives this proposal and reacts to it (Lines 20–35): if it is the source tower (`FROM == id`), it updates its local stack by removing the disk (decrementing `head`); if it is the destination tower (`TO == id`), it updates its local stack by placing the disk on top (incrementing `head` and writing into the corresponding slot); otherwise it ignores the message.

The main point here is that we use a blocking multicast channel. This means the send can only happen if all connected receivers have a matching receive transition enabled. We use this behavior to encode the validity of a move into the receive preconditions. For example, the destination tower only enables `rMoveTo_h1` if its current top disk is larger than the incoming disk (`s1 > DISK_ID`). Similarly, the source tower only enables `rMoveFrom_h2` if the disk being moved is exactly the current top disk (`head == 2 & s2 == DISK_ID`). If a disk attempts an illegal move (e.g., trying to move a

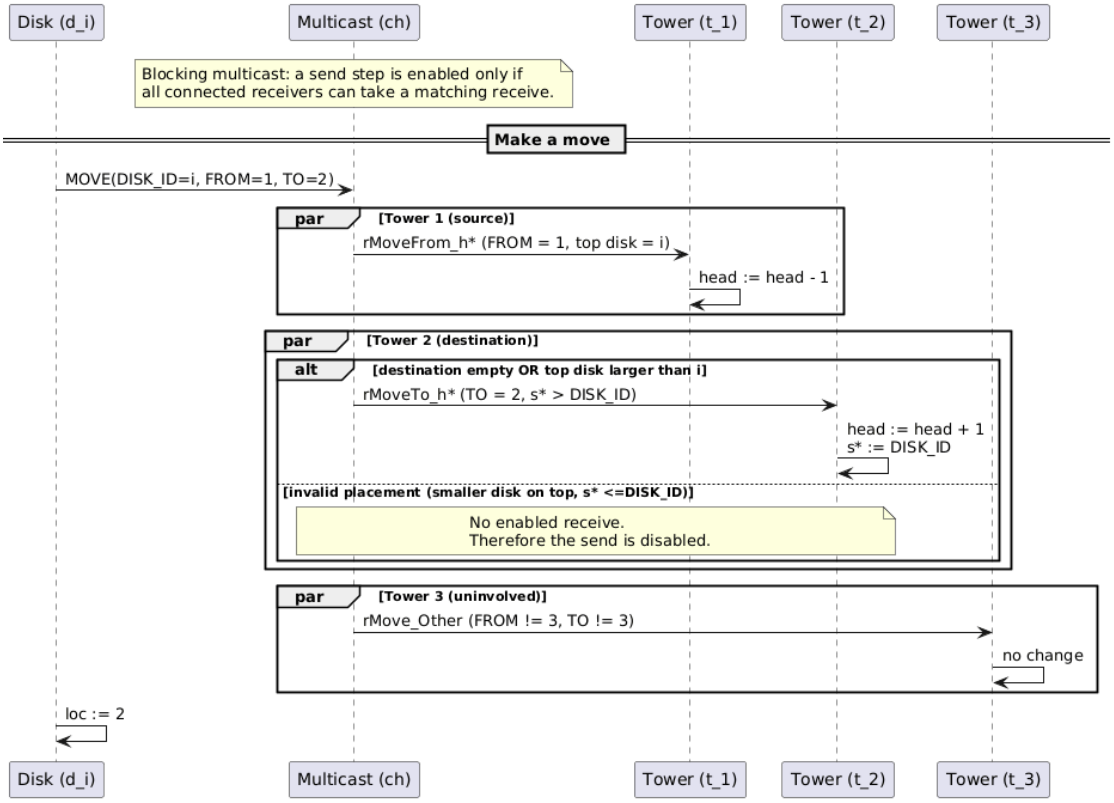

Figure 3.1: Sequence diagram of Hanoi R-CHECK move.

disk that is not on top, or trying to place a larger disk on a smaller one), then a relevant tower will have no enabled receive for that message, and because multicast is blocking, the send transition is disabled and no state change is performed. In this way, the validity checks that are expressed as `CanMove` in TLA+ are reconstructed in R-CHECK via the combination of message payload, receive preconditions, and the synchronous behavior of blocking multicast communication.

The UML Sequence diagram in Figure 3.1 illustrates one move in the described port. The disk agent `Disk(d_i)` proposes a move by multicasting `MOVE(DISK_ID=i, FROM=1, TO=2)` on channel `ch`. This step is enabled only if all connected towers can take a matching receive transition: the source tower `Tower(t_1)` executes the corresponding `rMoveFrom_h*` command and decrements its `head`, the destination tower `Tower(t_2)` executes `rMoveTo_h*` only if the placement is valid (empty destination or top disk larger than `i`) and then assigns the disk to the first free slot. At the same time, the uninvolved tower `Tower(t_3)` consumes the message via `rMove_Other` and remains unchanged. If the destination tower cannot accept the disk (invalid move), it has no enabled receive, resulting in the multicast send being blocked.

Validate the ported model against properties In the R-CHECK port, we do not need direct counterparts of the TLA+ invariants `TypeOK` and `NoNewElements`, because the R-CHECK's type system already enforces that the variables remain in the defined domains: message fields and variable can't contain values that do not belong their underlying types. Thus, we validate the port using two properties, that can be found in the listing with the model implementation. Property `P1` (Line 74) is the translation of of `TotalConstant` invariant: it checks that the sum of the three tower heads remains constant (3 in our case), which implies that the number of disks within the system remains constant throughout the run. Property `P2` (Line 78) does the same job as `NotSolved` in the TLA+ model. It asserts, that the goal configuration can never be reached. Therefore, once the system reaches the "solved" state, the model checker produces a counterexample trace, which corresponds to a valid solution of the Tower of Hanoi instance.

3.2.3 Semantic differences and justification

The main differences between the original Hanoi TLA+ model and the translation comes from the fact that we move from a global-state style (TLA+) to an agent-centric style (R-CHECK).

Agentification of the puzzle. In the R-CHECK version, we do not describe our whole system as one changing variable (`towers`) and one single transition relation over this variable. Instead, we explicitly introduce agents and make them the central modeling entity. This is the main structural change — the global state is now distributed and the execution is described through local commands and message exchange.

Initiation of transition relation The agent-centric structure also changes how state transitions are carried out. In TLA+, the next move is chosen by the global `Next` action. In R-CHECK, the move is driven by the `Disk` agent — a disk proposes a move by sending a message and the towers process the move using their receive preconditions and local updates. This also represents a semantic shift: we explicitly model the interaction between components that are responsible for the move. For this simple puzzle this is not really important, however, for realistic systems such a semantic shift forces developers of the model to make interaction structures explicit and think about the system in terms of "who can initiate", "who must agree", and "what conditions must hold locally". These are questions that are central in distributed multi-agent systems.

Overall, the main semantic difference in this port is in how the model is structured: TLA+ provides a compact description with a global state and move relation, but R-CHECK transforms the same representation and behavior into explicit agents and communication. This shift is exactly what we wanted to demonstrate in this example — even a non communication-centric models can be represented by R-CHECK, but it makes interaction structure explicit and therefore tends to be more structural than the original TLA+ model.

3.3 Key-value store with snapshot isolation

This example models a simple key-value store that uses a snapshot isolation technique [BBG+95] — each transaction maintains on its own snapshot of the store and performs updates on it. When a transaction closes, a write-write conflict detection is performed and, if no such a conflict is detected, the writes are committed to store. In other case, the writes are rejected.

We see here that the model is already about concurrency and interaction. This is different from the Tower of Hanoi puzzle addressed in the previous section, and makes this model a good candidate to show how we can translate a TLA+ system that is based around state transitions over global variables into a system with an explicit request/response protocol. In the TLA+ model, actions such as opening or closing a transaction are atomic and performed in a single state transition. In R-CHECK, we make communication explicit, so the same logical step becomes a small request/response handshake, which introduces additional states that are not present in original TLA+ model. This modeling choice is intentional — we want to make the interaction structure observable. As the result, the translation to R-CHECK is not a 1:1 mapping of the original TLA+ model.

3.3.1 Original TLA+ specification

The TLA+ specification of the key-value store with snapshot isolation [Hel23] is shown in Listing 5.5. The store is represented by the variable `store` (Lines 15–16), which maps keys to values. Transaction states are tracked in multiple ID-based maps. So, for example, the set `tx` contains exactly the currently open transaction IDs. Opening a transaction means choosing some `t` from the set if available IDs and adding it to `tx`.

In addition to tracking which transactions are open, the specification uses multiple per-transaction data structures:

- `snapshotStore[t]` stores the private snapshot of the committed store that transaction `t` operates on
- `written[t]` records which keys were written by `t`
- `missed[t]` tracks keys whose committed value changed since `t` took its snapshot and that were not overwritten by `t`

Next-state transitions are described by the `Next` action (Lines 114-120), which is a disjunction of possible transaction operations:

1. `OpenTx(t)` opens a new transaction and copies the current committed `store` into `snapshotStore[t]` (Lines 51–56)
2. `Add/Update/Remove` performs changes to `snapshotStore[t]` and update `written[t]` (Lines 58–80)

3. `RollbackTx(t)` closes the transaction without changing the `store` (Lines 82–90)

4. `CloseTx(t)` attempts to commit `t`'s writes into the `store` by checking if `t`'s set of missed writes overlaps with the set of successful writes and rejecting the close if a write-write conflict is detected. On a successful close, the model also updates `missed` for other open transactions to reflect the newly committed writes (Lines 92-112)

The TLC model used for the key-value store specification and its configuration file are shown in Listing 5.6 and Listing 5.7, respectively. The model is instantiated with three keys, three values and two transaction identifiers. The configuration file specifies the `TypeInvariant` and `TxLifecycle` as invariants, both of which are defined in Listing 5.6. `TypeInvariant` (Lines 31–36), as we have seen in the Hanoi example, ensures that the variables keep their expected types throughout the run. `TxLifecycle` property (Lines 38–49) checks the consistency of the transaction lifecycle: first, it ensures that if the committed state of the store is different from a transaction-local snapshot on a key and that key was not written by that transaction, then the key must be marked as missed; second, it checks that inactive transactions are in a clean state — snapshot of such transaction must contain only `NoVal`, and the `written` and `missed` sets must be empty.

All in all, this example again clearly shows the global-state style of TLA+: the state of all transactions is stored in globally shared variables indexed by transaction IDs. This keeps the model compact and declarative, but does not describe interactions explicitly. Instead, interactions are captured indirectly through updates to shared data structures.

3.3.2 Applying the porting recipe

The ported R-CHECK model is presented in Listing 5.8.

Identify agents and roles. As we have seen in the previous examples, in R-CHECK we are required to state all actors of the underlying system explicitly. Thus, as the first step of our translation, we introduce two straightforward agent types: `Store` and `Transaction` (Lines 52–112 and 114–290). This allows us to describe operations as interactions between transaction agents and the store agent, instead of updates to global variables.

Define agent states. In the original TLA+ model, the state of the store and all transaction-related information are stored in global maps indexed by transaction identifiers. In order to represent the same state in R-CHECK, we need to distribute it across the agents. So, in our R-CHECK port, the `Store` agent holds one committed value per key (`s_k*`, Lines 55–57) . Each `Transaction` agent holds an `active` flag and (i) one snapshot value (ii) one write flag (iii) one miss flag per each key (`snap_k*`, `w_k*`, `m_k*`) (Lines 116–125). This preserves the core idea of the underlying system: transactions

operate on local snapshots, tracking which keys they wrote and which keys were missed due to other commits.

Introduce communication. In the original TLA+ model, all the operations on the store are atomic. For example, a `CloseTx(t)` step both updates the store and marks the transaction as closed. In R-CHECK, we intentionally model the interactions between stores and transactions as a small request/response protocol. We introduce a multicast channel `ch` and model transaction lifecycle as `open`/`rollback`/`close` requests followed by `ack` responses from the store (Lines 130–142, 252–268 and 269–289).

In order for this protocol to work, we use messages to carry both control commands and data. A message contains a command tag `CMD` (which can be `open`, `rollback`, `close`, `ack`), the transaction identifier `TID`, snapshot values for each key (`K1..K3`), and per-key write and miss flags (`W1..W3`, `M1..M3`). Upon transaction `open`, the `ack` includes the current committed store values `s_k*`, which the transaction stores as its snapshot. After local modifications, the `close` request carries the transaction's current snapshot values together with the write/miss flags. The store then uses these fields to decide whether to accept the commit and which keys to update.

As in the previous example, we use multicast communication for the purpose of synchronization: a request/ack step is only possible when all the communicating endpoints can perform their corresponding receive/send transitions. For this reason, the transaction agents include "other message" receive branches (for example, `rOtherOpenReq` on Lines 138–142) so that a transaction does not block the channel.

The conflict detection is expressed in the receiving command of the store. For `close`, the store has two kinds of receive commands: rejecting commands that are executed if any conflict condition `W* & M*` holds, and accepting commands that executed only when there is no conflict. This mimics the TLA+ behavior where a transaction can commit only if it did not miss any writes to key that it wrote as well.

A simple diagram in Figure 3.2 depicts a typical scenario in the described protocol. Transaction `tx1` opens by sending an `open` request on channel `ch`. Then, the store receives it and replies with an `ack` carrying the current committed values, which `tx1` stores as its snapshot. After this, a number of local updates (adds, updates, or removals) are performed internally by `tx1` without any network communication. As the result of these updates, write flags are updated accordingly. To commit the values, `tx1` sends a `close` request that includes its snapshot values and the per-key write/miss flags. Then, the store either applies the written keys (in case no conflict is detected) or rejects the close (if a conflict is detected, the store remains unchanged). Finally, the store acknowledges the close request. The other transaction `tx2` consumes the same messages via its `rOther*` receives, and, importantly, updates its `missed` flags upon observing `tx1`'s close request.

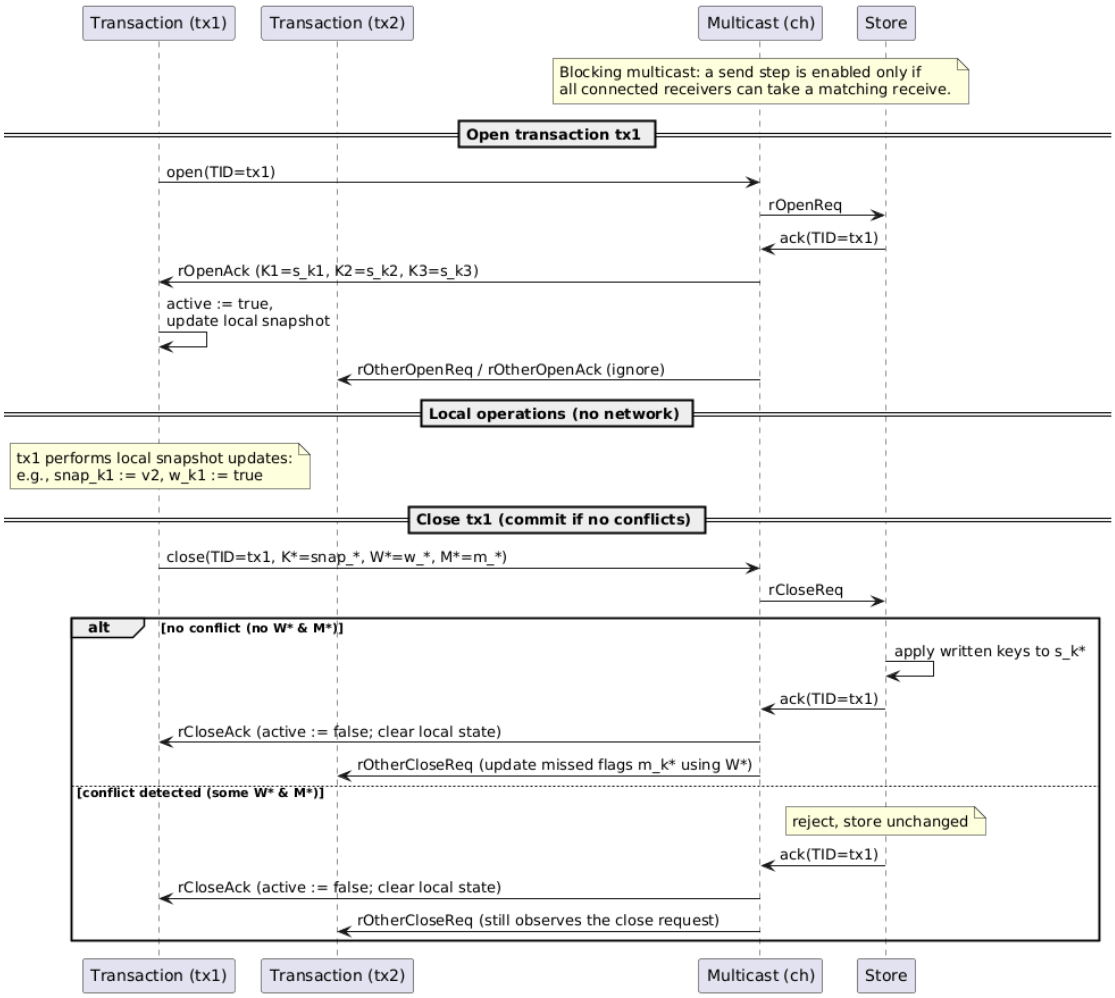

Figure 3.2: Sequence diagram of KeyValueStore R-CHECK model.

Validate the ported model against properties In the R-CHECK port, apart from the original properties, we specify additional properties for the model checker in order to make sure that the introduced protocol works correctly.

Properties `P1`–`P3` (Lines 297–332) check the basic protocol flow: an `open` request is acknowledged and activates the transaction, a `close` request is acknowledged and deactivates it, and an active transaction can always transition into the next state.

Properties `P4` and `P5` (Lines 334–428) check the effect of local snapshot updates and successful closes: a change to a local snapshot implies that it is different from the committed store, and if a transaction closes with changed snapshot without misses, the corresponding store values are updated.

Properties `P6` and `P7` (Lines 430–454) capture the TLA+ property `TxLifecycle`.

Property `P6` says that if a transaction is active, its snapshot differs from the committed store on some key, and that key was not written by the transaction, then the corresponding missed flag is set within this transaction. Property `P7` checks the cleanup part of the lifecycle: once a transaction becomes inactive, all snapshot values must be reset to `NoVal` and all write and miss flags must be false.

The remaining properties `P8`–`P11` (Lines 462–565) check the handling of close requests by other transactions and the store: inactive transactions must not update missed flags; active transactions must update missed flags when observing another transaction's writes; conflicting closes must not change the store; and closes without writes and misses must also leave the store unchanged.

3.3.3 Semantic differences and justification

In a typical distributed system, the essential behavior is expressed through interaction: requests, responses, and their order. Interaction is also the crucial difference between the original TLA+ model and our R-CHECK port. The TLA+ specification captures the system as atomic state transition, while the R-CHECK model makes the interactions explicit.

Atomic actions vs. explicit protocol. In the TLA+ model all the operations — opening, rolling back closing a transaction — are modeled as atomic global actions. However, in a real-world system there is no atomicity: communication can be delayed or parts of it reordered, introducing additional sources for errors and faults. In our R-CHECK port we therefore represent these operations as an explicit request/ack protocol and this is the main semantical difference between the original TLA+ and the R-CHECK model. In addition to that, this approach also helps to localize responsibility as it becomes clear where the functionality must be implemented in order for the model to work. In our case, for example, write-write conflict handling is implemented inside the `Store` agent, which reflects a typical real-world implementation for transactional stores.

In summary, the port models the intended high-level behavior of snapshot isolation for key-value store. The only difference is that it makes the transaction/store interaction explicit. With the help of this translation we see how TLA+ and R-CHECK differs in terms of expressing concurrency: in R-CHECK we express concurrency through interactions between agents, whereas in TLA+ this is done via single next-state action performing updates of the global variables.

3.4 MultiPaxos-SMR

The last port we present in this chapter is a MultiPaxos state machine replication [Lam02, Sch90] model written in PlusCal. We include it because it is a non-trivial distributed algorithm used in practice. It has a clear protocol structure (leader election, quorums, prepare/accept/commit phases), which makes it a good candidate to back up the claim

that R-CHECK is expressive enough to model classic distributed protocols, not only toy examples. In addition to that and most importantly, this port showcases the main semantic difference between the two frameworks: the original TLA+/PlusCal model relies on an asynchronous message buffer, whereas the R-CHECK port uses synchronous communication primitives supported directly by the language.

From a high-level perspective, the Paxos consensus protocol allows a set of replicas to agree on a value under the assumption that message can go missing or be delayed, and that multiple replicas can propose a new value concurrently. The consensus is achieved by electing a leader that then proposes a value to the followers. The protocol uses a ballot number to track the changes of a leader: the replica that wants to become a leader picks a new ballot number, for which a so-called *prepare* phase is executed, during which majority of followers must *promise* to follow the leader. Once the majority is reached, the leader starts the *accept* phase, during which it proposes a value. If then a majority of followers accepts this value, this value is considered to be committed and gets applied by all the replicas. MultiPaxos is an optimization of this idea: once a leader is elected, it can keep proposing the values without the need to repeatedly execute the prepare phase. In the case of state machine replication, the system agrees on a sequence of client commands, where each command is assigned to a slot in the log.

3.4.1 Original TLA+ / PlusCal specification

The original MultiPaxos-SMR model [Hu24] is written in PlusCal (see Listing 5.9), which is compiled into TLA+. In practice, PlusCal is often used for specifications because it makes the control structure explicit and more readable than a pure TLA+ encoding. In our case, this is particularly useful because the algorithm is already structured around communicating parties: the code defines `process Replica ∈ Replicas` (Line 382). A replica is modeled as an explicit action that repeatedly executes steps. Concurrency is expressed by interleaving steps of these replica processes, where at each step a replica non-deterministically chooses one of the available macro actions (e.g., becoming leader, handling prepares, handling accepts, committing, etc.) (Lines 384–400).

A replica can become a leader by choosing a new ballot number and broadcasting a `Prepare` message (Lines 206–229). Replicas that receive a prepare with a higher ballot update their local ballot information and reply with `PrepareReply` messages containing their current votes (Lines 232–249). The leader waits until it has collected a majority of prepare replies, after which it enters the accept phase: it picks a pending command, assigns it to the next empty slot, and broadcasts an `Accept` message (Lines 253–278, 281–306). Followers handle the accept by recording the proposed value as their vote for that slot and replying with `AcceptReply` (Lines 309–325). Finally, when the leader receives a majority of accept replies, it marks the slot committed and broadcasts a `CommitNotice` so that followers can advance their commit index as well (Lines 329–357 and 360–379).

There are two aspects of the PlusCal model that are important for the port. First, the

algorithm uses PlusCal macros such as `BecomeLeader`, `HandlePrepare`, `HandlePrepareReplies`, `HandleAcceptReplies`, etc. These macros are executed atomically in the generated TLA+ code. As a result, intermediate protocol states are abstracted away: a single macro step can both update local state and add multiple messages to the network. This keeps the specification compact, but it also hides parts of the protocol execution that would exist in an implementation. Second, communication is modeled via the shared global variable `msgs`, which represents the network as a set of messages currently en-route in the network. Sending is modeled by adding messages to `msgs`, and receiving is modeled by non-deterministically picking a message from `msgs` and processing it if guards allow it. This results in an asynchronous communication style: messages can accumulate, be processed later, and their delivery order is undefined.

The module used for running the MultiPaxos-SMR specification with TLC and the corresponding configuration file are shown in Listing 5.10 and Listing 5.11, respectively. The model is instantiated with three replicas, two write commands, one read command, and a maximum ballot value of two. The configuration file tells TLC to check the `TypeOK` and `Linearizability` invariants, which are defined in the TLC Module. As usual, `TypeOK` (Lines 22–30) checks that all variables are in the ranges defined by the model. The `Linearizability` property (Lines 67–71) specifies the correctness of the model. In simple terms, it says that once the execution terminates, the sequence of observed client requests and acknowledgments must be representable by a sequential order of commands. This order must both produce the same read results and respect real-time order, meaning that if one command is acknowledged before another command is requested, then the first one must appear earlier in the sequential order.

3.4.2 Applying the porting recipe

The R-CHECK port of the MultiPaxos model is shown in Listing 5.12.

Identify agents and roles. The PlusCal `process Replica` maps directly to an R-CHECK `Replica` agent (Lines 21–257). We instantiate three replicas in the system composition, which is the minimal non-trivial configuration where a majority is meaningful (Line 259–260). For purposes of this thesis, the port focuses only on a single command slot. Extending the model with additional slots would mainly introduce the same state and message field per slot. This is mostly mechanical work and does not change the main discussion points presented in the following sections.

Define agent state and assign ownership. In the TLA+ model, the state of each replica is stored in the global mapping `node[r]`. As it was the case with the previous ports, in the R-CHECK counterpart this global state becomes local state of the defined agents. Therefore, each `Replica` agent keeps the standard Paxos information: the current leader, the maximum known ballot, the prepared ballot, and counters used to track majority votes (Lines 24–29). In R-CHECK, we need to model each command slot explicitly. For this we use per-slot variables: we store the slot status, chosen command,

and vote information (statusS1, cmdS1, votedBalS1, votedCmdS1), together with a counter for accept replies (acceptsS1) (Lines 33–37). Apart from that, the port introduces a small set of bookkeeping variables that are used to implement internal control logic (e.g., selecting a new command to propose) (Lines 42–47). The important point here is that information ownership is now local to the replicas and all the coordination happens via explicit communication between replicas.

Introduce communication. The main point in this port is that communication becomes synchronized. As described previously, the original PlusCal model uses an asynchronous message set msgs: when a send happens, a message is added to a global buffer and upon receiving, a message is non-deterministically picked from that buffer. In R-CHECK, communication is expressed directly using broadcast and blocking multicast commands. In this port we use a single multicast channel ch. The crucial difference is that multicast is *blocking*: a send transition is enabled only if all connected receivers can perform a matching receive step.

From the structural point of view, in the R-CHECK model each protocol phase is implemented as blocks of send/receive commands inside the Replica agent. For example, when a replica want to become leader, it executes sPrepare, which results in a multicast send of the Prepare message (Lines 74–85). Receiving replicas (i.e., future followers) react to this message by executing rPrepare and send the reply using sPrepareReply (Lines 91–115). The leader collects replies by receiving PrepareReply messages (rPrepareReply) and updates its promise counter until a majority is reached (119–161). After the majority is reached, we continue the accept phase by selecting a new command (modeled as internal steps that update cmdS1/vote fields) and multicasting an Accept message (sAccept_S1) (Lines 164–189). Followers then execute rAccept_S1, store the voted value, and respond with sAcceptReply_S1 (Lines 192–214). Then, once the leader receives a majority of AcceptReply messages, it commits the slot and multicasts a CommitNotice (sCommitNotice_S1), which causes followers to commit their slots as well (Lines 217–239 and 242–247).

Because the communication is blocking, the model must ensure that the system never "gets stuck". For this reason, the port includes additional dummy receives that allow a replica to consume protocol messages not relevant for the current phase. In addition to that, leaders use a simple Continue message to signal followers about a phase switch in the protocol.

This overall simplified single-ballot flow is illustrated in the sequence diagram in Figure 3.3. Replica r1 tries to become leader by multicasting a Prepare message, after which r2 and r3 receive it and respond with PrepareReply messages and their current voted information for slot 1 (VOTED_BAL_S1, VOTED_CMD_S1). Once r1 receives the majority of replies, it proposes a command for slot 1 by multicasting an Accept message. After that, followers receive the command and reply with AcceptReply. Finally, after leader receives the majority of accept replies, it commits the value by sending a CommitNotice. This results in the committed command for slot 1.

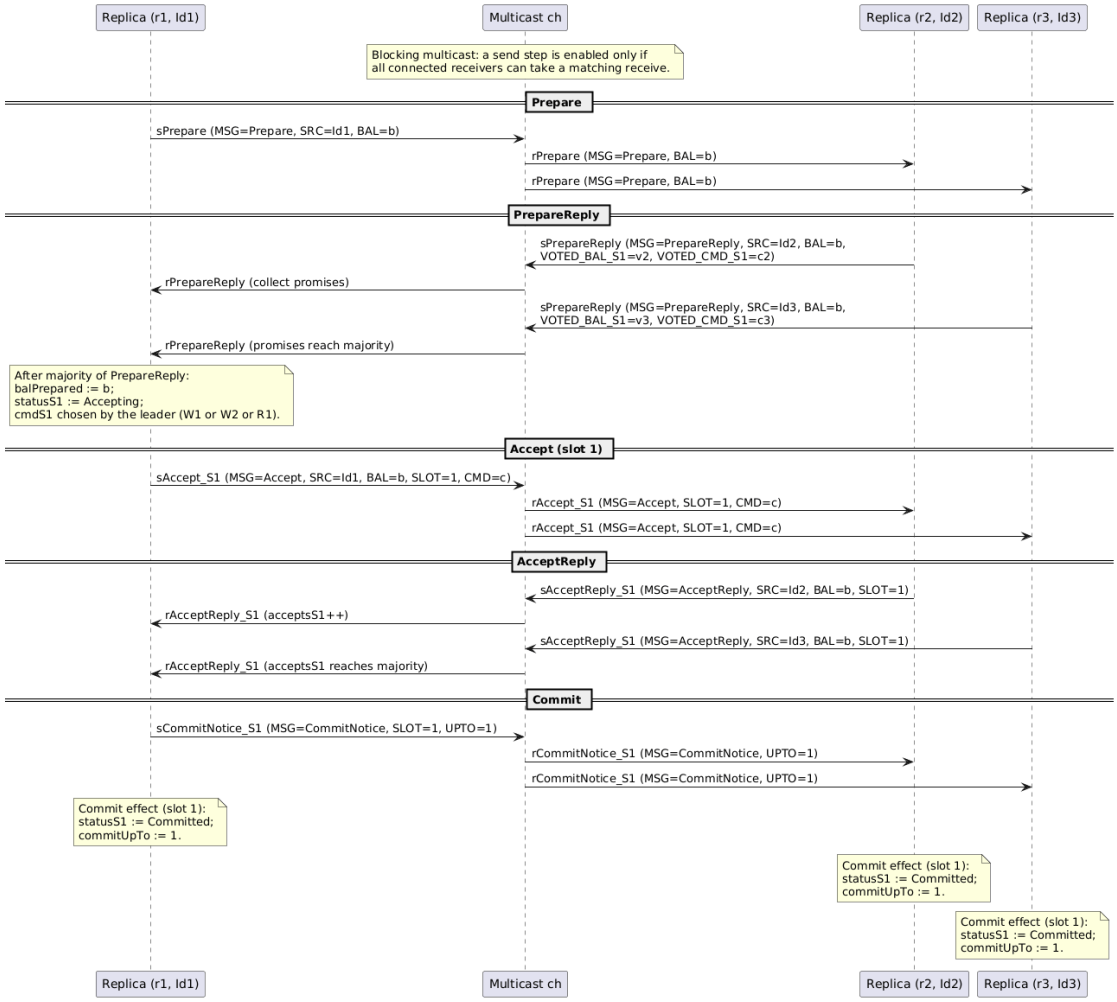

Figure 3.3: Sequence diagram of MultiPaxos-SMR R-CHECK model.

In summary, the port preserves the logic of MultiPaxos, but it does it using a different communication semantics: asynchronous message buffering in TLA+/PlusCal versus synchronized, blocking multicast in R-CHECK.

Validate the ported model against properties We validate our R-CHECK port using a set of protocol-oriented properties instead of directly reusing the original `Linearizability` property from the TLA+ model. There are two reasons for this choice. First, our R-CHECK model is intentionally simplified to a single slot, so the linear order here is trivial. Second, the original `Linearizability` property is much easier to express in TLA+ because of the nature of the language: it contains set-based definitions over global variables, and thus it is relatively easy to track the observed events. In R-CHECK, checking the same property directly would require introducing an

additional observer agent that records requests and acknowledgments and reconstructs the corresponding event history. For this reason, in the R-CHECK version we instead rely on the command labels feature of the language and check the correctness of the protocol flow in an even-based manner. In properties `P1`–`P6` (Lines 266–310) we verify the main stages of the protocol: a `Prepare` is received by the other replicas; the replicas always reply to `Prepare` with `PrepareReply`; a majority of prepare replies results in the leader taking a new request and sending the `Accept` message; the replicas always respond to `Accept` with `AcceptReply`; a majority of accept replies results in the leader sending `CommitNotice` and committing the slot; the replicas eventually reach the committed state. Although we do not check linearizability directly, we still verify that the main MultiPaxos protocol phases are correctly executed in the R-CHECK model.

3.4.3 Semantic differences and justification

This port is intentionally not 1:1. The main differences are communication and atomicity.

Asynchronous vs. synchronous communication. In the PlusCal model, the communication between replicas is represented as an asynchronous set of messages. In the R-CHECK port we use synchronous, blocking multicast commands. This changes the execution model: a state transition is only possible when all involved replicas have a corresponding receive transition. The port therefore introduces dummy receives and continuation steps to avoid deadlocks. Achieving asynchronous behavior in R-CHECK would require introducing an explicit agent for buffering the messages. For the purpose of this thesis, we keep the model focused on demonstrating feasibility of encoding the protocol structure rather than reconstructing the asynchronous communication.

Atomic macros vs. explicit intermediate steps. PlusCal's macro execution is atomic from the TLC model checker point of view. This means that intermediate protocol states are hidden (like choosing a next command for the slot). In the R-CHECK port we make these intermediate states visible. This makes the port more verbose and introduces additional implementation details, but it is also closer to how an implementation behaves, where protocol steps are not truly atomic.

Overall, this case study demonstrates that R-CHECK is expressive enough to model a real distributed consensus protocol. The price is that we must make a clear semantic decision regarding communication (asynchronous vs. synchronous) and we must refine atomic macro steps into explicit sequences. The payoff is that the resulting model exposes protocol events directly via labels, making it straightforward to state and check interaction-level properties over the protocol flow.

3.5 Discussion

3.5.1 Pitfalls

We saw that performing a translation from TLA+ to R-CHECK is not a mere task of syntactic conversion. Even in the cases where the overall behavior of the underlying system is preserved, there is a change in the modeling style. First, it changes from transitions over the global state to agent-local state: we must distribute the state across the agents and make explicit decisions about which agent stores which information. Second, the way state transitions work is different between the two frameworks: in TLA+ a state transition is defined by actions that change the global state, whereas in R-CHECK it is defined by the communication between agents. In this subsection we want to summarize the pitfalls we observed. Some of them result from the shift in the modeling paradigm, others are just limitations of the current state of R-CHECK language.

Distributed state and information duplication (Hanoi). The Tower of Hanoi Puzzle Model modeled in TLA+ remains compact because it is structured around a single global variable `towers`. In our R-CHECK port, same information is distributed across the agents: each `Tower` agent needs to know which disks it holds, and each `Disk` agent needs to know its current location. This leads to the fact that information in the model becomes duplicated and the main pitfall with such duplication is that it must be kept consistent. In case a disk updates its location, but the tower does not update its disk stack, the model can end up in impossible states. We solved this problem by encoding the moves as a multicast communication step, where all involved agents update their states together. In general, once global state is removed, we need to explicitly address such consistency problems. An additional, but slightly less important pitfall is the lack of high-level containers and data operators in R-CHECK, which make the puzzle easy to express in TLA+. In Hanoi, move checks are expressed using sequence operators such as `Head` and `Tail`. In the port, we had to implement equivalent "top of the stack" logic explicitly. This is not a big issue, but it is boilerplate code that comes from the fact that R-CHECK currently does not provide a support for containers and operations on them.

Atomic actions become explicit communication steps (KeyValueStore and MultiPaxos). Another pitfall that was encountered when working on the KeyValueStore and MultiPaxos ports is that TLA+ actions (and PlusCal macros) can change multiple parts of the system atomically. For example, in the KeyValueStore TLA+ model, operations such as open/close/rollback are single atomic transitions updating the global data structure. In the R-CHECK port, these operations become a request/ack protocol, introducing intermediate states that are not present in the original TLA+ model. This deliberate shift introduces complexity into the model: we need to define message structures, develop the protocol, and make sure that we never end in a deadlock state when using the multicast communication in R-CHECK. The same issue is present in the MultiPaxos model: PlusCal macros can update multiple local variables and send

multiple messages as one atomic step. When porting to R-CHECK, the macro logic must be split into multiple steps, which must be done with care. Overall, the pitfall in such cases is that with the shift from TLA+ to R-CHECK, the granularity of atomicity changes and the new model must be checked for unintended state transitions.

Lack of collection types leads to unrolling (KeyValueStore and MultiPaxos). Both KeyValueStore and MultiPaxos show another a practical limitation: the lack of support for basic collection types (sequences/sets/dictionaries) leads to the fact that parts of the model in R-CHECK must be unrolled into set of per-key/per-slot variables. In the KeyValueStore port, instead of a `Key -> Val` mapping, we explicitly introduce `snap_k1/snap_k2/snap_k3` and the corresponding `w_k*/m_k*` flags. In MultiPaxos, slot state is represented by per-slot variables such as `statusS1`, `cmdS1`, and vote fields. This unrolling is straightforward, but it still increases the size of the model and the number of places in the code that must be adapted when extending the model. This is not a limitation of R-CHECK/ReCiPe paradigm itself, but rather a limitation of the current state of the language. We still document it, as it explains why some ports may look blown up compared to the TLA+ counterparts.

The KeyValueStore port shows another pitfall connected to the lack of support for basic collection types. In TLA+, the rule applied before a commit is a compact combination of set operations (e.g., checking `missed[t] ∩ written[t] = {}` and merging written keys by quantifying over `k ∈ Key`). In the R-CHECK port, the store's `close` handling must cover combinations of write flags `W*` and miss flags `M*`. For an instance of this size, this is manageable. The pitfall here is a potentially lower maintainability: covering all combinations by hand is error-prone.

Synchronous communication requires "dummy" branches (Hanoi, KeyValueStore and MultiPaxos). Last but not least, when using blocking multicast, a communication can be blocked if one for the connected agents has no enabled and matching receive transition. In the original TLA+ model this does not represent a problem, because this blocking message will simply remain in the set of messages until the receiver eventually consumes it. In the R-CHECK ports, on the other hand, we used blocking multicast communication, so we must make sure that every connected receiver has an enabled matching receive step when a send occurs. For this reason, we introduce explicit "dummy" receive branches that allow agents to consume messages that would otherwise block the whole model. An alternative and more idiomatic approach would be to introduce multiple channels and use R-CHECK's receive-guards to dynamically "disconnect" agents from channels when they are not interested in receiving. Compared to the dummy-receive pattern, however, this approach would introduce additional channel-management logic and increase the semantic distance from the original TLA+ models even more. Thus, we decided to use dummy receives in our examples. This pattern appears in all of our ports, and the main pitfall is that mistakes in such logic are easier to overlook when modifying or extending the model.

To summarize, the presented pitfalls identify where the effort occurs when moving from global-state TLA+ to agent-centric R-CHECK models: distributing state, making protocols explicit, solving the problem of missing collection datatypes, and handling the consequences of synchronous communication.

3.5.2 Takeaways

The ports that we presented in this chapter show different aspects that must be taken care of when translating a model from TLA+ to R-CHECK. They support the main point of this chapter: TLA+ specifications can be ported to R-CHECK, but the effort and the resulting semantic differences are strongly defined by the structure and nature of the original model.

Agentification turns global updates into explicit responsibilities of the agents. We saw that, in R-CHECK, global variables become distributed agent-local variables. This forces developers to think about who owns the state and where certain actions and rules must be implemented (e.g., conflict detection and commit acceptance inside the store in the case of KeyValueStore). This demonstrates the main strength of the agent-centric view: it makes system structure and responsibilities of separate components of the underlying system explicit.

The choice of communication model becomes central design decision. The MultiPaxos port demonstrates that R-CHECK can encode a real non-trivial distributed consensus protocol. At the same time, it is also the clearest example showing that porting is not always a straightforward 1:1 copy. The original PlusCal model uses an asynchronous message buffer and atomic macros, whereas the R-CHECK port uses synchronous blocking multicast. As a result, the port has the same high-level protocol flow, but the internal execution semantics is a bit different. This represents another important takeaway: for communication-heavy models, the main task lays not in a simple translation of one syntax into another, but in choosing and documenting how communication between agents happens.

Some ports require adapted properties. Another important phase in the translation process is the validation of the resulting translations. Because the R-CHECK models are not always 1:1 copies of the original TLA+ specifications, it may not always be possible to transfer the original properties directly. Still, whenever this can be done naturally, we reuse the original TLA+ properties in the R-CHECK model. Otherwise, we add either adapted or more detailed protocol-level properties that check the intended behavior of the ported system. This gives an additional safety net that the port still captures the relevant behavior of the original model, even when the internal structure is different.

Table 3.1 summarizes the three ports and the main modeling decisions behind them. Overall, the ported examples show that TLA+ specifications can be translated to R-

Model	Port type	Key modeling decision	Main difference
Hanoi	close, agentified	Agents for `Tower` and `Disk`; move validity enforced via receive-guards (blocking multicast)	Introduction of agents and explicit interaction
KeyValueStore	agentified	Replace atomic actions by an explicit request/ack protocol between transactions and a store	Introduction of a protocol with intermediate states
MultiPaxos-SMR	semantic-shift	Replace async `msgs` set by synchronous blocking multicast; split atomic macros into explicit sequences	Async → sync communication shift; simplified scope (single slot in current port)

Table 3.1: Summary of the three ports and their main semantic choices.

CHECK, but the resulting semantic differences depend on the structure of the original model.

CHAPTER 4

R-CHECK features in TLA+

Main design aspect of R-CHECK is communication-centrism: communication primitives and the support for reconfiguration are the main modeling concepts. During the survey of available TLA+ case studies and example specifications, we did not find a model that ideally highlights these features and that is both concise enough and focuses on dynamic topology of the underlying communication. Therefore, we take a different route: we implement a minimal example of MAS in parallel in R-CHECK and in TLA+ step-by-step.

The goal of this experiment is not to show that one framework is universally better than the other. The goal is to show what effort it takes to express communication-centric MAS patterns in both tools. In essence, we discuss what is native in R-CHECK, what needs to be encoded (and maintained) in TLA+ explicitly. The evaluation is structured as a small running example: each step adds one feature or concept, and along the way we compare the resulting R-CHECK and TLA+ models.

4.1 Features to showcase

The running example we are going to present is specifically chosen to showcase the following R-CHECK features:

- **Agents and system composition.** Local state, initialization, state updates, and composition of a system consisting of multiple agents running in parallel.
- **Broadcast communication.** Non-blocking "announcement"-like communication.
- **Multicast communication.** Blocking multicast communication over dedicated channels.

- **Connection reconfigurability for multicast.** The ability for multicast endpoints change the channel in use.
- **Common variables, sender predicate, and relabeling.** Sender-side constraints over possible receivers.
- **Label-based LTL properties.** Writing and model checking LTL properties over command labels (system events) rather than only over state variables.

4.2 Running example: publishers and subscribers

4.2.1 Overview

The running example is a simple system where participants communicate using the publisher/subscriber communication pattern [RGS95]. Publishers periodically announce themselves via the broadcast channel. Subscribers receive announcements and then subscribe to a publisher. After subscription is carried out, a publisher pushes its internal state to subscribers using a dedicated multicast channel. The publisher state is a small cyclic state machine to keep the example simple. (i.e., $S0 \rightarrow S1 \rightarrow S2 \rightarrow SWITCHING \rightarrow S0 \rightarrow ...$).

To demonstrate reconfiguration, our system is then extended with a second multicast channel, which enables subscribers to switch their communication link when transitioning to the last state of the state machine. Each subscriber is notified about the intended change prior to this switch and must update its connectivity configuration accordingly. This introduces changing communication topology into our model.

As the final step, we show common variables, relabeling and sender predicates features of R-CHECK. We extend our system by a notion of access control: each subscriber has an access level and each publisher has a minimum required access level that is expected from the potential subscribers.

The sequence diagram in Figure 4.1 shows the overview of the system's behavior. For the sake of simplicity, we use only one instance of each agent type.

We finalize our example by model checking a small set of LTL properties. The properties we are going to check are written in an event-based style: instead of referring to the state variables in our system, we refer to concrete communication events like "publisher announces" or "subscriber received announcement". We will see that, with the help of R-CHECK's labeled commands, this becomes very direct: each label becomes a Boolean predicate that is true exactly on transitions where the labeled command fires.

Throughout the construction, we evolve the model in small steps. After each step, we discuss what the feature looks like in R-CHECK and what additional modeling effort is needed in TLA+. We particularly focus on the boilerplate code and the number of places that must be updated when the model changes.

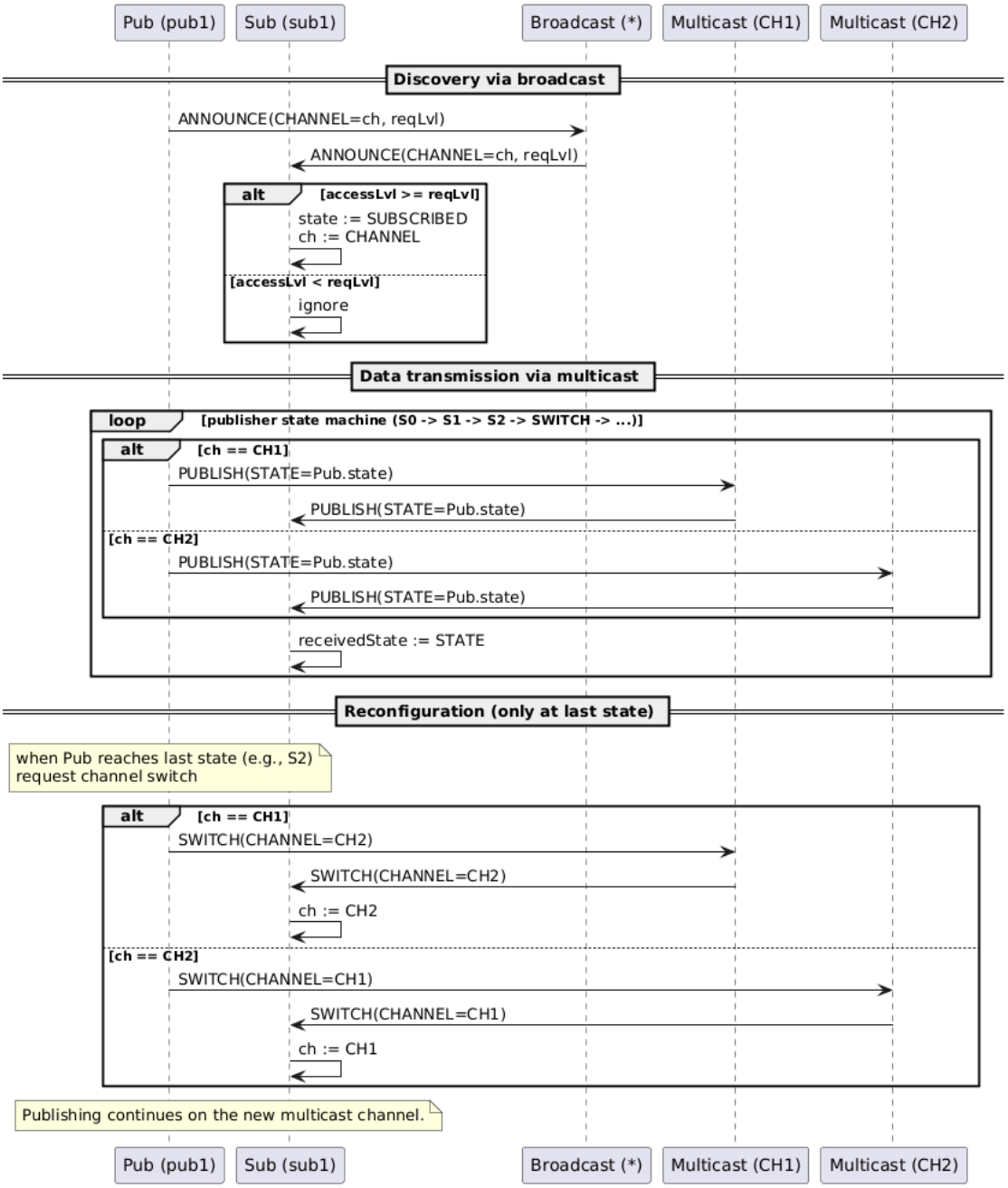

Figure 4.1: Sequence diagram of the running *PubSub* example.

4.2.2 Milestones and comparison protocol

As said earlier, we use an example where we start from a minimal baseline and add functionality to it in small steps. Each step introduces exactly one new feature or coupled

set of features. This allows us to assess the modeling effort of the proposed functionality to a specific feature.

We structure the running example into the following milestones:

- $PubSub_0$**, Base model.** A minimal system with one publisher and one subscriber agent.
- $PubSub_1$**, Broadcast announcement.** The publisher announces itself via the broadcast channel and starts a small cyclic internal state machine (i.e., $S0 \to S1 \to S2 \to S0 \to ...$). The subscriber listens on the broadcast channel and records the announcement.
- $PubSub_2$**, Multicast channel and publishing.** After discovery, publisher/subscriber communication runs over a dedicated multicast channel. The publisher pushes its internal state over this channel, and the subscriber receives and stores it. This milestone introduces the notions of channel connectivity and blocking multicast.
- $PubSub_3$**, Reconfiguration.** A second multicast channel is introduced and publisher's logic is extended with non-deterministic channel switch decision. As the result of the switch, subscribers must update their local connectivity configuration. This changes the system's communication topology.
- $PubSub_4$**, Sender Predicates and Relabeling** At this step we introduce a simple access control logic. Publishers now predefine the minimum required access level required from the subscribers.
- $PubSub_5$**, Model Checking** This milestone introduces basic checks for our system. We define a number of temporal properties and verify them using the respective model checkers.

For each milestone $PubSub_i$, we construct two models: one in R-CHECK and one in TLA+, such that both capture the same behavior. After that we measure what effort needs to be put in to come from $PubSub_{i-1}$ to $PubSub_i$. We use the following simple metrics:

- **State representation size.** For each milestone we count how many state-modeling constructs are introduced. In TLA+ this includes new state variables and record fields. In R-CHECK this includes new agent-local variables and message fields.
- **Behavior representation size.** For each milestone we count how many new behavior-modeling constructs are introduced. In TLA+ this includes new actions and predicates that are required to model particular feature. In R-CHECK this includes new agents and commands.

- **Touch points.** For each step from $PubSub_{i-1}$ to $PubSub_i$ we count how many distinct model blocks must be edited. This allows us to assess how scattered modifications are in case of evolving model. In TLA+, we treat each variable/constant/operator definition as separate blocks. In R-CHECK, we treat commands, enums, message-structure definitions, system compositions, etc. as separate blocks.

With the help of the defined metrics, we want to show each model's complexity and the effort required to introduce changes to the model. State representation size acts as a proxy for assessing the amount of information that needs to be tracked to describe the configuration of the system. With the growing state size, the cognitive load needed to understand and maintain the model increases.

Behavior representation size approximates the number of distinct places in the model that describe the system's transition logic — additional transitions in the system introduce additional operators that must be kept consistent and up-to-date with the evolving model.

Last but not least, touch points describe how scattered is a change across the system. High number of touch points for a change means that such a change is harder to implement and to reason about, because a deep and scattered understanding for the underlying system is required. Complexity and scattering of changes leads to error-proneness and can be used as a predictor of errors [Has09]. This leads to the fact that fewer touch points indicate higher evolvability of the underlying model, because addition of new features requires fewer adaptations across the whole model.

In the remainder of this chapter we present the models and results for each of the milestones.

4.2.3 Base model: $PubSub_0$

In the first model $PubSub_0$ we establish a minimal working base, so in this milestone we do not model any meaningful behavior yet: we simply set up the basic building blocks — agents — that are later used in the system composition.

R-CHECK. The R-CHECK code for $PubSub_0$ is presented in Listing 5.13. It defines two agent types, `Pub` and `Sub`, with trivial behavior: both agents execute a dummy broadcast command. We then compose our system from two agent instances — one publisher and one subscriber. After applying our metrics we get the following result:

- **Δ State representation size**
  0 agent-local variables; 0 message fields.

- **Δ Behavior representation size**
  +2 agents (`Pub`, `Sub`)
  +2 commands (one per agent).

- **Touch points**
  Not applicable (no previous version).

TLA+. The TLA+ code is shown in Listing 5.14. Since TLA+ does not provide an explicit *agent* construct, we encode agent populations as state variables. Concretely, we use two state variables `Pubs` and `Subs`, modeled as functions from instance identifiers (simple integers) to local state. After applying our metrics we get the following result:

- **Δ State representation size**
  +2 state variables (`Pubs`, `Subs`); 0 record fields.
- **Δ Behavior representation size**
  +5 operator blocks (`Init_Pub`, `Init_Sub`, `Init`, `Next`, `Spec`).
- **Touch points**
  Not applicable (no previous version).

A key difference already visible in $PubSub_0$ is that agents are a first-class concept in R-CHECK. The model explicitly declares `agent Pub` and `agent Sub` with their own local context (`init`, `receive-guard`, `repeat`), and system structure is expressed directly via parallel composition of agent instances - `system = Pub(pub1, true) || Sub(sub1, true)`. In contrast, TLA+ does not provide an agent construct out of the box. So agent construct must be encoded explicitly, for example, by representing collections of instances as state variables, which are functions over instance identifiers.

4.2.4 Broadcast announcement: $PubSub_1$

In $PubSub_1$, a first communication feature is introduced: broadcast announcements. The aim of this milestone is to model a simple publish/subscribe discovery behavior: a publisher announces itself on the broadcast channel, and a subscriber changes its local state in reaction to the announcement.

R-CHECK. The R-CHECK model (Listing 5.15) introduces simple local state machines and a communication scheme for both agent types (`PubState` and `SubState`). The modeled behavior is the following: The publisher starts in `IDLE` and executes a broadcast command `sAnnounce` on the broadcast channel `*`, which sets `MSG := ANNOUNCE` and moves the publisher to the next state (`state := S0`). After that, the publisher continues to evolve its internal state (`S0/S1/S2`) using additional always-enabled steps. The subscriber starts in `WAITING` and contains a single receive command `rAnnounce` on the broadcast channel. Upon receiving an `ANNOUNCE` message while in `WAITING`, it transitions to `SUBSCRIBED`. In essence, we use broadcast as a non-blocking announcement mechanism in this example: the publisher can always announce itself, and the subscriber only reacts when it receives the announcement. Listing 5.16 shows a diff view of $PubSub_1$ diffed against $PubSub_0$. After applying our metrics we get the following result:

- **Δ State representation size**
  +2 agent-local variables: Pub (1x), Sub (1x)
  F +2 message fields: `MSG`, `STATE`

- **Δ Behavior representation size**
  +3 commands: Pub (3x)

- **Touch points**
  11 blocks touched: updated enums, updated message structure, added local state variables (2x), initialized local state variables (2x), changed/added command (5x).

TLA+. In the TLA+ model (Listing 5.17), we encode publisher and subscriber populations in variables `Pubs` and `Subs` as functions that map instances to records holding the local states of agents, currently consisting only of a `state` field. The milestone introduces helper operators for initialization (`Init_Pub`, `Init_Sub`), send/receive preconditions (`SendPrecond_Pub`, `RecvPrecond_Sub`), and local state updates (`StateChange_Pub`, `StateChange_Sub`) as well as the internal publisher state machine (`NextState_Pub`). Broadcast communication is encoded as a special operator `BroadcastComm(m)` that fist picks an enabled sender and then updates both populations in the next state. Listing 5.18 shows a diff view of $PubSub_1$ diffed against $PubSub_0$. After applying our metrics we get the following result:

- **Δ State representation size**
  +2 record fields: Pubs (1x), Subs (1x)

- **Δ Behavior representation size**
  +7 actions: `SendPrecond_Pub`, `NextState_Pub`, `StateChange_Pub`, `RecvPrecond_Sub`, `StateChange_Sub`, `BroadcastComm`, `Announce`.

- **Touch points**
  11 blocks touched: updated list of constants, updated initialization action (2x), added communication actions (7x), updated next state action.

In this example, both specifications model the same simple idea: one agent announces on a broadcast channel and another agent reacts to the announcement. The effort it takes to model this behavior, however, looks different. In R-CHECK, broadcast communication is a language primitive, so the announcement and reaction to the reaction are expressed with only a small number of additional declarations. In TLA+, the same behavior is encoded as part of the global state, which forces us to introduce several helper operators for preconditions, state updates, and the broadcast communication step. This is not a disadvantage of TLA+, but for this communication-centric model, the R-CHECK model is closer to the intended interaction and requires less bookkeeping on the side.

4.2.5 Multicast channel and publishing: $PubSub_2$

In $PubSub_2$ our running example is extended with the multicast communication behavior. The broadcast announcement from the previous step is still used for discovery, with the following additional functionality: along with sending the announce message, the publisher also sends the channel identifier that it will use after the discovery as a dedicated multicast channel for data exchange. This example presents two coupled R-CHECK concepts: blocking multicast over channels, and explicit connectivity via the `receive-guard`.

R-CHECK. The R-CHECK model (Listing 5.19) adds a new channel type enum `Channels` with the multicast channel `CH1` and a default value `NULL`. The message structure is extended with a `CHANNEL` field, so that the publisher can announce on which multicast channel it is going to publish.

The key idea is simple: the publisher stores its multicast channel in a local variable `ch` (initialized to `CH1`). In the announcement step `sAnnounce`, the publisher broadcasts not only `MSG := ANNOUNCE` but also `CHANNEL := ch`. The subscriber receives the announcement in `rAnnounce` and stores the channel locally (`ch := CHANNEL`). From this point on, the publisher publishes its internal state via multicast sends on `ch!`, while the subscriber receives via `ch?` and stores the received state in `receivedState`.

Connectivity is modeled explicitly using the `receive-guard`. The subscriber uses `receive-guard: chan == ch`, meaning that it is connected exactly to the multicast channel stored in its local variable `ch`. Before the announcement, `ch == NULL`, so the subscriber is not connected to `CH1`. After storing the announced channel, the subscriber becomes connected and can participate in multicast communication. Overall, the multicast/channel part in R-CHECK stays local and explicit: define a channel value, send/receive on that channel, and define connectivity via the guard. Listing 5.20 shows a diff view of $PubSub_2$ diffed against $PubSub_1$. After applying our metrics we get the following result:

- **Δ State representation size**
  +3 agent-local variables: Pub (1x), Sub (2x)
  +1 message field: `CHANNEL`.
- **Δ Behavior representation size**
  +1 command: Sub (1x)
- **Touch points**
  13 blocks touched: updated enums, update message structure, added local state variables (2x), initialized local state variables (2x), updated receive guard (1x), changed/added commands (6x).

TLA+. The TLA+ model for $PubSub_2$ (Listing 5.21) extends $PubSub_1$ with explicit multicast publishing and channel-based connectivity.

Equivalently to `BroadcastComm`, we introduce a second communication operator to our model — `MulticastComm`. While both operators implement almost the same pattern (choose an enabled publisher, update agent populations), the multicast case a bit more complex. It must encode the blocking feature of multicast explicitly by requiring that all connected receivers satisfy the receive precondition. This is reflected in the quantification over subscribers inside `MulticastComm`, together with an explicit connectivity predicate `ConnCond_Sub(j, c)` (Lines 56-58).

Another effect is that previously simple local predicates become mixed across communication modes. In particular, `SendPrecond_Pub` and `RecvPrecond_Sub` are extended to cover multiple message types and multiple phases (announce and publish). This mixes separate protocol steps in a single predicate. Similarly, since subscriber updates now depend on the type of the received message, `StateChange_Sub` is generalized to take the message type as a parameter. As a result of these extensions, additional indexing and parameter passing is introduced into the specification (`i`, `j`, `k`, `m`, `c`, `s`), making the separation between local agent behavior and global coordination blurred. Listing 5.22 shows a diff view of $PubSub_2$ diffed against $PubSub_1$. After applying our metrics we get the following result:

- **Δ State representation size**
  +3 record fields: Pubs (1x), Subs (1x).
- **Δ Behavior representation size**
  +4 actions: `ConnCond_Sub`, `StateChange_Sub`, `MulticastComm`, `Publish`.
- **Touch points**
  11 blocks touched: updated list of constants, updated initialization action (2x), changed/added communication actions (7x), updated next state action.

When comparing the two models, we see that the R-CHECK model keeps the multicast behavior explicit and local: sending and receiving is expressed as an operation on a named channel (`ch!` and `ch?`), while connectivity condition is defined directly in the subscriber's `receive-guard`. In TLA+, the same concepts are fully expressible, but they must be encoded as part of the global transition relation, which introduces additional complexity requiring us to add new predicates for connectivity, message-based state update, and coordination logic. It also needs to be mentioned, that the entanglement of the changes introduced in the TLA+ is higher due to the need to keep track of the indices and new parameters.

4.2.6 Reconfigurable multicast channels: $PubSub_3$

In $PubSub_3$ we further evolve our $PubSub_2$ by adding reconfiguration to the system: subscribers can now switch between two multicast channels upon command. The publisher can instruct subscribers to switch between `CH1` and `CH2`. Subscribers must then update

their local connectivity configuration accordingly. The goal of this change is to introduce a simple dynamic topology.

R-CHECK. The R-CHECK model implements a second channel type `CH2` and a new message type `SWITCH` (Listing 5.23). Compared to $PubSub_2$, the publisher's send commands now explicitly set `MSG := PUBLISH` together with the payload `STATE := state`. In addition to that, the publisher instructs subscribers to switch the channel when reaching `S2` state. We still use multicast commands, where each command sends a `SWITCH` message on the current channel and adds the new channel to the message's payload. At the same time, the publisher updates its channel. The subscriber now implements the switch receive command — `rSwitch`. When executing it (see pre-condition), it also updates its local `ch` to the received one, resulting in receive guard immediately taking effect — `receive-guard: chan == ch`. This perfectly shows how reconfiguration remains fully local to agents: it is expressed as another send/receive command and connectivity is achieved via receive-guard and channel value stored in a variable. Listing 5.26 shows a diff view of $PubSub_3$ diffed against $PubSub_2$. After applying our metrics we get the following result:

- **Δ State representation size**
  No changes.
- **Δ Behavior representation size**
  +3 commands: Pub (2x), Sub (1x)
- **Touch points**
  5 blocks touched: updated enums (1x), changed/added commands (4x).

TLA+. The TLA+ $PubSub_3$ model is shown in Listing 5.25. The most crucial point of this change is that publishers now update their channel state variable `Pubs[i].ch` when a `SWITCH` message type is passed to `StateChange_Pub` function, and subscribers update their channel state variable `Subs[i].ch` when a `SWITCH` message type is passed to `StateChange_Sub` function. This means that reconfiguration is added by introducing another message type and injecting it in the global transition relation. 5.26 shows a diff view of $PubSub_3$ diffed against $PubSub_2$. After applying our metrics we get the following result:

- **Δ State representation size**
  No changes.
- **Δ Behavior representation size**
  +1 action: `Switch`.
- **Touch points**
  10 blocks touched: updated list of constants, changed/added communication actions (8x), updated next state action.

Compared to the R-CHECK model, the main difference from the TLA+ model is where the reconfiguration logic lives. In R-CHECK, reconfigurability is modeled as an explicit interaction between communicating parities (in our case this is the `SWITCH` send and a corresponding receive). Connectivity to a specific channel is derived from the `receive-guard`. In TLA+, this all is expressed by updating shared, parametrized functions and reusing the global operators. This introduces coupling between different phases of our protocol and adds extra indexing in the specification.

4.2.7 Access control via Sender Predicates and Relabeling: $PubSub_4$

In $PubSub_4$ we add one more feature to our model — sender predicates over common variables and relabeling. With the help of this feature we want to implement a simple access-control functionality: every subscriber now has an access level, every publisher specifies the required minimum access level and is discoverable only if the subscribers' access level satisfies this requirement.

R-CHECK. The R-CHECK model in Listing 5.27 introduces a common variable `accLvl` inside the `property-variables` block. It also extends the local variables of the subscribers with a local access variable `accessLvl`. Additionally, we define a trivial relabeling `accLvl <- accessLvl`, which just maps the value of accessLvl (local) to the accLvl property variable, making it available in the sender predicate section of other agent types.

Inside the publisher's agent type, we also add a new local variable `reqLvl`, which stores the required access level. Last but not least, inside the announcement step `sAnnounce`, we use the sender predicate `(@accLvl >= reqLvl)`. This results in the fact that only subscribers whose relabeled `accLvl` satisfies the c`(@accLvl >= reqLvl)` can see the broadcast message.

Finally, we use agent-specific initialization predicates inside the `system` composition for setting the added local variables to the desired values. Listing 5.28 shows a diff view of $PubSub_4$ diffed against $PubSub_3$. After applying our metrics we get the following result:

- **Δ State representation size**
  +2 agent-local variables: Pub (1x), Sub (1x).
- **Δ Behavior representation size**
  No changes.
- **Touch points**
  6 blocks touched: added property variable, added local state variable (2x), added relabeling rule, changed announce command, updated system composition.

TLA+. The TLA+ model for $PubSub_4$ in Listing 5.29 does the same as the R-CHECK one — adds simple acccess control. In contrast to R-CHECK, we do not need to implement

special relabeling mechanisms here due to the global state of TLA+: the required access levels and set access level variables are directly accessible from the global sets. Publishers get `reqLvl`, and subscribers get `accLvl` added to their states.

In the next step, we adapt agent initialization in TLA+ – we now want to initialize agent instances differently and need a mechanism to do so. Therefore, we introduce set initialization comprehensions `Init_Pubs` and `Init_Subs`, inside which we can cherry-pick the required agents with `EXCEPT` combined together with identification number. This is how we get instance-specifics values.

To finish the implementation, we update the behavior by integrating the access control logic into the broadcast receive logic of subscribers: we add a new parameter to the operator and add the guard `Subs[i].accLvl >= reqLvl` inside the announce case. Listing 5.30 shows a diff view of $PubSub_4$ diffed against $PubSub_3$. After applying our metrics we get the following result:

- **Δ State representation size**
  +2 record fields: Pubs (1x), Subs (1x).
- **Δ Behavior representation size**
  +2 actions: `Init_Pubs`, `Init_Subs`.
- **Touch points**
  6 blocks touched: update initialization actions (3x), changed/added communication action (3x).

When comparing both implementations, we can say that In R-CHECK, access control is expressed directly at the sender side as a predicate over a common variable with relabeling on the receiver side, keeping the logic agent-local. In TLA+, the same is achieved by updating the global state and wiring the access level values into the receive operator, which introduces additional parameter passing. This also centralizes the protocol logic in shared global operators.

4.2.8 Model checking the system: $PubSub_5$

In the previous section we mainly focused on expressing system's behavior, and, as the next step, we would like to address its correctness. We will do so by specifying a small set of temporal properties and check them with the respective model checkers. The point of this section is to show, that writing simple communication-centric properties in R-CHECK is really simple due to the labeling mechanism that it supports - we can talk about concrete communication events and their consequences.

R-CHECK properties R-CHECK exposes command labels as boolean predicates in the model checker, which makes it really convenient to express properties as "if event 1 happens, then eventually event 2 happens and some predicate on the state variables

holds". Listing 5.31 shows the properties we introduce to our model. Below we briefly explain them:

1. **P1: announcement results in subscription if access constraint holds** Whenever `pub1` performs an announcement and the subscriber's access level is sufficient, the subscriber must eventually execute the corresponding receive (`sub1-rAnnounce`) and end up subscribed on the same channel as the publisher.

2. **P2: subscribed implies connected.** This is a safety-style property: it should never be possible to be in `SUBSCRIBED` while no channel is selected.

3. **P3: publish results in a receive.** Whenever the publisher performs one of its publish steps (`s0/s1/s2`), the subscriber must eventually perform the corresponding receive step `rPublish`.

4. **P4: switch CH1 request leads to channel change at subscriber.** If the publisher requests a switch while on `CH1`, then the subscriber must react to it and change its channel to `CH2`.

5. **P5: switch CH2 request leads to channel change at subscriber.** Symmetric to P4.

6. **P6: switches occur only in the switching phase.** Whenever a switch command is taken, the publisher must be in the dedicated `SWITCHING` state. This ensures that "control" transitions are not interleaved arbitrarily with the publish cycle.

This change results us with the following metric from $PubSub_4$ to $PubSub_5$:

- **Δ State representation size**
  +0 (no changes to the actual model)

- **Δ Behavior representation size**
  +0 (no changes to the actual model)

- **Touch points**
  6 blocks touched: added 6 LTL `SPEC`s.

Any property from the added specifications can be turned into a test by negating the right-hand side. Doing so produces a counterexample trace that demonstrates the actual chain of events that makes the property fail. This is useful as a basic sanity check for the intended functionality of the system.

Transition to TLA+. In TLA+, we can also verify temporal properties of underlying system. In particular, TLA+ offers following standard temporal operators:

- `[]` ("always", G): a formula holds in every state of the behavior.
- <> ("eventually", F): a formula holds at some future state.

Therefore, when combined with logical implication =>, we can translate the structure of R-CHECK properties almost one-to-one: `G` becomes `[]`, `F` becomes <>, and `->` becomes =>.

A key difference is that TLA+ does not provide command labels out-of-the-box. First idea is to mimic the functionality of labels by inspecting state variables — for example, "`pub1-sAnnounce` happened" might be approximated using the "the publisher is now in state `S0`" or "the message type equals `ANNOUNCE`" reasoning. Unfortunately, this does not match the R-CHECK semantics. In R-CHECK, a label acts as an *event flag*: it is true only on the transition where the command fires. Predicates are over the state variable, on the contrary, are *persistent*: once such a predicate becomes true, it can remain true for more than one. This becomes crucial for the temporal properties. Replacing event predicates by a persistent state condition can result in over- or under-approximation of the desired labeling behavior and changes the meaning of the checked properties. This is why naive state-based checking is not equivalent to event-based checking.

To perform the same event-based checking in TLA+, we need to introduce explicit label tracking mechanism into the TLA+ model. We add a label field `l` to both publisher and subscriber agent records. The idea is simple: during the step where an action is "taken", we set the corresponding `l` value (e.g., to `sAnnounce`, `rPublish`) and we clear it afterwards. This makes `l` work like a one-step event flag.

Full code of the TLA+ side of $PubSub_5$ together with the properties is shown in Listing 5.32. At a high level, the TLA+ changes are:

1. Add additional constants for label values (e.g., `sAnnounce`, `s0`, `rPublish`, ...) and add a `l` field into the publisher and subscriber state records. (Lines 6, 11, 53)
2. Introduce two buffering variables, one for broadcast and one for multicast (`BroadcastBuf`, `MulticastBuf`). These are needed to represent "message in flight" state needed to model send/receive events (see next list item). (Lines 82–86)
3. Split both broadcast and multicast into explicit send and receive phases: `BroadcastSend`/ `BroadcastRcv` and `MulticastSend`/ `MulticastRcv`. The send phase only sets labels and stores message data in the buffer; the receive phase applies the actual state changes and clears the buffer. (Lines 94–181)
4. Add simultaneous send/receive to match R-CHECK's behavior: in R-CHECK, a receive of one message and the next enabled send may appear in the same global

step. With a simple two-phase buffering, a send that is carried out parallel to a receive would be delayed by one additional state. To avoid this mismatch, the receive actions optionally invoke a "send implementation" operator. (Lines 138–145, 178–179)

5. Add fairness constraints for the receive phases (e.g., `WF_vars(BroadcastRcv)` and `WF_vars(MulticastRcv)`). Without these fairness constraints, the TLA+ specification allows infinite stuttering and produces trivial counterexamples for our properties. (Lines 195–196.)

With these changes we can translate properties we check in R-CHECK into TLA+ in the expected way: occurrences such as `pub1-sAnnounce` become state predicates `Pubs[1].l = sAnnounce` that are true only for one step by construction, and the rest of the property remains the same (using `[]` and <>). Listing 5.33 shows a diff view of $PubSub_5$ diffed against $PubSub_4$. After applying our metrics, we get the following results:

- **Δ State representation size**
  +2 state variables (`BroadcastBuf`, `MulticastBuf`); +2 record fields in agent state (`Pubs[i].l`, `Subs[i].l`);
- **Δ Behavior representation size**
  +10 actions: `Label_Pub`, `Label_Sub`, `BroadcastBufInit`, `MulticastBufInit`, `MulticastSend_Impl`, `BroadcastSend`, `BroadcastRcv`, `MulticastSend`, `MulticastRcv`, `vars`.
- **Touch points**
  24 blocks touched: updated list of constants, updated list of variables, updated initialization actions (3x), changed/added communication actions (13x), changed main `Spec` action, added 6x properties (6x).

The $PubSub_5$ milestone supports our earlier statement: the R-CHECK model gives us the possibility to reason about the underlying system on the level of interaction between agents because command labels are directly available in the LTL specifications. In TLA+, checking the same event-based properties is possible, but for doing so additional modeling work is required: we need to introduce explicit label variables and find a way to mimic the labels behavior, which involves splitting of send/receive phase, thinking about operation pipelining and fairness. Each of these additions is of course straightforward, but still result in a non-trivial amount of bookkeeping code. This increases the surface for hard-to-find errors, such as forgetting to clear a label.

4.3 Discussion

Table 4.1 summarizes the metrics measured for each step of our example, and we can observe that the modeling effort is higher for TLA+ compared to R-CHECK. In particular,

	R-CHECK			TLA+		
Step	Δ State	Δ Behavior	Touch	Δ State	Δ Behavior	Touch
$0 \rightarrow 1$	4	3	11	2	7	11
$1 \rightarrow 2$	4	1	13	3	4	11
$2 \rightarrow 3$	0	3	5	0	1	10
$3 \rightarrow 4$	2	0	6	2	2	6
$4 \rightarrow 5$	0	0	6	2	10	24

Table 4.1: Summary of per-milestone metrics for running example.

communication features (multicast, reconfiguration, and later label-based LTL properties) typically require introducing additional components to the state and behavior of the TLA+ model. In R-CHECK, on the other hand, the same features are expressed through smaller changes due to the interaction-centric design of the framework.

The touch points metric also supports our assumption: in R-CHECK, introduced changes tend to be local to a small set of blocks (usually just the new command or commands), whereas in TLA+, the same feature addition often propagates through several places. This is especially visible in the last step: in R-CHECK, event-based LTL properties can directly reference the command labels, whereas explicit label mechanisms and the supporting bookkeeping logic need to be introduced to get to the same event-based observability of the system in TLA+.

It also has to be said that most of the changes introduced to the TLA+ model required changing the indexing and adding more parameters to the parametrized actions (for example, see the step from $PubSub_1$ to $PubSub_2$). It is to be expected that the number of such small but yet crucial changes increases when introducing new agent-local state variables and message types.

Overall, the table backs up the main conclusion we wanted to achieve with the help of this running example: both frameworks can model the same behaviors, but there is a systematic difference between their ease of application and modeling overhead. TLA+ focuses on global state updates, so interaction-level concepts must be encoded indirectly via additional state and helper operators. R-CHECK, on the contrary, provides dedicated primitives (agents, channels, multicast/broadcast, and labels) for communication-centric systems that let the model stay closer to the intended interaction structure without the need for auxiliary bookkeeping.

Finally, note that the running example uses only a minimal configuration (one publisher and one subscriber) and a very small control structure. Scaling the model to multiple interacting publishers/subscribers or trying to mirror other R-CHECK modeling constructs (e.g., command sequencing/loops) on the TLA+ side will require additional control variables and corresponding handling in the whole sub-system. This justifies the fact that the observed modeling overhead may become more noticeable as the model evolves.

CHAPTER 5

Conclusion and Future work

In this thesis, we compared TLA+ and R-CHECK while keeping the main focus on communication-centric multi-agent systems. Our objective was not to show that one framework is ultimately better than the other, but rather to find out what it takes in both tools to explicitly express communication-centric MAS patterns involving dynamic communication and coordination between agents. To achieve this, we carried out two complementary evaluations. First, in Chapter 3, we ported several representative TLA+ models to R-CHECK and documented the main modeling decisions and semantic differences which arose during the translation process. Second, in Chapter 4, we used a running publishers/subscribers example with dynamic connectivity in order to reconstruct selected R-CHECK concepts in TLA+ and to evaluate the resulting modeling overhead.

The first evaluation shows that R-CHECK is expressive enough for specifying different types of systems and that TLA+ specifications can be translated to R-CHECK. At the same time, the resulting ports also show that the translation is not always a straightforward one-to-one procedure and that the semantic differences between the original models and the ports strongly depend on the nature and structure of the original model. In some cases, the port strongly resembles the original model and only introduces agents and explicit communication between them (Hanoi). In other cases, the resulting translations introduce custom protocols with intermediate steps (KeyValueStore) or shift the communication model from an asynchronous communication model to synchronous blocking multicast message passing (MultiPaxos-SMR). This preserves the high-level idea behind the models, but changes their internal execution semantics. Together, the presented examples also make clear where the main translation effort occurs. The first recurring pattern is the *agentification* of the original model: global variables become agent-local variables (i.e., must be distributed across agents), and the decision must be made regarding where the state "lives" and which agent is responsible for different parts of the original behavior. The second recurring pattern is that the communication between agents becomes an explicit design decision: in TLA+, communication is usually

represented indirectly through shared global structures, whereas in R-CHECK, the modeler must define the communication pattern and encode it through commands, channels, sender predicates, and receive-guards. The third recurring pattern we encountered is that checked properties may require slight adaptations: whenever the resulting R-CHECK model is not a one-to-one copy of the original TLA+ model, some properties cannot be easily transferred and must be reformulated.

The second evaluation shows the comparison from the opposite direction and answers the question of what has to be introduced in TLA+ in order to mimic communication-centric features that are native in R-CHECK. The result of this experiment shows that both frameworks can model the same behaviors (agent-centric systems with dynamic communication patterns), but there is a systematic difference in modeling effort. In R-CHECK, agents, system composition, broadcast and blocking multicast communication, reconfiguration via receive-guards, sender predicates over common variables and relabeling, and event-based LTL model-checking are all first-class concepts. In TLA+, on the other hand, the same features can be expressed as well, but require additional operators, actions, and variables. This becomes especially visible in the final milestones of the presented examples: for multicast and reconfiguration, the model requires extra indexing and parameter passing; for sender predicates and relabeling — additional global-state information and communication operators; for event-based LTL properties — explicit label variables and additional logic to mimic the event logic. The metrics from Table 4.1 support these observations.

Taken together, the two presented evaluations answer the three research questions we defined for this thesis in Section 1:

1. *To what extent can representative TLA+ models be ported to R-CHECK? How hard is it to do so?*
   In Chapter 3 we showed that representative TLA+ models can be ported to R-CHECK in full extent, and that the resulting modeling overhead strongly depends on the nature of the original model — if the original model is already close to the agent- and communication-centric point of view, then the porting procedure is relatively direct. If, on the other hand, the original model relies on a global state, then additional design decisions are required and the port typically becomes more complex due to the introduction of explicit communication.

2. *What modeling trade-offs and semantic differences arise during the translation between the two frameworks?*
   In our ports in Chapter 3 we mainly encountered the following trade-offs and semantic differences. First, *global vs. local state ownership* addresses the fact that TLA+ models almost always keep the system's state in global variables. R-CHECK, on the other hand, requires assigning state to agents explicitly. Second, *asynchronous vs. synchronous communication* shows that TLA+ models communication using a global message pool with asynchronous delivery, while R-CHECK natively supports synchronous communication pattern. Third, *atomicity* also becomes a key decision

when performing a port, because in TLA+ one action can update multiple parts of the system, whereas in R-CHECK this can only be expressed via multiple receive steps.

3. *When reconstructing R-CHECK-style MAS features in TLA+, what additional modeling machinery must be introduced explicitly?*
   In Chapter 4, we demonstrated that native interaction-centric features of R-CHECK require additional state components in TLA+. This was most visible when we introduced multicast and reconfiguration, as well as event-based property checking. All in all, such features almost always require additional bookkeeping (labels/program-counter, indexing).

To summarize the findings, we present Table 5.1, which gives a compact overview of the observed comparison results.

Aspect	TLA+	R-CHECK
modeling view	Global-state variables and actions	Agents with local state and explicit system composition
Communication model	Indirect, via shared buffers / message sets / helper operators	Direct, via broadcast, blocking multicast, channels, and commands
Dynamic topology	Expressed indirectly through shared state updates	Expressed directly via receive-guards and channel connectivity
Event-based LTL model checking	Requires explicit label tracking and supporting bookkeeping logic	Command labels directly available in LTL specifications
Collection-heavy state	Convenient via sets / sequences / records	Current language state may require unrolling due to lack of collection types
Asynchronous protocol style	Natural fit for message-pool style models	Requires semantic shift when encoded using synchronous blocking multicast
Observed modeling overhead in this thesis	Higher for communication-centric MAS features, especially in $PubSub_5$	Usually more local changes due to interaction-centric primitives
Best fit suggested by this thesis	General-purpose distributed models with global-state abstractions	Communication-centric, reconfigurable MAS with event-level specifications

Table 5.1: Qualitative summary of the comparison results.

At the same time, the comparison presented in this thesis has several limitations. First, this thesis does not provide extensive benchmarks of runtime and memory consumption.

Since R-CHECK relies on symbolic verification techniques, whereas TLA+ case studies are typically checked using state-enumeration-based tools, we expect lower runtime on the R-CHECK side. Second, some of the studied configurations remain intentionally small. This was done in order to keep the examples readable and to isolate the modeling decisions and semantic differences more clearly. Third, the ports from TLA+ to R-CHECK are not always exact semantic copies of the original models. This follows from the fact that the two frameworks rely on different modeling views. Finally, the observed overhead in R-CHECK appears in part due to the current state of the language rather than the underlying paradigm itself (e.g., lack of collection types). This means that the overhead is partially only caused by tooling limitations and may be reduced by future extensions of the language. All these limitations suggest directions for future work. So, for example, one natural extension of this thesis would be to pick a broader corpus of communication-centric and reconfigurable MAS case studies and systematically measure the performance and resource usage in order to obtain a quantitative comparison. Also, on the R-CHECK side, the ports suggest several improvements that would reduce modeling overhead even further, such as support for collection types, idiomatic ways of ignoring messages without dummy receives, and, possibly, reusable modeling patterns for asynchronous buffering. Finally, communication reconfiguration patterns beyond simple channel switching — such as dynamic membership or coalition-style connectivity changes — would be a particularly interesting goal for further comparison between the two frameworks.

Overall, the thesis shows that the comparison between TLA+ and R-CHECK mostly plays out at the level of modeling perspective. Both frameworks can model the same class of behaviors considered in this work, but they do so using completely different abstractions. TLA+ offers a broad and flexible state-based framework. R-CHECK provides dedicated interaction-centric primitives that are well aligned with communication-centric MAS. In this sense, the strongest contribution of the thesis is to make these differences explicit and to show for which class of systems the modeling view of R-CHECK provides the clearest practical benefit.

Code Listings

```
1  ----------------------- MODULE HanoiSeq -----------------------
2  EXTENDS TLC, Sequences, Integers
3
4  CONSTANTS A, B, C
5  VARIABLES towers
6
7  (*************************************************************)
8  (* We model the three positions where a "tower" of disks can  *)
9  (* be present as sequences of natural numbers.  The numbers   *)
10 (* represent the sizes of the disks.                          *)
11 (*                                                            *)
12 (* A, B, and C are the initial configurations of the towers.  *)
13 (* For example:                                               *)
14 (*   A == <<1,2,3>>                                           *)
15 (*   B == <<>>                                                *)
16 (*   C == <<>>                                                *)
17 (*************************************************************)
18 ASSUME A \in [1..Len(A) -> Nat]
19 ASSUME B \in [1..Len(B) -> Nat]
20 ASSUME C \in [1..Len(C) -> Nat]
21
22 Init ==
23   towers = <<A, B, C>>
24
25 (*************************************************************)
26 (* A disk can be moved if:                                    *)
27 (*  - The source position is different from the destination.  *)
28 (*  - The source tower is not empty.                          *)
29 (*  - The top disk of the source tower is smaller than the    *)
30 (*    top disk of the destination tower.                      *)
31 (*************************************************************)
32 CanMove(from, to) ==
33   /\ from /= to
34   /\ towers[from] /= <<>>
35   /\ IF
36       towers[to] = <<>>
37     THEN
38       TRUE
39     ELSE
40       Head(towers[from]) < Head(towers[to])
41
42 (*************************************************************)
43 (* Moving a disk means the source tower is left with all but  *)
```

```
44  (* the top disk, which is added to the destination tower.     *)
45  (*************************************************************)
46  Move(from, to) ==
47    towers' = [
48      towers EXCEPT
49        ![from] = Tail(towers[from]),
50        ![to] = <<Head(towers[from])>> \o towers[to]
51    ]
52
53  Next ==
54    \E from, to \in 1..Len(towers):
55      /\ CanMove(from, to)
56      /\ Move(from, to)
57
58  (*************************************************************)
59  (* This finishes the spec. The next section are the          *)
60  (* invariants to check.                                      *)
61  (*************************************************************)
62
63  (*************************************************************)
64  (* Helper to get the elements of a sequence.                 *)
65  (*************************************************************)
66  Range(sequence) ==
67    {sequence[i]: i \in DOMAIN sequence}
68
69  (*************************************************************)
70  (* `towers` has 3 elements, each a sequence of numbers.      *)
71  (*************************************************************)
72  TypeOK ==
73    /\ DOMAIN towers = 1..3
74    /\ \A sequence \in Range(towers):
75        sequence \in [1..Len(sequence) -> Nat]
76
77  (*************************************************************)
78  (* In all towers there should never be elements which were   *)
79  (* not initially present.                                    *)
80  (*************************************************************)
81  NoNewElements ==
82    LET
83      originalElements ==
84        UNION {Range(A), Range(B), Range(C)}
85      towerElements ==
86        UNION {Range(towers[1]), Range(towers[2]), Range(towers[3])}
87    IN
88      towerElements = originalElements
89
90  (*************************************************************)
91  (* The total number of disks should stay constant.           *)
92  (*************************************************************)
93  TotalConstant ==
94    LET
95      originalTotal ==
96        Len(A) + Len(B) + Len(C)
97      towerTotal==
98        Len(towers[1]) + Len(towers[2]) + Len(towers[3])
99    IN
```

```
100      towerTotal = originalTotal
101
102  (*************************************************************)
103  (* The final configuration has all disks on the right tower   *)
104  (* with the disks ordered by size.  If a violation of this    *)
105  (* invariant can be found, the stack trace shows the steps to *)
106  (* solve the Hanoi problem.                                  *)
107  (*************************************************************)
108  NotSolved ==
109    ~(
110      /\ towers[1] = <<>>
111      /\ towers[2] = <<>>
112      /\ towers[3] = [i \in 1..Len(towers[3]) |-> i]
113    )
114  ============================================================================
```

Listing 5.1: Hanoi model, TLA+

```
1  ---- MODULE MC_HanoiSeq ----
2  EXTENDS HanoiSeq
3
4  config_a == <<1,2,3>>
5  config_b == <<>>
6  config_c == <<>>
7  ====
```

Listing 5.2: Hanoi MC module, TLA+

```
1  INIT Init
2  NEXT Next
3
4  CONSTANTS
5    A <- config_a
6    B <- config_b
7    C <- config_c
8
9  INVARIANT
10   TypeOK
11   NoNewElements
12   TotalConstant
13   NotSolved
```

Listing 5.3: Hanoi MC configuration file, TLA+

```
enum channel {ch}

message-structure:
  DISK_ID: 0..3,
  FROM: 0..3,
  TO: 0..3

agent Tower
  local:
    id: 0..3,
    // slots + ptr, 0 means empty
    s1: 0..3,
    s2: 0..3,
    s3: 0..3,
    head: 0..3
  init: true
  receive-guard: chan == ch
  repeat:
  (
    rMoveFrom_h1: {FROM == id & head == 1 & s1 == DISK_ID} ch?
      [head := head - 1]
    +
    rMoveFrom_h2: {FROM == id & head == 2 & s2 == DISK_ID} ch?
      [head := head - 1]
    +
    rMoveFrom_h3: {FROM == id & head == 3 & s3 == DISK_ID} ch?
      [head := head - 1]
    +
    rMoveTo_h0: {TO == id & head == 0} ch?
      [head := head + 1, s1 := DISK_ID]
    +
    rMoveTo_h1: {TO == id & head == 1 & s1 > DISK_ID} ch?
      [head := head + 1, s2 := DISK_ID]
    +
    rMoveTo_h2: {TO == id & head == 2 & s2 > DISK_ID} ch?
      [head := head + 1, s3 := DISK_ID]
    +
    rMove_Other: {FROM != id & TO != id} ch? []
  )


agent Disk
  local:
    id: 0..3,
    loc: 0..3
  init: true
  receive-guard: chan == ch
  repeat:
  (
    sMoveToTower_1: {loc != 1} ch!
      (true)(DISK_ID := id, FROM := loc, TO := 1)[loc := 1]
    +
    sMoveToTower_2: {loc != 2} ch!
      (true)(DISK_ID := id, FROM := loc, TO := 2)[loc := 2]
    +
    sMoveToTower_3: {loc != 3} ch!
```

```
57        (true)(DISK_ID := id, FROM := loc, TO := 3)[loc := 3]
58      +
59      rMoveToTower_Other: {true} ch? []
60    )
61
62  system = Tower(t_1, id == 1 & s1 == 3 & s2 == 2 & s3 == 1
63                  & head == 3) ||
64           Tower(t_2, id == 2 & s1 == 0 & s2 == 0 & s3 == 0
65                  & head == 0) ||
66           Tower(t_3, id == 3 & s1 == 0 & s2 == 0 & s3 == 0
67                  & head == 0) ||
68           Disk(d_3, id == 3 & loc == 1) ||
69           Disk(d_2, id == 2 & loc == 1) ||
70           Disk(d_1, id == 1 & loc == 1)
71
72  // P1
73  // Sum of the heads is always 3
74  SPEC G ((t_1-head + t_2-head + t_3-head) == 3)
75
76  // P2
77  // Negation of the goal state, i.e., not all disks are on tower 3
78  SPEC (G !F (d_1-loc == 3 & d_2-loc == 3 & d_3-loc == 3))
```

Listing 5.4: Hanoi model, R-CHECK

```
---------------------- MODULE KeyValueStore ----------------------
CONSTANTS   Key, \* The set of all keys.
 Val, \* The set of all values.
 TxId  \* The set of all transaction IDs.
VARIABLES
 store, \* A data store mapping keys to values.
 tx, \* The set of open snapshot transactions.
 snapshotStore, \* Snapshots of the store for each transaction.
 written, \* A log of writes performed within each transaction.
 missed \* The set of writes invisible to each transaction.
-----------------------------------------------------------------------
NoVal ==    \* Choose something to represent the absence of a value.
    CHOOSE v : v \notin Val

Store ==    \* The set of all key-value stores.
    [Key -> Val \cup {NoVal}]

Init == \* The initial predicate.
    \* All store values are initially NoVal.
    /\ store = [k \in Key |-> NoVal]
    \* The set of open transactions is initially empty.
    /\ tx = {}
    \* All snapshotStore values are initially NoVal.
    /\ snapshotStore =
        [t \in TxId |-> [k \in Key |-> NoVal]]
    \* All write logs are initially empty.
    /\ written = [t \in TxId |-> {}]
    \* All missed writes are initially empty.
    /\ missed = [t \in TxId |-> {}]

TypeInvariant ==    \* The type invariant.
    /\ store \in Store
    /\ tx \subseteq TxId
    /\ snapshotStore \in [TxId -> Store]
    /\ written \in [TxId -> SUBSET Key]
    /\ missed \in [TxId -> SUBSET Key]

TxLifecycle ==
    \* If store != snapshot & we haven't written it,
    \* we must have missed a write.
    /\ \A t \in tx :
        \A k \in Key :
            (store[k] /= snapshotStore[t][k] /\
            k \notin written[t]) => k \in missed[t]
    \* Checks transactions are cleaned up after disposal.
    /\ \A t \in TxId \ tx :
        /\ \A k \in Key : snapshotStore[t][k] = NoVal
        /\ written[t] = {}
        /\ missed[t] = {}

\* Open a new transaction.
OpenTx(t) ==
    /\ t \notin tx
    /\ tx' = tx \cup {t}
    /\ snapshotStore' = [snapshotStore EXCEPT ![t] = store]
    /\ UNCHANGED <<written, missed, store>>
```

```
57
58  \* Using transaction t, add value v to the store under key k.
59  Add(t, k, v) ==
60      /\ t \in tx
61      /\ snapshotStore[t][k] = NoVal
62      /\ snapshotStore' = [snapshotStore EXCEPT ![t][k] = v]
63      /\ written' = [written EXCEPT ![t] = @ \cup {k}]
64      /\ UNCHANGED <<tx, missed, store>>
65
66  \* Using transaction t, update the value associated with key k to v.
67  Update(t, k, v) ==
68      /\ t \in tx
69      /\ snapshotStore[t][k] \notin {NoVal, v}
70      /\ snapshotStore' = [snapshotStore EXCEPT ![t][k] = v]
71      /\ written' = [written EXCEPT ![t] = @ \cup {k}]
72      /\ UNCHANGED <<tx, missed, store>>
73
74  \* Using transaction t, remove key k from the store.
75  Remove(t, k) ==
76      /\ t \in tx
77      /\ snapshotStore[t][k] /= NoVal
78      /\ snapshotStore' = [snapshotStore EXCEPT ![t][k] = NoVal]
79      /\ written' = [written EXCEPT ![t] = @ \cup {k}]
80      /\ UNCHANGED <<tx, missed, store>>
81
82  \* Close the transaction without merging writes into store.
83  RollbackTx(t) ==
84      /\ t \in tx
85      /\ tx' = tx \ {t}
86      /\ snapshotStore' =
87          [snapshotStore EXCEPT ![t] = [k \in Key |-> NoVal]]
88      /\ written' = [written EXCEPT ![t] = {}]
89      /\ missed' = [missed EXCEPT ![t] = {}]
90      /\ UNCHANGED store
91
92  \* Close transaction t, merging writes into store.
93  CloseTx(t) ==
94      /\ t \in tx
95      \* Detection of write-write conflicts.
96      /\ missed[t] \cap written[t] = {}
97      \* Merge snapshotStore writes into store.
98      /\ store' =
99          [k \in Key |->
100             IF k \in written[t]
101             THEN snapshotStore[t][k]
102             ELSE store[k]]
103     /\ tx' = tx \ {t}
104     \* Update the missed writes for other open transactions.
105     /\ missed' =
106         [otherTx \in TxId |->
107             IF otherTx \in tx'
108             THEN missed[otherTx] \cup written[t]
109             ELSE {}]
110     /\ snapshotStore' =
111         [snapshotStore EXCEPT ![t] = [k \in Key |-> NoVal]]
112     /\ written' = [written EXCEPT ![t] = {}]
```

```
113
114 Next == \* The next-state relation.
115     \/ \E t \in TxId : OpenTx(t)
116     \/ \E t \in tx : \E k \in Key : \E v \in Val : Add(t, k, v)
117     \/ \E t \in tx : \E k \in Key : \E v \in Val : Update(t, k, v)
118     \/ \E t \in tx : \E k \in Key : Remove(t, k)
119     \/ \E t \in tx : RollbackTx(t)
120     \/ \E t \in tx : CloseTx(t)
121
122 Spec == \* Initialize state with Init and transition with Next.
123     Init /\ [][Next]_<<store, tx, snapshotStore, written, missed>>
124 ----------------------------------------------------------------------
125 THEOREM Spec => [](TypeInvariant /\ TxLifecycle)
126 ======================================================================
```

Listing 5.5: Key-Value Store model, TLA+

```
---- MODULE MCKVS ----
EXTENDS KeyValueStore, TLC
TxIdSymmetric == Permutations(TxId)
====
```

Listing 5.6: KeyValueStore MC module, TLA+

```
CONSTANTS
    Key = {k1, k2, k3}
    Val = {v1, v2, v3}
    TxId = {t1, t2}
    NoVal = NoVal

SYMMETRY
    TxIdSymmetric

SPECIFICATION
    Spec

INVARIANTS
    TypeInvariant
    TxLifecycle
```

Listing 5.7: KeyValueStore MC configuration file, TLA+

```
// ------------------------------------------------------------------
// KeyValueStore - R-CHECK translation
// Source (TLA+):
// https://github.com/tlaplus/Examples/tree/master/specifications/KeyValueStore
//
// What it models
//   - Two transactions (tx1, tx2) interacting with a single store
//     over channel ch.
//   - Commands: open, rollback, close, ack. Each close may carry
//     per-key write (W*) and miss (M*) flags and snapshot values
//     for three keys (K1..K3).
//   - A simple request/ack protocol: in TLA+ actions are atomic,
//     but in R-CHECK we must model the communication explicitly
//     (send + ack).
//
// Agents & channels
//   - Store: keeps committed values s_k1..s_k3; replies with acks
//     and applies updates.
//   - Transaction: keeps local snapshot (snap_k*), write flags
//     (w_k*), miss flags (m_k*), and an active bit;
//     uses local "*" broadcasts only to label internal updates.
//   - ch is a multicast channel (send blocks until connected
//      receivers can step).
//
// Protocol sketch
//   - open  -> ack: activates the tx and returns current store
//     snapshot.
//   - rollback -> ack: deactivates the tx and clears local state.
//   - close -> ack: if any (W* & M*) then reject (no store change);
//                   else apply written keys (W* & !M*) and
//                   leave others unchanged.
// ------------------------------------------------------------------

enum channel { ch }
enum cmd { open, rollback, close, ack }
enum key { k1, k2, k3 }
enum val { NoVal, v1, v2, v3 }

message-structure:
  CMD: cmd,
  TID: location,
  K1: val,
  K2: val,
  K3: val,
  W1: bool,
  W2: bool,
  W3: bool,
  M1: bool,
  M2: bool,
  M3: bool

agent Store
  local:
    src: location,
    s_k1: val,
    s_k2: val,
```

```
57      s_k3: val
58    init: s_k1 == NoVal & s_k2 == NoVal & s_k3 == NoVal
59    receive-guard: chan == ch
60    repeat:
61    (
62      // ------ Open ------
63      (
64        rOpenReq: {CMD == open} ch? [src := TID]
65        ;
66        sOpenAck: {true} ch!
67          (true)(CMD := ack, TID := src, K1 := s_k1, K2 := s_k2,
68            K3 := s_k3)[]
69      )
70      // ------ Rollback ------
71      +
72      (
73        rRollbackReq: {CMD == rollback} ch? [src := TID]
74        ;
75        sRollbackAck: {true} ch! (true)(CMD := ack, TID := src)[]
76      )
77      // ------ Close ------
78      +
79      (
80        (
81          {CMD == close & ((W1 & M1) | (W2 & M2) | (W3 & M3))} ch?
82            [src := TID]
83          +
84          {CMD == close & !((W1 & M1) | (W2 & M2) | (W3 & M3))
85            & W1 & !W2 & !W3} ch? [src := TID, s_k1 := K1]
86          +
87          {CMD == close & !((W1 & M1) | (W2 & M2) | (W3 & M3))
88            & !W1 & W2 & !W3} ch? [src := TID, s_k2 := K2]
89          +
90          {CMD == close & !((W1 & M1) | (W2 & M2) | (W3 & M3))
91            & !W1 & !W2 & W3} ch? [src := TID, s_k3 := K3]
92          +
93          {CMD == close & !((W1 & M1) | (W2 & M2) | (W3 & M3))
94            & W1 & W2 & !W3} ch? [src := TID, s_k1 := K1, s_k2 := K2]
95          +
96          {CMD == close & !((W1 & M1) | (W2 & M2) | (W3 & M3))
97            & W1 & !W2 & W3} ch? [src := TID, s_k1 := K1, s_k3 := K3]
98          +
99          {CMD == close & !((W1 & M1) | (W2 & M2) | (W3 & M3))
100           & !W1 & W2 & W3} ch? [src := TID, s_k2 := K2, s_k3 := K3]
101         +
102         {CMD == close & !((W1 & M1) | (W2 & M2) | (W3 & M3))
103           & W1 & W2 & W3}
104           ch? [src := TID, s_k1 := K1, s_k2 := K2, s_k3 := K3]
105         +
106         {CMD == close & !((W1 & M1) | (W2 & M2) | (W3 & M3))
107           & !W1 & !W2 & !W3} ch? [src := TID]
108       )
109       ;
110       sCloseAck: {true} ch! (true)(CMD := ack, TID := src)[]
111     )
112   )
```

```
113
114 agent Transaction
115   local:
116     active: bool,
117     snap_k1: val,
118     snap_k2: val,
119     snap_k3: val,
120     w_k1: bool,
121     m_k1: bool,
122     w_k2: bool,
123     m_k2: bool,
124     w_k3: bool,
125     m_k3: bool
126   init: !active & !w_k1 & !m_k1 & !w_k2 & !m_k2 & !w_k3 & !m_k3 &
127     snap_k1 == NoVal & snap_k2 == NoVal & snap_k3 == NoVal
128   receive-guard: chan == ch
129   repeat:
130   (
131     // ------ Open ------
132     (
133       sOpenReq: {!active} ch! (true)(CMD := open, TID := myself)[]
134       ;
135       rOpenAck: {CMD == ack & TID == myself} ch?
136       [active := true, snap_k1 := K1, snap_k2 := K2, snap_k3 := K3]
137     )
138     +
139     (
140     rOtherOpenReq: {CMD == open & TID != myself} ch? []
141     ;
142     rOtherOpenAck: {CMD == ack & TID != myself} ch? []
143     )
144     // ------ Local operations (internal, nobody receives) ------
145     +
146     (
147       snapUpdates_k1: {active} *! (false)()[]
148       ;
149       (
150         sAdd_k1_v1: {active & snap_k1 == NoVal} *!
151           (false)()[snap_k1 := v1, w_k1 := true]
152         +
153         sAdd_k1_v2: {active & snap_k1 == NoVal} *!
154           (false)()[snap_k1 := v2, w_k1 := true]
155         +
156         sAdd_k1_v3: {active & snap_k1 == NoVal} *!
157           (false)()[snap_k1 := v3, w_k1 := true]
158         +
159         sUpdate_k1_v1_to_v2: {active & snap_k1 == v1} *!
160           (false)()[snap_k1 := v2, w_k1 := true]
161         +
162         sUpdate_k1_v1_to_v3: {active & snap_k1 == v1} *!
163           (false)()[snap_k1 := v3, w_k1 := true]
164         +
165         sUpdate_k1_v2_to_v1: {active & snap_k1 == v2} *!
166           (false)()[snap_k1 := v1, w_k1 := true]
167         +
168         sUpdate_k1_v2_to_v3: {active & snap_k1 == v2} *!
```

```
169           (false)()[snap_k1 := v3, w_k1 := true]
170         +
171         sUpdate_k1_v3_to_v1: {active & snap_k1 == v3} *!
172           (false)()[snap_k1 := v1, w_k1 := true]
173         +
174         sUpdate_k1_v3_to_v2: {active & snap_k1 == v3} *!
175           (false)()[snap_k1 := v2, w_k1 := true]
176         +
177         sRemove_k1: {active & snap_k1 != NoVal} *!
178           (false)()[snap_k1 := NoVal, w_k1 := true]
179       )
180     )
181     +
182     (
183       snapUpdates_k2: {active} *! (false)()[]
184       ;
185       (
186         sAdd_k2_v1: {active & snap_k2 == NoVal} *!
187           (false)()[snap_k2 := v1, w_k2 := true]
188         +
189         sAdd_k2_v2: {active & snap_k2 == NoVal} *!
190           (false)()[snap_k2 := v2, w_k2 := true]
191         +
192         sAdd_k2_v3: {active & snap_k2 == NoVal} *!
193           (false)()[snap_k2 := v3, w_k2 := true]
194         +
195         sUpdate_k2_v1_to_v2: {active & snap_k2 == v1} *!
196           (false)()[snap_k2 := v2, w_k2 := true]
197         +
198         sUpdate_k2_v1_to_v3: {active & snap_k2 == v1} *!
199           (false)()[snap_k2 := v3, w_k2 := true]
200         +
201         sUpdate_k2_v2_to_v1: {active & snap_k2 == v2} *!
202           (false)()[snap_k2 := v1, w_k2 := true]
203         +
204         sUpdate_k2_v2_to_v3: {active & snap_k2 == v2} *!
205           (false)()[snap_k2 := v3, w_k2 := true]
206         +
207         sUpdate_k2_v3_to_v1: {active & snap_k2 == v3} *!
208           (false)()[snap_k2 := v1, w_k2 := true]
209         +
210         sUpdate_k2_v3_to_v2: {active & snap_k2 == v3} *!
211           (false)()[snap_k2 := v2, w_k2 := true]
212         +
213         sRemove_k2: {active & snap_k2 != NoVal} *!
214           (false)()[snap_k2 := NoVal, w_k2 := true]
215       )
216     )
217     +
218     (
219       snapUpdates_k3: {active} *! (false)()[]
220       ;
221       (
222         sAdd_k3_v1: {active & snap_k3 == NoVal} *!
223           (false)()[snap_k3 := v1, w_k3 := true]
224         +
```

```
225           sAdd_k3_v2: {active & snap_k3 == NoVal} *!
226             (false)()[snap_k3 := v2, w_k3 := true]
227           +
228           sAdd_k3_v3: {active & snap_k3 == NoVal} *!
229             (false)()[snap_k3 := v3, w_k3 := true]
230           +
231           sUpdate_k3_v1_to_v2: {active & snap_k3 == v1} *!
232             (false)()[snap_k3 := v2, w_k3 := true]
233           +
234           sUpdate_k3_v1_to_v3: {active & snap_k3 == v1} *!
235             (false)()[snap_k3 := v3, w_k3 := true]
236           +
237           sUpdate_k3_v2_to_v1: {active & snap_k3 == v2} *!
238             (false)()[snap_k3 := v1, w_k3 := true]
239           +
240           sUpdate_k3_v2_to_v3: {active & snap_k3 == v2} *!
241             (false)()[snap_k3 := v3, w_k3 := true]
242           +
243           sUpdate_k3_v3_to_v1: {active & snap_k3 == v3} *!
244             (false)()[snap_k3 := v1, w_k3 := true]
245           +
246           sUpdate_k3_v3_to_v2: {active & snap_k3 == v3} *!
247             (false)()[snap_k3 := v2, w_k3 := true]
248           +
249           sRemove_k3: {active & snap_k3 != NoVal} *!
250             (false)()[snap_k3 := NoVal, w_k3 := true]
251       )
252     )
253     // ------ Rollback ------
254     +
255     (
256       sRollbackReq: {active} ch!
257         (true)(CMD := rollback, TID := myself)[]
258       ;
259       rRollbackAck: {CMD == ack & TID == myself} ch?
260         [active := false, snap_k1 := NoVal, snap_k2 := NoVal,
261           snap_k3 := NoVal, w_k1 := false, m_k1 := false,
262           w_k2 := false, m_k2 := false, w_k3 := false, m_k3 := false]
263     )
264     +
265     (
266       rOtherRollbackReq: {CMD == rollback & TID != myself} ch? []
267       ;
268       rOtherRollbackAck: {CMD == ack & TID != myself} ch? []
269     )
270     // ------ Close ------
271     +
272     (
273       sCloseReq: {active} ch!
274         (true)(CMD := close, TID := myself, K1 := snap_k1,
275           K2 := snap_k2, K3 := snap_k3, W1 := w_k1, W2 := w_k2,
276           W3 := w_k3, M1 := m_k1, M2 := m_k2, M3 := m_k3)[]
277       ;
278       rCloseAck: {CMD == ack & TID == myself} ch?
279         [active := false, snap_k1 := NoVal, snap_k2 := NoVal,
280           snap_k3 := NoVal, w_k1 := false, m_k1 := false,
```

```
281             w_k2 := false, m_k2 := false, w_k3 := false, m_k3 := false]
282       )
283       +
284       (
285       rOtherCloseReq: {CMD == close & TID != myself} ch?
286         [m_k1 := active & (m_k1 | W1), m_k2 := active & (m_k2 | W2),
287           m_k3 := active & (m_k3 | W3)]
288       ;
289       rOtherCloseAck: {CMD == ack & TID != myself} ch? []
290       )
291     )
292
293   system = Transaction(tx1, true) || Transaction(tx2, true)
294     || Store(store, true)
295
296
297   // P1
298   // tx open -> will be acked and tx becomes active
299   // tx1
300   SPEC G tx1-sOpenReq ->
301     ((X (store-rOpenReq & store-sOpenAck)) &
302       (X X (tx1-rOpenAck & tx1-active)))
303   // tx2
304   SPEC G tx2-sOpenReq ->
305     ((X (store-rOpenReq & store-sOpenAck)) &
306       (X X (tx2-rOpenAck & tx2-active)))
307
308   // P2
309   // tx close -> will be acked and tx becomes inactive
310   // tx1
311   SPEC G tx1-sCloseReq
312     -> ((X store-sCloseAck) & (X X (tx1-rCloseAck & !tx1-active)))
313   // tx2
314   SPEC G tx2-sCloseReq
315     -> ((X store-sCloseAck) & (X X (tx2-rCloseAck & !tx2-active)))
316
317   // P3
318   // tx active -> finally one of the defined actions will be executed
319   // tx1
320   SPEC G tx1-active
321     -> F (tx1-sRollbackReq | tx1-sCloseReq | tx1-rOtherOpenReq |
322           tx1-rOtherRollbackReq | tx1-rOtherCloseReq |
323           tx1-snapUpdates_k1 | tx2-snapUpdates_k1 |
324           tx1-snapUpdates_k2 | tx2-snapUpdates_k2 |
325           tx1-snapUpdates_k3 | tx2-snapUpdates_k3)
326   // tx2
327   SPEC G tx2-active
328     -> F (tx2-sRollbackReq | tx2-sCloseReq | tx2-rOtherOpenReq |
329           tx2-rOtherRollbackReq | tx2-rOtherCloseReq |
330           tx1-snapUpdates_k1 | tx2-snapUpdates_k1 |
331           tx1-snapUpdates_k2 | tx2-snapUpdates_k2 |
332           tx1-snapUpdates_k3 | tx2-snapUpdates_k3)
333
334   // P4
335   // tx active and does a snapshot change
336   // -> after change snapshot != store
```

```
// tx1
SPEC G (tx1-active & !tx2-active & tx1-snapUpdates_k1 &
        tx1-snap_k1 == store-s_k1)
          -> (X X (tx1-snap_k1 != store-s_k1)) &
         (tx1-active & !tx2-active & tx1-snapUpdates_k2 &
          tx1-snap_k2 == store-s_k2)
          -> (X X (tx1-snap_k2 != store-s_k2)) &
         (tx1-active & !tx2-active & tx1-snapUpdates_k3 &
          tx1-snap_k3 == store-s_k3)
          -> (X X (tx1-snap_k3 != store-s_k3))
// tx2
SPEC G (!tx1-active & tx2-active & tx2-snapUpdates_k1 &
          tx2-snap_k1 == store-s_k1)
          -> (X X (tx2-snap_k1 != store-s_k1)) &
         (!tx1-active & tx2-active & tx2-snapUpdates_k2 &
          tx2-snap_k2 == store-s_k2)
          -> (X X (tx2-snap_k2 != store-s_k2)) &
         (!tx1-active & tx2-active & tx2-snapUpdates_k3 &
          tx2-snap_k3 == store-s_k3)
          -> (X X (tx2-snap_k3 != store-s_k3))

// P5
// close with a changed snapshot and no misses
// -> store will be updated
// tx1
SPEC G (((tx1-sCloseReq & tx1-w_k1 & !tx1-m_k1 & !tx1-m_k2 &
           !tx1-m_k3 & tx1-snap_k1 != store-s_k1)
             -> ((X X tx1-rCloseAck) &
                   ((tx1-snap_k1 == NoVal)
                     -> (X store-s_k1 == NoVal)) &
                   ((tx1-snap_k1 == v1)
                     -> (X store-s_k1 == v1)) &
                   ((tx1-snap_k1 == v2)
                     -> (X store-s_k1 == v2)) &
                 ((tx1-snap_k1 == v3)
                     -> (X store-s_k1 == v3)))) &
         ((tx1-sCloseReq & tx1-w_k2 & !tx1-m_k1 & !tx1-m_k2 &
           !tx1-m_k3 & tx1-snap_k2 != store-s_k2)
             -> ((X X tx1-rCloseAck) &
                 ((tx1-snap_k2 == NoVal)
                     -> (X store-s_k2 == NoVal)) &
                 ((tx1-snap_k2 == v1)
                     -> (X store-s_k2 == v1)) &
                 ((tx1-snap_k2 == v2)
                     -> (X store-s_k2 == v2)) &
                 ((tx1-snap_k2 == v3)
                     -> (X store-s_k2 == v3)))) &
         ((tx1-sCloseReq & tx1-w_k3 & !tx1-m_k1 & !tx1-m_k2 &
           !tx1-m_k3 & tx1-snap_k3 != store-s_k3)
             -> ((X X tx1-rCloseAck) &
                 ((tx1-snap_k3 == NoVal)
                     -> (X store-s_k3 == NoVal)) &
                 ((tx1-snap_k3 == v1)
                     -> (X store-s_k3 == v1)) &
                 ((tx1-snap_k3 == v2)
                     -> (X store-s_k3 == v2)) &
```

```
                ((tx1-snap_k3 == v3)
                    -> (X store-s_k3 == v3)))))
// tx2
SPEC G (((tx2-sCloseReq & tx2-w_k1 & !tx2-m_k1 & !tx2-m_k2 &
          !tx2-m_k3 & tx2-snap_k1 != store-s_k1)
            -> ((X X tx2-rCloseAck) &
                ((tx2-snap_k1 == NoVal)
                  -> (X store-s_k1 == NoVal)) &
                ((tx2-snap_k1 == v1)
                  -> (X store-s_k1 == v1)) &
                ((tx2-snap_k1 == v2)
                  -> (X store-s_k1 == v2)) &
                ((tx2-snap_k1 == v3)
                  -> (X store-s_k1 == v3)))) &
         ((tx2-sCloseReq & tx2-w_k2 & !tx2-m_k1 & !tx2-m_k2 &
          !tx2-m_k3 & tx2-snap_k2 != store-s_k2)
            -> ((X X tx2-rCloseAck) &
                ((tx2-snap_k2 == NoVal)
                  -> (X store-s_k2 == NoVal)) &
                ((tx2-snap_k2 == v1)
                  -> (X store-s_k2 == v1)) &
                ((tx2-snap_k2 == v2)
                  -> (X store-s_k2 == v2)) &
                ((tx2-snap_k2 == v3) ->
                  (X store-s_k2 == v3)))) &
         ((tx2-sCloseReq & tx2-w_k3 & !tx2-m_k1 & !tx2-m_k2 &
          !tx2-m_k3 & tx2-snap_k3 != store-s_k3)
            -> ((X X tx2-rCloseAck) &
                ((tx2-snap_k3 == NoVal)
                  -> (X store-s_k3 == NoVal)) &
                ((tx2-snap_k3 == v1)
                  -> (X store-s_k3 == v1)) &
                ((tx2-snap_k3 == v2)
                  -> (X store-s_k3 == v2)) &
                ((tx2-snap_k3 == v3)
                  -> (X store-s_k3 == v3)))))

// P6
// snapshot != to store -> missed flag must be set
// tx1
SPEC G ((tx1-active & tx1-snap_k1 != store-s_k1 & !tx1-w_k1)
          -> tx1-m_k1) &
         ((tx1-active & tx1-snap_k2 != store-s_k2 & !tx1-w_k2)
          -> tx1-m_k2) &
         ((tx1-active & tx1-snap_k3 != store-s_k3 & !tx1-w_k3)
          -> tx1-m_k3)
// tx2
SPEC G ((tx2-active & tx2-snap_k1 != store-s_k1 & !tx2-w_k1)
          -> tx2-m_k1) &
         ((tx2-active & tx2-snap_k2 != store-s_k2 & !tx2-w_k2)
          -> tx2-m_k2) &
         ((tx2-active & tx2-snap_k3 != store-s_k3 & !tx2-w_k3)
          -> tx2-m_k3)

// P7
// tx inactive -> all values must be NoVal and all flags must be false
```

```
449 // tx1
450 SPEC G ((!tx1-active)
451           -> (!tx1-w_k1 & !tx1-m_k1 & !tx1-w_k2 &
452               !tx1-m_k2 & !tx1-w_k3 & !tx1-m_k3 &
453               tx1-snap_k1 == NoVal & tx1-snap_k2 == NoVal &
454               tx1-snap_k3 == NoVal))
455 // tx2
456 SPEC G ((!tx2-active)
457           -> (!tx2-w_k1 & !tx2-m_k1 & !tx2-w_k2 &
458               !tx2-m_k2 & !tx2-w_k3 & !tx2-m_k3 &
459               tx2-snap_k1 == NoVal & tx2-snap_k2 == NoVal &
460               tx2-snap_k3 == NoVal))
461
462 // P8
463 // tx inactive and sees other close request
464 // -> don't update missed flags
465 // tx1
466 SPEC G (tx1-rOtherCloseReq & !tx1-active)
467           -> (!tx1-m_k1 & !tx1-m_k2 & !tx1-m_k3)
468 // tx2
469 SPEC G (tx2-rOtherCloseReq & !tx2-active)
470           -> (!tx2-m_k1 & !tx2-m_k2 & !tx2-m_k3)
471
472 // tx active and sees other close request with write flags
473 // -> update missed flags
474 // tx1
475 SPEC G ((tx1-rOtherCloseReq & tx1-active & tx2-w_k1) -> (tx1-m_k1)) &
476         ((tx1-rOtherCloseReq & tx1-active & tx2-w_k2) -> (tx1-m_k2)) &
477         ((tx1-rOtherCloseReq & tx1-active & tx2-w_k3) -> (tx1-m_k3))
478 // tx2
479 SPEC G ((tx2-rOtherCloseReq & tx2-active & tx1-w_k1) -> (tx2-m_k1)) &
480         ((tx2-rOtherCloseReq & tx2-active & tx1-w_k2) -> (tx2-m_k2)) &
481         ((tx2-rOtherCloseReq & tx2-active & tx1-w_k3) -> (tx2-m_k3))
482
483 // P9
484 // conflicts upon closure -> updates are rejected
485 // tx1, k1
486 SPEC G (tx1-sCloseReq & ((tx1-w_k1 & tx1-m_k1) |
487        (tx1-w_k2 & tx1-m_k2) | (tx1-w_k3 & tx1-m_k3)))
488           -> ((store-s_k1 == NoVal -> X store-s_k1 == NoVal) &
489               (store-s_k1 == v1 -> X store-s_k1 == v1) &
490               (store-s_k1 == v2 -> X store-s_k1 == v2) &
491               (store-s_k1 == v3 -> X store-s_k1 == v3))
492 // tx1, k2
493 SPEC G (tx1-sCloseReq & ((tx1-w_k1 & tx1-m_k1) |
494        (tx1-w_k2 & tx1-m_k2) | (tx1-w_k3 & tx1-m_k3)))
495           -> ((store-s_k2 == NoVal -> X store-s_k2 == NoVal) &
496               (store-s_k2 == v1 -> X store-s_k2 == v1) &
497               (store-s_k2 == v2 -> X store-s_k2 == v2) &
498               (store-s_k2 == v3 -> X store-s_k2 == v3))
499 // tx1, k3
500 SPEC G (tx1-sCloseReq & ((tx1-w_k1 & tx1-m_k1) |
501         (tx1-w_k2 & tx1-m_k2) | (tx1-w_k3 & tx1-m_k3)))
502           -> ((store-s_k3 == NoVal -> X store-s_k3 == NoVal) &
503               (store-s_k3 == v1 -> X store-s_k3 == v1) &
504               (store-s_k3 == v2 -> X store-s_k3 == v2) &
```

```
505               (store-s_k3 == v3 -> X store-s_k3 == v3))
506  // tx2, k1
507  SPEC G (tx2-sCloseReq & ((tx2-w_k1 & tx2-m_k1) |
508          (tx2-w_k2 & tx2-m_k2) | (tx2-w_k3 & tx2-m_k3)))
509            -> ((store-s_k1 == NoVal -> X store-s_k1 == NoVal) &
510                (store-s_k1 == v1 -> X store-s_k1 == v1) &
511                (store-s_k1 == v2 -> X store-s_k1 == v2) &
512                (store-s_k1 == v3 -> X store-s_k1 == v3))
513  // tx2, k2
514  SPEC G (tx2-sCloseReq & ((tx2-w_k1 & tx2-m_k1) |
515          (tx2-w_k2 & tx2-m_k2) | (tx2-w_k3 & tx2-m_k3)))
516            -> ((store-s_k2 == NoVal -> X store-s_k2 == NoVal) &
517                (store-s_k2 == v1 -> X store-s_k2 == v1) &
518                (store-s_k2 == v2 -> X store-s_k2 == v2) &
519                (store-s_k2 == v3 -> X store-s_k2 == v3))
520  // tx2, k3
521  SPEC G (tx2-sCloseReq & ((tx2-w_k1 & tx2-m_k1) |
522          (tx2-w_k2 & tx2-m_k2) | (tx2-w_k3 & tx2-m_k3)))
523            -> ((store-s_k3 == NoVal -> X store-s_k3 == NoVal) &
524                (store-s_k3 == v1 -> X store-s_k3 == v1) &
525                (store-s_k3 == v2 -> X store-s_k3 == v2) &
526                (store-s_k3 == v3 -> X store-s_k3 == v3))
527
528  // P10
529  // no writes and no misses -> no changes
530  // tx1, k1
531  SPEC G (tx1-sCloseReq & !tx1-w_k1 & !tx1-m_k1 & !tx1-m_k2 & !tx1-m_k3)
532            -> ((store-s_k1 == NoVal -> X store-s_k1 == NoVal) &
533                (store-s_k1 == v1 -> X store-s_k1 == v1) &
534                (store-s_k1 == v2 -> X store-s_k1 == v2) &
535                (store-s_k1 == v3 -> X store-s_k1 == v3))
536  // tx1, k2
537  SPEC G (tx1-sCloseReq & !tx1-w_k2 & !tx1-m_k1 & !tx1-m_k2 & !tx1-m_k3)
538            -> ((store-s_k2 == NoVal -> X store-s_k2 == NoVal) &
539                (store-s_k2 == v1 -> X store-s_k2 == v1) &
540                (store-s_k2 == v2 -> X store-s_k2 == v2) &
541                (store-s_k2 == v3 -> X store-s_k2 == v3))
542  // tx1, k3
543  SPEC G (tx1-sCloseReq & !tx1-w_k3 & !tx1-m_k1 & !tx1-m_k2 & !tx1-m_k3)
544            -> ((store-s_k3 == NoVal -> X store-s_k3 == NoVal) &
545                (store-s_k3 == v1 -> X store-s_k3 == v1) &
546                (store-s_k3 == v2 -> X store-s_k3 == v2) &
547                (store-s_k3 == v3 -> X store-s_k3 == v3))
548  // tx2, k1
549  SPEC G (tx2-sCloseReq & !tx2-w_k1 & !tx2-m_k1 & !tx2-m_k2 & !tx2-m_k3)
550            -> ((store-s_k1 == NoVal -> X store-s_k1 == NoVal) &
551                (store-s_k1 == v1 -> X store-s_k1 == v1) &
552                (store-s_k1 == v2 -> X store-s_k1 == v2) &
553                (store-s_k1 == v3 -> X store-s_k1 == v3))
554  // tx2, k2
555  SPEC G (tx2-sCloseReq & !tx2-w_k2 & !tx2-m_k1 & !tx2-m_k2 & !tx2-m_k3)
556            -> ((store-s_k2 == NoVal -> X store-s_k2 == NoVal) &
557                (store-s_k2 == v1 -> X store-s_k2 == v1) &
558                (store-s_k2 == v2 -> X store-s_k2 == v2) &
559                (store-s_k2 == v3 -> X store-s_k2 == v3))
560  // tx2, k3
```

```
561  SPEC G (tx2-sCloseReq & !tx2-w_k3 & !tx2-m_k1 & !tx2-m_k2 & !tx2-m_k3)
562             -> ((store-s_k3 == NoVal -> X store-s_k3 == NoVal) &
563                 (store-s_k3 == v1 -> X store-s_k3 == v1) &
564                 (store-s_k3 == v2 -> X store-s_k3 == v2) &
565                 (store-s_k3 == v3 -> X store-s_k3 == v3))
```

Listing 5.8: Key-Value Store model, R-CHECK

```
1  ---- MODULE MultiPaxos ----
2  EXTENDS FiniteSets, Sequences, Integers, TLC
3
4  (******************************)
5  (* Model inputs & assumptions. *)
6  (******************************)
7  CONSTANT Replicas,  \* symmetric set of server nodes
8   Writes,    \* symmetric set of write commands (each w/ unique value)
9   Reads,     \* symmetric set of read commands
10  MaxBallot  \* maximum ballot pickable for leader preemption
11
12 ReplicasAssumption == /\ IsFiniteSet(Replicas)
13                       /\ Cardinality(Replicas) >= 1
14
15 WritesAssumption == /\ IsFiniteSet(Writes)
16                     /\ Cardinality(Writes) >= 1
17                     /\ "nil" \notin Writes
18                     \* a write command model value serves as both the
19                     \* ID of the command and the value to be written
20
21 ReadsAssumption == /\ IsFiniteSet(Reads)
22                   /\ Cardinality(Reads) >= 0
23                   /\ "nil" \notin Writes
24
25 MaxBallotAssumption == /\ MaxBallot \in Nat
26                       /\ MaxBallot >= 2
27
28 ASSUME /\ ReplicasAssumption
29        /\ WritesAssumption
30        /\ ReadsAssumption
31        /\ MaxBallotAssumption
32
33 ----------
34
35 (********************************)
36 (* Useful constants & typedefs. *)
37 (********************************)
38 Commands == Writes \cup Reads
39
40 NumCommands == Cardinality(Commands)
41
42 Range(seq) == {seq[i]: i \in 1..Len(seq)}
43
44 \* Client observable events.
45 ClientEvents ==      [type: {"Req"}, cmd: Commands]
46                \cup [type: {"Ack"}, cmd: Commands,
47                                     val: {"nil"} \cup Writes]
48
49 ReqEvent(c) == [type |-> "Req", cmd |-> c]
50
51 AckEvent(c, v) == [type |-> "Ack", cmd |-> c, val |-> v]
52                       \* val is the old value for a write command
53
54 InitPending ==    (CHOOSE ws \in [1..Cardinality(Writes) -> Writes]
55                       : Range(ws) = Writes)
56               \o (CHOOSE rs \in [1..Cardinality(Reads) -> Reads]
```

```
57                          : Range(rs) = Reads)
58              \* W.L.O.G., choose any sequence contatenating writes
59              \* commands and read commands as the sequence of reqs;
60              \* all other cases are either symmetric or less useful
61              \* than this one
62
63  \* Server-side constants & states.
64  MajorityNum == (Cardinality(Replicas) \div 2) + 1
65
66  Ballots == 1..MaxBallot
67
68  Slots == 1..NumCommands
69
70  Statuses == {"Preparing", "Accepting", "Committed"}
71
72  InstStates == [status: {"Empty"} \cup Statuses,
73                 cmd: {"nil"} \cup Commands,
74                 voted: [bal: {0} \cup Ballots,
75                         cmd: {"nil"} \cup Commands]]
76
77  NullInst == [status |-> "Empty",
78               cmd |-> "nil",
79               voted |-> [bal |-> 0, cmd |-> "nil"]]
80
81  NodeStates == [leader: {"none"} \cup Replicas,
82                 kvalue: {"nil"} \cup Writes,
83                 commitUpTo: {0} \cup Slots,
84                 balPrepared: {0} \cup Ballots,
85                 balMaxKnown: {0} \cup Ballots,
86                 insts: [Slots -> InstStates]]
87
88  NullNode == [leader |-> "none",
89               kvalue |-> "nil",
90               commitUpTo |-> 0,
91               balPrepared |-> 0,
92               balMaxKnown |-> 0,
93               insts |-> [s \in Slots |-> NullInst]]
94
95  FirstEmptySlot(insts) ==
96      CHOOSE s \in Slots:
97          /\ insts[s].status = "Empty"
98          /\ \A t \in 1..(s-1): insts[t].status # "Empty"
99
100 \* Service-internal messages.
101 PrepareMsgs == [type: {"Prepare"}, src: Replicas,
102                                    bal: Ballots]
103
104 PrepareMsg(r, b) == [type |-> "Prepare", src |-> r,
105                                           bal |-> b]
106
107 InstsVotes == [Slots -> [bal: {0} \cup Ballots,
108                          cmd: {"nil"} \cup Commands]]
109
110 VotesByNode(n) == [s \in Slots |-> n.insts[s].voted]
111
112 PrepareReplyMsgs == [type: {"PrepareReply"}, src: Replicas,
```

```
113                                             bal: Ballots,
114                                             votes: InstsVotes]
115
116 PrepareReplyMsg(r, b, iv) ==
117     [type |-> "PrepareReply", src |-> r,
118                               bal |-> b,
119                               votes |-> iv]
120
121 PeakVotedCmd(prs, s) ==
122     IF \A pr \in prs: pr.votes[s].bal = 0
123         THEN "nil"
124         ELSE LET bc == CHOOSE bc \in (Ballots \X Commands):
125                         /\ \E pr \in prs: /\ pr.votes[s].bal = bc[1]
126                                           /\ pr.votes[s].cmd = bc[2]
127                         /\ \A pr \in prs: pr.votes[s].bal =< bc[1]
128              IN  bc[2]
129
130 AcceptMsgs == [type: {"Accept"}, src: Replicas,
131                                   bal: Ballots,
132                                   slot: Slots,
133                                   cmd: Commands]
134
135 AcceptMsg(r, b, s, c) == [type |-> "Accept", src |-> r,
136                                               bal |-> b,
137                                               slot |-> s,
138                                               cmd |-> c]
139
140 AcceptReplyMsgs == [type: {"AcceptReply"}, src: Replicas,
141                                            bal: Ballots,
142                                            slot: Slots]
143
144 AcceptReplyMsg(r, b, s) == [type |-> "AcceptReply", src |-> r,
145                                                     bal |-> b,
146                                                     slot |-> s]
147
148 CommitNoticeMsgs == [type: {"CommitNotice"}, upto: Slots]
149
150 CommitNoticeMsg(u) == [type |-> "CommitNotice", upto |-> u]
151
152 Messages ==      PrepareMsgs
153             \cup PrepareReplyMsgs
154             \cup AcceptMsgs
155             \cup AcceptReplyMsgs
156             \cup CommitNoticeMsgs
157
158 ----------
159
160 (*****************************)
161 (* Main algorithm in PlusCal. *)
162 (*****************************)
163 (*--algorithm MultiPaxos
164
165 variable msgs = {}, \* messages in the network
166          node = [r \in Replicas |-> NullNode], \* replica node state
167          pending = InitPending, \* sequence of pending reqs
168          observed = <<>>; \* client observed events
```

```
169
170 define
171     UnseenPending(insts) ==
172         LET filter(c) == c \notin {insts[s].cmd: s \in Slots}
173         IN  SelectSeq(pending, filter)
174
175     RemovePending(cmd) ==
176         LET filter(c) == c # cmd
177         IN  SelectSeq(pending, filter)
178
179     reqsMade == {e.cmd: e \in {e \in Range(observed): e.type = "Req"}}
180
181     acksRecv == {e.cmd: e \in {e \in Range(observed): e.type = "Ack"}}
182
183     terminated == /\ Len(pending) = 0
184                   /\ Cardinality(reqsMade) = NumCommands
185                   /\ Cardinality(acksRecv) = NumCommands
186 end define;
187
188 \* Send a set of messages helper.
189 macro Send(set) begin
190     msgs := msgs \cup set;
191 end macro;
192
193 \* Observe a client event helper.
194 macro Observe(e) begin
195     if e \notin Range(observed) then
196         observed := Append(observed, e);
197     end if;
198 end macro;
199
200 \* Resolve a pending command helper.
201 macro Resolve(c) begin
202     pending := RemovePending(c);
203 end macro;
204
205 \* Someone steps up as leader and sends Prepare message to followers.
206 macro BecomeLeader(r) begin
207     \* if I'm not a leader
208     await node[r].leader # r;
209     \* pick a greater ballot number
210     with b \in Ballots do
211         await /\ b > node[r].balMaxKnown
212               /\ ~\E m \in msgs: (m.type = "Prepare") /\ (m.bal = b);
213                 \* W.L.O.G., using this clause to model that ballot
214                 \* numbers from different proposers be unique
215 \* update states and restart Prepare phase for in-progress instances
216         node[r].leader := r ||
217         node[r].balPrepared := 0 ||
218         node[r].balMaxKnown := b ||
219         node[r].insts :=
220             [s \in Slots |->
221                 [node[r].insts[s]
222                     EXCEPT !.status = IF @ = "Accepting"
223                                       THEN "Preparing"
224                                       ELSE @]];
```

```
        \* broadcast Prepare and reply to myself instantly
        Send({PrepareMsg(r, b),
              PrepareReplyMsg(r, b, VotesByNode(node[r]))});
    end with;
end macro;

\* Replica replies to a Prepare message.
macro HandlePrepare(r) begin
\* if receiving a Prepare message with larger ballot than ever seen
    with m \in msgs do
        await /\ m.type = "Prepare"
              /\ m.bal > node[r].balMaxKnown;
        \* update states and reset statuses
        node[r].leader := m.src ||
        node[r].balMaxKnown := m.bal ||
        node[r].insts :=
            [s \in Slots |->
                [node[r].insts[s]
                    EXCEPT !.status = IF @ = "Accepting"
                                      THEN "Preparing"
                                      ELSE @]];
        \* send back PrepareReply with my voted list
        Send({PrepareReplyMsg(r, m.bal, VotesByNode(node[r]))});
    end with;
end macro;

\* Leader gathers PrepareReply messages until condition met, then marks
\* the corresponding ballot as prepared and saves highest voted commands.
macro HandlePrepareReplies(r) begin
    \* if I'm waiting for PrepareReplies
    await /\ node[r].leader = r
          /\ node[r].balPrepared = 0;
    \* when there are enough number of PrepareReplies of desired ballot
    with prs = {m \in msgs: /\ m.type = "PrepareReply"
                            /\ m.bal = node[r].balMaxKnown}
    do
        await Cardinality(prs) >= MajorityNum;
        \* marks this ballot as prepared and saves highest voted command
        \* in each slot if any
        node[r].balPrepared := node[r].balMaxKnown ||
        node[r].insts :=
            [s \in Slots |->
                [node[r].insts[s]
                    EXCEPT !.status = IF \/ @ = "Preparing"
                                         \/ /\ @ = "Empty"
                                            /\ PeakVotedCmd(prs, s) # "nil"
                                      THEN "Accepting"
                                      ELSE @,
                           !.cmd = PeakVotedCmd(prs, s)]];
        \* send Accept messages for in-progress instances
        Send({AcceptMsg(r, node[r].balPrepared, s, node[r].insts[s].cmd):
              s \in {s \in Slots: node[r].insts[s].status = "Accepting"}});
    end with;
end macro;

\* A prepared leader takes a new request to fill the next empty slot.
```

```
281 macro TakeNewRequest(r) begin
282     \* if I'm a prepared leader and there's pending request
283     await /\ node[r].leader = r
284           /\ node[r].balPrepared = node[r].balMaxKnown
285           /\ \E s \in Slots: node[r].insts[s].status = "Empty"
286           /\ Len(UnseenPending(node[r].insts)) > 0;
287     \* find the next empty slot and pick a pending request
288     with s = FirstEmptySlot(node[r].insts),
289          c = Head(UnseenPending(node[r].insts))
290                 \* W.L.O.G., only pick a command not seen in current
291                 \* prepared log to have smaller state space; in practice,
292                 \* duplicated client requests should be treated by some
293                 \* idempotency mechanism such as using request IDs
294     do
295         \* update slot status and voted
296         node[r].insts[s].status := "Accepting" ||
297         node[r].insts[s].cmd := c ||
298         node[r].insts[s].voted.bal := node[r].balPrepared ||
299         node[r].insts[s].voted.cmd := c;
300         \* broadcast Accept and reply to myself instantly
301         Send({AcceptMsg(r, node[r].balPrepared, s, c),
302               AcceptReplyMsg(r, node[r].balPrepared, s)});
303         \* append to observed events sequence if haven't yet
304         Observe(ReqEvent(c));
305     end with;
306 end macro;
307
308 \* Replica replies to an Accept message.
309 macro HandleAccept(r) begin
310     \* if receiving an unreplied Accept message with valid ballot
311     with m \in msgs do
312         await /\ m.type = "Accept"
313               /\ m.bal >= node[r].balMaxKnown
314               /\ m.bal > node[r].insts[m.slot].voted.bal;
315         \* update node states and corresponding instance's states
316         node[r].leader := m.src ||
317         node[r].balMaxKnown := m.bal ||
318         node[r].insts[m.slot].status := "Accepting" ||
319         node[r].insts[m.slot].cmd := m.cmd ||
320         node[r].insts[m.slot].voted.bal := m.bal ||
321         node[r].insts[m.slot].voted.cmd := m.cmd;
322         \* send back AcceptReply
323         Send({AcceptReplyMsg(r, m.bal, m.slot)});
324     end with;
325 end macro;
326
327 \* Leader gathers AcceptReply messages for a slot until condition met, then
328 \* marks the slot as committed and acknowledges the client.
329 macro HandleAcceptReplies(r) begin
330     \* if I think I'm a current leader
331     await /\ node[r].leader = r
332           /\ node[r].balPrepared = node[r].balMaxKnown
333           /\ node[r].commitUpTo < NumCommands
334           /\ node[r].insts[node[r].commitUpTo+1].status = "Accepting";
335                 \* W.L.O.G., only enabling the next slot after commitUpTo
336                 \* here to make the body of this macro simpler
```

```
337     \* for this slot, when there are enough number of AcceptReplies
338     with s = node[r].commitUpTo + 1,
339          c = node[r].insts[s].cmd,
340          v = node[r].kvalue,
341          ars = {m \in msgs: /\ m.type = "AcceptReply"
342                             /\ m.slot = s
343                             /\ m.bal = node[r].balPrepared}
344     do
345         await Cardinality(ars) >= MajorityNum;
346         \* marks this slot as committed and apply command
347         node[r].insts[s].status := "Committed" ||
348         node[r].commitUpTo := s ||
349         node[r].kvalue := IF c \in Writes THEN c ELSE @;
350         \* append to observed events sequence if haven't yet, and remove
351         \* the command from pending
352         Observe(AckEvent(c, v));
353         Resolve(c);
354         \* broadcast CommitNotice to followers
355         Send({CommitNoticeMsg(s)});
356     end with;
357 end macro;
358
359 \* Replica receives new commit notification.
360 macro HandleCommitNotice(r) begin
361     \* if I'm a follower waiting on CommitNotice
362     await /\ node[r].leader # r
363           /\ node[r].commitUpTo < NumCommands
364           /\ node[r].insts[node[r].commitUpTo+1].status = "Accepting";
365                 \* W.L.O.G., only enabling the next slot after commitUpTo
366                 \* here to make the body of this macro simpler
367     \* for this slot, when there's a CommitNotice message
368     with s = node[r].commitUpTo + 1,
369          c = node[r].insts[s].cmd,
370          m \in msgs
371     do
372         await /\ m.type = "CommitNotice"
373               /\ m.upto = s;
374         \* marks this slot as committed and apply command
375         node[r].insts[s].status := "Committed" ||
376         node[r].commitUpTo := s ||
377         node[r].kvalue := IF c \in Writes THEN c ELSE @;
378     end with;
379 end macro;
380
381 \* Replica server node main loop.
382 process Replica \in Replicas
383 begin
384     rloop: while ~terminated do
385         either
386             BecomeLeader(self);
387         or
388             HandlePrepare(self);
389         or
390             HandlePrepareReplies(self);
391         or
392             TakeNewRequest(self);
```

```
393             or
394                 HandleAccept(self);
395             or
396                 HandleAcceptReplies(self);
397             or
398                 HandleCommitNotice(self);
399             end either;
400         end while;
401     end process;
402
403     end algorithm; *)
```

Listing 5.9: MultiPaxos-SMR model, TLA+

```
1  ---- MODULE MultiPaxos_MC ----
2  EXTENDS MultiPaxos
3
4  (****************************)
5  (* TLC config-related defs. *)
6  (****************************)
7  ConditionalPerm(set) == IF Cardinality(set) > 1
8                          THEN Permutations(set)
9                          ELSE {}
10
11 SymmetricPerms ==      ConditionalPerm(Replicas)
12                   \cup ConditionalPerm(Writes)
13                   \cup ConditionalPerm(Reads)
14
15 ConstMaxBallot == 2
16
17 ----------
18
19 (*************************)
20 (* Type check invariant. *)
21 (*************************)
22 TypeOK == /\ \A m \in msgs: m \in Messages
23          /\ \A s \in Replicas: node[s] \in NodeStates
24          /\ Len(pending) =< NumCommands
25          /\ Cardinality(Range(pending)) = Len(pending)
26          /\ \A c \in Range(pending): c \in Commands
27          /\ Len(observed) =< 2 * NumCommands
28          /\ Cardinality(Range(observed)) = Len(observed)
29          /\ Cardinality(reqsMade) >= Cardinality(acksRecv)
30          /\ \A e \in Range(observed): e \in ClientEvents
31
32 THEOREM Spec => []TypeOK
33
34 ----------
35
36 (*******************************)
37 (* Linearizability constraint. *)
38 (*******************************)
39 ReqPosOfCmd(c) == CHOOSE i \in 1..Len(observed):
40                         /\ observed[i].type = "Req"
41                         /\ observed[i].cmd = c
42
43 AckPosOfCmd(c) == CHOOSE i \in 1..Len(observed):
44                         /\ observed[i].type = "Ack"
45                         /\ observed[i].cmd = c
46
47 ResultOfCmd(c) == observed[AckPosOfCmd(c)].val
48
49 OrderIdxOfCmd(order, c) == CHOOSE j \in 1..Len(order): order[j] = c
50
51 LastWriteBefore(order, j) ==
52     LET k == CHOOSE k \in 0..(j-1):
53                   /\ (k = 0 \/ order[k] \in Writes)
54                   /\ \A l \in (k+1)..(j-1): order[l] \in Reads
55     IN  IF k = 0 THEN "nil" ELSE order[k]
56
```

```
57  IsLinearOrder(order) ==
58      /\ {order[j]: j \in 1..Len(order)} = Commands
59      /\ \A j \in 1..Len(order):
60              ResultOfCmd(order[j]) = LastWriteBefore(order, j)
61
62  ObeysRealTime(order) ==
63      \A c1, c2 \in Commands:
64          (AckPosOfCmd(c1) < ReqPosOfCmd(c2))
65              => (OrderIdxOfCmd(order, c1) < OrderIdxOfCmd(order, c2))
66
67  Linearizability ==
68      terminated =>
69          \E order \in [1..NumCommands -> Commands]:
70              /\ IsLinearOrder(order)
71              /\ ObeysRealTime(order)
72
73  THEOREM Spec => Linearizability
74
75  ====
```

Listing 5.10: MultiPaxos-SMR MC module, TLA+

```
1   SPECIFICATION Spec
2
3   CONSTANTS
4       Replicas = {s1, s2, s3}
5       Writes = {w1, w2}
6       Reads = {r1}
7       MaxBallot <- ConstMaxBallot
8
9   SYMMETRY SymmetricPerms
10
11  INVARIANTS
12      TypeOK
13      Linearizability
14
15  CHECK_DEADLOCK TRUE
```

Listing 5.11: MultiPaxos-SMR MC configuration file, TLA+

```
enum SlotStatus { Empty, Preparing, Accepting, Committed }
enum Command { Nil, W1, W2, R1 }
enum ReplicaId { NoId, Id1, Id2, Id3 }
enum MsgType { Prepare, PrepareReply, Accept, AcceptReply,
  CommitNotice, Continue }
enum channel { ch }

message-structure:
  MSG:  MsgType,
  SRC:  ReplicaId,
  BAL:  0..2,
  CMD:  Command,
  SLOT: 0..1,
  UPTO: 0..1,

  // Slots info messages
  // Slot 1:
  VOTED_BAL_S1: 0..2,
  VOTED_CMD_S1: Command

agent Replica
  local:
    // Replica state
    id:          ReplicaId,
    leader:      ReplicaId,
    balMaxKnown: 0..2,
    balPrepared: 0..2,
    promises:    0..2,
    commitUpTo:  0..1,

    // Slots states
    // Slot 1:
    statusS1:   SlotStatus,
    cmdS1:      Command,
    votedBalS1: 0..2,
    votedCmdS1: Command,
    acceptsS1:  0..2,
    // Slot 2:
    // TODO: support for multiple slots

    // bookkeeping variables intended for logic
    votedBalBufferS1: 0..2,
    votedCmdBufferS1: Command,
    cmd1_ready:       bool,
    cmd2_ready:       bool,
    cmd3_ready:       bool,
    cmdBuffer:        Command
  init:
    leader      == NoId &
    balMaxKnown == 0    &
    balPrepared == 0    &
    promises    == 0    &
    commitUpTo  == 0    &

    // Slot 1:
    statusS1   == Empty &
```

```
    acceptsS1  == 0     &
    cmdS1      == Nil   &
    votedBalS1 == 0     &
    votedCmdS1 == Nil   &

    // bookkeeping variables intended for logic
    votedBalBufferS1  == 0     &
    votedCmdBufferS1  == Nil   &
    cmd1_ready        == true  &
    cmd2_ready        == true  &
    cmd3_ready        == true  &
    cmdBuffer         == Nil
  receive-guard: chan == ch
  repeat:
  (
    // BecomeLeader
    (
      sPrepare: {leader != id & (balMaxKnown + 1) <= 5} ch! (true)
          (MSG := Prepare, SRC := id, BAL := balMaxKnown + 1)
          [leader := id, balPrepared := 0,
            balMaxKnown := balMaxKnown + 1, promises := 1,
            acceptsS1 := 0]
      ;
      // internal state update
      (
        {statusS1 == Accepting} *! (false)() [statusS1 := Preparing]
        +
        {statusS1 != Accepting} *! (false)() []
      )
    )
    +
    // HandlePrepare
    (
      // Receive Prepare message, change slot states accordingly
      rPrepare: {MSG == Prepare & BAL > balMaxKnown}
          ch?
          [leader := SRC, balMaxKnown := BAL]
      ;
      (
        {statusS1 == Accepting} *! (false)() [statusS1 := Preparing]
        +
        {statusS1 != Accepting} *! (false)() []
      )
      ;
      // Send PrepareReply message including voted values
      (
        (
          sPrepareReply: {true}
            ch! (true)
            (MSG := PrepareReply, BAL := balMaxKnown,
              VOTED_BAL_S1 := votedBalS1, VOTED_CMD_S1 := votedCmdS1)
            []
          +
          rContinue_PrepareReply: {MSG == Continue} ch? []
        )
        +
```

```
113           // dummy receive to avoid blocking
114           (rep {MSG == PrepareReply} ch? [])
115         )
116       )
117       +
118       // HandlePrepareReplies
119       (
120         rPrepareReply: {leader == id & balPrepared == 0 &
121             MSG == PrepareReply & BAL == balMaxKnown} ch?
122           [promises := promises + 1, votedBalBufferS1 := VOTED_BAL_S1,
123             votedCmdBufferS1 := VOTED_CMD_S1]
124         ;
125         // internal state update
126         (
127           (
128             {votedBalBufferS1 >= votedBalS1} *! (false)()
129               [cmdS1 := votedCmdBufferS1]
130             +
131             {votedBalBufferS1 < votedBalS1} *! (false)() []
132           )
133           ;
134           (
135             {promises < 2} *! (false)() []
136             +
137             (
138               {promises >= 2} *! (false)() [balPrepared := balMaxKnown]
139               ;
140               (
141                 (
142                   {statusS1 == Preparing |
143                       (statusS1 == Empty & cmdS1 != Nil)} *! (false)()
144                     [statusS1 := Accepting]
145                   ;
146                   sContinue_PrepareReply: {true} ch!
147                     (true)(MSG := Continue) []
148                   ;
149                   sAccept_S1_PrepareReply: {statusS1 == Accepting} ch!
150                     (true) (MSG := Accept, SRC := id,
151                       BAL := balPrepared, SLOT := 1, CMD := cmdS1)
152                     [acceptsS1 := 1]
153                 )
154                 +
155                 {!(statusS1 == Preparing |
156                     (statusS1 == Empty & cmdS1 != Nil))} *! (false)() []
157               )
158             )
159           )
160         )
161       )
162       +
163       // TakeNewRequest
164       (
165         iTakeNewRequest_S1: {statusS1 == Empty & leader == id &
166             balPrepared == balMaxKnown} *! (false) () []
167         ;
168         (
```

```
169           {cmd1_ready} *! (false)()
170             [cmdBuffer := W1, cmd1_ready := false]
171           +
172           {cmd2_ready} *! (false)()
173             [cmdBuffer := W2, cmd2_ready := false]
174           +
175           {cmd3_ready} *! (false)()
176             [cmdBuffer := R1, cmd3_ready := false]
177         )
178         ;
179         {true} *! (false) ()
180           [cmdS1 := cmdBuffer, votedBalS1 := balPrepared,
181             votedCmdS1 := cmdBuffer, statusS1 := Accepting]
182         ;
183         sContinue_iTakeNewRequest_S1: {true} ch! (true)
184           (MSG := Continue) []
185         ;
186         sAccept_S1: {statusS1 == Accepting} ch! (true)
187             (MSG := Accept, SRC := id, BAL := balPrepared,
188               SLOT := 1, CMD := cmdS1) [acceptsS1 := 1]
189       )
190       +
191       // HandleAccept
192       (
193         // we need to change global state to WaitForSlots because a
194         // replica may still be in SendPrepareReply state when leader
195         // sends `accept` message (i.e., quorum of replicas replied
196         // to prepre message, leader continued before a lagging replica
197         // has sent its reply)
198         rAccept_S1: {SLOT == 1 & MSG  == Accept & BAL >= balMaxKnown &
199             BAL > votedBalS1}
200           ch? [leader := SRC, balMaxKnown := BAL, statusS1 := Accepting,
201              cmdS1 := CMD, votedBalS1 := BAL, votedCmdS1 := CMD]
202         ;
203         (
204           (
205             sAcceptReply_S1: {true} ch! (true)
206                 (MSG := AcceptReply, SRC := id, BAL := balMaxKnown,
207                   SLOT := 1) []
208             +
209             rContinue_AcceptReply_S1: {MSG == Continue} ch? []
210           )
211           +
212           (rep {MSG == AcceptReply} ch? [])
213         )
214       )
215       +
216       // HandleAcceptReplies
217       (
218         rAcceptReply_S1: {MSG == AcceptReply & leader == id &
219             commitUpTo < 1 & SLOT == 1 & BAL == balMaxKnown} ch?
220           [acceptsS1 := acceptsS1 + 1]
221           ;
222           (
223             {acceptsS1 < 2} *! (false)() []
224             +
```

```
225             (
226               {acceptsS1 >= 2} *! (false)() []
227               ;
228               sContinue_AcceptReply_S1: {true} ch!
229                   (true)(MSG := Continue) []
230               ;
231               // HandleAcceptReplies (2)
232               (
233                 sCommitNotice_S1: {true}
234                   ch! (true) (MSG := CommitNotice, SLOT := 1, UPTO := 1)
235                   [statusS1 := Committed, commitUpTo := 1]
236               )
237             )
238           )
239       )
240       +
241       // HandleCommitNotice
242       (
243         rCommitNotice_S1:
244           {leader != id & commitUpTo < 1 & statusS1 == Accepting &
245             MSG == CommitNotice & SLOT == 1 & UPTO == 1} ch?
246           [statusS1 := Committed, commitUpTo := 1]
247       )
248       // dummy receives to avoid blocking
249       +
250       rPrepareReply_NonLeader: {MSG == PrepareReply &
251           (leader != id | promises >= 2)} ch? []
252       +
253       rAcceptReply_NonLeader: {MSG == AcceptReply &
254           (leader != id | acceptsS1 >= 2)} ch? []
255       +
256       rContinue_General: {MSG == Continue} ch? []
257     )
258
259   system = Replica(r1, id == Id1) || Replica(r2, id == Id2) ||
260     Replica(r3, id == Id3)
261
262   // P1
263   // If a replica sends Prepare message (= tries to become a leader),
264   // then other replicas receive
265   // it in the next step
266   SPEC G (r1-sPrepare -> X (r2-rPrepare & r3-rPrepare))
267
268   // P2
269   // If non-leader replicas receive Prepare message and none of them
270   // sends another Prepare message
271   // (=tries to become leader), then they send PrepareReply message
272   SPEC G (r1-rPrepare &
273           !(F r1-sPrepare) & r2-rPrepare & !(F r2-sPrepare))
274               -> F (r1-sPrepareReply | r2-sPrepareReply)
275
276   // P3
277   // If a replica receives majority of PrepareReply messages, slot is
278   // empty and other replicas do not interfere, it picks a new request
279   // for the slot and sends Accept message
280   SPEC  G (r1-rPrepareReply & r1-statusS1 == Empty & r1-promises >= 2 &
```

```
281              !(F r1-sPrepare | r2-sPrepare | r2-sPrepareReply |
282                  r3-sPrepare | r3-sPrepareReply))
283          -> F (r1-iTakeNewRequest_S1 &
284                (F r1-sAccept_S1
285                  -> X X (r1-acceptsS1 == 1 & r2-rAccept_S1
286                          & r3-rAccept_S1)))
287
288  // P4
289  // If non-leader replicas receive Accept message and none of them
290  // sends another Prepare message (=tries to become leader),
291  // then they send AcceptReply message
292  SPEC G ((r1-rAccept_S1 & !r1-sPrepare & r2-rAccept_S1 & !r2-sPrepare)
293          -> (r1-sAcceptReply_S1 | r2-sAcceptReply_S1))
294
295  // P5
296  // If a replica receives majority of AcceptReply messages and other
297  // replicas do not interfere, it sends CommitNotice message
298  SPEC G ((r1-rAcceptReply_S1 & r1-acceptsS1 >= 2 &
299            !(F r1-sPrepare | r2-sPrepare | r2-sPrepareReply |
300              r2-sAcceptReply_S1 | r3-sPrepare |
301              r3-sPrepareReply | r3-sAcceptReply_S1))
302         -> F (r1-sCommitNotice_S1 &
303                X (r2-rCommitNotice_S1 & r3-rCommitNotice_S1 &
304                   r1-statusS1 == Committed & r2-statusS1 == Committed &
305                   r3-statusS1 == Committed)))
306
307  // P6
308  // All replicas eventually commit
309  SPEC G F (r1-statusS1 == Committed & r2-statusS1 == Committed &
310            r3-statusS1 == Committed)
```

Listing 5.12: MultiPaxos-SMR model, R-CHECK

```
1  agent Pub
2    receive-guard: true
3    init: true
4    repeat:
5    (
6      {true} *!
7        (true)() []
8    )
9
10 agent Sub
11   receive-guard: true
12   init: true
13   repeat:
14   (
15     {true} *!
16       (true)() []
17   )
18
19 system = Pub(pub1, true) || Sub(sub1, true)
20
21 SPEC true
```

Listing 5.13: $PubSub_0$ model, R-CHECK

```
1  ------------------------- MODULE PubSub_0 ---------------------------
2  EXTENDS Naturals
3
4  CONSTANTS PubsCount, SubsCount
5  VARIABLE Pubs, Subs
6
7  Init_Pub ==
8      TRUE
9  Init_Sub ==
10     TRUE
11
12 Init ==
13     /\ Pubs = [ i \in 1..PubsCount |-> Init_Pub ]
14     /\ Subs = [ i \in 1..SubsCount |-> Init_Sub ]
15
16 Next ==
17     UNCHANGED <<Pubs, Subs>>
18
19 Spec ==
20     Init /\ [][Next]_<<Pubs, Subs>>
21
22 INV_1 ==
23     TRUE
24 =============================================================================
```

Listing 5.14: $PubSub_0$ model, TLA+

```
1  enum PubState {IDLE, S0, S1, S2}
2  enum SubState {WAITING, SUBSCRIBED}
3  enum MsgType  {ANNOUNCE, PUBLISH}
4
5  message-structure: MSG: MsgType, STATE: PubState
6
7  agent Pub
8    local: state: PubState
9    receive-guard: true
10   init: state == IDLE
11   repeat:
12   (
13     sAnnounce: {state == IDLE} *!
14       (true)(MSG := ANNOUNCE)[state := S0]
15     +
16     s0: {state == S0} *!
17       (true)() [state := S1]
18     +
19     s1: {state == S1} *!
20       (true)() [state := S2]
21     +
22     s2: {state == S2} *!
23       (true)() [state := S0]
24   )
25
26 agent Sub
27   local: state: SubState
28   receive-guard: true
29   init: state == WAITING
30   repeat:
31   (
32     rAnnounce: {state == WAITING & MSG == ANNOUNCE} *?
33       [state := SUBSCRIBED]
34   )
35
36 system = Pub(pub1, true) || Sub(sub1, true)
37
38 SPEC true
```

Listing 5.15: $PubSub_1$ model, R-CHECK

```
1  @@ -1,8 +1,24 @@
2  +enum PubState {IDLE, S0, S1, S2}
3  +enum SubState {WAITING, SUBSCRIBED}
4  +enum MsgType  {ANNOUNCE, PUBLISH}
5  +
6  +message-structure: MSG: MsgType, STATE: PubState
7  +
8   agent Pub
9  +  local: state: PubState
10    receive-guard: true
11 -  init: true
12 +  init: state == IDLE
13    repeat:
14    (
15 -    {true} *!
16 -      (true)() []
17 +    sAnnounce: {state == IDLE} *!
18 +      (true)(MSG := ANNOUNCE)[state := S0]
19 +    +
20 +    s0: {state == S0} *!
21 +      (true)() [state := S1]
22 +    +
23 +    s1: {state == S1} *!
24 +      (true)() [state := S2]
25 +    +
26 +    s2: {state == S2} *!
27 +      (true)() [state := S0]
28    )
29 @@ -10,8 +26,9 @@ agent Pub
30  agent Sub
31 +  local: state: SubState
32    receive-guard: true
33 -  init: true
34 +  init: state == WAITING
35    repeat:
36    (
37 -    {true} *!
38 -      (true)() []
39 +    rAnnounce: {state == WAITING & MSG == ANNOUNCE} *?
40 +      [state := SUBSCRIBED]
41    )
```

Listing 5.16: $PubSub_0$ and $PubSub_1$ diff, R-CHECK

```
1  ------------------------- MODULE PubSub_1 ----------------------------
2  EXTENDS Naturals
3
4  CONSTANTS PubsCount, SubsCount, IDLE, S0, S1, S2, WAITING, SUBSCRIBED,
5      ANNOUNCE
6  VARIABLE Pubs, Subs
7
8  Init_Pub ==
9      [ state |-> IDLE ]
10 SendPrecond_Pub(i, m) ==
11     (Pubs[i].state = IDLE /\ m = ANNOUNCE)
12 NextState_Pub(s) ==
13     IF s = IDLE THEN S0
14     ELSE IF s = S0 THEN S1
15     ELSE IF s = S1 THEN S2
16     ELSE IF s = S2 THEN S0
17     ELSE s
18 StateChange_Pub(i) ==
19     [ Pubs EXCEPT ![i].state = NextState_Pub(@) ]
20
21 Init_Sub ==
22     [ state |-> WAITING ]
23 RecvPrecond_Sub(i, m) ==
24     /\ Subs[i].state = WAITING
25     /\ m = ANNOUNCE
26 StateChange_Sub(i) ==
27     [ Subs EXCEPT ![i].state = SUBSCRIBED ]
28
29 Init ==
30     /\ Pubs = [ i \in 1..PubsCount |-> Init_Pub ]
31     /\ Subs = [ i \in 1..SubsCount |-> Init_Sub ]
32
33 BroadcastComm(m) ==
34     \E i \in 1..PubsCount:
35         /\ SendPrecond_Pub(i, m)
36         /\ Pubs' = [ j \in 1..PubsCount |->
37             StateChange_Pub(i)[j] ]
38         /\ Subs' = [ j \in 1..SubsCount |->
39             IF RecvPrecond_Sub(j, ANNOUNCE)
40             THEN StateChange_Sub(j)[j]
41             ELSE Subs[j] ]
42
43 Announce ==
44     BroadcastComm(ANNOUNCE)
45
46 Next ==
47     Announce \/ UNCHANGED <<Pubs, Subs>>
48
49 Spec ==
50     Init /\ [][Next]_<<Pubs, Subs>>
51
52 INV_1 ==
53     TRUE
54 =============================================================================
```

Listing 5.17: $PubSub_1$ model, TLA+

```
@@ -1,5 +1,6 @@
---------------------------- MODULE PubSub_0 ----------------------------
+---------------------------- MODULE PubSub_1 ----------------------------
 EXTENDS Naturals

-CONSTANTS PubsCount, SubsCount
+CONSTANTS PubsCount, SubsCount, IDLE, S0, S1, S2, WAITING, SUBSCRIBED,
+    ANNOUNCE
 VARIABLE Pubs, Subs
@@ -7,5 +8,21 @@ VARIABLE Pubs, Subs
 Init_Pub ==
-    TRUE
+    [ state |-> IDLE ]
+SendPrecond_Pub(i, m) ==
+    (Pubs[i].state = IDLE /\ m = ANNOUNCE)
+NextState_Pub(s) ==
+    IF s = IDLE THEN S0
+    ELSE IF s = S0 THEN S1
+    ELSE IF s = S1 THEN S2
+    ELSE IF s = S2 THEN S0
+    ELSE s
+StateChange_Pub(i) ==
+    [ Pubs EXCEPT ![i].state = NextState_Pub(@) ]
+
 Init_Sub ==
-    TRUE
+    [ state |-> WAITING ]
+RecvPrecond_Sub(i, m) ==
+    /\ Subs[i].state = WAITING
+    /\ m = ANNOUNCE
+StateChange_Sub(i) ==
+    [ Subs EXCEPT ![i].state = SUBSCRIBED ]

@@ -15,4 +32,17 @@ Init ==

+BroadcastComm(m) ==
+    \E i \in 1..PubsCount:
+        /\ SendPrecond_Pub(i, m)
+        /\ Pubs' = [ j \in 1..PubsCount |->
+            StateChange_Pub(i)[j] ]
+        /\ Subs' = [ j \in 1..SubsCount |->
+            IF RecvPrecond_Sub(j, ANNOUNCE)
+            THEN StateChange_Sub(j)[j]
+            ELSE Subs[j] ]
+
+Announce ==
+    BroadcastComm(ANNOUNCE)
+
 Next ==
-    UNCHANGED <<Pubs, Subs>>
+    Announce \/ UNCHANGED <<Pubs, Subs>>
```

Listing 5.18: $PubSub_0$ and $PubSub_1$ diff, TLA+

```
1  enum PubState {IDLE, S0, S1, S2}
2  enum SubState {WAITING, SUBSCRIBED}
3  enum MsgType {ANNOUNCE, PUBLISH}
4  enum Channels {NULL, CH1}
5
6  message-structure: MSG: MsgType, STATE: PubState, CHANNEL: Channels
7
8  agent Pub
9    local: state: PubState, ch: Channels
10   receive-guard: true
11   init: state == IDLE & ch == CH1
12   repeat:
13   (
14     sAnnounce: {state == IDLE} *!
15       (true)(MSG := ANNOUNCE, CHANNEL := ch) [state := S0]
16     +
17     s0: {state == S0} ch!
18       (true)(MSG := PUBLISH, STATE := state) [state := S1]
19     +
20     s1: {state == S1} ch!
21       (true)(MSG := PUBLISH, STATE := state) [state := S2]
22     +
23     s2: {state == S2} ch!
24       (true)(MSG := PUBLISH, STATE := state) [state := S0]
25   )
26
27 agent Sub
28   local: state: SubState, ch: Channels, receivedState: PubState
29   receive-guard: chan == ch
30   init: state == WAITING & ch == NULL
31   repeat:
32   (
33     rAnnounce: {state == WAITING & MSG == ANNOUNCE} *?
34       [state := SUBSCRIBED, ch := CHANNEL]
35     +
36     rPublish: {state == SUBSCRIBED & MSG == PUBLISH} ch?
37       [receivedState := STATE]
38   )
39
40 system = Pub(pub1, true) || Sub(sub1, true)
41
42 SPEC true
```

Listing 5.19: $PubSub_2$ model, R-CHECK

```
@@ -3,9 +3,10 @@ enum SubState {WAITING, SUBSCRIBED}
 enum MsgType {ANNOUNCE, PUBLISH}
+enum Channels {NULL, CH1}

-message-structure: MSG: MsgType, STATE: PubState
+message-structure: MSG: MsgType, STATE: PubState, CHANNEL: Channels

 agent Pub
-  local: state: PubState
+  local: state: PubState, ch: Channels
   receive-guard: true
-  init: state == IDLE
+  init: state == IDLE & ch == CH1
   repeat:
@@ -13,12 +14,12 @@ agent Pub
     sAnnounce: {state == IDLE} *!
-      (true)(MSG := ANNOUNCE)[state := S0]
+      (true)(MSG := ANNOUNCE, CHANNEL := ch) [state := S0]
     +
-    s0: {state == S0} *!
-      (true)() [state := S1]
+    s0: {state == S0} ch!
+      (true)(MSG := PUBLISH, STATE := state) [state := S1]
     +
-    s1: {state == S1} *!
-      (true)() [state := S2]
+    s1: {state == S1} ch!
+      (true)(MSG := PUBLISH, STATE := state) [state := S2]
     +
-    s2: {state == S2} *!
-      (true)() [state := S0]
+    s2: {state == S2} ch!
+      (true)(MSG := PUBLISH, STATE := state) [state := S0]
   )
@@ -26,5 +27,5 @@ agent Pub
 agent Sub
-  local: state: SubState
-  receive-guard: true
-  init: state == WAITING
+  local: state: SubState, ch: Channels, receivedState: PubState
+  receive-guard: chan == ch
+  init: state == WAITING & ch == NULL
   repeat:
@@ -32,3 +33,6 @@ agent Sub
     rAnnounce: {state == WAITING & MSG == ANNOUNCE} *?
-      [state := SUBSCRIBED]
+      [state := SUBSCRIBED, ch := CHANNEL]
+    +
+    rPublish: {state == SUBSCRIBED & MSG == PUBLISH} ch?
+      [receivedState := STATE]
   )
```

Listing 5.20: $PubSub_1$ and $PubSub_2$ diff, R-CHECK

```
1  -------------------------- MODULE PubSub_2 ----------------------------
2  EXTENDS Naturals
3
4  CONSTANTS PubsCount, SubsCount, IDLE, S0, S1, S2, WAITING, SUBSCRIBED,
5      ANNOUNCE, PUBLISH, NULL, CH1
6  VARIABLE Pubs, Subs
7
8  Init_Pub ==
9      [ state |-> IDLE, ch |-> CH1 ]
10 SendPrecond_Pub(i, m) ==
11     \/ (Pubs[i].state = IDLE /\ m = ANNOUNCE)
12     \/ ((Pubs[i].state = S0
13         \/ Pubs[i].state = S1
14         \/ Pubs[i].state = S2) /\ m = PUBLISH)
15 NextState_Pub(s) ==
16     IF s = IDLE THEN S0
17     ELSE IF s = S0 THEN S1
18     ELSE IF s = S1 THEN S2
19     ELSE IF s = S2 THEN S0
20     ELSE s
21 StateChange_Pub(i) ==
22     [ Pubs EXCEPT ![i].state = NextState_Pub(@) ]
23
24 Init_Sub ==
25     [ state |-> WAITING, ch |-> CH1, receivedState |->  IDLE ]
26 RecvPrecond_Sub(i, m) ==
27     \/ (Subs[i].state = WAITING /\ m = ANNOUNCE)
28     \/ (Subs[i].state = SUBSCRIBED /\ m = PUBLISH)
29 ConnCond_Sub(i, c) ==
30     Subs[i].ch = c
31 StateChange_Sub(i, m, s) ==
32     [ Subs EXCEPT
33     ![i].state =
34         IF m = ANNOUNCE THEN SUBSCRIBED
35         ELSE Subs[i].state,
36     ![i].receivedState =
37         IF m = PUBLISH THEN s
38         ELSE Subs[i].receivedState ]
39
40 Init ==
41     /\ Pubs = [ i \in 1..PubsCount |-> Init_Pub ]
42     /\ Subs = [ i \in 1..SubsCount |-> Init_Sub ]
43
44 BroadcastComm(m) ==
45     \E i \in 1..PubsCount:
46         /\ SendPrecond_Pub(i, m)
47         /\ Pubs' = [ j \in 1..PubsCount |->
48             StateChange_Pub(i)[j] ]
49         /\ Subs' = [ j \in 1..SubsCount |->
50             IF RecvPrecond_Sub(j, ANNOUNCE)
51             THEN StateChange_Sub(j, m, "dummy")[j]
52             ELSE Subs[j] ]
53
54 MulticastComm(m) ==
55     \E i \in 1..PubsCount:
56         /\ SendPrecond_Pub(i, m)
```

```
57          /\ \A j \in 1..SubsCount:
58              ConnCond_Sub(j, Pubs[i].ch) =>
59                  RecvPrecond_Sub(j, m)
60          /\ Subs' = [ k \in 1..SubsCount |->
61              IF ConnCond_Sub(k, Pubs[i].ch)
62              THEN StateChange_Sub(k, m, Pubs[i].state)[k]
63              ELSE Subs[k] ]
64          /\ Pubs' = [ j \in 1..PubsCount |->
65              StateChange_Pub(j)[j] ]
66
67  Announce ==
68      BroadcastComm(ANNOUNCE)
69  Publish ==
70      MulticastComm(PUBLISH)
71
72  Next ==
73      Announce \/ Publish
74
75  Spec ==
76      Init /\ [][Next]_<<Pubs, Subs>>
77
78  INV_1 ==
79      TRUE
80  ============================================================================
```

Listing 5.21: $PubSub_2$ model, TLA+

```
@@ -1,2 +1,2 @@
--------------------------- MODULE PubSub_1 ---------------------------
+--------------------------- MODULE PubSub_2 ---------------------------
 EXTENDS Naturals
@@ -4,3 +4,3 @@ EXTENDS Naturals
 CONSTANTS PubsCount, SubsCount, IDLE, S0, S1, S2, WAITING, SUBSCRIBED,
-    ANNOUNCE
+    ANNOUNCE, PUBLISH, NULL, CH1
 VARIABLE Pubs, Subs
@@ -8,5 +8,8 @@ VARIABLE Pubs, Subs
 Init_Pub ==
-    [ state |-> IDLE ]
+    [ state |-> IDLE, ch |-> CH1 ]
 SendPrecond_Pub(i, m) ==
-    (Pubs[i].state = IDLE /\ m = ANNOUNCE)
+    \/ (Pubs[i].state = IDLE /\ m = ANNOUNCE)
+    \/ ((Pubs[i].state = S0
+        \/ Pubs[i].state = S1
+        \/ Pubs[i].state = S2) /\ m = PUBLISH)
 NextState_Pub(s) ==
@@ -21,8 +24,16 @@ StateChange_Pub(i) ==
 Init_Sub ==
-    [ state |-> WAITING ]
+    [ state |-> WAITING, ch |-> CH1, receivedState |->  IDLE ]
 RecvPrecond_Sub(i, m) ==
-    /\ Subs[i].state = WAITING
-    /\ m = ANNOUNCE
-StateChange_Sub(i) ==
-    [ Subs EXCEPT ![i].state = SUBSCRIBED ]
+    \/ (Subs[i].state = WAITING /\ m = ANNOUNCE)
+    \/ (Subs[i].state = SUBSCRIBED /\ m = PUBLISH)
+ConnCond_Sub(i, c) ==
+    Subs[i].ch = c
+StateChange_Sub(i, m, s) ==
+    [ Subs EXCEPT
+    ![i].state =
+        IF m = ANNOUNCE THEN SUBSCRIBED
+        ELSE Subs[i].state,
+    ![i].receivedState =
+        IF m = PUBLISH THEN s
+        ELSE Subs[i].receivedState ]

@@ -39,10 +50,25 @@ BroadcastComm(m) ==
                IF RecvPrecond_Sub(j, ANNOUNCE)
-               THEN StateChange_Sub(j)[j]
+               THEN StateChange_Sub(j, m, "dummy")[j]
                ELSE Subs[j] ]

+MulticastComm(m) ==
+    \E i \in 1..PubsCount:
+        /\ SendPrecond_Pub(i, m)
+        /\ \A j \in 1..SubsCount:
+            ConnCond_Sub(j, Pubs[i].ch) =>
+                RecvPrecond_Sub(j, m)
+        /\ Subs' = [ k \in 1..SubsCount |->
+            IF ConnCond_Sub(k, Pubs[i].ch)
```

```
57 +             THEN StateChange_Sub(k, m, Pubs[i].state)[k]
58 +             ELSE Subs[k] ]
59 +         /\ Pubs' = [ j \in 1..PubsCount |->
60 +             StateChange_Pub(j)[j] ]
61 +
62  Announce ==
63      BroadcastComm(ANNOUNCE)
64 +Publish ==
65 +    MulticastComm(PUBLISH)
66
67  Next ==
68 -    Announce \/ UNCHANGED <<Pubs, Subs>>
69 +    Announce \/ Publish
```

Listing 5.22: $PubSub_1$ and $PubSub_2$ diff, TLA+

```
enum PubState {IDLE, S0, S1, S2, SWITCHING}
enum SubState {WAITING, SUBSCRIBED}
enum MsgType {ANNOUNCE, PUBLISH, SWITCH}
enum Channels {NULL, CH1, CH2}

message-structure: MSG: MsgType, STATE: PubState, CHANNEL: Channels

agent Pub
  local: state: PubState, ch: Channels
  receive-guard: true
  init: state == IDLE & ch == CH1
  repeat:
  (
    sAnnounce: {state == IDLE} *!
      (true)(MSG := ANNOUNCE, CHANNEL := ch) [state := S0]
    +
    s0: {state == S0} ch!
      (true)(MSG := PUBLISH, STATE := state) [state := S1]
    +
    s1: {state == S1} ch!
      (true)(MSG := PUBLISH, STATE := state) [state := S2]
    +
    s2: {state == S2} ch!
      (true)(MSG := PUBLISH, STATE := state) [state := SWITCHING]
    +
    sSwitch_CH1: {state == SWITCHING & ch == CH1} ch!
      (true)(MSG := SWITCH, CHANNEL := CH2) [ch := CH2, state := S0]
    +
    sSwitch_CH2: {state == SWITCHING & ch == CH2} ch!
      (true)(MSG := SWITCH, CHANNEL := CH1) [ch := CH1, state := S0]
  )

agent Sub
  local: state: SubState, ch: Channels, receivedState: PubState
  receive-guard: chan == ch
  init: state == WAITING & ch == NULL
  repeat:
  (
    rAnnounce: {state == WAITING & MSG == ANNOUNCE} *?
      [state := SUBSCRIBED, ch := CHANNEL]
    +
    rPublish: {state == SUBSCRIBED & MSG == PUBLISH} ch?
      [receivedState := STATE]
    +
    rSwitch: {state == SUBSCRIBED & MSG == SWITCH} ch?
      [ch := CHANNEL]
  )

system = Pub(pub1, true) || Sub(sub1, true)

SPEC true
```

Listing 5.23: $PubSub_3$ model, R-CHECK

```
@@ -1,5 +1,5 @@
-enum PubState {IDLE, S0, S1, S2}
```

```
+enum PubState {IDLE, S0, S1, S2, SWITCHING}
 enum SubState {WAITING, SUBSCRIBED}
-enum MsgType {ANNOUNCE, PUBLISH}
-enum Channels {NULL, CH1}
+enum MsgType {ANNOUNCE, PUBLISH, SWITCH}
+enum Channels {NULL, CH1, CH2}

@@ -23,3 +23,9 @@ agent Pub
     s2: {state == S2} ch!
-      (true)(MSG := PUBLISH, STATE := state) [state := S0]
+      (true)(MSG := PUBLISH, STATE := state) [state := SWITCHING]
+    +
+    sSwitch_CH1: {state == SWITCHING & ch == CH1} ch!
+      (true)(MSG := SWITCH, CHANNEL := CH2) [ch := CH2, state := S0]
+    +
+    sSwitch_CH2: {state == SWITCHING & ch == CH2} ch!
+      (true)(MSG := SWITCH, CHANNEL := CH1) [ch := CH1, state := S0]
   )
@@ -37,2 +43,5 @@ agent Sub
       [receivedState := STATE]
+    +
+    rSwitch: {state == SUBSCRIBED & MSG == SWITCH} ch?
+      [ch := CHANNEL]
   )
```

Listing 5.24: $PubSub_2$ and $PubSub_3$ diff, R-CHECK

```
1  -------------------------- MODULE PubSub_3 ----------------------------
2  EXTENDS Naturals
3
4  CONSTANTS PubsCount, SubsCount, IDLE, S0, S1, S2, SWITCHING, WAITING,
5      SUBSCRIBED, ANNOUNCE, PUBLISH, NULL, CH1, CH2, SWITCH
6  VARIABLE Pubs, Subs
7
8  Init_Pub ==
9      [ state |-> IDLE, ch |-> CH1 ]
10 SendPrecond_Pub(i, m) ==
11     \/ (Pubs[i].state = IDLE /\ m = ANNOUNCE)
12     \/ ((Pubs[i].state = S0
13         \/ Pubs[i].state = S1
14         \/ Pubs[i].state = S2) /\ m = PUBLISH)
15     \/ Pubs[i].state = SWITCHING /\ m = SWITCH
16 NextState_Pub(s) ==
17     IF s = IDLE THEN S0
18     ELSE IF s = S0 THEN S1
19     ELSE IF s = S1 THEN S2
20     ELSE IF s = S2 THEN SWITCHING
21     ELSE IF s = SWITCHING THEN S0
22     ELSE s
23 StateChange_Pub(i, m) ==
24     [ Pubs EXCEPT
25     ![i].state =
26         IF m = PUBLISH \/ m = ANNOUNCE THEN NextState_Pub(@)
27         ELSE Pubs[i].state,
28     ![i].ch =
29         IF m = SWITCH THEN
30             (IF Pubs[i].ch = CH1 THEN CH2 ELSE CH1)
31         ELSE Pubs[i].ch ]
32
33 Init_Sub ==
34     [ state |-> WAITING, ch |-> CH1, receivedState |->  IDLE ]
35 RecvPrecond_Sub(i, m) ==
36     \/ (Subs[i].state = WAITING /\ m = ANNOUNCE)
37     \/ (Subs[i].state = SUBSCRIBED /\ m = PUBLISH)
38     \/ (Subs[i].state = SUBSCRIBED /\ m = SWITCH)
39 ConnCond_Sub(i, c) ==
40     Subs[i].ch = c
41 StateChange_Sub(i, m, s, c) ==
42     [ Subs EXCEPT
43     ![i].state =
44         IF m = ANNOUNCE THEN SUBSCRIBED
45         ELSE Subs[i].state,
46     ![i].receivedState =
47         IF m = PUBLISH THEN s
48         ELSE Subs[i].receivedState,
49     ![i].ch =
50         IF m = ANNOUNCE THEN c
51         ELSE IF m = SWITCH THEN
52             (IF Subs[i].ch = CH1 THEN CH2 ELSE CH1)
53         ELSE Subs[i].ch ]
54
55 Init ==
56     /\ Pubs = [ i \in 1..PubsCount |-> Init_Pub ]
```

```
57      /\ Subs = [ i \in 1..SubsCount |-> Init_Sub ]
58
59  BroadcastComm(m) ==
60      \E i \in 1..PubsCount:
61          /\ SendPrecond_Pub(i, m)
62          /\ Pubs' = [ j \in 1..PubsCount |->
63              StateChange_Pub(i, m)[j] ]
64          /\ Subs' = [ j \in 1..SubsCount |->
65              IF RecvPrecond_Sub(j, ANNOUNCE)
66              THEN StateChange_Sub(j, m, "dummy", Pubs[i].ch)[j]
67              ELSE Subs[j] ]
68
69  MulticastComm(m) ==
70      \E i \in 1..PubsCount:
71          /\ SendPrecond_Pub(i, m)
72          /\ \A j \in 1..SubsCount:
73              ConnCond_Sub(j, Pubs[i].ch) =>
74                  RecvPrecond_Sub(j, m)
75          /\ Subs' = [ k \in 1..SubsCount |->
76              IF ConnCond_Sub(k, Pubs[i].ch)
77              THEN StateChange_Sub(k, m, Pubs[i].state, Pubs[i].ch)[k]
78              ELSE Subs[k] ]
79          /\ Pubs' = [ j \in 1..PubsCount |->
80              StateChange_Pub(i, m)[j] ]
81
82  Announce ==
83      BroadcastComm(ANNOUNCE)
84  Publish ==
85      MulticastComm(PUBLISH)
86  Switch ==
87      MulticastComm(SWITCH)
88
89  Next ==
90      Announce \/ Publish \/ Switch
91
92  Spec ==
93      Init /\ [][Next]_<<Pubs, Subs>>
94
95  INV_1 ==
96      TRUE
97  ============================================================================
```

Listing 5.25: $PubSub_3$ model, TLA+

```
@@ -1,6 +1,6 @@
--------------------------- MODULE PubSub_2 ---------------------------
+--------------------------- MODULE PubSub_3 ---------------------------
 EXTENDS Naturals

-CONSTANTS PubsCount, SubsCount, IDLE, S0, S1, S2, WAITING, SUBSCRIBED,
-    ANNOUNCE, PUBLISH, NULL, CH1
+CONSTANTS PubsCount, SubsCount, IDLE, S0, S1, S2, SWITCHING, WAITING,
+    SUBSCRIBED, ANNOUNCE, PUBLISH, NULL, CH1, CH2, SWITCH
 VARIABLE Pubs, Subs
@@ -14,2 +14,3 @@ SendPrecond_Pub(i, m) ==
         \/ Pubs[i].state = S2) /\ m = PUBLISH)
+    \/ Pubs[i].state = SWITCHING /\ m = SWITCH
 NextState_Pub(s) ==
@@ -18,6 +19,14 @@ NextState_Pub(s) ==
     ELSE IF s = S1 THEN S2
-    ELSE IF s = S2 THEN S0
+    ELSE IF s = S2 THEN SWITCHING
+    ELSE IF s = SWITCHING THEN S0
     ELSE s
-StateChange_Pub(i) ==
-    [ Pubs EXCEPT ![i].state = NextState_Pub(@) ]
+StateChange_Pub(i, m) ==
+    [ Pubs EXCEPT
+    ![i].state =
+        IF m = PUBLISH \/ m = ANNOUNCE THEN NextState_Pub(@)
+        ELSE Pubs[i].state,
+    ![i].ch =
+        IF m = SWITCH THEN
+            (IF Pubs[i].ch = CH1 THEN CH2 ELSE CH1)
+        ELSE Pubs[i].ch ]

@@ -28,5 +37,6 @@ RecvPrecond_Sub(i, m) ==
     \/ (Subs[i].state = SUBSCRIBED /\ m = PUBLISH)
+    \/ (Subs[i].state = SUBSCRIBED /\ m = SWITCH)
 ConnCond_Sub(i, c) ==
     Subs[i].ch = c
-StateChange_Sub(i, m, s) ==
+StateChange_Sub(i, m, s, c) ==
     [ Subs EXCEPT
@@ -37,3 +47,8 @@ StateChange_Sub(i, m, s) ==
         IF m = PUBLISH THEN s
-        ELSE Subs[i].receivedState ]
+        ELSE Subs[i].receivedState,
+    ![i].ch =
+        IF m = ANNOUNCE THEN c
+        ELSE IF m = SWITCH THEN
+            (IF Subs[i].ch = CH1 THEN CH2 ELSE CH1)
+        ELSE Subs[i].ch ]

@@ -47,6 +62,6 @@ BroadcastComm(m) ==
         /\ Pubs' = [ j \in 1..PubsCount |->
-            StateChange_Pub(i)[j] ]
+            StateChange_Pub(i, m)[j] ]
         /\ Subs' = [ j \in 1..SubsCount |->
             IF RecvPrecond_Sub(j, ANNOUNCE)
```

```
-             THEN StateChange_Sub(j, m, "dummy")[j]
+             THEN StateChange_Sub(j, m, "dummy", Pubs[i].ch)[j]
              ELSE Subs[j] ]
@@ -61,6 +76,6 @@ MulticastComm(m) ==
              IF ConnCond_Sub(k, Pubs[i].ch)
-             THEN StateChange_Sub(k, m, Pubs[i].state)[k]
+             THEN StateChange_Sub(k, m, Pubs[i].state, Pubs[i].ch)[k]
              ELSE Subs[k] ]
          /\ Pubs' = [ j \in 1..PubsCount |->
-             StateChange_Pub(j)[j] ]
+             StateChange_Pub(i, m)[j] ]

@@ -70,5 +85,7 @@ Publish ==
     MulticastComm(PUBLISH)
+Switch ==
+    MulticastComm(SWITCH)

 Next ==
-    Announce \/ Publish
+    Announce \/ Publish \/ Switch
```

Listing 5.26: $PubSub_2$ and $PubSub_3$ diff, TLA+

```
enum PubState {IDLE, S0, S1, S2, SWITCHING}
enum SubState {WAITING, SUBSCRIBED}
enum MsgType {ANNOUNCE, PUBLISH, SWITCH}
enum Channels {NULL, CH1, CH2}

message-structure: MSG: MsgType, STATE: PubState, CHANNEL: Channels


property-variables: accLvl : 0..2

agent Pub
  local: state: PubState, ch: Channels, reqLvl: 0..2
  receive-guard: true
  init: state == IDLE & ch == CH1
  repeat:
  (
    sAnnounce: {state == IDLE} *!
      (@accLvl >= reqLvl)(MSG := ANNOUNCE, CHANNEL := ch) [state := S0]
    +
    s0: {state == S0} ch!
      (true)(MSG := PUBLISH, STATE := state) [state := S1]
    +
    s1: {state == S1} ch!
      (true)(MSG := PUBLISH, STATE := state) [state := S2]
    +
    s2: {state == S2} ch!
      (true)(MSG := PUBLISH, STATE := state) [state := SWITCHING]
    +
    sSwitch_CH1: {state == SWITCHING & ch == CH1} ch!
      (true)(MSG := SWITCH, CHANNEL := CH2) [ch := CH2, state := S0]
    +
    sSwitch_CH2: {state == SWITCHING & ch == CH2} ch!
      (true)(MSG := SWITCH, CHANNEL := CH1) [ch := CH1, state := S0]
  )

agent Sub
  local: state: SubState, ch: Channels, receivedState: PubState,
    accessLvl: 0..2
  relabel:
    accLvl <- accessLvl
  receive-guard: chan == ch
  init: state == WAITING & ch == NULL
  repeat:
  (
    rAnnounce: {state == WAITING & MSG == ANNOUNCE} *?
      [state := SUBSCRIBED, ch := CHANNEL]
    +
    rPublish: {state == SUBSCRIBED & MSG == PUBLISH} ch?
      [receivedState := STATE]
    +
    rSwitch: {state == SUBSCRIBED & MSG == SWITCH} ch?
      [ch := CHANNEL]
  )

system = Pub(pub1, reqLvl == 1) || Sub(sub1, accessLvl == 2)

```

```
57  SPEC true
```

Listing 5.27: $PubSub_4$ model, R-CHECK

```
@@ -7,4 +7,7 @@ message-structure: MSG: MsgType, STATE: PubState, CHANNEL: Channels

+
+property-variables: accLvl : 0..2
+
 agent Pub
-  local: state: PubState, ch: Channels
+  local: state: PubState, ch: Channels, reqLvl: 0..2
   receive-guard: true
@@ -14,3 +17,3 @@ agent Pub
     sAnnounce: {state == IDLE} *!
-      (true)(MSG := ANNOUNCE, CHANNEL := ch) [state := S0]
+      (@accLvl >= reqLvl)(MSG := ANNOUNCE, CHANNEL := ch) [state := S0]
     +
@@ -33,3 +36,6 @@ agent Pub
 agent Sub
-  local: state: SubState, ch: Channels, receivedState: PubState
+  local: state: SubState, ch: Channels, receivedState: PubState,
+    accessLvl: 0..2
+  relabel:
+    accLvl <- accessLvl
   receive-guard: chan == ch
@@ -48,3 +54,3 @@ agent Sub

-system = Pub(pub1, true) || Sub(sub1, true)
+system = Pub(pub1, reqLvl == 1) || Sub(sub1, accessLvl == 2)
```

Listing 5.28: $PubSub_3$ and $PubSub_4$ diff, R-CHECK

```
1  ------------------------- MODULE PubSub_4 ---------------------------
2  EXTENDS Naturals
3
4  CONSTANTS PubsCount, SubsCount, IDLE, S0, S1, S2, SWITCHING, WAITING,
5      SUBSCRIBED, ANNOUNCE, PUBLISH, NULL, CH1, CH2, SWITCH
6  VARIABLE Pubs, Subs
7
8  Init_Pubs ==
9      [ [ i \in 1..PubsCount |-> [ state |-> IDLE, ch |-> CH1,
10         reqLvl |-> 0 ] ]
11     EXCEPT
12         ![1].reqLvl = 1 ]
13 SendPrecond_Pub(i, m) ==
14     \/ (Pubs[i].state = IDLE /\ m = ANNOUNCE)
15     \/ ((Pubs[i].state = S0
16         \/ Pubs[i].state = S1
17         \/ Pubs[i].state = S2) /\ m = PUBLISH)
18     \/ Pubs[i].state = SWITCHING /\ m = SWITCH
19 NextState_Pub(s) ==
20     IF s = IDLE THEN S0
21     ELSE IF s = S0 THEN S1
22     ELSE IF s = S1 THEN S2
23     ELSE IF s = S2 THEN SWITCHING
24     ELSE IF s = SWITCHING THEN S0
25     ELSE s
26 StateChange_Pub(i, m) ==
27     [ Pubs EXCEPT
28     ![i].state =
29         IF m = PUBLISH \/ m = ANNOUNCE THEN NextState_Pub(@)
30         ELSE Pubs[i].state,
31     ![i].ch =
32         IF m = SWITCH THEN
33             (IF Pubs[i].ch = CH1 THEN CH2 ELSE CH1)
34         ELSE Pubs[i].ch ]
35
36 Init_Subs ==
37     [ [ i \in 1..SubsCount |-> [ state |-> WAITING, ch |-> NULL,
38         receivedState |-> IDLE, accLvl |-> 0 ] ]
39     EXCEPT
40        ![1].accLvl = 2 ]
41 RecvPrecond_Sub(i, m, reqLvl) ==
42     \/ (Subs[i].state = WAITING /\ m = ANNOUNCE
43         /\ Subs[i].accLvl >= reqLvl)
44     \/ (Subs[i].state = SUBSCRIBED /\ m = PUBLISH)
45     \/ (Subs[i].state = SUBSCRIBED /\ m = SWITCH)
46 ConnCond_Sub(i, c) ==
47     Subs[i].ch = c
48 StateChange_Sub(i, m, s, c) ==
49     [ Subs EXCEPT
50     ![i].state =
51         IF m = ANNOUNCE THEN SUBSCRIBED
52         ELSE Subs[i].state,
53     ![i].receivedState =
54         IF m = PUBLISH THEN s
55         ELSE Subs[i].receivedState,
56     ![i].ch =
```

```
57          IF m = ANNOUNCE THEN c
58          ELSE IF m = SWITCH THEN
59              (IF Subs[i].ch = CH1 THEN CH2 ELSE CH1)
60          ELSE Subs[i].ch ]
61
62
63  Init ==
64      /\ Pubs = Init_Pubs
65      /\ Subs = Init_Subs
66
67  BroadcastComm(m) ==
68      \E i \in 1..PubsCount:
69          /\ SendPrecond_Pub(i, m)
70          /\ Pubs' = [ j \in 1..PubsCount |->
71              StateChange_Pub(i, m)[j] ]
72          /\ Subs' = [ j \in 1..SubsCount |->
73              IF RecvPrecond_Sub(j, ANNOUNCE, Pubs[i].reqLvl)
74              THEN StateChange_Sub(j, m, "dummy", Pubs[i].ch)[j]
75              ELSE Subs[j] ]
76
77  MulticastComm(m) ==
78      \E i \in 1..PubsCount:
79          /\ SendPrecond_Pub(i, m)
80          /\ \A j \in 1..SubsCount:
81              ConnCond_Sub(j, Pubs[i].ch) =>
82                  RecvPrecond_Sub(j, m, Pubs[i].reqLvl)
83          /\ Subs' = [ k \in 1..SubsCount |->
84              IF ConnCond_Sub(k, Pubs[i].ch)
85              THEN StateChange_Sub(k, m, Pubs[i].state, Pubs[i].ch)[k]
86              ELSE Subs[k] ]
87          /\ Pubs' = [ j \in 1..PubsCount |->
88              StateChange_Pub(i, m)[j] ]
89
90  Announce ==
91      BroadcastComm(ANNOUNCE)
92  Publish ==
93      MulticastComm(PUBLISH)
94  Switch ==
95      MulticastComm(SWITCH)
96
97  Next ==
98      Announce \/ Publish \/ Switch
99
100 Spec ==
101     Init /\ [][Next]_<<Pubs, Subs>>
102
103 INV_1 ==
104     TRUE
105 ============================================================================
```

Listing 5.29: $PubSub_4$ model, TLA+

```
@@ -1,2 +1,2 @@
--------------------------- MODULE PubSub_3 ---------------------------
+--------------------------- MODULE PubSub_4 ---------------------------
 EXTENDS Naturals
@@ -7,4 +7,7 @@ VARIABLE Pubs, Subs

-Init_Pub ==
-    [ state |-> IDLE, ch |-> CH1 ]
+Init_Pubs ==
+    [ [ i \in 1..PubsCount |-> [ state |-> IDLE, ch |-> CH1,
+        reqLvl |-> 0 ] ]
+    EXCEPT
+        ![1].reqLvl = 1 ]
 SendPrecond_Pub(i, m) ==
@@ -32,6 +35,10 @@ StateChange_Pub(i, m) ==

-Init_Sub ==
-    [ state |-> WAITING, ch |-> CH1, receivedState |->  IDLE ]
-RecvPrecond_Sub(i, m) ==
-    \/ (Subs[i].state = WAITING /\ m = ANNOUNCE)
+Init_Subs ==
+    [ [ i \in 1..SubsCount |-> [ state |-> WAITING, ch |-> NULL,
+        receivedState |-> IDLE, accLvl |-> 0 ] ]
+    EXCEPT
+       ![1].accLvl = 2 ]
+RecvPrecond_Sub(i, m, reqLvl) ==
+    \/ (Subs[i].state = WAITING /\ m = ANNOUNCE
+        /\ Subs[i].accLvl >= reqLvl)
     \/ (Subs[i].state = SUBSCRIBED /\ m = PUBLISH)
@@ -54,5 +61,6 @@ StateChange_Sub(i, m, s, c) ==

+
 Init ==
-    /\ Pubs = [ i \in 1..PubsCount |-> Init_Pub ]
-    /\ Subs = [ i \in 1..SubsCount |-> Init_Sub ]
+    /\ Pubs = Init_Pubs
+    /\ Subs = Init_Subs

@@ -64,3 +72,3 @@ BroadcastComm(m) ==
         /\ Subs' = [ j \in 1..SubsCount |->
-            IF RecvPrecond_Sub(j, ANNOUNCE)
+            IF RecvPrecond_Sub(j, ANNOUNCE, Pubs[i].reqLvl)
             THEN StateChange_Sub(j, m, "dummy", Pubs[i].ch)[j]
@@ -73,3 +81,3 @@ MulticastComm(m) ==
             ConnCond_Sub(j, Pubs[i].ch) =>
-                RecvPrecond_Sub(j, m)
+                RecvPrecond_Sub(j, m, Pubs[i].reqLvl)
         /\ Subs' = [ k \in 1..SubsCount |->
```

Listing 5.30: $PubSub_3$ and $PubSub_4$ diff, TLA+

```
1  // P1
2  SPEC G ((pub1-sAnnounce & (sub1-accessLvl >= pub1-reqLvl)) ->
3            F (sub1-rAnnounce &
4               sub1-ch = pub1-ch &
5               sub1-state = SUBSCRIBED))
6
7  // P2
8  SPEC G ((sub1-state = SUBSCRIBED) -> (sub1-ch != NULL))
9
10 // P3
11 SPEC G ((pub1-s0 | pub1-s1 | pub1-s2) -> (F sub1-rPublish))
12
13 // P4
14 SPEC G (pub1-sSwitch_CH1 -> F (sub1-rSwitch & sub1-ch = CH2))
15
16 // P5
17 SPEC G (pub1-sSwitch_CH2 -> F (sub1-rSwitch & sub1-ch = CH1))
18
19 // P6
20 SPEC G ((pub1-sSwitch_CH1 | pub1-sSwitch_CH2) ->
21          pub1-state == SWITCHING)
```

Listing 5.31: LTL SPECs $PubSub_5$, R-CHECK

```
1  ------------------------ MODULE PubSub_5 -----------------------
2  EXTENDS Naturals
3
4  CONSTANTS PubsCount, SubsCount, IDLE, S0, S1, S2, SWITCHING, WAITING,
5      SUBSCRIBED, ANNOUNCE, PUBLISH, NULL, CH1, CH2, SWITCH, sAnnounce,
6      s0, s1, s2, sSwitch_CH1, sSwitch_CH2, rAnnounce, rPublish, rSwitch
7  VARIABLE Pubs, Subs, BroadcastBuf, MulticastBuf
8
9  Init_Pubs ==
10     [ [ i \in 1..PubsCount |-> [ state |-> IDLE, ch |-> CH1,
11         reqLvl |-> 0 , l |-> NULL ] ]
12     EXCEPT
13         ![1].reqLvl = 1 ]
14 SendPrecond_Pub(P, i, m) ==
15     \/ (P[i].state = IDLE /\ m = ANNOUNCE)
16     \/ ((P[i].state = S0
17         \/ P[i].state = S1
18         \/ P[i].state = S2) /\ m = PUBLISH)
19     \/ P[i].state = SWITCHING /\ m = SWITCH
20 NextState_Pub(s) ==
21     IF s = IDLE THEN S0
22     ELSE IF s = S0 THEN S1
23     ELSE IF s = S1 THEN S2
24     ELSE IF s = S2 THEN SWITCHING
25     ELSE IF s = SWITCHING THEN S0
26     ELSE s
27 StateChange_Pub(i, m) ==
28     [ Pubs EXCEPT
29     ![i].state =
30         IF m = PUBLISH \/ m = ANNOUNCE THEN NextState_Pub(@)
31         ELSE Pubs[i].state,
32     ![i].ch =
33         IF m = SWITCH THEN
34             (IF Pubs[i].ch = CH1 THEN CH2 ELSE CH1)
35         ELSE Pubs[i].ch,
36     ![i].l = NULL ]
37 Label_Pub(P, i, m) ==
38     [ P EXCEPT
39     ![i].l =
40         IF m = ANNOUNCE THEN sAnnounce
41         ELSE IF m = PUBLISH THEN
42             IF P[i].state = S0 THEN s0
43             ELSE IF P[i].state = S1 THEN s1
44             ELSE IF P[i].state = S2 THEN s2
45             ELSE NULL
46         ELSE IF m = SWITCH THEN
47             IF P[i].ch = CH1 THEN sSwitch_CH1
48             ELSE sSwitch_CH2
49         ELSE NULL ]
50
51 Init_Subs ==
52     [ [ i \in 1..SubsCount |-> [ state |-> WAITING, ch |-> NULL,
53         receivedState |-> IDLE, accLvl |-> 0, l |-> NULL ] ]
54     EXCEPT
55        ![1].accLvl = 2 ]
56 RecvPrecond_Sub(i, m, reqLvl) ==
```

```
57      \/ (Subs[i].state = WAITING /\ m = ANNOUNCE
58          /\ Subs[i].accLvl >= reqLvl)
59      \/ (Subs[i].state = SUBSCRIBED /\ m = PUBLISH)
60      \/ (Subs[i].state = SUBSCRIBED /\ m = SWITCH)
61  ConnCond_Sub(i, c) ==
62      Subs[i].ch = c
63  StateChange_Sub(i, m, s, c) ==
64      [ Subs EXCEPT
65      ![i].state =
66          IF m = ANNOUNCE THEN SUBSCRIBED
67          ELSE Subs[i].state,
68      ![i].receivedState =
69          IF m = PUBLISH THEN s
70          ELSE Subs[i].receivedState,
71      ![i].ch =
72          IF m = ANNOUNCE THEN c
73          ELSE IF m = SWITCH THEN
74              (IF Subs[i].ch = CH1 THEN CH2 ELSE CH1)
75          ELSE Subs[i].ch ]
76  Label_Sub(m) ==
77      IF m = ANNOUNCE THEN rAnnounce
78      ELSE IF m = PUBLISH THEN rPublish
79      ELSE IF m = SWITCH THEN rSwitch
80      ELSE NULL
81
82  BroadcastBufInit == [ ready |-> FALSE, sender |-> NULL, m |-> NULL,
83      ch |-> NULL, state |-> NULL, reqLvl |-> 0 ]
84
85  MulticastBufInit == [ ready |-> FALSE, sender |-> NULL, m |-> NULL,
86      ch |-> NULL, state |-> NULL, reqLvl |-> 0 ]
87
88  Init ==
89      /\ Pubs = Init_Pubs
90      /\ Subs = Init_Subs
91      /\ BroadcastBuf = BroadcastBufInit
92      /\ MulticastBuf = MulticastBufInit
93
94  MulticastSend_Impl(P, S, m) ==
95      \E i \in 1..PubsCount:
96          /\ SendPrecond_Pub(P, i, m)
97          /\ \A j \in 1..SubsCount:
98              ConnCond_Sub(j, P[i].ch) =>
99                  RecvPrecond_Sub(j, m, P[i].reqLvl)
100         /\ Pubs' = Label_Pub(P, i, m)
101         /\ Subs' = S
102         /\ MulticastBuf' = [ ready  |-> TRUE, sender |-> i, m |-> m,
103                              ch |-> P[i].ch,
104                              state |-> P[i].state,
105                              reqLvl |-> P[i].reqLvl ]
106
107 BroadcastSend(m) ==
108     /\ ~BroadcastBuf.ready
109     /\ ~MulticastBuf.ready
110     /\ \E i \in 1..PubsCount:
111         /\ SendPrecond_Pub(Pubs, i, m)
112         /\ Pubs' = Label_Pub(Pubs, i, m)
```

```
113          /\ Subs' = [ j \in 1..SubsCount |->
114                       [ Subs[j] EXCEPT !.l = NULL ] ]
115          /\ BroadcastBuf' = [ ready |-> TRUE, sender |-> i, m |-> m,
116              ch |-> Pubs[i].ch,
117              reqLvl |-> Pubs[i].reqLvl ]
118      /\ UNCHANGED MulticastBuf
119
120  BroadcastRcv ==
121      /\ BroadcastBuf.ready
122      /\ LET P == [ j \in 1..PubsCount |->
123                       StateChange_Pub(BroadcastBuf.sender,
124                           BroadcastBuf.m)[j] ]
125             S == [ j \in 1..SubsCount |->
126                       IF RecvPrecond_Sub(j, BroadcastBuf.m,
127                           BroadcastBuf.reqLvl)
128                       THEN [ StateChange_Sub(j, BroadcastBuf.m, "dummy",
129                                   BroadcastBuf.ch)[j]
130                              EXCEPT !.l = Label_Sub(BroadcastBuf.m) ]
131                       ELSE [ Subs[j] EXCEPT !.l = NULL ] ]
132          IN
133          \/
134              /\ Pubs' = P
135              /\ Subs' = S
136              /\ BroadcastBuf' = BroadcastBufInit
137              /\ UNCHANGED MulticastBuf
138          \/
139              /\ ~MulticastBuf.ready
140              /\ MulticastSend_Impl(P, S, PUBLISH)
141              /\ BroadcastBuf' = BroadcastBufInit
142          \/
143              /\ ~MulticastBuf.ready
144              /\ MulticastSend_Impl(P, S, SWITCH)
145              /\ BroadcastBuf' = BroadcastBufInit
146
147  BroadcastComm(m) == BroadcastSend(m) \/ BroadcastRcv
148
149  MulticastSend(m) ==
150      /\ ~MulticastBuf.ready
151      /\ ~BroadcastBuf.ready
152      /\ MulticastSend_Impl(Pubs,
153          [ j \in 1..SubsCount |-> [ Subs[j] EXCEPT !.l = NULL ] ],
154          m)
155      /\ UNCHANGED BroadcastBuf
156
157  MulticastRcv ==
158    /\ MulticastBuf.ready
159    /\ LET P == [ j \in 1..PubsCount |->
160                    StateChange_Pub(MulticastBuf.sender,
161                        MulticastBuf.m)[j] ]
162           S == [ j \in 1..SubsCount |->
163                    IF ConnCond_Sub(j, MulticastBuf.ch)
164                    THEN [ StateChange_Sub(j, MulticastBuf.m,
165                             MulticastBuf.state, MulticastBuf.ch)[j]
166                             EXCEPT !.l = Label_Sub(MulticastBuf.m) ]
167                    ELSE [ Subs[j] EXCEPT !.l = NULL ] ]
168       IN
```

```
169     \/
170         /\ Pubs' = P
171         /\ Subs' = S
172         /\ MulticastBuf' = MulticastBufInit
173         /\ UNCHANGED BroadcastBuf
174     \/
175         /\ MulticastSend_Impl(P, S, PUBLISH)
176         /\ BroadcastBuf' = BroadcastBufInit
177     \/
178         /\ MulticastSend_Impl(P, S, SWITCH)
179         /\ BroadcastBuf' = BroadcastBufInit
180
181 MulticastComm(m) == MulticastSend(m) \/ MulticastRcv
182
183 Announce ==
184     BroadcastComm(ANNOUNCE)
185 Publish ==
186     MulticastComm(PUBLISH)
187 Switch ==
188     MulticastComm(SWITCH)
189
190 Next ==
191     Announce \/ Publish \/ Switch
192
193 vars == <<Pubs, Subs, BroadcastBuf, MulticastBuf>>
194 Spec ==
195   Init /\ [][Next]_vars /\ WF_vars(BroadcastRcv)
196     /\ WF_vars(MulticastRcv)
197
198 P1 ==
199     [](Pubs[1].l = sAnnounce /\ Subs[1].accLvl >= Pubs[1].reqLvl)
200         => <>(/\ Subs[1].l = rAnnounce
201               /\ Subs[1].ch = Pubs[1].ch
202               /\ Subs[1].state = SUBSCRIBED))
203
204 P2 ==
205     [](Subs[1].state = SUBSCRIBED)
206         => (Subs[1].ch # NULL))
207
208 P3 ==
209     [](Pubs[1].l = s0 \/ Pubs[1].l = s1 \/ Pubs[1].l = s2)
210             => <>(Subs[1].l = rPublish))
211
212 P4 ==
213     [](Pubs[1].l = sSwitch_CH1)
214         => <>(Subs[1].l = rSwitch /\ Subs[1].ch = CH2))
215
216 P5 ==
217     [](Pubs[1].l = sSwitch_CH2)
218         => <>(Subs[1].l = rSwitch /\ Subs[1].ch = CH1))
219
220 P6 ==
221     [](Pubs[1].l = sSwitch_CH1 \/ Pubs[1].l = sSwitch_CH2)
222         => <>(Pubs[1].state = SWITCHING))
```

```
223 ============================================================================
```

Listing 5.32: $PubSub_5$, TLA+

```
@@ -1,2 +1,2 @@
--------------------------- MODULE PubSub_4 ---------------------------
+------------------------ MODULE PubSub_5 ------------------------
 EXTENDS Naturals
@@ -4,4 +4,5 @@ EXTENDS Naturals
 CONSTANTS PubsCount, SubsCount, IDLE, S0, S1, S2, SWITCHING, WAITING,
-    SUBSCRIBED, ANNOUNCE, PUBLISH, NULL, CH1, CH2, SWITCH
-VARIABLE Pubs, Subs
+    SUBSCRIBED, ANNOUNCE, PUBLISH, NULL, CH1, CH2, SWITCH, sAnnounce,
+    s0, s1, s2, sSwitch_CH1, sSwitch_CH2, rAnnounce, rPublish, rSwitch
+VARIABLE Pubs, Subs, BroadcastBuf, MulticastBuf

@@ -9,11 +10,11 @@ Init_Pubs ==
     [ [ i \in 1..PubsCount |-> [ state |-> IDLE, ch |-> CH1,
-        reqLvl |-> 0 ] ]
+        reqLvl |-> 0 , l |-> NULL ] ]
     EXCEPT
         ![1].reqLvl = 1 ]
-SendPrecond_Pub(i, m) ==
-    \/ (Pubs[i].state = IDLE /\ m = ANNOUNCE)
-    \/ ((Pubs[i].state = S0
-        \/ Pubs[i].state = S1
-        \/ Pubs[i].state = S2) /\ m = PUBLISH)
-    \/ Pubs[i].state = SWITCHING /\ m = SWITCH
+SendPrecond_Pub(P, i, m) ==
+    \/ (P[i].state = IDLE /\ m = ANNOUNCE)
+    \/ ((P[i].state = S0
+        \/ P[i].state = S1
+        \/ P[i].state = S2) /\ m = PUBLISH)
+    \/ P[i].state = SWITCHING /\ m = SWITCH
 NextState_Pub(s) ==
@@ -33,3 +34,17 @@ StateChange_Pub(i, m) ==
             (IF Pubs[i].ch = CH1 THEN CH2 ELSE CH1)
-        ELSE Pubs[i].ch ]
+        ELSE Pubs[i].ch,
+    ![i].l = NULL ]
+Label_Pub(P, i, m) ==
+    [ P EXCEPT
+    ![i].l =
+        IF m = ANNOUNCE THEN sAnnounce
+        ELSE IF m = PUBLISH THEN
+            IF P[i].state = S0 THEN s0
+            ELSE IF P[i].state = S1 THEN s1
+            ELSE IF P[i].state = S2 THEN s2
+            ELSE NULL
+        ELSE IF m = SWITCH THEN
+            IF P[i].ch = CH1 THEN sSwitch_CH1
+            ELSE sSwitch_CH2
+        ELSE NULL ]

@@ -37,3 +52,3 @@ Init_Subs ==
     [ [ i \in 1..SubsCount |-> [ state |-> WAITING, ch |-> NULL,
-        receivedState |-> IDLE, accLvl |-> 0 ] ]
+        receivedState |-> IDLE, accLvl |-> 0, l |-> NULL ] ]
     EXCEPT
@@ -60,3 +75,13 @@ StateChange_Sub(i, m, s, c) ==
```

```
57            ELSE Subs[i].ch ]
58  +Label_Sub(m) ==
59  +     IF m = ANNOUNCE THEN rAnnounce
60  +     ELSE IF m = PUBLISH THEN rPublish
61  +     ELSE IF m = SWITCH THEN rSwitch
62  +     ELSE NULL
63  +
64  +BroadcastBufInit == [ ready |-> FALSE, sender |-> NULL, m |-> NULL,
65  +     ch |-> NULL, state |-> NULL, reqLvl |-> 0 ]
66
67  +MulticastBufInit == [ ready |-> FALSE, sender |-> NULL, m |-> NULL,
68  +     ch |-> NULL, state |-> NULL, reqLvl |-> 0 ]
69
70  @@ -65,25 +90,93 @@ Init ==
71        /\ Subs = Init_Subs
72  +     /\ BroadcastBuf = BroadcastBufInit
73  +     /\ MulticastBuf = MulticastBufInit
74
75  -BroadcastComm(m) ==
76  +MulticastSend_Impl(P, S, m) ==
77        \E i \in 1..PubsCount:
78  -         /\ SendPrecond_Pub(i, m)
79  -         /\ Pubs' = [ j \in 1..PubsCount |->
80  -             StateChange_Pub(i, m)[j] ]
81  +         /\ SendPrecond_Pub(P, i, m)
82  +         /\ \A j \in 1..SubsCount:
83  +             ConnCond_Sub(j, P[i].ch) =>
84  +                 RecvPrecond_Sub(j, m, P[i].reqLvl)
85  +         /\ Pubs' = Label_Pub(P, i, m)
86  +         /\ Subs' = S
87  +         /\ MulticastBuf' = [ ready  |-> TRUE, sender |-> i, m |-> m,
88  +                              ch |-> P[i].ch,
89  +                              state |-> P[i].state,
90  +                              reqLvl |-> P[i].reqLvl ]
91  +
92  +BroadcastSend(m) ==
93  +     /\ ~BroadcastBuf.ready
94  +     /\ ~MulticastBuf.ready
95  +     /\ \E i \in 1..PubsCount:
96  +         /\ SendPrecond_Pub(Pubs, i, m)
97  +         /\ Pubs' = Label_Pub(Pubs, i, m)
98            /\ Subs' = [ j \in 1..SubsCount |->
99  -             IF RecvPrecond_Sub(j, ANNOUNCE, Pubs[i].reqLvl)
100 -             THEN StateChange_Sub(j, m, "dummy", Pubs[i].ch)[j]
101 -             ELSE Subs[j] ]
102 +                     [ Subs[j] EXCEPT !.l = NULL ] ]
103 +         /\ BroadcastBuf' = [ ready |-> TRUE, sender |-> i, m |-> m,
104 +             ch |-> Pubs[i].ch,
105 +             reqLvl |-> Pubs[i].reqLvl ]
106 +     /\ UNCHANGED MulticastBuf
107
108 -MulticastComm(m) ==
109 -     \E i \in 1..PubsCount:
110 -         /\ SendPrecond_Pub(i, m)
111 -         /\ \A j \in 1..SubsCount:
112 -             ConnCond_Sub(j, Pubs[i].ch) =>
```

```
113 -                RecvPrecond_Sub(j, m, Pubs[i].reqLvl)
114 -        /\ Subs' = [ k \in 1..SubsCount |->
115 -            IF ConnCond_Sub(k, Pubs[i].ch)
116 -            THEN StateChange_Sub(k, m, Pubs[i].state, Pubs[i].ch)[k]
117 -            ELSE Subs[k] ]
118 -        /\ Pubs' = [ j \in 1..PubsCount |->
119 -            StateChange_Pub(i, m)[j] ]
120 +BroadcastRcv ==
121 +    /\ BroadcastBuf.ready
122 +    /\ LET P == [ j \in 1..PubsCount |->
123 +                    StateChange_Pub(BroadcastBuf.sender,
124 +                        BroadcastBuf.m)[j] ]
125 +           S == [ j \in 1..SubsCount |->
126 +                    IF RecvPrecond_Sub(j, BroadcastBuf.m,
127 +                        BroadcastBuf.reqLvl)
128 +                    THEN [ StateChange_Sub(j, BroadcastBuf.m, "dummy",
129 +                                BroadcastBuf.ch)[j]
130 +                           EXCEPT !.l = Label_Sub(BroadcastBuf.m) ]
131 +                    ELSE [ Subs[j] EXCEPT !.l = NULL ] ]
132 +        IN
133 +        \/
134 +            /\ Pubs' = P
135 +            /\ Subs' = S
136 +            /\ BroadcastBuf' = BroadcastBufInit
137 +            /\ UNCHANGED MulticastBuf
138 +        \/
139 +            /\ ~MulticastBuf.ready
140 +            /\ MulticastSend_Impl(P, S, PUBLISH)
141 +            /\ BroadcastBuf' = BroadcastBufInit
142 +        \/
143 +            /\ ~MulticastBuf.ready
144 +            /\ MulticastSend_Impl(P, S, SWITCH)
145 +            /\ BroadcastBuf' = BroadcastBufInit
146 +
147 +BroadcastComm(m) == BroadcastSend(m) \/ BroadcastRcv
148 +
149 +MulticastSend(m) ==
150 +    /\ ~MulticastBuf.ready
151 +    /\ ~BroadcastBuf.ready
152 +    /\ MulticastSend_Impl(Pubs,
153 +        [ j \in 1..SubsCount |-> [ Subs[j] EXCEPT !.l = NULL ] ],
154 +        m)
155 +    /\ UNCHANGED BroadcastBuf
156 +
157 +MulticastRcv ==
158 +  /\ MulticastBuf.ready
159 +  /\ LET P == [ j \in 1..PubsCount |->
160 +                StateChange_Pub(MulticastBuf.sender,
161 +                    MulticastBuf.m)[j] ]
162 +         S == [ j \in 1..SubsCount |->
163 +                IF ConnCond_Sub(j, MulticastBuf.ch)
164 +                THEN [ StateChange_Sub(j, MulticastBuf.m,
165 +                        MulticastBuf.state, MulticastBuf.ch)[j]
166 +                        EXCEPT !.l = Label_Sub(MulticastBuf.m) ]
167 +                ELSE [ Subs[j] EXCEPT !.l = NULL ] ]
168 +    IN
```

```
169 +    \/
170 +        /\ Pubs' = P
171 +        /\ Subs' = S
172 +        /\ MulticastBuf' = MulticastBufInit
173 +        /\ UNCHANGED BroadcastBuf
174 +    \/
175 +        /\ MulticastSend_Impl(P, S, PUBLISH)
176 +        /\ BroadcastBuf' = BroadcastBufInit
177 +    \/
178 +        /\ MulticastSend_Impl(P, S, SWITCH)
179 +        /\ BroadcastBuf' = BroadcastBufInit
180 +
181 +MulticastComm(m) == MulticastSend(m) \/ MulticastRcv
182
183 @@ -99,7 +192,32 @@ Next ==
184
185 +vars == <<Pubs, Subs, BroadcastBuf, MulticastBuf>>
186  Spec ==
187 -    Init /\ [][Next]_<<Pubs, Subs>>
188 +  Init /\ [][Next]_vars /\ WF_vars(BroadcastRcv)
189 +    /\ WF_vars(MulticastRcv)
190 +
191 +P1 ==
192 +    [](Pubs[1].l = sAnnounce /\ Subs[1].accLvl >= Pubs[1].reqLvl)
193 +        => <>(/\ Subs[1].l = rAnnounce
194 +              /\ Subs[1].ch = Pubs[1].ch
195 +              /\ Subs[1].state = SUBSCRIBED))
196 +
197 +P2 ==
198 +    [](Subs[1].state = SUBSCRIBED)
199 +        => (Subs[1].ch # NULL))
200 +
201 +P3 ==
202 +    [](Pubs[1].l = s0 \/ Pubs[1].l = s1 \/ Pubs[1].l = s2)
203 +            => <>(Subs[1].l = rPublish))
204 +
205 +P4 ==
206 +    [](Pubs[1].l = sSwitch_CH1)
207 +        => <>(Subs[1].l = rSwitch /\ Subs[1].ch = CH2))
208 +
209 +P5 ==
210 +    [](Pubs[1].l = sSwitch_CH2)
211 +        => <>(Subs[1].l = rSwitch /\ Subs[1].ch = CH1))
212
213 -INV_1 ==
214 -    TRUE
215 +P6 ==
216 +    [](Pubs[1].l = sSwitch_CH1 \/ Pubs[1].l = sSwitch_CH2)
217 +        => <>(Pubs[1].state = SWITCHING))
218  ================================================================
```

Listing 5.33: $PubSub_4$ and $PubSub_5$ diff, TLA+

Overview of Generative AI Tools Used

For the translation of the abstract from English to German, ChatGPT in the currently available web version at `https://chatgpt.com` (09.2026–05.2026) was used. The following prompt was used: "Translate this text to German. This should be a 1:1 translation. Do not change wording, style and tone."

For spelling correction and grammar improvement of selected paragraphs, ChatGPT in the currently available web version at `https://chatgpt.com` (09.2026–05.2026) was used. The following prompt was used: "Proof-read the following paragraph and provide results in the format of a list containing the original mistake/inaccuracy and the proposed fix. Never change the original intent, style and tone."

List of Figures

List of Tables

List of Code Listings

Bibliography

[AAP22] Yehia Abd Alrahman, Shaun Azzopardi, and Nir Piterman. R-CHECK: A model checker for verifying reconfigurable MAS. In Piotr Faliszewski, Viviana Mascardi, Catherine Pelachaud, and Matthew E. Taylor, editors, *21st International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2022, Auckland, New Zealand, May 9-13, 2022*, pages 1518–1520. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2022.

[AMP06] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. In Antti Valmari, editor, *Model Checking Software*, pages 146–162, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[AP21] Yehia Abd Alrahman and Nir Piterman. Modelling and verification of reconfigurable multi-agent systems. *Auton. Agents Multi Agent Syst.*, 35(2):47, 2021.

[BBG+95] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 1–10. ACM Press, 1995.

[BBK+20] Sean Braithwaite, Ethan Buchman, Ismail Khoffi, Igor Konnov, Zarko Milosevic, Romain Ruetschi, and Josef Widder. A tendermint light client. *CoRR*, abs/2010.07031, 2020.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[BLTK24] Roman Bögli, Leandro Lerena, Christos Tsigkanos, and Timo Kehrer. A systematic literature review on a decade of industrial tla$^+$ practice. In Nikolai Kosmatov and Laura Kovács, editors, *Integrated Formal Methods - 19th International Conference, IFM 2024, Manchester, UK, November 13-15, 2024, Proceedings*, volume 15234 of *Lecture Notes in Computer Science*, pages 24–34. Springer, 2024.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 70–87, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, page 334–342, Berlin, Heidelberg, 2014. Springer-Verlag.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[DKJ18] Ali Dorri, Salil S. Kanhere, and Raja Jurdak. Multi-agent systems: A survey. *IEEE Access*, 6:28573–28593, 2018.

[DNZZ22] Luming Dong, Zhi Niu, Yong Zhu, and Wei Zhang. Specifying and verifying SDP protocol based zero trust architecture using TLA+. In *Proceedings of the 7th International Conference on Cyber Security and Information Engineering, ICCSIE 2022, Brisbane, QLD, Australia, September 23-25, 2022*, pages 35–43. ACM, 2022.

[dT26] dblp Team. dblp computer science bibliography – Monthly Snapshot XML Release of March 2026, March 2026.

[GGM18] Antonios Gouglidis, Christos Grompanopoulos, and Anastasia Mavridou. Formal verification of usage control models: A case study of usecon using TLA+. In Simon Bliudze and Saddek Bensalem, editors, *Proceedings of the 1st International Workshop on Methods and Tools for Rigorous System Design, MeTRiD@ETAPS 2018, Thessaloniki, Greece, 15th April 2018*, volume 272 of *EPTCS*, pages 52–64, 2018.

[GH22] Matthias Grundmann and Hannes Hartenstein. Verifying payment channels with tla$^+$. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2022, Shanghai, China, May 2-5, 2022*, pages 1–3. IEEE, 2022.

[GJZ24] Hua Guo, Yunhong Ji, and Xuan Zhou. The development of a tla$^+$ verified correctness raft consensus protocol. In Wenjie Zhang, Anthony K. H. Tung, Zhonglong Zheng, Zhengyi Yang, and Xiaoyang Wang, editors, *Web and Big Data - 8th International Joint Conference, APWeb-WAIM 2024, Jinhua, China, August 30 - September 1, 2024, Proceedings, Part V*, volume 14965 of *Lecture Notes in Computer Science*, pages 459–469. Springer, 2024.

[GZL+21] Song Gao, Bohua Zhan, Depeng Liu, Xuechao Sun, Yanan Zhi, David N. Jansen, and Lijun Zhang. Formal verification of consensus in the taurus distributed database. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, volume 13047 of *Lecture Notes in Computer Science*, pages 741–751. Springer, 2021.

[Has09] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *31st International Conference on Software Engineering (ICSE)*, pages 78–88, Vancouver, Canada, 2009. IEEE.

[Hel23] Andrew Helwer. Key Value Store. `https://github.com/tlaplus/Examples/blob/master/specifications/KeyValueStore/KeyValueStore.tla`, 2023. Accessed 2026-03-01.

[HRK23] A. Finn Hackett, Joshua Rowe, and Markus Alexander Kuppe. Understanding inconsistency in azure cosmos DB with TLA+. In *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1–12. IEEE, 2023.

[Hu24] Guanzhou Hu. MultiPaxos-SMR. `https://github.com/tlaplus/Examples/blob/master/specifications/MultiPaxos-SMR/MultiPaxos.tla`, 2024. Accessed 2026-03-01.

[JWSH21] Christine Jakobs, Matthias Werner, Karsten Schmidt, and Gerhard Hansch. Following the white rabbit: Integrity verification based on risk analysis results. In Björn Brücher, Christoph Krauß, Mario Fritz, Hans-Joachim Hof, and Oliver Wasenmüller, editors, *CSCS '21: Computer Science in Cars Symposium, Ingolstadt, Germany, 30 November 2021*, pages 6:1–6:9. ACM, 2021.

[KK20] Young-Mi Kim and Miyoung Kang. Formal verification of sdn-based firewalls by using TLA+. *IEEE Access*, 8:52100–52112, 2020.

[KKT19] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ model checking made symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA):123:1–123:30, 2019.

[KLPP19] Panagiotis Kouvaros, Alessio Lomuscio, Edoardo Pirovano, and Hashan Punchihewa. Formal verification of open multi-agent systems. In Edith Elkind, Manuela Veloso, Noa Agmon, and Matthew E. Taylor, editors, *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13-17, 2019*, pages 179–187. International Foundation for Autonomous Agents and Multiagent Systems, 2019.

[LAKM+] Leslie Lamport, Markus A. Kuppe, Stephan Merz, Andrew Helwer, William Schultz, Jeff Hemphill, Mariusz Ryndzionek, Igor Konnov, Thanh Hai Tran, Josef Widder, Jim Gray, Murat Demirbas, Guanzhou Hu, Giuliano Losa, Ron Pressler, Younes Akhouayri, Luming Dong, Zhi Niu, Lim Ngian Xin Terry, Gaurav Gandhi, Isaac DeFrain, Martin Harrison, Santhosh Raju, Cherry G. Mathew, Fransisca Andriani, and Ludovic Yvoz. TLA+ Examples. `https://github.com/tlaplus/Examples`. Accessed 2026-03-01.

[Lam02] Leslie Lamport. Paxos made simple, fast, and byzantine. In Alain Bui and Hacène Fouchal, editors, *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*, volume 3 of *Studia Informatica Universalis*, pages 7–9. Suger, Saint-Denis, rue Catulienne, France, 2002.

[Lam03] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2003.

[Lam09] Leslie Lamport. The pluscal algorithm language. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer, 2009.

[LQR17] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: an open-source model checker for the verification of multi-agent systems. *Int. J. Softw. Tools Technol. Transf.*, 19(1):9–30, 2017.

[LS21] Leslie Lamport and Fred B. Schneider. Verifying hyperproperties with TLA. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–16. IEEE, 2021.

[Nie20] Alexander Niederbuehl. Tower of Hanoi. `https://github.com/tlaplus/Examples/blob/master/specifications/tower_of_hanoi/HanoiSeq.tla`, 2020. Accessed 2026-03-01.

[Niy25] Rajdeep Niyogi. Tla$^+$based specification and verification of a team formation protocol with message loss. In Leonard Barolli, editor, *Advanced Information Networking and Applications - Proceedings of the 39th International Conference on Advanced Information Networking and Applications, AINA 2025, Barcelona, Spain, 9-11 April 2025, Volume 7*, volume 251 of *Lecture Notes on Data Engineering and Communications Technologies*, pages 314–325. Springer, 2025.

[OHH+23] Lingzhi Ouyang, Yu Huang, Binyu Huang, Hengfeng Wei, and Xiaoxing Ma. Leveraging TLA+ specifications to improve the reliability of the zookeeper coordination service. *CoRR*, abs/2302.02703, 2023.

[OP20] Nawar H. Obeidat and Carla Purdy. Modeling a smart school building system using UML and TLA+. In *3rd International Conference on Information and Computer Technologies, ICICT 2020, San Jose, CA, USA, March 9-12, 2020*, pages 131–136. IEEE, 2020.

[OP21] Nawar H. Obeidat and Carla Purdy. Improving security in SCADA systems through model-checking with TLA+. In *64th IEEE International Midwest Symposium on Circuits and Systems, MWSCAS 2021, Lansing, MI, USA, August 9-11, 2021*, pages 832–835. IEEE, 2021.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977.

[PS21] Smruti Padhy and Joe Stubbs. Designing and proving properties of the abaco autoscaler using TLA+. In Roderick Bloem, Rayna Dimitrova, Chuchu Fan, and Natasha Sharygina, editors, *Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers*, volume 13124 of *Lecture Notes in Computer Science*, pages 86–103. Springer, 2021.

[PSB18] Pedro Yuri Arbs Paiva, Osamu Saotome, and Christof Brandauer. Specification and verification of a multi-agent coordination protocol with TLA+. In *VIII Brazilian Symposium on Computing Systems Engineering, SBESC 2018, Salvador, Brazil, November 5-8, 2018*, pages 207–212. IEEE, 2018.

[RGS95] Ragunathan Rajkumar, Michael Gagliardi, and Lui Sha. The real-time publisher/subscriber inter-process communication model for distributed real-time systems: Design and implementation. In *1st Real-Time Technology and Applications Symposium (RTAS)*, pages 66–75, Chicago, IL, USA, 1995. IEEE.

[RRZB19] Mohammad Roohitavaf, Kun Ren, Gene Zhang, and Sami Ben-Romdhane. Logplayer: Fault-tolerant exactly-once delivery using grpc asynchronous streaming. *CoRR*, abs/1911.11286, 2019.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[SvdSV20] Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. Automated validation of state-based client-centric isolation with tla$^+$. In Loek Cleophas and Mieke Massink, editors, *Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops - ASYDE, CIFMA, and CoSim-CPS, Amsterdam, The Netherlands, September 14-15, 2020, Revised Selected Papers*, volume 12524 of *Lecture Notes in Computer Science*, pages 43–57. Springer, 2020.

[SZDT21] William Schultz, Siyuan Zhou, Ian Dardik, and Stavros Tripakis. Design and analysis of a logless dynamic reconfiguration protocol. In Quentin Bramas, Vincent Gramoli, and Alessia Milani, editors, *25th International Conference on Principles of Distributed Systems, OPODIS 2021, Strasbourg, France, December 13-15, 2021*, volume 217 of *LIPIcs*, pages 26:1–26:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[Woo92] Michael John Wooldridge. *The logical modelling of computational multi-agent systems*. The University of Manchester (United Kingdom), 1992.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla$^+$ specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999.

[YZF20] Jiaqi Yin, Huibiao Zhu, and Yuan Fei. Specification and verification of the zab protocol with TLA+. *J. Comput. Sci. Technol.*, 35(6):1312–1323, 2020.