University of Basel

Formal Specification and Decentralized Implementation of Monopoly Using TLA+ and Git

Bachelor Thesis

University of Basel - Faculty of Science
Department of Mathematics and Computer Science
Computer Networks Group
https://cn.dmi.unibas.ch

Examiner: Prof. Dr. Christian Tschudin
Supervisor: Dr. Erick Lavoie

Luca Gloor
luca.gloor@stud.unibas.ch

17.06.2025

Acknowledgments

First and foremost, I want to thank Prof. Dr. Tschudin and Dr. Erick Lavoie for the opportunity to work on this thesis. I especially thank Dr. Erick Lavoie for his guidance, expertise, and patience in supervising this thesis.

I would also like to express my gratitude to Tim Matter for introducing me to his implementation of the board game Catan. This helped me a lot to quickly setup a working foundation for this project.

Lastly, I want to thank Miriam Märki for her constant support and encouragement.

Abstract

Guaranteeing correctness of decentralized applications is a big challenge due to the complexity of concurrent behaviours. In this thesis, we formally specify and implement the board game Monopoly as a decentralized application to address this problem. Using the formal specification language TLA+, we model the core mechanics of the game, including randomness and auctions. We further introduce a verifiable auction protocol which is refined in multiple abstraction levels, with each refinement preserving the correctness properties of termination, agreement, validity, and integrity.

We implement the game using Git to synchronize the game state between players, omitting the need for a centralized server. The state is represented in a YAML file and player actions are modelled as Git commits, where each commit contains the new state.

Correctness of the design is validated by model checking with TLC and we evaluate the implementation using simulations. Additionally, we analyze the performance of the implementation on metrics including action latency and storage use. The results show the feasibility of using formal methods for multiplayer game logic, providing a framework for decentralized application development.

Table of Contents

1 Introduction

Ensuring correctness of decentralized applications is a well-known challenge in Computer Science. As these applications grow in complexity, so does the difficulty of verifying their behaviour in concurrent environments. This challenge is particularly relevant in systems that require consistency and coordination, as is the case in multiplayer games, where unexpected behaviour can easily emerge from seemingly harmless local actions. An option to manage this difficulty is to use formal specification languages to mathematically model applications and verify their possible behaviours before implementation.

The goal of this thesis is to formally specify and implement the board game Monopoly as a decentralized application while ensuring, that the core mechanics of the game are verifiable. This includes interactions such as auctions, turn-based play, and randomness. Additionally, we show how a decentralized version of Monopoly can be implemented using append-only logs and Git for synchronization between participants.

To address these challenges, we present several novel contributions. First, we develop a TLA+ specification of Monopoly. This includes a verifiable auction protocol which is refined in multiple abstraction levels. In each step, we prove that the correctness properties of *termination, agreement, validity*, and *integrity*, which we formally define, are preserved. Second, we show how this specification is implemented using YAML to represent the state, encoding player actions as Git commits and utilizing Git replication operations between repositories to synchronize the game state between participants. With this approach, we eliminate the need for a centralized server while maintaining correct game progression.

We structure the thesis as follows: In Chapter 2 we provide background on Monopoly, TLA+, append-only logs and Git. Chapter 3 shows the design of the game and the auction protocol using TLA+. We present the implementation from the specification to a playable multiplayer-game in Chapter 4. In Chapter 5 we evaluate both the design and its implementation through model checking and simulation tests while also checking the performance of the implementation in terms of delay and storage consumption. We discuss related work in Chapter 6, including contributions in model specification, and distributed applications. Finally, Chapter 7 concludes the thesis and provides possible directions for future work. In Appendix A, we provide the TLA+ model for Monopoly and the specifications of the auction procotol in Appendix B, C, and D. All source code is provided on Github.[1]

[1] Specifications: https://github.com/lgloor/bachelor-thesis-tla-specs,
Implementation: https://github.com/lgloor/monopoly

2
Background

In this chapter we give an overview of the topics and concepts which are used in the later chapters of this thesis. We provide a brief outline of the core mechanics of the board game Monopoly. Afterwards, we introduce the specification language TLA+ and a summary of its most important concepts and notations. Lastly, we present the append-only log data structure before showing how we use Git, an implementation of append-only logs, to update replicas in our decentralized Monopoly application.

2.1 Mechanics of Monopoly

Monopoly is a competitive turn-based board game. The goal of each player is to prevent going bankrupt by obtaining properties, on which other participants must pay rent when landing on them. In the following sections, we summarize the core concepts we use to later formally specify and implement the game.

2.1.1 Components

A standard Monopoly board consists of 40 spaces which are categorized into property (streets, railroads, and utilities), Chance, Community Chest, Jail, Go to Jail, Free Parking, and Go spaces. The amount of money is limited to a pre-defined amount ($118,660 in modern versions). It comes with two sets of cards, one for Community Chest and the other for Chance which are drawn when landing on the respective squares.

2.1.2 Setup

In the beginning, the token of each player is placed on the Go square. Every participant gets a fixed amount of money (usually $1,500). All properties belong to the bank and the cards of the Chance and Community Chest decks are shuffled.

2.1.3 Phases of a Turn

The turn of a player, hereafter called the *active player*, is divided into four phases:

1. Before rolling, the active player can mortgage, unmortgage, upgrade, or downgrade any properties they own.

2. Roll the two dice and move the corresponding amount of squares.
3. Perform any action that is associated with the square landed on. This may involve:
    - Purchasing a property if it is unowned and the player has enough money to afford it.
    - Auctioning a property when unowned and the player cannot afford or does not want to buy it.
    - Paying rent if the property is owned by another player.
    - Drawing a card from the Chance or Community Chest deck.
    - Paying tax.
    - Doing nothing, if the space belongs to the active player, is mortgaged or inherently requires no action.

    If the active player has rolled the same number with both dice, they will get to roll again, unless they have rolled doubles for the third consecutive time, in which case they must go to jail.
4. Before the active turn is given to the next player, all participants are allowed to mortgage, unmortgage, upgrade, or downgrade any of their owned properties.

2.1.4 Win Condition

When a player cannot come up with enough money to pay off an owed debt, e.g. rent when landing on a owned property or tax on a tax square, they go bankrupt to the corresponding creditor of the debt, which can either be another player or the bank. In this case, all of their assets are transferred to the creditor, and they are out of the game. The game ends when all participants except one have gone bankrupt. The remaining player is declared the winner.

2.2 TLA+

TLA+ [8] is a language used to formally specify the behaviours of concurrent systems. It was invented by Leslie Lamport and is based on discrete mathematics and temporal logic. Specifications can be written in the TLA+ Toolbox, an IDE for TLA+ which also comes with a model checker called TLC. The model checker allows us to simulate all possible behaviours of the specified system. This helps identifying errors and violations of defined properties. In the following subsections, we introduce the notation we use in Chapter 3 for specifying the design.

2.2.1 Definition

For definitions, TLA+ uses "$\triangleq$". This is used when defining new operators or local variables in `LET ... IN ...` blocks. In code, the definition symbol is typed ==.

2.2.2 Sets

Sets of elements are defined with $\{e_1, e_2, \dots, e_n\}$. For the set of integers from `lower` up to and including `upper`, we can use the integer range notation `lower..upper`. If `lower` $>$ `upper`, we obtain the empty set. To get the cardinality of set S, we write `Cardinality(S)`.

2.2.3 Functions

Functions, or mappings, can be explicitly defined by specifying what elements of the domain map to in the corresponding range e.g., we can define the function f from natural numbers to strings only at two points using $f \triangleq [2 \mapsto \text{"two"}, 7 \mapsto \text{"seven"}]$.

Another way is to specify the result of a function for all values in the domain. For this, we use $[x \in S \mapsto e]$, where S is the domain of the function and e is an expression. For example, to define the function g, mapping the numbers 1..5 to their square, we write $g \triangleq [x \in 1..5 \mapsto x * x]$.

Additionally, to obtain the set of all possible mappings from S to T, with S, T both sets, defining the domain and the range, we use $[S \rightarrow T]$. Notice that the arrows used are different. While for explicit definitions we use "`|->`", to specify the set of all functions we write "`->`".

2.2.4 Function Operators

To simplify working with functions, TLA+ offers a couple of operators:
Using the `DOMAIN` operator, we get the set of all values, for which the function is defined. To illustrate, `DOMAIN f` would give us $\{2, 7\}$. We retrieve values of a function with $f[i]$, where $i \in$ `DOMAIN f`.

The `EXCEPT` operator allows us to define the function $\hat{f}$ equal to f with the exception of some changes. This omits the need to repeat all values of the function that should remain the same. For example, to define function h, equal to g from Section 2.2.3 except that $h[1] = 0$ and $h[3] = -1$, we write:

$$\begin{aligned} h \triangleq [f \texttt{ EXCEPT } &![1] = 0, \\ &![3] = -1] \end{aligned}$$

2.2.5 Sequences

Sequences, also called tuples or lists, in TLA+ are special functions whose domain is the integer range $1..n$ where n is the length of the sequence. They are created using $\langle s_1, s_2, \dots, s_n \rangle$ which is written in code as `<<`$s_1, s_2, \dots, s_n$`>>`. Unlike most programming languages, indexing of sequences starts at one. Elements of sequences are retrieved the same way as in functions by using their corresponding index. Thus, to access s_i of sequence `S` $\triangleq \langle s_1, s_2, \dots, s_n \rangle$, we write `S[i]`, where $i \in 1..n$.

2.2.6 Records

Another special subset of functions are records. Records are functions whose domain consists only of strings. They can be compared to structs in C-like programming languages and are very helpful for grouping attributes together in a coherent way. For records, TLA+

offers syntactic sugar which simplifies accessing their values. For example, to obtain the value of field ”f1” in record r, we either use r[”f1”] or the simplified r.f1, omitting the quotation marks and square brackets.

2.2.7 Constants

In TLA+, constants are defined using `CONSTANTS` $C_1, C_2, \ldots, C_n$ where C_i is a placeholder for any identifying name. There may be no two constants with the same name. The value of a constant is not allowed to change at any point in the specification. When model checking with TLC, the values of the declared constants must be assigned in the *What is the model?* section of the TLA+ Toolbox.

2.2.8 Variables

Similar to defining constant values, variables are declared using the `VARIABLES` keyword. Unlike constants, variables are allowed to change when stepping from one state to the next. Their initial values are assigned in the `Init` operator of the specification.

2.2.9 Primed Variables

In order to define how a variable is changed after an action has been complete, we use the name of the variable with an apostrophe (prime). For example, to specify that variable x is incremented by one in the next step, we write $x' = x + 1$. In contrast to usual programming languages, we are also required to specify all variables that do not change in an action. Because writing $v' = v$ for each unchanged variable can be tedious, TLA+ offers the operator `UNCHANGED` $\langle v_1, v_2, \ldots, v_m \rangle$ as shorthand notation for $v_1' = v_1 \wedge v_2' = v2 \wedge \ldots \wedge v_m' = v_m$.

2.2.10 Stuttering Steps

A stuttering step is a transition where the state of the system does not change, meaning that the value of all variables remain the same. Stuttering steps are a necessary feature of TLA+, which allows specifications to abstract over time and implementation details. In multiprocess systems, it often occurs that some processes make progress while others remain idle. Essentially, these idle processes experience stuttering steps.

2.2.11 Weak Fairness

TLA+ allows an infinite amount of consecutive stuttering steps to occur. This can make it hard to make assertions on things that must *eventually* happen in a system. For this, TLA+ offers the concept of weak fairness assumptions written as $\text{WF}_{vars}(A)$ where *vars* is the sequence of all variables of the specification. This allows us to assert that if action A remains continuously enabled, then an A step must eventually occur. In code, we write this as `WF_vars(A)`.

2.3 Append-Only Logs

An append-only log is a data structure in the form of a ”totally-ordered replicated list of messages, such that each index has at most one unique message associated to it and indexes are associated with new messages from lowest to greatest.” [10]. Its key property is that new messages can only be appended to the end of the log. After a message is appended it must never be modified or removed.

This makes append-only logs a suitable data structure for decentralized applications. When receiving updates from peers it becomes immediately clear, whether all received messages have already been replicated or if some are new compared to the local replica. In the latter case, the missing messages are simply appended to one’s own replica of the log.

2.4 Git

Its official website defines Git as ”a free and open source distributed version control system [. . .].” [1]. It is based on append-only directed acyclic graphs of commits, in the sense that each change (i.e. addition, modification, or removal) to any files requires appending a new message. This way, Git can maintain a reliable history of all modifications. In the following sections, we assume basic knowledge of Git and only introduce concepts which we use for the implementation in Chapter 4.

2.4.1 Commit

Commits in Git are objects that represent a snapshot of a replica at a given point in time. While they are very intricate in detail, it is only relevant for us that they contain the following information:

1. References to zero or more *parent* commits specifying the previous state.
2. The new state of file(s) after the applied changes.
3. A message used to describe the applied changes.

When adding a new commit at the end of linear commit history, this corresponds to appending a new message in an append-only log. In our implementation, we use commit objects to represent all the actions that players have taken in the game.

2.4.2 Remote

A Git remote is a reference to a Git repository behind a corresponding URL, which is typically hosted on a remote server (e.g. Github). The most common remote is called *origin* which is the default name when cloning a Git repository from a server. While the origin remote is created automatically when cloning a remote repository, others can manually be added using `git remote add <name> <URL>`. To propagate changes between local and remote git repositories, Git offers the following operations:

2.4.2.1 Push

The `git push` operation is used to upload local changes to a remote repository. For example, `git push origin main` replicates the latest commits from the local main branch to the main branch of the remote called origin.

2.4.2.2 Fetch

In contrast to pushing, `git fetch` is used to download updates from a remote repository. The retrieved changes are not directly included in the local replica and must be merged manually. To fetch from the main branch of the origin remote, we execute `git fetch origin main`.

3 Design

This chapter introduces the design of Monopoly as a whole and in a more specific fashion, an auction protocol which can be used in the game's implementation. The designs are written in TLA+, a formal specification language created by Leslie Lamport which not only helps us to define the applications behaviour and structure but also allows us to verify the correctness of the design in Section 5.1. We first show how we approach system specifications in general. AFterwards, we go into more detail for the specific applications of Monopoly and the auction mechanism. For the auction protocol in particular, we also demonstrate how to use refinement to prove that one specification implies another.

3.1 General Approach Using TLA+

When specifying a system or algorithm in TLA+ it is important to first think about how the current state is represented. We do this using constant values that stay the same over all behaviors, and variables that are allowed to change when transitioning from one state to the next.

Next we define the properties we want the design to have:

1. **Safety properties** are assertions on what the system *is allowed to* do in each possible behavior of the system. They may apply on a single state, in which case they are called invariants, or steps (that is pairs of subsequent states) to constrain allowed transitions. An example in Monopoly would be that the total amount of money in play is always equal to some pre-defined number. Convention is that every specification defines a `TypeOK` invariant which is responsible for checking that the values of all variables are within their expected range.

2. **Liveness properties** define conditions that must *eventually* hold in all possible behaviors of the system. One such property could be termination which specifies that all behaviors must eventually halt, i.e. end in an infinite sequence of stuttering steps.

In a third step, we define the set of initial states the system can take on. We do this by specifying the range of values that each variable can take on in a valid initial state.

Lastly, we specify all possible actions that lead from one state to a next one. We do this by describing the required pre-conditions for an action to be enabled, and the ways in which the variables do or do not change when the action is taken.

3.2 Monopoly

In this section we specify the design for Monopoly using the four-step approach from Section 3.1. We do not cover all actions of the specification as this would go into too much detail. For reference, the full design can be seen in Appendix A.

3.2.1 State Representation

The most important constant values for the state of Monopoly are:

1. **NumPlayers**: The amount of players participating in the game.
2. **StartingMoney**: The amount of money that each player starts with.
3. **TotalMoney**: The total amount of money in play.

The other constants can be seen in Appendix A. They are either model values, relevant for other parts of the state or help in constraining the size of the state space e.g. *DiceMax* defines the highest value one die can show which influences the amount of possible outcomes of two dice.

For variables we differentiate between three different types.

1. **Player Attributes** are responsible for keeping track of properties that refer to a player's state. They are all defined as mappings from players to a defined range of values. As an example, *inJail* is a mapping from players to the set {`TRUE, FALSE`} which indicates, whether each player is in jail at the current moment.
2. **Global Attributes** track values that are important not only to players but to the game as a whole. An example is *turnPlayer* which defines who's turn it is in the current state. Another is *board*, responsible for the attributes of all squares of the board such as their order, whether a property is mortgaged, or the amount of tax/rent that is owed when landing on them.
3. **Pseudo-Variables** could be defined as constants but are defined as variables to make their values more transparent to readers of the specification. The variable *chanceCards* for instance, never changes. Since it is not trivial to a reader, however, that it should be a sequence of records we set its value in the initial state but never change it afterwards.

3.2.2 Properties

In the specification for Monopoly, we only define safety properties since termination cannot be guaranteed. For further elaboration on this claim, see Section 5.1.1.3. Since the amount of variables for Monopoly is rather large, so is the `TypeOK` invariant which is the reason we will only look at some of the constraints.

In Figure 3.1 we show the value ranges for each of the player attributes. Player's *positions* are constrained such that they cannot be outside of the board. The amount of *money* per player must be a positive number while not being higher than the total amount of money in the game. The values of *inJail* and *isBankrupt* can only be true or false for each player, and the *jailTime* of a player cannot exceed 2 since they should latest get out of jail in the third round after going to jail.

$$\begin{array}{l} \land \forall p \in 1 \,..\, NumPlayers : \\ \quad \land positions[p] \in 1 \,..\, Len(board) \\ \quad \land money[p] \in 0 \,..\, TotalMoney \\ \quad \land inJail[p] \in \text{BOOLEAN} \\ \quad \land isBankrupt[p] \in \text{BOOLEAN} \\ \quad \land jailTime[p] \in 0 \,..\, 2 \end{array}$$

Figure 3.1: TypeOK Definition of Player Attributes in Monopoly

We depict the constraints on two global attributes in Figure 3.2. Each turn is modeled to consist of multiple phases, each of which enables different actions. The possible values are constrained in the range assertion on *phase*. Similarly to the player attribute *money*, also the *bankMoney* must not be negative or higher than the total amount of money available.

$$\begin{array}{l} \land phase \in \{\text{“pre-roll”}, \text{“roll”}, \text{“post-roll”}, \text{“bankruptcy-prevention”}, \\ \qquad\qquad\quad \text{“doubles-check”}, \text{“free-4-all”}\} \\ \land bankMoney \in 0 \,..\, TotalMoney \end{array}$$

Figure 3.2: TypeOK Definition of Some Global Attributes in Monopoly

3.2.3 Initial State

The initial state of the game is defined in the *Init* formula of the specification in Appendix A. Every player is placed on the first square (= Go), and has a certain amount of *money* which is given by the *StartingMoney* constant. The remaining money stays with the bank. None of the players are in jail, have not been in jail for any rounds, or are bankrupt. Action is given to the first player in the *pre-roll* phase. The order and types of the squares on the board is initialized as are the Chance and Community Chest cards. For the two card decks it is vital that they each only contain one Get Out Of Jail Free card and that this is placed at the last position of the deck. We explain this constraint in Section 3.2.4.5. No player owns any of the Get Out Of Jail Free cards yet. Since it is not the *free-4-all* phase yet, the *free4AllOrder* is not initialized and there exists no *debt* that must be payed off yet.

3.2.4 Actions

Because there are many possible actions in Monopoly, we will limit ourselves to show only those that we think are the most interesting or might need some explanation to be fully understood. To see all available actions, please refer to the full specification in Appendix A. Experienced Monopoly players may notice the absence of trading. Since it is not in the scope of this project to derive a protocol for trading assets between players, this is not provided as an action in the specification.

3.2.4.1 Mortgaging a Property

For a player to mortgage a property as in Figure 3.3 the pre-conditions are that said property is owned by the player and that it is not already mortgaged. Additionally, if the property is a street then none of the streets of the same set are allowed to be higher than level one, which corresponds to them not having any buildings on them. Using the `LET ... IN ...` structure of TLA+, we define the local variable *mortgageValue* to be half of the property's value using floor division, as indicated by the $\div$ symbol. After execution, the property should be mortgaged and the player gets the mortgage value of the property from the bank. If the bank does not have the full amount available, the player only gets the amount of money that remains in the bank. Thus, it is possible that a player mortgages a property for no monetary gain if the bank has no money left. Since mortgaging properties is allowed in multiple phases of the game, `MortgageProperty` is a sub-action of `PreRollMortage`, `BankruptcyPreventionMortage`, and `F4AMortage` which all set constraints on the current phase and specify the unchanged variables of the action.

$$
\begin{array}{l}
\mathit{MortgageProperty}(\mathit{player}) \triangleq \exists\, \mathit{idx} \in \mathit{ownedPropertyIdxs}(\mathit{player}) : \\
\quad \land \neg \mathit{board}[\mathit{idx}].\mathit{mortgaged} \\
\quad \land \mathit{board}[\mathit{idx}].\mathit{type} = \text{“street”} \Rightarrow \mathit{noStreetFromSameSetHasBuildings}(\mathit{idx}) \\
\quad \land \textsc{let } \mathit{mortgageValue} \triangleq \mathit{board}[\mathit{idx}].\mathit{value} \div 2 \\
\qquad \textsc{in} \quad \land \mathit{board}' = [\mathit{board} \textsc{ except } ![\mathit{idx}].\mathit{mortgaged} = \textsc{true}] \\
\qquad \qquad \land \mathit{CollectFromBank}(\mathit{player}, \mathit{mortgageValue})
\end{array}
$$

Figure 3.3: TLA+ Definition of MortgageProperty Action

3.2.4.2 Rolling While Not in Jail

In Figure 3.4, we show the specification for when it is a player's turn to roll the dice while they are not in jail. The result of the action depends on whether or not they roll the same number with both dice. If the two dice show different numbers then the player will get to move the sum of the dice before moving to the next phase, and the count of consecutive doubles is reset to zero.

When the results of the dice are equal and the player has already rolled doubles twice before, then they will go straight to jail. If it is not their third consecutive doubles they will progress normally but the doubles count increases by one.

```
$RollAndMove \triangleq$
    $\exists\, d1,\ d2 \in 1 \,..\, DiceMax :$
      $\land\ \neg terminated$
      $\land\ phase = \text{“roll”}$
      $\land\ inJail[turnPlayer] = \text{FALSE}$
      $\land$ IF $d1 \neq d2$
         THEN $\land\ MoveAfterRoll(d1 + d2)$
              $\land\ doublesCount' = 0$
              $\land\ phase' = \text{“post-roll”}$
              $\land$ UNCHANGED $\langle inJail,\ free4AllOrder \rangle$
         ELSE IF $doublesCount = 2$  Current throw is 3rd consecutive doubles
                 THEN $\land\ GoToJail$
                      $\land$ UNCHANGED $\langle bankMoney,\ money \rangle$
                 ELSE $\land\ MoveAfterRoll(d1 + d2)$
                      $\land\ doublesCount' = doublesCount + 1$
                      $\land\ phase' = \text{“post-roll”}$
                      $\land$ UNCHANGED $\langle inJail,\ free4AllOrder \rangle$
      $\land$ UNCHANGED $\langle board,\ chanceCards,\ communityChestCards,\ debt,\ goojfCcOwner,$
                      $goojfChOwner,\ isBankrupt,\ jailIndex,\ jailTime,\ turnPlayer \rangle$
```

Figure 3.4: TLA+ Definition of RollAndMove Action

3.2.4.3 Rolling While in Jail

Figure 3.5 shows the action of a player rolling when they are in jail. The outcome of this action again relies on the results of the dice but also on the amount of money they own. In case the player rolls doubles, they are released from jail for free and continue as normal, with the exception that they will not get to roll again despite having rolled doubles. Thus, the *doublesCount* value is not increased.

When the dice don't show the same result and the player has missed doubles for the third round while in jail, they are released from jail while moving the amount of the two dice. Additionally they will have to pay a pre-defined fine for having missed doubles three times in a row. Should they not have enough money to do so at the moment, they will enter the bankruptcy-prevention phase with an open debt to the bank. This debt must be payed off before they can continue their turn.

```
RollInJail ≜
    ∃ d1, d2 ∈ 1 . . DiceMax :
        ∧ ¬terminated
        ∧ phase = "roll"
        ∧ inJail[turnPlayer] = TRUE
        ∧ IF d1 ≠ d2
            THEN IF jailTime[turnPlayer] = 2  has missed doubles for the 3rd time
                    THEN ∧ MoveAfterRoll(d1 + d2)
                         ∧ jailTime′ = [jailTime EXCEPT ![turnPlayer] = 0]
                         ∧ inJail′ = [inJail EXCEPT ![turnPlayer] = FALSE]
                         ∧ IF money[turnPlayer] ≥ JailFine
                             THEN ∧ PayBank(turnPlayer, JailFine)
                                  ∧ phase′ = "post-roll"
                                  ∧ UNCHANGED ⟨debt⟩
                             ELSE ∧ phase′ = "bankruptcy-prevention"
                                  ∧ debt′ = [creditor ↦ NULL,
                                             amount ↦ JailFine,
                                             nextPhase ↦ "post-roll"]
                         ∧ UNCHANGED ⟨free4AllOrder⟩
                    ELSE ∧ jailTime′ = [jailTime EXCEPT ![turnPlayer] = @ + 1]
                         ∧ initializeFree4All
                         ∧ UNCHANGED ⟨money, bankMoney, positions, inJail, debt⟩
            ELSE ∧ MoveAfterRoll(d1 + d2)  Player will not get to roll again even if they rolled doubles.
                 ∧ jailTime′ = [jailTime EXCEPT ![turnPlayer] = 0]
                 ∧ inJail′ = [inJail EXCEPT ![turnPlayer] = FALSE]
                 ∧ phase′ = "post-roll"
                 ∧ UNCHANGED ⟨free4AllOrder, debt⟩
        ∧ UNCHANGED ⟨board, chanceCards, communityChestCards, doublesCount,
                     goojfCcOwner, goojfChOwner, isBankrupt, jailIndex, turnPlayer⟩
```

Figure 3.5: TLA+ Definition of RollInJail Action

3.2.4.4 Paying Rent for Utilities

Paying rent for utilities is another action that depends on dice rolls. This time, their sum defines the amount that the player landing on the utility owes the owner. Similarly to Section 3.2.4.3, the player must come up with enough money to pay the rent by mortgaging or downgrading properties should they not have the required amount. The specification of the action can be seen in Figure 3.6.

```
$TryPayUtilRent \triangleq$
    IF $currentSquare.type \neq$ "util"
     THEN FALSE
     ELSE $\exists\, d1, d2 \in 1\,..\,DiceMax$ :
               $\land \lnot terminated$
               $\land phase =$ "post-roll"
               $\land$ LET $owner \triangleq currentSquare.owner$
                      $multiplier \triangleq$ IF $ownsBothUtilities(owner)$ THEN 10 ELSE 4
                      $rentCost \triangleq (d1 + d2) * multiplier$
                  IN  $\land owner \notin \{NULL, turnPlayer\}$
                      $\land$ IF $money[turnPlayer] \geq rentCost$
                          THEN $\land PayPlayer(turnPlayer, owner, rentCost)$
                               $\land phase' =$ "doubles-check"
                               $\land$ UNCHANGED $\langle debt \rangle$
                          ELSE $\land debt = NULL$
                               $\land debt' = [creditor \mapsto owner,$
                                            $amount \mapsto rentCost,$
                                            $nextPhase \mapsto$ "doubles-check"$]$
                               $\land phase' =$ "bankruptcy-prevention"
                               $\land$ UNCHANGED $\langle money \rangle$
               $\land$ UNCHANGED $\langle bankMoney, board, chanceCards, communityChestCards, doublesCount,$
                              $free4AllOrder, goojfCcOwner, goojfChOwner, inJail, isBankrupt,$
                              $jailIndex, jailTime, positions, turnPlayer \rangle$
```

Figure 3.6: TLA+ Definition of TryPayUtilRent Action

3.2.4.5 Drawing Chance/Community Chest Cards

As we mentioned in Section 3.2.3, it is crucial that the Get Out of Jail Free cards in each card deck are placed at the very end. We set this constraint because it simplifies preventing the card from being drawn while another player already owns it. We do this by excluding the corresponding index from the range of eligible cards. This can be seen in Figure 3.7. In our design, we model that the *turnPlayer* draws any card from the deck. In the board game version, the player will always draw the top-most card which is placed at the bottom of the pile after execution or kept by the player if it is a Get Out of Jail Free card. We make this design choice to imitate drawing a card at random. In the implementation, this ensures that the order in which cards are drawn remains partial-information even when all cards are known.

```
$DrawAndExecuteChanceCard \triangleq$
    IF $currentSquare.type \neq$ "chance"
     THEN FALSE
     ELSE   $\land \neg terminated$
            $\land phase =$ "post-roll"
            $\land \exists cardIdx \in$ IF $goojfChOwner = NULL$
                        THEN $1 \,..\, Len(chanceCards)$
                          ELSE $1 \,..\, (Len(chanceCards) - 1)$ :
                LET $card \triangleq chanceCards[cardIdx]$
                IN  $ExecuteCard(card)$
            $\land$ UNCHANGED $\langle board, chanceCards, communityChestCards, isBankrupt,$
                             $jailIndex, jailTime, turnPlayer \rangle$
```

Figure 3.7: TLA+ Definition of DrawAndExecuteChanceCard Action

3.3 Auction

In this section we present the design of an auction protocol that we used to implement auctions in Monopoly. Our specification is general enough that we believe it could also be used for other decentralized applications. We approach the specification with a top-down strategy in three steps. In the first version we define the required properties of the protocol and derive a minimal set of actions that fulfill these conditions. From there on we introduce additional features until we arrive at a refined design using append-only logs as a data structure for message publication, which corresponds to our implementation. We verify that each refinement still follows the required properties.

3.3.1 Properties

We introduce here the correctness properties of the auction protocol. For each, we give a descriptive definition and additionally provide the formal specification.

3.3.1.1 Termination

In order for an auction to reach *termination*, all participants of the auction must eventually make a decision regarding the outcome of an auction. Moreover, once a participant makes a decision, they never revert. The formal specification of this property can be seen in Figure 3.8. The specification is given in two steps. First, *terminated* verifies whether all participants have made a decision in the current state. Second, *termination* asserts that eventually, *terminated* will stay true forever, as captured by the *eventually* ($\diamond$) *always* ($\Box$) operator. Since the termination property contains assertions on something that must *eventually* happen, this is a liveness property.

$$terminated \triangleq \forall p \in Participants : winner[p] \neq UNKNOWN$$

$$termination \triangleq \diamond\Box terminated \quad \text{liveness}$$

Figure 3.8: TLA+ Definition of the Termination Property for Auctions

3.3.1.2 Agreement

The *agreement* property is a safety property which specifies that participants may at no point disagree on the winner of the auction. More formally it asserts that always, for each pair of participants, if both have picked a winner this must be the same. To see the TLA+ definition, refer to Figure 3.9.

$$\begin{aligned} \mathit{agreed} \triangleq\ & \forall\, p,\ p2 \in \mathit{Participants} : \\ & \quad \lor\ \mathit{winner}[p] = \mathit{UNKNOWN} \\ & \quad \lor\ \mathit{winner}[p2] = \mathit{UNKNOWN} \\ & \quad \lor\ \mathit{winner}[p] = \mathit{winner}[p2] \end{aligned}$$

$$\mathit{agreement} \triangleq \Box \mathit{agreed} \quad \text{safety}$$

Figure 3.9: TLA+ Definition of the Agreement Property for Auctions

3.3.1.3 Validity

To achieve *validity*, two conditions must be met. Firstly, the winner of the auction must have the highest bid of all participants. In Figure 3.10 we call this *winWithHigherBid*. Additionally, to be *solvable*, no participant can bid a negative amount of money or more than they possess. This is shown in figure Figure 3.10.

$$\begin{aligned} \mathit{solvable} \triangleq\ & \forall\, p \in \mathit{Participants} : \\ & \quad \mathit{lastBid}[p] \in 0 \,..\, (\mathit{initialMoney}[p]) \end{aligned}$$

$$\begin{aligned} \mathit{winWithHigherBid} \triangleq\ & \forall\, p,\ p2 \in \mathit{Participants} : \\ & \quad \mathit{winner}[p2] = p \Rightarrow \mathit{lastBid}[p] > \mathit{lastBid}[p2] \lor p = p2 \end{aligned}$$

$$\mathit{valid} \triangleq \mathit{solvable} \land \mathit{winWithHigherBid}$$

$$\mathit{validity} \triangleq \Box \mathit{valid} \quad \text{safety}$$

Figure 3.10: TLA+ Definition of the Validity Property for Auctions

3.3.1.4 Integrity

While we already specified in Section 3.3.1.1 that participants can never revert from having chosen a winner, their choice is not binding. By introducing *integrity* as seen in Figure 3.11, we assert that in any step, participants may not later select a different winner if they have already decided on the winner of the auction.

$$\begin{aligned} \mathit{winnerStaysSame} \triangleq\ & \forall\, p \in \mathit{Participants} : \\ & \quad \mathit{winner}[p] \neq \mathit{UNKNOWN} \Rightarrow \mathit{winner}'[p] = \mathit{winner}[p] \end{aligned}$$

$$\mathit{integrity} \triangleq \Box[\mathit{winnerStaysSame}]_{A1vars} \quad \text{safety}$$

Figure 3.11: TLA+ Definition of the Integrity Property for Auctions

3.3.2 Abstract Model (Auction1)

We begin with an abstract model of the auction procedure that implements the minimal amount of state required to fulfill the properties of the last section. The purpose of this specification is to serve as a baseline for later refinements in Section 3.3.3 and Section 3.3.4, which will progressively add more state to reflect our concrete implementation. The full design can be seen in Appendix B.

3.3.2.1 State Representation

To represent the state of the abstract auction, we define the following constants:

1. **UNKNOWN** is a model value for signifying that a participant has not yet decided who (if anyone) has won the auction.
2. **NONE** is a model value with which they determine that no participant has won the auction. In contrast to *UNKNOWN*, this means that the participants made the decision that no participant has won the auction.
3. **Participants** defines the set of participants of the auction.
4. **MaxAmount** $\in \mathbb{N}$ determines the maximum amount of money a participant can have at the start of the auction which in turn determines the highest bid they can place.

Additionally we define the following variables:

1. The **initialMoney** pseudo-variable is a mapping from participants to the range $0..\mathit{MaxAmount}$ indicating the amount of money each participant has when the auction starts.
2. **lastBid** stands for the last bid each participant has made and is thus modeled as a mapping from participants to the set of natural numbers.
3. The mapping **winner** from participants to the set $\mathit{Participants} \cup \{\mathit{UNKNOWN},\ \mathit{NONE}\}$ denotes who each participant has decided is a winner, with the possibility of choosing none of the participants.

3.3.2.2 Actions

For a participant to take the bid action visible in Figure 3.12, the only pre-condition is that no participant has already determined a winner. They can then choose a new bid that is higher than their previous bid. Notice that we do not set any restrictions regarding the bets of the other participants: If we did the specification would force participants to synchronize prior to updating their bid.

$$\begin{array}{l}
A1Bid \triangleq \\
\quad \land \forall p \in Participants : winner[p] = UNKNOWN \\
\quad \land \exists p \in Participants : \\
\qquad\quad \exists newBid \in (lastBid[p] + 1) \,..\, initialMoney[p] : \\
\qquad\qquad lastBid' = [lastBid \text{ EXCEPT } ![p] = newBid] \\
\quad \land \text{UNCHANGED } \langle winner, initialMoney \rangle
\end{array}$$

Figure 3.12: TLA+ Definition of the Bid Action in Auction1

After some time, a first participant will choose a winner for the auction. We show this process in the *FirstChooseWinner* action in Figure 3.13. Again, the only pre-condition states that no participant has chosen a winner yet. There are two possible outcomes for choosing a winner: Either the winner is a participant or none. In the first case the winner must have the single highest lastBid value at the moment. In the second case, there are no conditions on the status of bids to allow the largest possible range of later implementations.

In any case, after the first participant has chosen the winner, the other participants copy their winner in the *OthersChooseWinner* action. This ensures that the *agreement* property is always fulfilled. This corresponds to the requirement that once a participant has made a decision, no participant will later contradict that decision.

$$\begin{array}{l}
A1FirstChooseWinner \triangleq \\
\quad \land \forall p \in Participants : winner[p] = UNKNOWN \\
\quad \land \lor \exists p, p2 \in Participants : \\
\qquad\qquad \land \forall p3 \in Participants \setminus \{p\} : lastBid[p] > lastBid[p3] \\
\qquad\qquad \land winner' = [winner \text{ EXCEPT } ![p2] = p] \\
\qquad \lor \exists p \in Participants : winner' = [winner \text{ EXCEPT } ![p] = NONE] \\
\quad \land \text{UNCHANGED } \langle lastBid, initialMoney \rangle
\end{array}$$

$$\begin{array}{l}
A1OthersChooseWinner \triangleq \\
\quad \land \exists p, p2 \in Participants : \\
\qquad \land winner[p] \neq UNKNOWN \\
\qquad \land winner[p2] = UNKNOWN \\
\qquad \land winner' = [winner \text{ EXCEPT } ![p2] = winner[p]] \\
\quad \land \text{UNCHANGED } \langle lastBid, initialMoney \rangle
\end{array}$$

Figure 3.13: TLA+ Definition of the ChooseWinner Actions in Auction1

3.3.2.3 Weak Fairness Assumptions

Since TLA+ allows for an infinite amount of stuttering steps, it might occur that the *termination* property is violated when some participant does not choose a winner by taking stuttering steps only. To prevent this from happening, we assert weak fairness on the actions *FirstChooseWinner* and *OthersChooseWinner*, shown in Figure 3.14. This ensures that these actions must eventually occur if they remain continuously enabled. With this we guarantee that termination can only be violated by an error in the specification.

$$\begin{aligned}
&A1FairSpec \triangleq \\
&\quad \land A1Init \\
&\quad \land \Box[A1Next]_{A1vars} \\
&\quad \land \mathrm{WF}_{A1vars}(A1FirstChooseWinner) \\
&\quad \land \mathrm{WF}_{A1vars}(A1OthersChooseWinner)
\end{aligned}$$

Figure 3.14: Assumptions on Weak Fairness in Auction1

3.3.3 Round-Based Model (Auction2)

In this refinement of the abstract model we present the first version of the protocol that might seem applicable in a real-world situation. We do this by introducing the concept of rounds and by constraining the bids of the participants to depend on previous bids of the other participants. Additionally, participants may pass to indicate that they withdraw from the auction. Using these additions, we then set deterministic boundaries for choosing the winner of the auction. The whole specification can be seen in Appendix C.

3.3.3.1 Withdrawing from an Auction

When a participant no longer wants to increase their bid either because they don't have the required funds to do so or for strategic reasons, they have the opportunity to pass. With this, they mark their permanent withdrawal from the auction. The consequences are that they will no longer be allowed to place any new bids. All pre and post-conditions can be seen in Figure 3.15.

$$\begin{aligned}
&A2Pass \triangleq \exists\, p \in Participants : \\
&\quad \land\ winner[p] = UNKNOWN \\
&\quad \land\ \neg passed[p] \\
&\quad \land\ readyForAction(p) \\
&\quad \land\ bid[p] = NULL \\
&\quad \land\ \exists\, p2 \in Participants \setminus \{p\} : round[p2] = round[p] \\
&\quad \land\ passed' = [passed \text{ EXCEPT } ![p] = \text{TRUE}] \\
&\quad \land\ \text{UNCHANGED } \langle bid, lastBid, round, initialMoney, winner \rangle
\end{aligned}$$

Figure 3.15: TLA+ Definition of the Pass Action in Auction2

3.3.3.2 Concept of a Round

We define a round of an auction as a synchronization frame in which each active (i.e. not passed) participant must take exactly one action. A participant may move on to the next round if and only if:

1. they themselves have not passed and

2. all active participants (including themselves) have taken action in the current round.

This prevents participants from placing multiple consecutive bids without other participants being able to react to them.

We keep track of each player's round by adding the *round* variable to the model. At the beginning, this is initialized to 1 for all participants and will increment when they decide to move to the next round.

3.3.3.3 Constraints on Bidding

In Section 3.3.2.1, we introduced the *lastBid* variable as the last bid each participant has placed. In the round-based we redefine the meaning of this variable to be the bid that each participant has placed in the *previous round*. With this change comes the need for a new variable *bid* to keep track of the bids for each participant in the current round. Participants are only allowed to place one bid per round. Unlike in Section 3.3.2.2 we assert that bids of the current round may not be lower than any bids of the previous round.

However, we don't want to force the leading bidder to increase their bid in a new round. For this, we add the ability to *stand*. This is very similar to bidding except that a participant is allowed to bid the same amount as in the previous round if all other bids were lower. Standing is not forced so the leading bidder may increase their bid even further.

To prevent a participant from placing new bids when they are the only remaining active participant of the auction, we further add the pre-condition that at least one other participant must be in the same round.

3.3.3.4 Determining a Winner

In Section 3.3.2.2 the determination of a winner is somewhat arbitrary, with some participant determining a winner and the others just blindly copying the first. This changes in the round-based model. All participants individually determine the winner based on the current state of the auction. If all players have decided to pass, then noone wins the auction. On the other hand, a participant wins the auction if they are the only remaining active participant. In order to win, they must have the highest bid of all.

With the individual choice of winners, it is no longer obvious that *agreement* still holds in this model. We explain how we verified that the property does hold in Section 5.1.1.2.

3.3.4 Append-Only Log Refinement (Auction3)

In the last refinement of the auction protocol we introduce the append-only log data structure to keep track of participants' actions during the auction. We simulate a decentralized environment by presenting frontiers as a way to reflect which messages of other participants each participant currently replicates. To achieve progress, participants must propagate their states to their competitors. Using append-only logs prepares us for the implementation of the protocol using Git in Chapter 4.

3.3.4.1 Mapping of Actions from Round-Based Model

This version of the auction procedure still maintains the concept of rounds and the restrictions they bring. However, we step away from the notion that all participants immediately know what actions their competitors have taken. For this, we introduce two new variables:

1. The **msgs** variable is a mapping from participants to the sequence of all messages the participant has published. These messages can either be natural numbers indicating their bids, *PASS* to signify their withdrawal, or *CHANGE* to show that they have moved on to the next round.

2. **frontiers** is a mapping from participants to another mapping from participants to the set of natural numbers. It indicates the latest message replicated from other participants. To give an example, $msgs[p2][frontiers[p][p2]]$ refers to the last message by $p2$ that p has replicated, where $p, p2 \in Participants$.

Each time a participant takes an action, they append a new message to their log and increment the frontier value for themselves.

3.3.4.2 Further Bidding Restrictions

In Section 3.3.3.3 we constrained the bids of a participant such they must be higher than the last bid of all other participants. We restrict this even further in the log-based protocol by forcing bids of a participant to be higher than all bids of the other participants they have replicated, including bids of the current round. The definition of the last bid by $p2$ that p has replicated can be seen in Figure 3.16. Similarly, participants are no longer allowed to stand if they had the highest bid in the last round, but already know that they have been outbid in the current round by some other participant.

This additional constraint is not enforcable in peer-to-peer systems because a participant may always pretend to not have replicated the other messages. However, purposefully placing a lower bid than the leading one will never lead to an advantage, because the only way to win is by having the exclusive highest bid. Thus, participants have no incentive to deliberately do so.

```
$knownLastBid(p, p2) \triangleq$
    LET $frontier \triangleq frontiers[p]$
    IN  IF $frontier[p2] = 0$ THEN 0   $p$ knows nothing about $p2$
        ELSE LET $lastKnownMsg \triangleq msgs[p2][frontier[p2]]$
             IN  IF $lastKnownMsg = PASS$
                 THEN IF $frontier[p2] = 1$ THEN 0   $p2$ passed immediately
                      ELSE $msgs[p2][frontier[p2] - 2]$   $PASS$ must be preceeded by $CHANGE$
                 ELSE IF $lastKnownMsg \in Nat$
                      THEN $lastKnownMsg$
                      ELSE $msgs[p2][frontier[p2] - 1]$   $lastKnownMsg = CHANGE$
```

Figure 3.16: TLA+ Definition of the knownLastbid Operator in Auction3

3.3.4.3 Merging Frontiers

For the auction to progress, participants must eventually learn of the actions that the other participants have taken. We do this by adding the *Merge* action, visible in Figure 3.17, to the specification. In this action we simulate *sender* propagating their replicated messages to *receiver* by updating the values of the receivers frontier to be the maximum of their current values and the values received by *sender*.

$$\begin{array}{l}
A3Merge \triangleq \exists\, sender, receiver \in Participants : \\
\quad \land \text{ LET } sFrontier \triangleq frontiers[sender] \\
\qquad\qquad rFrontier \triangleq frontiers[receiver] \\
\qquad\qquad newRFrontier \triangleq [p \in Participants \mapsto max(sFrontier[p], rFrontier[p])] \\
\qquad\; \text{IN } \; frontiers' = [frontiers \text{ EXCEPT } ![receiver] = newRFrontier] \\
\quad \land \text{ UNCHANGED } \langle initialMoney, msgs, bid, lastBid, passed, round, winner \rangle
\end{array}$$

Figure 3.17: TLA+ Definition of the Merge Action in Auction3

With this final refinement, the design of the auction protocol and Monopoly as a whole is complete. In the next chapter, we demonstrate how we turn the specifications into a real application.

4 Implementation

This chapter outlines the implementation of the TLA+ specifications into a real application. We focus on four main aspects: showing how the game state is represented in YAML format such that it remains close to the structure of the TLA+ models, describing how actions are executed, multiplayer synchronization with Git, and explaining how the game is initialized, hereby differentiating between human-playable and simulation versions. The source code can be found at https://github.com/lgloor/monopoly.

4.1 State Representation

We store the whole game state in a YAML file, closely mapping the structure from the TLA+ specifications. The only difference lies in the player attributes. Instead of top-level mappings each player contains its own set of attributes. In Figure 4.1 we show the structure of the whole state file. Players (here arbitrarily chosen *p1, p2*, and *p3*) are identified by unique strings.

In order to read and manipulate the game state in Python, we use the PyYAML library which can convert YAML to Python data types and vice versa. In our case, the YAML file is mapped to a nested Python dictionary, where the top-level keys correspond to major game variables.

To access different parts of the state, we use Python's dictionary operations. For example, to access the bankruptcy status of *p1* we use `state['players']['p1']['bankrupt']`. Ordered collections of objects, like the spaces of the board, are mapped to lists, which are accessed using zero-based indexing. Thus, to retrieve details of the third square on the board, we use `state['board'][2]`.

4.2 Action Execution

The execution of an action is divided into five steps:

1. First, we load the current game state from the YAML file to Python format.
2. Next, we gather all enabled actions by checking their pre-conditions.
3. Third, we select an action to be executed.
4. Then, we apply the effects of the selected action to the Python state.

```yaml
active: int
order: [p1, p2, p3] # turnPlayer = order[active]
phase: {pre-roll, roll, post-roll, etc.}
goojf_cc_owner: player | null
goojf_ch_owner: player | null
bank_money: int
winner: player | null

players:
  p1:
    url: remote url for other participants
    money: int
    bankrupt: boolean
    in_jail: boolean
    jail_time: int
    position: int
    consecutive_doubles: int
  p2_p3: same things

board: list of squares with attributes (type, value, etc.)
community_chest: list of cards with attributes
chance: list of cards with attributes

debt:
  creditor: player | bank
  amount: int
  next_phase: {pre-roll, roll, post-roll, etc.}
  # or null if current player has no debt

auction:
  asset: int (index of square)
  initiator: player
  players:
    p1:
      bid: int
      last_action: {bid, pass, stand, change}
      round: int
      winner: {unknown, none, player}
    p2: same attributes
  # or null if no auction is in progress
```

Figure 4.1: Structure of the Game State in YAML

5. Finally, we write the changed state back to the YAML file.

To collect all enabled actions, we initialize an empty list. For each possible action, we check whether the pre-conditions are met. In case they are, we add a tuple containing a short description of the action and the lambda function responsible for applying the state changes of the corresponding action to the list.

Should there be multiple actions whose pre-conditions are satisfied, we ask the user to choose one of them. In simulation runs, one is chosen uniformly at random. If only one single action is enabled, it is picked automatically.

To execute the selected action we call its lambda function and print the short description to notify the user of which action has been executed. The lambda function returns a more detailed version of the effects that the action has which is used as the message for the Git commit which is created after each action has been executed. This allows all participants to follow the events of the game using Git's log function.

4.3 Action Broadcasting and Synchronization

As described in Section 4.2 we create a separate Git commit for each action that was executed. To enable all participants to observe the progression of the game state, every player must have access to the repositories of the other players. We accomplish this by adding the URL to each participant's repository as a Git remote, named after the corresponding player. This process is further explained in Section 4.4.

To synchronize changes from other repositories, we use a fetch-only model, placing the responsibility of retrieving updates on each individual player. The only exception to this rule is the *origin* remote, which the participants use to share their local replica with the others. In this case, it is required that local changes are pushed to the *origin* remote to ensure that the other players can access the latest updates.

When new updates are received, we add them to the local repository using `git merge`. In our implementation, we ensure that no merge conflicts can occur. This is because concurrency is only present in auctions, where participants can concurrently bid, stand or pass. However, each participant can only take one action per round, only affecting parts of the state file that are not changed by any other participants. All other actions are executed sequentially and can thereby inherently not lead to any issues.

4.3.1 Asynchronous Push

While the obligation of using a hosted repository to prevent access to one's local machine might be useful to users, it comes at the expense of having to push all actions to the origin remote. This operation can be time consuming if done in a synchronous manner. This is especially apparent when a player is able to execute multiple actions consecutively.

To circumvent this issue, we outsource the push operation to a separate thread. This allows sharing the current state of the game with peers while still maintaining minimal overhead in case of consecutive action execution.

The only time we force a synchronous push is upon game termination to prevent the program from exiting before the last relevant changes are accessible to the other players. This guarantees that all participants will eventually be able to conclude that the game has ended, allowing them to exit as well.

4.4 Game Initialization

When initializing a new game, we differentiate between instances playable for real people and ones that are used for simulations. We will later use simulations in Section 5.2 to evaluate the performance of the implementation.

4.4.1 Human-Playable

4.4.1.1 Creating a New Game

For a player to create a new instance of monopoly, we assume that the following conditions are met:

1. All players have created a new, empty Git repository on a hosting platform like Github.
2. All participants are authorized to fetch from the other players' repositories.
3. The initiator of the game knows the requested nicknames and repository URLs for each player, including themselves.
4. The nicknames are all unique, are not equal to the string *origin*, and comply with all other requirements for naming a Git remote.

When these requirements are met, we prompt the initiator to first enter the URL of their remote repository as well as their nickname, and afterwards the same information for each other participant. In the current directory of the user, we clone the remote repository to `./monopoly_<name>` where `<name>` is replaced by the chosen nickname of the initiator. Additionally, we save the nickname in `.git/.name` of the newly created repository such that in can be retrieved in case the program should disgracefully terminate. This allows the user to rejoin the game without having to enter their nickname again. Since we persist the nickname in the `.git` folder, we automatically prevent that the corresponding `.name` file is tracked by git.

All other players are added as Git remotes using the provided nicknames and URLS. This enables us to retrieve updates from them during the course of the game.

Finally, we create and commit the initial state in the repository of the initiator. The play order will always be the same as the order in which the initiator has provided the player information, meaning that the initiator will always have the first turn.

4.4.1.2 Joining a New Game

After a new game instance has been created, the other players are able to join. While not as many as for creating a new one, some conditions are still required to join a game:

1. The joining player has access to the URL of their own, as well as the initiator's repository.
2. Their own repository URL does not differ from the one that the initiator knows of.
3. The joining player will choose the nickname which the iniator has assigned for them.

We prompt the user to enter the URL of their own repository and their nickname, as well as the URL of the initiator's repository. Equivalent to Section 4.4.1.1, we clone the the repository and persist the player's nickname.

To retrieve the game state from the initiator, we first create a temporary *initiator* remote with the initiator's repository URL from which we pull the main branch. After successfully receiving the game state, the *initiator* remote is removed. Finally, we add the remotes to the other players based on the nicknames and corresponding URLs from the state file.

4.4.1.3 Rejoining an Active Game

Should the program unexpectedly exit before the game is terminated, players have the option to rejoin games that are already in progress. In order to do so, they only need to provide the path, under which the repository for this game is saved. Since their previously chosen nickname is already persisted when creating or first joining the game, they can directly continue from where they left off.

4.4.2 Simulation

To run simulations, we create instances of the game with three participants. The corresponding repositories are isolated in directories named `monopoly_<gameID>` with subdirectories like `m<gameID>_p<playerID>`. For example, the repository of player 0 in simulation 2 is stored under `./monopoly_2/m2_p0`.

Using this structure, we add the other partcipants' repositories as remotes by using relative paths. For example, `m2_p0` can refer to `m2_p1`'s repository by adding `../m2_p1` as a remote. This allows us to fully use Git's fetch operation without requiring separate Github repositories for each simulation participant.

5 Evaluation

In this chapter we evaluate the design from Chapter 3 and the implementation introduced in Chapter 4 on their correctness. The primary goal is to show that there exists no reachable state which violates some pre-defined properties.

Additionally we look at the performance of the implementation to evaluate the playability of the game for end-users. For this, we measure the time elapsed from the point of choosing an action to the time of it having been commited to the local Git repository as well as the amount of time saved by using asynchronous pushing to remote repositories.

To analyze the length of a game we count the number of Git commits in simulation repositories. Furthermore we examine the storage consumption of repositories.

We present and discuss all results in their corresponding sections.

5.1 Correctness

This section focuses on the evaluation of correctness of the application design and its corresponding implementation. We define a collection of safety and liveness properties and show that no sequence of actions, neither in the design nor the implementation, can lead to violations of these properties. To achieve this, we apply model-checking using TLC on the design and run simulations on the implementation while checking for violations.

5.1.1 Verifying the Design

5.1.1.1 Model-Checking with TLC

To evaluate the correctness of the application design we use TLC, the model checker for TLA+ that ships directly with the TLA+ Toolbox. For smaller specifications it allows us to expand all possible state sequences while checking both safety and liveness properties using a breadth-first-search approach. For specifications with a large amount of possible states, this thorough checking can take a long time. TLC offers the simulation mode for such cases where it generates random traces up to a specified maximum length. In this mode it will not check liveness properties.

All following results are obtained using version 1.7.4 of the TLA+ Toolbox which is running on a machine with a Ryzen 5 1600 processor with 6 cores clocked at 3.2 GHz and 16 GB of RAM with Linux Mint as its operating system while allocating 8 worker threads and 10.3 GB of RAM to the TLA+ Toolbox.

5.1.1.2 Auction

In Section 3.3.1, we previously defined the four liveness and safety properties of *termination, agreement, validity*, and *integrity* required for an auction protocol to be correct. To verify that these properties hold at all abstraction levels of the auction design we first need to prove that they hold at the most abstract level of `Auction1` and then use an incremental approach to show that `Auction2` implies `Auction1` and lastly to check that `Auction3` implies `Auction2`.

The only customizable constants (i.e. not pure model values) are the *Participants, and MaxAmount* values. We define *Participants* as a symmetry set of model values $\{p1, p2, p3\}$ and $MaxAmount \triangleq 4$ because this enables all bidding possibilities, i.e. one participant could bet everything in the beginning or all participants incrementally outbid each other, while still enforcing that the amount of possible states cannot grow too large.

To ensure that TLC verifies the properties, we add *termination, agreement, validity*, and *integrity* to the `Properties` section under *What to check?* in the TLA+ Toolbox. As visible in Table 5.1, TLC explores the entire state space of `Auction1` after one second. During this, it finds no state or sequence of steps that violate any of the defined properties. The amount of times each action is enabled is displayed in Table 5.2.

Time	Diameter	States Found	Distinct States	Queue Size
00:00:01	7	18,585	7,982	0
00:00:00	0	125	35	35

Table 5.1: State Space Progress of TLC in Auction1

Action	States Found	Distinct States
A1Init	125	125
A1Bid	2,720	645
A1FirstChooseWinner	3,408	3,083
A1OthersChooseWinner	12,332	4,219

Table 5.2: Actions Taken in Auction1

The next step is to show that `Auction2` implements `Auction1`. For this, we add `Auction1` as a local instance in `Auction2` and run TLC while checking `A1FairSpec` as a property. It takes TLC two seconds to explore all possible sequences of steps. Again, it finds no errors meaning that all desired properties hold in `Auction2` as well. The details of the TLC run can be seen in Table 5.3 and Table 5.4

Time	Diameter	States Found	Distinct States	Queue Size
00:00:02	31	74,647	23,289	0
00:00:00	0	125	35	35

Table 5.3: State Space Progress of TLC in Auction2

In the final step, we validate that `Auction3` implies `Auction2`. Similarly to before, we run TLC on `Auction3` while checking the property `A2FairSpec`. This time TLC takes about two minutes but is still able to explore all possible traces. Also here, TLC cannot find behaviors that violate any of the properties. The detailed results are displayed in Table 5.5 and Table 5.6.

We have now shown that *termination, agreement, validity*, and *integrity* hold on all levels

Action	**States Found**	**Distinct States**
A2Init	125	125
A2Stand	4,173	644
A2Bid	19,270	2,194
A2NextRound	8,054	4,239
A2Pass	26,723	6,521
A2ChooseWinner	16,752	9,656

Table 5.4: Actions Taken in Auction2

Time	**Diameter**	**States Found**	**Distinct States**	**Queue Size**
00:01:58	57	8,496,284	836,326	0
00:01:23	27	7,093,650	726,526	30,258
00:00:04	7	224,838	33,901	16,099
00:00:01	0	125	35	35

Table 5.5: State Space Progress of TLC in Auction3

Action	**States Found**	**Distinct States**
A3Init	125	125
A3Stand	57,796	21,638
A3Bid	157,317	32,136
A3NextRound	144,132	116,081
A3Pass	356,891	120,039
A3ChooseWinner	253,089	179,109
A3Merge	7,526,934	367,295

Table 5.6: Actions Taken in Auction3

of abstraction in the auction protocol design. Since `Auction3` is based on append-only-logs, we can implement this specification using Git or any other similar version control system. The design, however, sets no limitations about security constraints meaning that it assumes all participants only append to their local logs, which is something that may have to be regarded in a real-world application.

5.1.1.3 Monopoly

Monopoly has a very large state space due to its inherent randomness. On each dice roll, all distinct sums of the two dice lead to different outcomes in the next state and each card drawn from a community chest or chance deck has different effects. To decrease the state-space, we make the following changes for the monopoly specification:

1. We decrease the total amount of money in the game to 150$. Because of this, we also reduce the following quantities:
   a) Reward for passing Go to 4$
   b) Amount of money per player at the start of the game to 30$
   c) Fine for getting out of jail to 8$
2. For each type of property (i.e. street, railroad, utility) we only add one set consisting of two properties. This allows us to test all the logic while minimizing the amount of squares on the board.

3. With the same reasoning, we add only one of each type of Community-Chest/Chance card to the corresponding deck.

4. We limit the number of Tax, Community-Chest and Chance squares to one per type.

5. Because the number of squares is now much lower, we limit each die to show a maximum value of 2.

6. A player can only win an auction with a bid of 1$, 5$, or 10$ instead of any amount in his money range.

Since for Go, Free Parking, and Go To Jail there exist only one square on the original board we cannot reduce these any further without having some untested logic.

Even with the previous simplifications, during a full model checking run with three players, TLC finds $2.153 \cdot 10^9$ distinct states after 13 hours while still increasing the amount of distinct states found by more than $1.8 \cdot 10^6$ each minute. Due to time reasons, we decided to abort further checking. The last few minutes of this run can be seen in Table 5.7.

Time	**Diameter**	**States Found**	**Distinct States**	**Queue Size**
13:20:32	44	4,211,134,936	2,153,375,757	650,237,606
13:19:32	44	4,207,404,019	2,151,457,320	649,673,252
13:18:32	44	4,203,812,203	2,149,568,139	649,121,609
13:17:32	44	4,200,207,697	2,147,652,229	648,524,957
13:16:32	44	4,196,566,507	2,145,635,399	647,843,795
13:15:32	44	4,192,759,996	2,143,628,465	647,219,782
13:14:32	44	4,189,088,642	2,141,615,842	646,544,839

Table 5.7: Last 7 Minutes of State Space Progress of TLC in Monopoly

In this exhaustive search, we additionally checked using liveness that all possible behaviours eventually terminate (i.e. only one non-bankrupt player remains). However, TLC provides a sequence of states which contains a cycle.[2] This cycle relies on a specific sequence of dice rolls and player actions such as mortgaging and unmortgaging properties. Hence, this cycle alone does not prove that there exist instances of our monopoly model that will not terminate since the cycle can always be escaped by one different dice roll which is bound to happen due to the inherent randomness of the dice. Nonetheless, we stopped checking for termination after the counterexample has been found.

After removing the only liveness property of the specification it is now possible to check the specification using the simulation mode of TLC. In this mode, TLC does not check liveness properties or the amount of distinct states visited, but still checks invariants and keeps track of the total amount of states seen and the sub-actions that were taken. This is not as strong a proof as the full model checking but TLC is not able to find any invariant violations after two more hours of exploring random traces of length up to 250 while each possible action has been taken at least 339 times. The detailed amounts of sub-actions can be seen in Table 5.8

Because there were no violations found by TLC, we can quite confidently say that the specification is correct. While it is unfortunate that we are not able to check the whole state space even after reducing it to our best abilities, it may be possible that we missed ways to

[2] https://github.com/lgloor/bachelor-thesis-tla-specs/blob/main/counterexamples/cycle_proof

Action	**States Found**
EndPreRoll	16,165,633
PlayGoofjCh	25,675
PlayGoofjCc	26,718
PayJailFine	3,125,725
PreRollUnmortgage	7,245,206
PreRollMortgage	5,918,443
PreRollUpgrade	104,101
PreRollDowngrade	84,351
RollAndMove	19,307,352
RollInJail	1,590,074
BuyProperty	861,375
PayStreetRent	168,739
PreventBankruptcyOnStreetRent	339
PayRailRent	25,807
PreventBankruptcyOnRailRent	20,587
TryPayUtilRent	2,196,430
PayTax	357,742
PreventBankruptcyOnTax	496,527
AuctionProperty	18,454,269
LandOnGoToJail	4,097,300
DrawAndExecuteChanceCard	12,003,911
DrawAndExecuteCommunityChestCard	13,616,858
DoNothingOnOwnProperty	1,327,840
DoNothingOnJailSquare	418,158
DoNothingOnGo	1,475,965
DoNothingOnFreeParking	1,592,992
DoNothingOnMortgagedProperty	1,036,925
DoublesCheck	25,185,080
PayOffDebt	183,767
BankruptcyPreventionMortgage	610,636
BankruptcyPreventionDowngrade	10,728
GoBankrupt	907,423
ConcludeFree4AllActions	22,409,000
F4AUnmortgage	8,607,142
F4AMortgage	7,798,143
F4AUpgrade	133,479
F4ADowngrade	107,809
EndTurn	8,717,697
Init	1

Table 5.8: Actions Taken in Simulation Runs of Monopoly after 2 Hours

further limit the possible actions such that a full check would become feasible and confirm or refute our expectation.

Model checking with TLC helped us to find errors in our specification. For instance, when initializing the free-4-all phase, we did not discard the players that have already went bankrupt meaning that bankrupt players were still technically able to take some actions. TLC recognized this as a violation of the invariant `InvNoActionsPossibleIfBankrupt`, allowing us to quickly resolve the error.

5.1.2 Verifying the Implementation

The implementation using Python and Git for the actions and YAML for the state representation closely follows the TLA+ specification to ensure correctness. While the structure of the state differs slightly from the specification (for example, bankruptcy is now represented as a player attribute rather than a top-level mapping), all actions are implemented such that they can only be taken if their enabling conditions are valid and that have the same effects as defined in the TLA+ specification. These differences are purely for improving the readability of the state and do not lead to any semantic changes.

To verify the correctness of the implementation, we again applied simulation testing using three players. For each player, we create a separate Git repository on the same machine to speed up fetch operations. Before each commit, the Python translations of the TLA+ invariants are checked, raising Exceptions in case they are violated. We use Python's `Rand` class, seeded with the hash of the initial commit of a game to create the random traces. This technique also enables us to exactly replay all actions leading up to an invariant violation.

Using this technique, we simulate 200 games until termination. While TLC proved that this is not guaranteed, it requires very specific chains of actions and is thus highly unlikely when executing actions, drawing cards, and throwing dice pseudo-randomly. All simulations do in fact terminate and in no pass could any violation be found.

Since the implementation is strongly following the design and simulations did not find any errors, we are confident that the implementation is also correct. The focus of this thesis lies more on showing correctness on the design rather than the implementation. We thus present no further results regarding the correctness of the implementation. Measurements regarding action execution and game length can be found in Section 5.2.

5.2 Performance

While correctness is the main focus of this thesis, it is still important that the implementation performs well enough for players to enjoy using the program. In this section, we take measurements regarding time and storage usage to evaluate the implementation on its performance.

5.2.1 Local Action Execution

To compute the time taken for executing an action and subsequently committing the changes to the local repository, we measure the time before and after said steps and write the difference to a file. It is most efficient to do this in the simulation environment since this way we are not reliant on a user picking an enabled action manually. The resulting boxplot of the analysis over 6500 action executions can be seen in Fig. 5.1.

The results show that more than 75% of actions can be executed and their results committed within 0.019 seconds. Even for the greatest outlier, this takes only 0.034 seconds. This is good news, because it means that any noticable latency will not stem from the implementation's action execution nor from Git's commit protocol. We did not analyze, from which action each execution time comes from so there might be a correlation between action and execution times that could explain the outliers that we are not aware of.

5.2.2 Network Considerations

Measuring the propagation delay between one player's push operation and another player's reception of the update presents significant challenges. This latency mostly de-

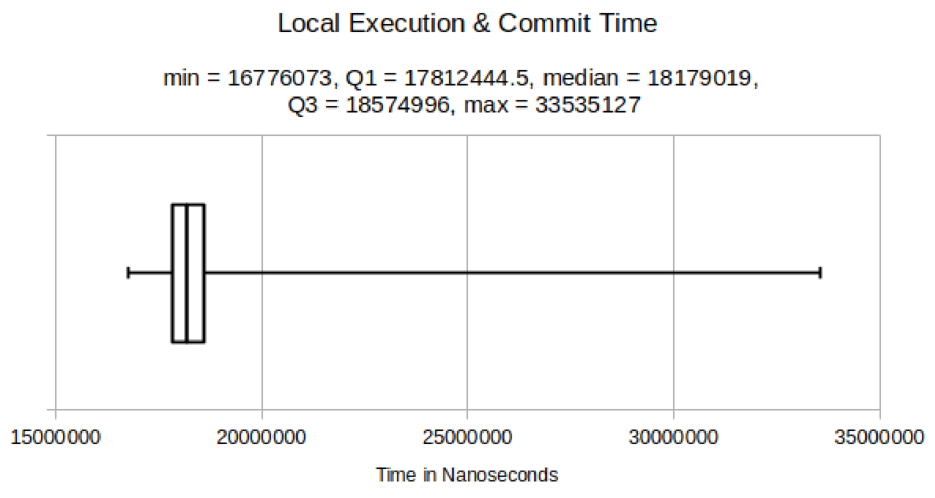

Figure 5.1: Local Action Execution and Commit Times

pends on external factors beyond our control, such as:

- Git push and fetch operations
- Network infrastructure and bandwidth limitations
- Remote repository hosting service performance

It is not in the scope of this project to analyze and/or optimize these uncertainties. Hence, we provide no further elaboration for this part.

5.2.3 Number of Commits per Game

To evaluate the estimated duration per game, we count the number of commits in Git repositories. For this, we use the command `git rev-list --count <branch>`. This gives us an overview on the number of actions taken until a game completes. The analysis over 50 games can be seen in Figure 5.2.

The results suggest that an average game has around 3000 actions that are taken. However, we assume that the number of actions taken is much lower when the game is played with real players instead of running simulations. In simulation runs we observed that in the *pre-roll and free-4-all* phases, players frequently mortgage and unmortgage or upgrade and then downgrade their properties multiple times consecutively. This is unrealistic in a real game since players do not gain any advantage by doing so. For more accurate simulation results, it may make sense to only allow for mortgaging and downgrading properties in the *bankruptcy-prevention* phase.

5.2.4 Disk Usage

To estimate the spacial cost of a monopoly instance we measure the size of git repositories when the game has terminated. This can be done using the command `du -sk <path_of_directory>` which yields the size of a directory in kilobytes. Similarly as in Section 5.2.3, we look at the same 50 simulations and display the results in Fig. 5.3.

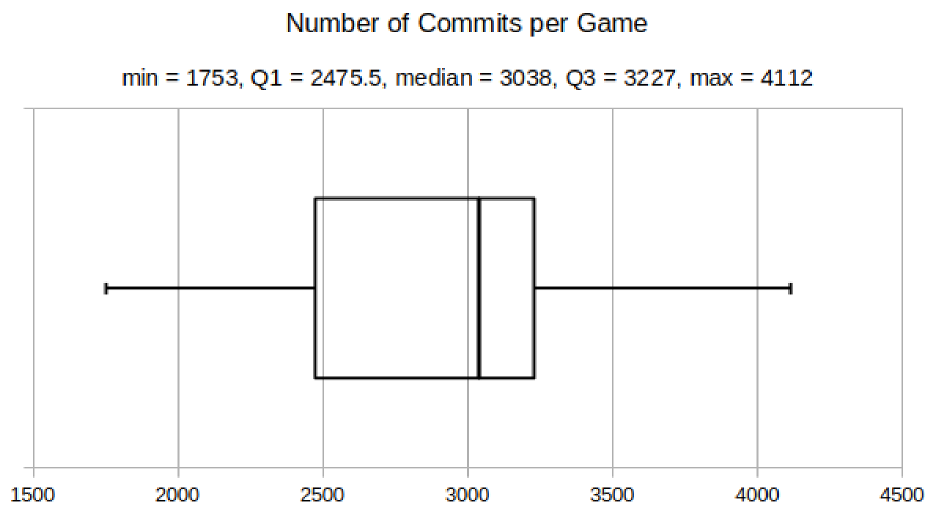

Figure 5.2: Number of Commits per Monopoly Repository

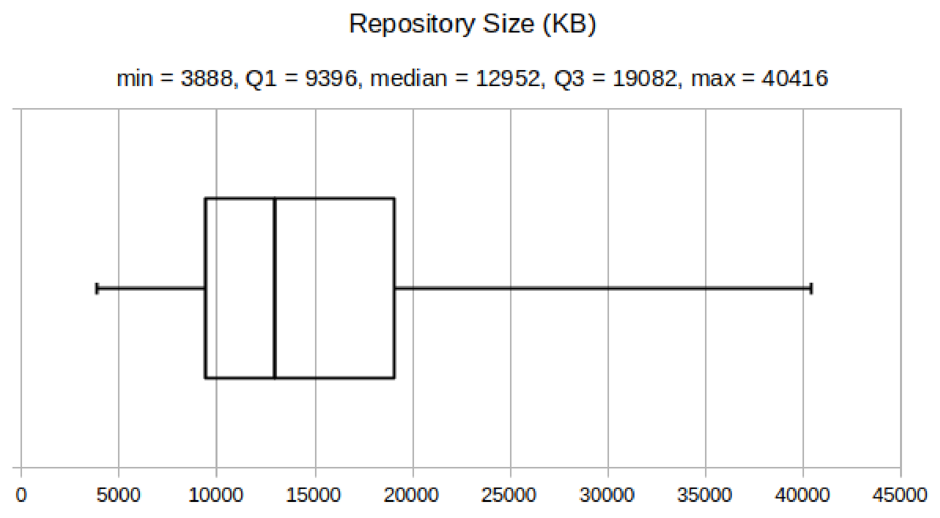

Figure 5.3: Size of Monopoly Git Repositories

Together with the results from Section 5.2.3, we additionally look at the increase in disk usage that each commit causes. We obtain this value as follows:

$$\Delta\text{size} = \frac{\text{Repository size}}{\text{Number of Commits in Repository}}$$

We show the results acquired results in Figure 5.4.

Figure 5.3 shows that while most of the games remain under the size of 20 megabytes it can occur that they take up more than 40 megabytes. However, considering that most machines today have hundreds of gigabytes of storage this is still a rather small footprint.

Seeing that the `state.yml` file in each repository is approximately 5.7 KB large it makes sense that each commit increases the amount of space used by almost the same amount. This is because when a file changes in a commit, Git stores the complete new file in its object database. From a disk usage perspective, this is definitely not optimal and could be improved by splitting the game state over multiple files, each with their own responsibility.

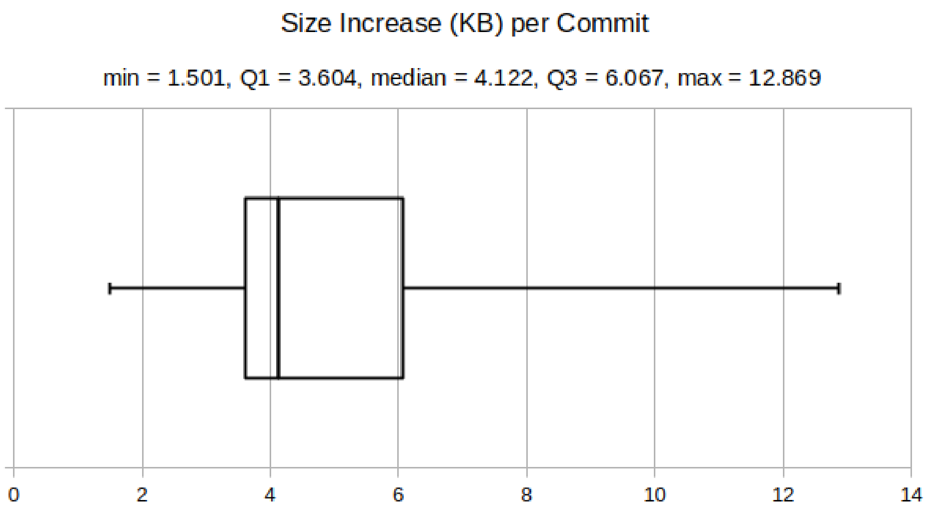

Figure 5.4: Size Increase per Commit in Kilobytes

Since each action only changes a small portion of the state, this would lead to a great size decrease.

6 Related Work

In this chapter, we look at publications by other people which are related to our own project. We discuss what similarities and differences they show in terms of what they achieve and how they are implemented.

6.1 Monopoly Implementations

On Github, we found multiple implementations of Monopoly. Here we want to specifically highlight the versions of Levalleux [11], Tomya [14], and Xu [16], all of which chose a decentralized approach like we did. In contrast to our implementation, however, they all use blockchain technology and smart contracts to propagate game states instead of Git.

6.2 Append-Only Log Applications

Append-only log technology is not only applicable for implementing games. It can also be used in other applications. A-Run [3] has created a chat application called Collab-App in Javascript. To find and connect to peers, the program uses a library called discovery-swarm making use of TCP sockets.

Another example is Tremola [2], a fully decentralized chat application using the tinySSB protocol [15] which is built on append-only logs. In contrast to our implementation, however, SSB uses authenticated append-only logs to prevent participants from impersonating one another [7] which is something we do not enforce.

Also, while unnecessary in the context of a game because the players have an incentive to not fork the game state in order to finish it, append-only logs can be extended to become equivocation-tolerant to prevent partitioning of replicas, which Lavoie [10] proposed with 2p-bft-log. Furthermore, append-only logs may be extended to eventually exclude Byzantine participants, e.g. with a Blocklace as presented by Almeida and Shapiro [4].

6.3 Formal Specifications of Games

To the best of our knowledge, there do not exist any previous specifications of Monopoly. Nonetheless, other games have been formally specified using TLA+ or other specification languages like PDDL (Planning Domain Definition Language).

Ramisetti [13] formalizes a role-playing game where a hero tries to escape from a dungeon containing monsters and other traps in PDDL. The goal is to find a solution for various initial configurations whereas we focus on showing correctness.

Additionally, we found a specification of the classic video game Pong using TLA+ [6]. It describes the movement of the paddles as well as the ball and keeps track of the player's score terminating when a player reaches a certain amount of points.

Similarly to our project, Matter [12] presents both a TLA+ specification as well as an implementation of the board game Catan. The game is also implemented with Git. In contrast to Monopoly, Catan does not use auctions. Moreover, Matter's implementation does not provide a way to interactively play Catan. However, users can run local simulations with a graphical user interface displaying the progress of the game's state.

6.4 Relationship of Auction Protocols to Consensus

Our auction protocol is closely related to the Regular Consensus abstraction described by Cachin et al. [5], because it shares the same properties except that *validity* requires *solvability* of participants. The valid combinations of winning bet and winner upon termination, i.e. $\mathit{Amount} \times (\mathit{Participant} \cup \{\mathit{NONE}\})$, can be seen as the set of proposed values and the auction algorithm guarantees only one of those is selected.

6.5 Other Applications of TLA+

While TLA+ can be used to formally specify games, it is more often used to formalize and prove protocols or other algorithms. We found a Github repository providing many examples using TLA+, several of which are written by Leslie Lamport himself [9]. Examples include a resource allocation algorithm or a consensus protocol called Cesar.

7 Conclusion

In this thesis we addressed the broader challlenge of designing correct distributed systems by presenting a way to formally specify, verify, and implement a decentralized version of the board game Monopoly. For this, we developed a TLA+ specification of Monopoly, capturing its core mechanics including player turns, randomized events, asset ownership, and auctions. For auctions specifically, we introduced a protocol which was refined across three abstraction levels. In each, we proved that the correctness properties of *termination, agreement, validity,* and *integrity* are maintained by using the model checker TLC.

In the implementation we translated these formal specifications into a working application. The game state was represented in YAML and we used Git for synchronizing actions between distributed players. This choice allowed us to create both a human-playable and a simulated version of the game used for verifying correctness of our implementation.

7.1 Future Work

Despite the successes, there remain opportunities for future work. First, the current implementation assumes honest participants and does not prevent any security threats. Later extensions could implement self-certifying Git commits [10] using public-private key pair encryption to prevent a malicious participant from impersonating others.

A mechanic missing from our implementation is trading of assets between participants. To resolve this, a trading protocol could be established using TLA+ to prove correctness properties, one example being the prevention of double spending when the same asset is involved in multiple trade offers.

Finally, the existing implementation runs entirely through command-line interactions and YAML state files. While functional, this can be unintuitive for end users. Extending the program with a graphical user interface would improve usability by making the game state and immediate effects of actions more apparent to users.

7.2 Impact of this Thesis

The impact of this thesis lies in demonstrating that proving correctness of concurrent systems can be made accessible using formal methods. Applications of this approach can range from entertainment contexts such as games to financial environments where correct auction protocols are very important. With decentralized programs becoming more and more prevalent, methods like the one presented here offer a valuable framework for building verifiable systems.

Bibliography

[1] Git, 2008. URL https://git-scm.com/. Accessed: 2025-06-17.

[2] cn-uofbasel/tremola, May 2025. URL https://github.com/cn-uofbasel/tremola. Accessed: 2025-06-17.

[3] A-Run. runionow/Collab-App, May 2019. URL https://github.com/runionow/Collab-App. Accessed: 2025-06-17.

[4] Paulo Sérgio Almeida and Ehud Shapiro. The Blocklace: A Byzantine-repelling and Universal Conflict-free Replicated Data Type, January 2025. URL http://arxiv.org/abs/2402.08068. Accessed: 2025-06-17.

[5] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. Introduction to Reliable and Secure Distributed Programming. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-15259-7 978-3-642-15260-3. URL http://link.springer.com/10.1007/978-3-642-15260-3. Accessed: 2025-06-17.

[6] René Dudfield. illume/pong-tlaplus: TLA+ model for checking a pong game, October 2023. URL https://github.com/illume/pong-tlaplus. Accessed: 2025-06-17.

[7] Anne-Marie Kermarrec, Erick Lavoie, and Christian Tschudin. Gossiping with Append-Only Logs in Secure-Scuttlebutt. In Proceedings of the 1st International Workshop on Distributed Infrastructure for Common Good, DICG'20, pages 19–24, New York, NY, USA, January 2021. Association for Computing Machinery. ISBN 978-1-4503-8197-0. doi: 10.1145/3428662.3428794. URL https://dl.acm.org/doi/10.1145/3428662.3428794. Accessed: 2025-06-17.

[8] Leslie Lamport. Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley, Boston, Mass., 2003. ISBN 978-0-321-14306-8. Accessed: 2025-06-17.

[9] 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, June 2025. URL https://github.com/tlaplus/Examples. Accessed: 2025-06-17.

[10] Erick Lavoie. 2P-BFT-Log: 2-Phase Single-Author Append-Only Log for Adversarial Environments, July 2023. URL http://arxiv.org/abs/2307.08381. Accessed: 2025-06-17.

[11] Ludovic Levalleux. levalleux-ludo/DeFi-Venture, June 2022. URL https://github.com/levalleux-ludo/DeFi-Venture. Accessed: 2025-06-17.

[12] Tim Matter. Modelling and Implementing the "Catan" Boardgame as a Replicated State Machine for Peer-to-Peer Systems. Master's thesis, Universität Basel, Basel, May 2025.

[13] Nikhila Ramisetti. NikhilaRamisetti/Dungeon_game-Role_playing_games, October 2023. URL https://github.com/NikhilaRamisetti/Dungeon_Game-Role_playing_games. Accessed: 2025-06-17.

[14] Somar Tomya. somya-15/de-monopoly, April 2023. URL https://github.com/somya-15/de-monopoly. Accessed: 2025-06-17.

[15] Christian Tschudin. ssbc/tinySSB, June 2025. URL https://github.com/ssbc/tinySSB. Accessed: 2025-06-17.

[16] Michael Xu. SirNeural/monopoly, October 2024. URL https://github.com/SirNeural/monopoly. Accessed: 2025-06-17.

A Monopoly TLA+ Specification

Starts on next page.

```
──────────────────── MODULE Monopoly ────────────────────
EXTENDS Integers, Sequences, FiniteSets

CONSTANTS NULL,             Model value because TLA+ does not have built in support for null
          NumPlayers,       Number of players participating
          StartingMoney,    Amount of money that each player starts with
          TotalMoney,       Total money available
          DiceMax,          Highest number one die can show
          JailFine,         Fine for getting out of jail
          BaseRailRent,     Rent when owning 1 railroad
          PassGoReward      Reward for passing Go

VARIABLES positions,        Board position of each player
          money,            Cash amount of each player
          inJail,           Jail status of each player
          jailTime,         Amount of rounds that each player is already in jail for
          isBankrupt,       Bankruptcy status of each player
          board,            Current status of the board (e.g. owner of properties, current level of properties etc.)
          turnPlayer,       Player taking turn at the moment
          phase,            Current phase of the game (determines the possible actions)
          bankMoney,        Amount of money is left in the bank
          goojfChOwner,     Owner of the "Get out of Jail free" card of the Chance deck
          goojfCcOwner,     Owner of the "Get out of Jail free" card of the Community Chest deck
          doublesCount,     Number of consecutive doubles rolled by the current player
          free4AllOrder,    Order of players in free-4-all phase
          debt,             Debt of a player in bankruptcy-prevention phase
          chanceCards,      Will never change, more transparent than constant
          communityChestCards,  Will never change
          jailIndex         Index of jail square on board, will never change

vars ≜ ⟨positions, money, inJail, jailTime, isBankrupt, board,
  turnPlayer, phase, bankMoney, goojfChOwner, goojfCcOwner,
  doublesCount, free4AllOrder, debt, chanceCards,
  communityChestCards, jailIndex⟩

─────────────────────────────────────────────────────────
abs(n) ≜ IF n < 0 THEN −n ELSE n

RECURSIVE SeqSum(_)
SeqSum(sq) ≜ IF sq = ⟨⟩ THEN 0
                 ELSE Head(sq) + SeqSum(Tail(sq))

incrCirc(initial, amount, maxIdx) ≜ ((initial + amount − 1)%maxIdx) + 1

currentSquare ≜ board[positions[turnPlayer]]

isProperty(field) ≜ field.type ∈ {"street", "rail", "util"}
```

```
$PayBank(player,\ amount) \triangleq$
    $\land\ amount \in 0\ ..\ money[player]$
    $\land\ money' = [money\ \text{EXCEPT}\ ![player] = @ - amount]$
    $\land\ bankMoney' = bankMoney + amount$

$CollectFromBank(player,\ amount) \triangleq$
    $\land\ amount > 0$
    $\land$ IF $bankMoney \geq amount$
        THEN $\land\ money' = [money\ \text{EXCEPT}\ ![player] = @ + amount]$
             $\land\ bankMoney' = bankMoney - amount$
        ELSE $\land\ money' = [money\ \text{EXCEPT}\ ![player] = @ + bankMoney]$
             $\land\ bankMoney' = 0$

$ownedPropertyIdxs(player) \triangleq$
    $\{i \in 1\ ..\ Len(board) :$
      IF $\neg isProperty(board[i])$
       THEN FALSE
       ELSE $board[i].owner = player\}$

$noStreetFromSameSetHasBuildings(strIdx) \triangleq$
    LET $p\_set \triangleq board[strIdx].set$
    IN  $Cardinality($
          $\{i \in 1\ ..\ Len(board) :$
            IF $\neg board[i].type =$ "street"
             THEN FALSE
             ELSE $\land\ board[i].level > 1$
                  $\land\ board[i].set = p\_set$
    $\}) = 0$

$ownsAllOfSet(owner,\ set) \triangleq$
    $\forall\ idx \in \text{DOMAIN}\ board :$
      IF $board[idx].type \neq$ "street"
       THEN TRUE
       ELSE $board[idx].set = set \Rightarrow board[idx].owner = owner$

$permutationSequences(S) \triangleq$
    $\{p \in \text{UNION}\ \{[1\ ..\ Cardinality(S) \rightarrow S]\} :$
      $\forall\ i1,\ i2 \in \text{DOMAIN}\ p :$
        $i1 \neq i2 \Rightarrow p[i1] \neq p[i2]\}$

$initializeFree4All \triangleq$
    $\land\ phase' =$ "free-4-all"
    $\land\ \exists\ order \in permutationSequences(\{p \in 1\ ..\ NumPlayers : \neg isBankrupt[p]\}) :$
         $free4AllOrder' = order$

$terminated \triangleq$
    $Cardinality(\{i \in 1\ ..\ NumPlayers : \neg isBankrupt[i]\}) = 1$
```

```
$EndPreRoll \triangleq$
    $\land \neg terminated$
    $\land phase = $ "pre-roll"
    $\land phase' = $ "roll"
    $\land$ UNCHANGED $\langle positions, money, inJail, isBankrupt, board, turnPlayer,$
        $bankMoney, goojfCcOwner, goojfChOwner, doublesCount, jailTime,$
        $chanceCards, communityChestCards, debt, free4AllOrder, jailIndex\rangle$

$PlayGoojfCh \triangleq$
    $\land \neg terminated$
    $\land phase = $ "pre-roll"
    $\land goojfChOwner = turnPlayer$
    $\land inJail[turnPlayer]$
    $\land inJail' = [inJail$ EXCEPT $![turnPlayer] = $ FALSE]
    $\land jailTime' = [jailTime$ EXCEPT $![turnPlayer] = 0]$
    $\land goojfChOwner' = NULL$
    $\land$ UNCHANGED $\langle bankMoney, board, chanceCards, communityChestCards, debt,$
                    $doublesCount, free4AllOrder, goojfCcOwner, isBankrupt,$
                    $jailIndex, money, phase, positions, turnPlayer\rangle$

$PlayGoojfCc \triangleq$
    $\land \neg terminated$
    $\land phase = $ "pre-roll"
    $\land goojfCcOwner = turnPlayer$
    $\land inJail[turnPlayer]$
    $\land inJail' = [inJail$ EXCEPT $![turnPlayer] = $ FALSE]
    $\land jailTime' = [jailTime$ EXCEPT $![turnPlayer] = 0]$
    $\land goojfCcOwner' = NULL$
    $\land$ UNCHANGED $\langle bankMoney, board, chanceCards, communityChestCards, debt,$
                    $doublesCount, free4AllOrder, goojfChOwner, isBankrupt,$
                    $jailIndex, money, phase, positions, turnPlayer\rangle$

$PayJailFine \triangleq$
    $\land \neg terminated$
    $\land inJail[turnPlayer]$
    $\land money[turnPlayer] \geq JailFine$
    $\land PayBank(turnPlayer, JailFine)$
    $\land inJail' = [inJail$ EXCEPT $![turnPlayer] = $ FALSE]
    $\land jailTime' = [jailTime$ EXCEPT $![turnPlayer] = 0]$
    $\land$ UNCHANGED $\langle board, chanceCards, communityChestCards, debt,$
                    $doublesCount, free4AllOrder, goojfCcOwner, goojfChOwner,$
                    $isBankrupt, jailIndex, phase, positions, turnPlayer\rangle$

$UnmortgageProperty(player) \triangleq \exists idx \in ownedPropertyIdxs(player) :$
```

```
    $\land$ $board[idx].mortgaged$
    $\land$ LET $mortgageValue \triangleq board[idx].value \div 2$
            $unmortgageCost \triangleq mortgageValue + (mortgageValue \div 10)$
      IN  $\land$ $money[player] \geq unmortgageCost$
          $\land$ $board' = [board$ EXCEPT $![idx].mortgaged =$ FALSE$]$
          $\land$ $PayBank(player, unmortgageCost)$

$MortgageProperty(player) \triangleq \exists\, idx \in ownedPropertyIdxs(player) :$
    $\land$ $\neg board[idx].mortgaged$
    $\land$ $board[idx].type =$ "street" $\Rightarrow noStreetFromSameSetHasBuildings(idx)$
    $\land$ LET $mortgageValue \triangleq board[idx].value \div 2$
      IN  $\land$ $board' = [board$ EXCEPT $![idx].mortgaged =$ TRUE$]$
          $\land$ $CollectFromBank(player, mortgageValue)$


$allFromSetAreHigherOrEqualLevel(set, level) \triangleq$
    $\forall\, idx \in$ DOMAIN $board$ :
      IF $board[idx].type \neq$ "street"
       THEN TRUE
       ELSE $board[idx].set = set \Rightarrow board[idx].level \geq level$

$UpgradeStreet(player) \triangleq \exists\, idx \in ownedPropertyIdxs(player) :$
    $\land$ IF $\neg board[idx].type =$ "street"
       THEN FALSE
       ELSE LET $street \triangleq board[idx]$
              IN  $\land$ $\neg street.mortgaged$
                  $\land$ $ownsAllOfSet(player, street.set)$
                  $\land$ $street.level < Len(board[idx].rent)$
                  $\land$ $allFromSetAreHigherOrEqualLevel(street.set, street.level)$
                  $\land$ $money[player] \geq street.houseCost$
                  $\land$ $board' = [board$ EXCEPT $![idx].level = @ + 1]$
                  $\land$ $PayBank(player, street.houseCost)$

$allFromSetAreLowerOrEqualLevel(set, level) \triangleq$
    $\forall\, idx \in$ DOMAIN $board$ :
      IF $board[idx].type \neq$ "street"
       THEN TRUE
       ELSE $board[idx].set = set \Rightarrow board[idx].level \leq level$

$DowngradeStreet(player) \triangleq \exists\, idx \in ownedPropertyIdxs(player) :$
    $\land$ IF $\neg board[idx].type =$ "street"
       THEN FALSE
       ELSE LET $street \triangleq board[idx]$
              IN  $\land$ $street.level > 1$
                  $\land$ $allFromSetAreLowerOrEqualLevel(street.set, street.level)$
                  $\land$ $board' = [board$ EXCEPT $![idx].level = @ - 1]$
                  $\land$ $CollectFromBank(player, street.houseCost \div 2)$
```

```
PreRollUnmortgage ≜
    ∧ ¬terminated
    ∧ phase = "pre-roll"
    ∧ UnmortgageProperty(turnPlayer)
    ∧ UNCHANGED ⟨chanceCards, communityChestCards, debt,
                  doublesCount, free4AllOrder, goojfCcOwner,
                  goojfChOwner, inJail, isBankrupt, jailIndex,
                  jailTime, phase, positions, turnPlayer⟩

PreRollMortgage ≜
    ∧ ¬terminated
    ∧ phase = "pre-roll"
    ∧ MortgageProperty(turnPlayer)
    ∧ UNCHANGED ⟨chanceCards, communityChestCards, debt,
                  doublesCount, free4AllOrder, goojfCcOwner,
                  goojfChOwner, inJail, isBankrupt, jailIndex,
                  jailTime, phase, positions, turnPlayer⟩

PreRollUpgrade ≜
    ∧ ¬terminated
    ∧ phase = "pre-roll"
    ∧ UpgradeStreet(turnPlayer)
    ∧ UNCHANGED ⟨chanceCards, communityChestCards, debt, doublesCount,
                  free4AllOrder, goojfCcOwner, goojfChOwner, inJail,
                  isBankrupt, jailIndex, jailTime, phase, positions, turnPlayer⟩

PreRollDowngrade ≜
    ∧ ¬terminated
    ∧ phase = "pre-roll"
    ∧ DowngradeStreet(turnPlayer)
    ∧ UNCHANGED ⟨chanceCards, communityChestCards, debt, doublesCount,
                  free4AllOrder, goojfCcOwner, goojfChOwner, inJail,
                  isBankrupt, jailIndex, jailTime, phase, positions, turnPlayer⟩

TakePreRollAction ≜
    ∨ EndPreRoll
    ∨ PlayGoojfCh
    ∨ PlayGoojfCc
    ∨ PayJailFine
    ∨ PreRollUnmortgage
    ∨ PreRollMortgage
    ∨ PreRollUpgrade
    ∨ PreRollDowngrade

ChangeOwnerOfProperties(from, to) ≜
    ∧ board′ = [i ∈ DOMAIN board ↦
```

```
                    LET field ≜ board[i]
                    IN   IF isProperty(field)
                           THEN IF field.owner = from
                                  THEN [field EXCEPT !.owner = to]
                                  ELSE field
                           ELSE field]
       ∧ IF goojfCcOwner = from
           THEN goojfCcOwner' = to
           ELSE UNCHANGED ⟨goojfCcOwner⟩
       ∧ IF goojfChOwner = from
           THEN goojfChOwner' = to
           ELSE UNCHANGED ⟨goojfChOwner⟩

CollectIfPassGo ≜
    IF positions'[turnPlayer] < positions[turnPlayer]
     THEN CollectFromBank(turnPlayer, PassGoReward)
     ELSE UNCHANGED ⟨money, bankMoney⟩

MoveAfterRoll(amount) ≜
    ∧ positions' = [positions EXCEPT ![turnPlayer] = incrCirc(@, amount, Len(board))]
    ∧ CollectIfPassGo

GoToJail ≜
    ∧ inJail' = [inJail EXCEPT ![turnPlayer] = TRUE]
    ∧ doublesCount' = 0
    ∧ positions' = [positions EXCEPT ![turnPlayer] = jailIndex]
    ∧ initializeFree4All

RollAndMove ≜
    ∃ d1, d2 ∈ 1 .. DiceMax :
      ∧ ¬terminated
      ∧ phase = "roll"
      ∧ inJail[turnPlayer] = FALSE
      ∧ IF d1 ≠ d2
          THEN ∧ MoveAfterRoll(d1 + d2)
               ∧ doublesCount' = 0
               ∧ phase' = "post-roll"
               ∧ UNCHANGED ⟨inJail, free4AllOrder⟩
          ELSE IF doublesCount = 2  Current throw is 3rd consecutive doubles
                 THEN ∧ GoToJail
                      ∧ UNCHANGED ⟨bankMoney, money⟩
                 ELSE ∧ MoveAfterRoll(d1 + d2)
                      ∧ doublesCount' = doublesCount + 1
                      ∧ phase' = "post-roll"
                      ∧ UNCHANGED ⟨inJail, free4AllOrder⟩
      ∧ UNCHANGED ⟨board, chanceCards, communityChestCards, debt, goojfCcOwner,
                    goojfChOwner, isBankrupt, jailIndex, jailTime, turnPlayer⟩

RollInJail ≜
  ∃ d1, d2 ∈ 1 .. DiceMax :
    ∧ ¬terminated
    ∧ phase = "roll"
    ∧ inJail[turnPlayer] = TRUE
    ∧ IF d1 ≠ d2
      THEN IF jailTime[turnPlayer] = 2  has missed doubles for the 3rd time
           THEN ∧ MoveAfterRoll(d1 + d2)
                ∧ jailTime' = [jailTime EXCEPT ![turnPlayer] = 0]
                ∧ inJail' = [inJail EXCEPT ![turnPlayer] = FALSE]
                ∧ IF money[turnPlayer] ≥ JailFine
                    THEN ∧ PayBank(turnPlayer, JailFine)
                         ∧ phase' = "post-roll"
                         ∧ UNCHANGED ⟨debt⟩
                    ELSE ∧ phase' = "bankruptcy-prevention"
                         ∧ debt' = [creditor ↦ NULL,
                                    amount ↦ JailFine,
                                    nextPhase ↦ "post-roll"]
                ∧ UNCHANGED ⟨free4AllOrder⟩
           ELSE ∧ jailTime' = [jailTime EXCEPT ![turnPlayer] = @ + 1]
                ∧ initializeFree4All
                ∧ UNCHANGED ⟨money, bankMoney, positions, inJail, debt⟩
      ELSE ∧ MoveAfterRoll(d1 + d2)  Player will not get to roll again even if they rolled doubles.
           ∧ jailTime' = [jailTime EXCEPT ![turnPlayer] = 0]
           ∧ inJail' = [inJail EXCEPT ![turnPlayer] = FALSE]
           ∧ phase' = "post-roll"
           ∧ UNCHANGED ⟨free4AllOrder, debt⟩
    ∧ UNCHANGED ⟨board, chanceCards, communityChestCards, doublesCount,
                    goojfCcOwner, goojfChOwner, isBankrupt, jailIndex, turnPlayer⟩

TakeRollAction ≜ ∨ RollAndMove
                 ∨ RollInJail

BuyProperty ≜
  IF ¬isProperty(currentSquare)
   THEN FALSE
   ELSE ∧ ¬terminated
        ∧ phase = "post-roll"
        ∧ currentSquare.owner = NULL
        ∧ money[turnPlayer] ≥ currentSquare.value
        ∧ PayBank(turnPlayer, currentSquare.value)
        ∧ board' = [board EXCEPT ![positions[turnPlayer]].owner = turnPlayer]
        ∧ phase' = "doubles-check"
        ∧ UNCHANGED ⟨inJail, positions, turnPlayer, doublesCount, jailTime,
                                    isBankrupt, goojfCcOwner, goojfChOwner, chanceCards,
                                    communityChestCards, debt, free4AllOrder, jailIndex⟩

PayPlayer(from, to, amount) ≜
    money′ = [money EXCEPT ![from] = @ − amount,
                           ![to] = @ + amount]

getStreetRent(street) ≜
    IF street.level > 1
     THEN street.rent[street.level]
     ELSE IF ownsAllOfSet(street.owner, street.set)
           THEN street.rent[1] ∗ 2
           ELSE street.rent[1]

PayStreetRent ≜
    IF currentSquare.type ≠ "street"
     THEN FALSE
     ELSE ∧ ¬terminated
          ∧ phase = "post-roll"
          ∧ LET rentCost ≜ getStreetRent(currentSquare)
                owner ≜ currentSquare.owner
             IN ∧ owner ∉ {NULL, turnPlayer}
                ∧ ¬currentSquare.mortgaged
                ∧ money[turnPlayer] ≥ rentCost
                ∧ PayPlayer(turnPlayer, owner, rentCost)
                ∧ phase′ = "doubles-check"
          ∧ UNCHANGED ⟨bankMoney, board, chanceCards, communityChestCards,
                       debt, doublesCount, free4AllOrder, goojfCcOwner, goojfChOwner,
                       inJail, isBankrupt, jailIndex, jailTime, positions, turnPlayer⟩

PreventBankruptcyOnStreetRent ≜
    IF currentSquare.type ≠ "street"
     THEN FALSE
     ELSE ∧ ¬terminated
          ∧ phase = "post-roll"
          ∧ LET rentCost ≜ getStreetRent(currentSquare)
                owner ≜ currentSquare.owner
             IN ∧ owner ∉ {NULL, turnPlayer}
                ∧ ¬currentSquare.mortgaged
                ∧ money[turnPlayer] < rentCost
                ∧ debt = NULL
                ∧ debt′ = [creditor ↦ owner,
                           amount ↦ rentCost,
                           nextPhase ↦ "doubles-check"]
                ∧ phase′ = "bankruptcy-prevention"
       ∧ UNCHANGED ⟨bankMoney, board, chanceCards, communityChestCards, doublesCount,
                         free4AllOrder, goojfCcOwner, goojfChOwner, inJail, isBankrupt,
                         jailIndex, jailTime, money, positions, turnPlayer⟩

getRailRent(owner) ≜
    LET ownedRails ≜ Cardinality({
        i ∈ DOMAIN board :
            IF ¬board[i].type = "rail"
             THEN FALSE
             ELSE board[i].owner = owner
        })
    IN  BaseRailRent * 2^(ownedRails−1)

PayRailRent ≜
    IF currentSquare.type ≠ "rail"
     THEN FALSE
     ELSE ∧ ¬terminated
          ∧ phase = "post-roll"
          ∧ LET owner ≜ currentSquare.owner
                rentCost ≜ getRailRent(owner)
             IN   ∧ owner ∉ {NULL, turnPlayer}
                  ∧ ¬currentSquare.mortgaged
                  ∧ money[turnPlayer] ≥ rentCost
                  ∧ PayPlayer(turnPlayer, owner, rentCost)
                  ∧ phase′ = "doubles-check"
          ∧ UNCHANGED ⟨bankMoney, board, chanceCards, communityChestCards, debt,
                       doublesCount, free4AllOrder, goojfCcOwner, goojfChOwner, inJail,
                       isBankrupt, jailIndex, jailTime, positions, turnPlayer⟩

PreventBankruptcyOnRailRent ≜
    IF currentSquare.type ≠ "rail"
     THEN FALSE
     ELSE ∧ ¬terminated
          ∧ phase = "post-roll"
          ∧ LET owner ≜ currentSquare.owner
                rentCost ≜ getRailRent(owner)
             IN   ∧ owner ∉ {NULL, turnPlayer}
                  ∧ ¬currentSquare.mortgaged
                  ∧ money[turnPlayer] < rentCost
                  ∧ debt = NULL
                  ∧ debt′ = [creditor ↦ owner,
                             amount ↦ rentCost,
                             nextPhase ↦ "doubles-check"]
                  ∧ phase′ = "bankruptcy-prevention"
       ∧ UNCHANGED ⟨bankMoney, board, chanceCards, communityChestCards, doublesCount,
                    free4AllOrder, goojfCcOwner, goojfChOwner, inJail, isBankrupt, jailIndex,
                    jailTime, money, positions, turnPlayer⟩
```

```
$ownsBothUtilities(owner) \triangleq$
    LET $ownedUtils \triangleq Cardinality(\{$
          $i \in$ DOMAIN $board$ :
             IF $board[i].type \neq$ "util"
              THEN FALSE
              ELSE $board[i].owner = owner$
          $\})$
    IN   $ownedUtils = 2$

$TryPayUtilRent \triangleq$
    IF $currentSquare.type \neq$ "util"
     THEN FALSE
     ELSE $\exists\, d1, d2 \in 1 \,..\, DiceMax$ :
              $\land \neg terminated$
              $\land phase =$ "post-roll"
              $\land$ LET $owner \triangleq currentSquare.owner$
                      $multiplier \triangleq$ IF $ownsBothUtilities(owner)$ THEN 10 ELSE 4
                      $rentCost \triangleq (d1 + d2) * multiplier$
                 IN  $\land owner \notin \{NULL, turnPlayer\}$
                     $\land$ IF $money[turnPlayer] \geq rentCost$
                         THEN $\land PayPlayer(turnPlayer, owner, rentCost)$
                              $\land phase' =$ "doubles-check"
                              $\land$ UNCHANGED $\langle debt \rangle$
                         ELSE $\land debt = NULL$
                              $\land debt' = [creditor \mapsto owner,$
                                            $amount \mapsto rentCost,$
                                            $nextPhase \mapsto$ "doubles-check"$]$
                              $\land phase' =$ "bankruptcy-prevention"
                              $\land$ UNCHANGED $\langle money \rangle$
              $\land$ UNCHANGED $\langle bankMoney, board, chanceCards, communityChestCards, doublesCount,$
                            $free4AllOrder, goojfCcOwner, goojfChOwner, inJail, isBankrupt,$
                            $jailIndex, jailTime, positions, turnPlayer \rangle$

$PayTax \triangleq$
    IF $currentSquare.type \neq$ "tax"
     THEN FALSE
     ELSE $\land \neg terminated$
          $\land phase =$ "post-roll"
          $\land money[turnPlayer] \geq currentSquare.value$
          $\land PayBank(turnPlayer, currentSquare.value)$
          $\land phase' =$ "doubles-check"
          $\land$ UNCHANGED $\langle board, chanceCards, communityChestCards, debt,$
                        $doublesCount, free4AllOrder, goojfCcOwner, goojfChOwner,$
                        $inJail, isBankrupt, jailIndex, jailTime, positions, turnPlayer \rangle$

$PreventBankruptcyOnTax \triangleq$
```

```
    IF $currentSquare.type \neq$ "tax"
     THEN FALSE
     ELSE  $\land \neg terminated$
           $\land phase =$ "post-roll"
           $\land money[turnPlayer] < currentSquare.value$
           $\land debt' = [creditor \mapsto NULL,$
                        $amount \mapsto currentSquare.value,$
                        $nextPhase \mapsto$ "doubles-check"]
           $\land phase' =$ "bankruptcy-prevention"
           $\land$ UNCHANGED $\langle bankMoney, board, chanceCards, communityChestCards,$
                         $doublesCount, free4AllOrder, goojfCcOwner, goojfChOwner,$
                         $inJail, isBankrupt, jailIndex, jailTime, money, positions, turnPlayer\rangle$

$AuctionProperty \triangleq$
    IF $\neg isProperty(currentSquare)$
     THEN FALSE
     ELSE  $\land \neg terminated$
           $\land phase =$ "post-roll"
           $\land currentSquare.owner = NULL$
           $\land \lor \exists winner \in 1 \,..\, NumPlayers :$
                 $\exists bid \in \{1, 5, 10\} :$  should theoretically be 1 .. $money[winner]$ but this makes state space explode
                   $\land money[winner] \geq bid$  would be unnecessary with 1 .. $money[winner]$
                   $\land \neg isBankrupt[winner]$
                   $\land PayBank(winner, bid)$
                   $\land board' = [board$ EXCEPT $![positions[turnPlayer]].owner = winner]$
              $\lor$ UNCHANGED $\langle board, bankMoney, money\rangle$
           $\land phase' =$ "doubles-check"
           $\land$ UNCHANGED $\langle chanceCards, communityChestCards, debt, doublesCount,$
                         $free4AllOrder, goojfCcOwner, goojfChOwner, inJail, isBankrupt,$
                         $jailIndex, jailTime, positions, turnPlayer\rangle$

$LandOnGoToJail \triangleq$
    $\land \neg terminated$
    $\land currentSquare.type =$ "go-to-jail"
    $\land GoToJail$
    $\land$ UNCHANGED $\langle bankMoney, board, chanceCards, communityChestCards,$
                  $debt, goojfCcOwner, goojfChOwner, isBankrupt, jailIndex,$
                  $jailTime, money, turnPlayer\rangle$

$AdvanceTo(destinationIdx) \triangleq$
    $\land positions' = [positions$ EXCEPT $![turnPlayer] = destinationIdx]$
    $\land CollectIfPassGo$

$ExecuteCard(card) \triangleq$
    LET $type \triangleq card.type$
```

```
IN  CASE $type = \text{"collect"} \rightarrow$ $\land\ CollectFromBank(turnPlayer,\ card.amount)$
                                $\land\ phase' = \text{"doubles-check"}$
                                $\land$ UNCHANGED $\langle debt,\ positions,\ goojfCcOwner,\ goojfChOwner,$
                                                $doublesCount,\ inJail,\ free4AllOrder\rangle$
     $\Box$ $type = \text{"pay"} \rightarrow$ IF $money[turnPlayer] \geq card.amount$
                                THEN $\land\ PayBank(turnPlayer,\ card.amount)$
                                     $\land\ phase' = \text{"doubles-check"}$
                                     $\land$ UNCHANGED $\langle debt,\ positions,\ goojfCcOwner,\ goojfChOwner,$
                                                     $doublesCount,\ inJail,\ free4AllOrder\rangle$
                                ELSE $\land\ debt' = [creditor \mapsto NULL,$
                                                 $amount \mapsto card.amount,$
                                                 $nextPhase \mapsto \text{"doubles-check"}]$
                                     $\land\ phase' = \text{"bankruptcy-prevention"}$
                                     $\land$ UNCHANGED $\langle money,\ bankMoney,\ goojfCcOwner,$
                                                     $goojfChOwner,\ positions,\ doublesCount,$
                                                     $inJail,\ free4AllOrder\rangle$
     $\Box$ $type = \text{"advance"} \rightarrow$ $\land\ AdvanceTo(card.square)$
                                $\land$ UNCHANGED $\langle debt,\ goojfCcOwner,\ goojfChOwner,\ phase,$
                                                $doublesCount,\ inJail,\ free4AllOrder\rangle$
     $\Box$ $type = \text{"go-to-jail"} \rightarrow$ $\land\ GoToJail$
                                $\land$ UNCHANGED $\langle debt,\ goojfCcOwner,\ goojfChOwner,$
                                                $money,\ bankMoney\rangle$
     $\Box$ $type = \text{"goojf-cc"} \rightarrow$ $\land\ goojfCcOwner' = turnPlayer$
                                $\land\ phase' = \text{"doubles-check"}$
                                $\land$ UNCHANGED $\langle money,\ bankMoney,\ positions,\ goojfChOwner,$
                                                $debt,\ doublesCount,\ inJail,\ free4AllOrder\rangle$
     $\Box$ $type = \text{"goojf-ch"} \rightarrow$ $\land\ goojfChOwner' = turnPlayer$
                                $\land\ phase' = \text{"doubles-check"}$
                                $\land$ UNCHANGED $\langle money,\ bankMoney,\ positions,\ goojfCcOwner,$
                                                $debt,\ doublesCount,\ inJail,\ free4AllOrder\rangle$

$DrawAndExecuteChanceCard \triangleq$
    IF $currentSquare.type \neq \text{"chance"}$
     THEN FALSE
     ELSE $\land\ \neg terminated$
          $\land\ phase = \text{"post-roll"}$
          $\land\ \exists\ cardIdx \in$ IF $goojfChOwner = NULL$
                   THEN $1\ ..\ Len(chanceCards)$
                      ELSE $1\ ..\ (Len(chanceCards) - 1)$ :
               LET $card \triangleq chanceCards[cardIdx]$
               IN  $ExecuteCard(card)$
          $\land$ UNCHANGED $\langle board,\ chanceCards,\ communityChestCards,\ isBankrupt,$
                          $jailIndex,\ jailTime,\ turnPlayer\rangle$

$DrawAndExecuteCommunityChestCard \triangleq$
```

```
    IF $currentSquare.type \neq$ "community-chest"
     THEN FALSE
     ELSE  $\land \neg terminated$
           $\land$ $phase =$ "post-roll"
           $\land \exists cardIdx \in$ IF $goojfCcOwner = NULL$
                      THEN $1 \,..\, Len(communityChestCards)$
                        ELSE $1 \,..\, (Len(communityChestCards) - 1)$ :
                 LET $card \triangleq communityChestCards[cardIdx]$
                 IN   $ExecuteCard(card)$
           $\land$ UNCHANGED $\langle board, chanceCards, communityChestCards, isBankrupt,$
                            $jailIndex, jailTime, turnPlayer\rangle$

$EndPostRoll \triangleq$
    $\land$ $phase =$ "post-roll"
    $\land$ $phase' =$ "doubles-check"

$DoNothingOnOwnProperty \triangleq$
    IF $\neg isProperty(currentSquare)$
     THEN FALSE
     ELSE  $\land \neg terminated$
           $\land currentSquare.owner = turnPlayer$
           $\land EndPostRoll$
           $\land$ UNCHANGED $\langle bankMoney, board, chanceCards, communityChestCards, debt,$
                            $doublesCount, free4AllOrder, goojfCcOwner, goojfChOwner, inJail,$
                            $isBankrupt, jailIndex, jailTime, money, positions, turnPlayer\rangle$

$DoNothingOnJailSquare \triangleq$
    $\land \neg terminated$
    $\land currentSquare.type =$ "jail"
    $\land EndPostRoll$
    $\land$ UNCHANGED $\langle bankMoney, board, chanceCards, communityChestCards, debt,$
                     $doublesCount, free4AllOrder, goojfCcOwner, goojfChOwner, inJail,$
                     $isBankrupt, jailIndex, jailTime, money, positions, turnPlayer\rangle$

$DoNothingOnGo \triangleq$
    $\land \neg terminated$
    $\land currentSquare.type =$ "go"
    $\land EndPostRoll$
    $\land$ UNCHANGED $\langle bankMoney, board, chanceCards, communityChestCards, debt,$
                     $doublesCount, free4AllOrder, goojfCcOwner, goojfChOwner, inJail,$
                     $isBankrupt, jailIndex, jailTime, money, positions, turnPlayer\rangle$

$DoNothingOnFreeParking \triangleq$
    $\land \neg terminated$
    ∧ $currentSquare.type$ = “free-parking”
    ∧ $EndPostRoll$
    ∧ UNCHANGED ⟨$bankMoney$, $board$, $chanceCards$, $communityChestCards$, $debt$,
                        $doublesCount$, $free4AllOrder$, $goojfCcOwner$, $goojfChOwner$, $inJail$,
                        $isBankrupt$, $jailIndex$, $jailTime$, $money$, $positions$, $turnPlayer$⟩

$DoNothingOnMortgagedProperty$ ≜
    IF ¬$isProperty(currentSquare)$
     THEN FALSE
     ELSE  ∧ ¬$terminated$
           ∧ $currentSquare.owner \neq turnPlayer$
           ∧ $currentSquare.mortgaged$
           ∧ $EndPostRoll$
           ∧ UNCHANGED ⟨$bankMoney$, $board$, $chanceCards$, $communityChestCards$, $debt$,
                              $doublesCount$, $free4AllOrder$, $goojfCcOwner$, $goojfChOwner$, $inJail$,
                              $isBankrupt$, $jailIndex$, $jailTime$, $money$, $positions$, $turnPlayer$⟩

$TakePostRollAction$ ≜
    ∨ $DoNothingOnOwnProperty$
    ∨ $DoNothingOnJailSquare$
    ∨ $DoNothingOnGo$
    ∨ $DoNothingOnFreeParking$
    ∨ $DoNothingOnMortgagedProperty$
    ∨ $PayStreetRent$
    ∨ $PayRailRent$
    ∨ $PreventBankruptcyOnStreetRent$
    ∨ $PreventBankruptcyOnRailRent$
    ∨ $TryPayUtilRent$
    ∨ $BuyProperty$
    ∨ $AuctionProperty$
    ∨ $PayTax$
    ∨ $PreventBankruptcyOnTax$
    ∨ $LandOnGoToJail$
    ∨ $DrawAndExecuteChanceCard$
    ∨ $DrawAndExecuteCommunityChestCard$

$DoublesCheck$ ≜
    ∧ ¬$terminated$
    ∧ $phase$ = “doubles-check”
    ∧ IF $doublesCount > 0$
        THEN  ∧ $phase'$ = “pre-roll”
              ∧ UNCHANGED ⟨$free4AllOrder$⟩
        ELSE  $initializeFree4All$
    ∧ UNCHANGED ⟨$bankMoney$, $board$, $chanceCards$, $communityChestCards$, $debt$,
                    $doublesCount$, $goojfCcOwner$, $goojfChOwner$, $inJail$, $isBankrupt$,
                          jailIndex, jailTime, money, positions, turnPlayer⟩

PayOffDebt ≜
    IF phase ≠ "bankruptcy-prevention"
     THEN FALSE
     ELSE  ∧ ¬terminated
           ∧ money[turnPlayer] ≥ debt.amount
           ∧ IF debt.creditor = NULL
                THEN PayBank(turnPlayer, debt.amount)
                ELSE  ∧ PayPlayer(turnPlayer, debt.creditor, debt.amount)
                      ∧ UNCHANGED ⟨bankMoney⟩
           ∧ phase′ = debt.nextPhase
           ∧ debt′ = NULL
           ∧ UNCHANGED ⟨board, chanceCards, communityChestCards, doublesCount,
                          free4AllOrder, goojfCcOwner, goojfChOwner, inJail, isBankrupt,
                          jailIndex, jailTime, positions, turnPlayer⟩

BankruptcyPreventionMortgage ≜
    IF phase ≠ "bankruptcy-prevention"
     THEN FALSE
     ELSE  ∧ ¬terminated
           ∧ money[turnPlayer] < debt.amount
           ∧ MortgageProperty(turnPlayer)
           ∧ UNCHANGED ⟨chanceCards, communityChestCards, debt,
                          doublesCount, free4AllOrder, goojfCcOwner,
                          goojfChOwner, inJail, isBankrupt, jailIndex,
                          jailTime, phase, positions, turnPlayer⟩

BankruptcyPreventionDowngrade ≜
    IF phase ≠ "bankruptcy-prevention"
     THEN FALSE
     ELSE  ∧ ¬terminated
           ∧ money[turnPlayer] < debt.amount
           ∧ DowngradeStreet(turnPlayer)
           ∧ UNCHANGED ⟨chanceCards, communityChestCards, debt,
                          doublesCount, free4AllOrder, goojfCcOwner,
                          goojfChOwner, inJail, isBankrupt, jailIndex,
                          jailTime, phase, positions, turnPlayer⟩

RECURSIVE BoardAfterBankruptcyToBank(_)
BoardAfterBankruptcyToBank(currentBoard) ≜
    IF currentBoard = ⟨⟩ THEN ⟨⟩
     ELSE LET field ≜ Head(currentBoard)
           IN   IF ¬isProperty(field)
                 THEN ⟨field⟩ ∘ BoardAfterBankruptcyToBank(Tail(currentBoard))
                 ELSE  IF field.owner = turnPlayer
```

```
                                  THEN LET newField ≜ [field EXCEPT !.owner = NULL,
                                                                    !.mortgaged = FALSE]
                                           IN   ⟨newField⟩ ∘ BoardAfterBankruptcyToBank(Tail(currentBoard))
                                  ELSE ⟨field⟩ ∘ BoardAfterBankruptcyToBank(Tail(currentBoard))

TransferAllAssetsToBank ≜
    ∧ PayBank(turnPlayer, money[turnPlayer])
    ∧ IF goojfChOwner = turnPlayer
       THEN goojfChOwner' = NULL
       ELSE UNCHANGED ⟨goojfChOwner⟩
    ∧ IF goojfCcOwner = turnPlayer
       THEN goojfCcOwner' = NULL
       ELSE UNCHANGED ⟨goojfCcOwner⟩
    ∧ board' = BoardAfterBankruptcyToBank(board)

RECURSIVE BoardAfterBankruptcyToPlayer(_, _)
BoardAfterBankruptcyToPlayer(creditor, currentBoard) ≜
    IF currentBoard = ⟨⟩ THEN ⟨⟩
     ELSE LET field ≜ Head(currentBoard)
            IN  IF ¬isProperty(field)
                  THEN ⟨field⟩ ∘ BoardAfterBankruptcyToPlayer(creditor, Tail(currentBoard))
                  ELSE IF field.owner = turnPlayer
                         THEN LET newField ≜ [field EXCEPT !.owner = creditor]
                                IN  ⟨newField⟩ ∘ BoardAfterBankruptcyToPlayer(creditor, Tail(currentBoard))
                         ELSE ⟨field⟩ ∘ BoardAfterBankruptcyToPlayer(creditor, Tail(currentBoard))

TransferAllAssetsToPlayer(creditor) ≜
    ∧ PayPlayer(turnPlayer, creditor, money[turnPlayer])
    ∧ IF goojfChOwner = turnPlayer
       THEN goojfChOwner' = creditor
       ELSE UNCHANGED ⟨goojfChOwner⟩
    ∧ IF goojfCcOwner = turnPlayer
       THEN goojfCcOwner' = creditor
       ELSE UNCHANGED ⟨goojfCcOwner⟩
    ∧ board' = BoardAfterBankruptcyToPlayer(creditor, board)

RECURSIVE GiveTurnToNextLivePlayer(_)
GiveTurnToNextLivePlayer(curr) ≜
    ∧ LET next ≜ incrCirc(curr, 1, NumPlayers)
      IN  IF isBankrupt[next]
                THEN GiveTurnToNextLivePlayer(next)
                ELSE ∧ turnPlayer' = next
                     ∧ doublesCount' = 0
                     ∧ free4AllOrder' = NULL
                     ∧ phase' = "pre-roll"
```

```
$GoBankrupt \triangleq$
    IF $phase \neq$ “bankruptcy-prevention”
     THEN FALSE
     ELSE  $\land \neg terminated$
           $\land money[turnPlayer] < debt.amount$
           $\land \forall idx \in$ DOMAIN $board$ :
                IF $\neg isProperty(board[idx])$ THEN TRUE
                 ELSE $board[idx].owner = turnPlayer \Rightarrow board[idx].mortgaged$
           $\land$ IF $debt.creditor = NULL$
               THEN $TransferAllAssetsToBank$
               ELSE  $\land TransferAllAssetsToPlayer(debt.creditor)$
                     $\land$ UNCHANGED $\langle bankMoney \rangle$
           $\land isBankrupt' = [isBankrupt$ EXCEPT $![turnPlayer] =$ TRUE$]$
           $\land GiveTurnToNextLivePlayer(turnPlayer)$
           $\land$ UNCHANGED $\langle chanceCards, communityChestCards, debt, inJail,$
                          $jailIndex, jailTime, positions \rangle$

$TakeBankruptcyPreventionAction \triangleq$
    $\lor PayOffDebt$
    $\lor BankruptcyPreventionMortgage$
    $\lor BankruptcyPreventionDowngrade$
    $\lor GoBankrupt$

$ConcludeFree4AllActions \triangleq$
    $\land \neg terminated$
    $\land phase =$ “free-4-all”
    $\land$ IF $free4AllOrder = \langle\rangle$
        THEN FALSE
        ELSE  $\land free4AllOrder' = Tail(free4AllOrder)$
              $\land$ UNCHANGED $\langle bankMoney, board, chanceCards, communityChestCards,$
                             $debt, doublesCount, goojfCcOwner, goojfChOwner, inJail,$
                             $isBankrupt, jailIndex, jailTime, money, phase,$
                             $positions, turnPlayer \rangle$

$F4AUnmortgage \triangleq$
    $\land \neg terminated$
    $\land phase =$ “free-4-all”
    $\land$ IF $free4AllOrder = \langle\rangle$
        THEN FALSE
        ELSE LET $player \triangleq Head(free4AllOrder)$
              IN   $UnmortgageProperty(player)$
    $\land$ UNCHANGED $\langle chanceCards, communityChestCards, debt, doublesCount,$
                   $free4AllOrder, goojfCcOwner, goojfChOwner, inJail, isBankrupt,$
                   $jailIndex, jailTime, phase, positions, turnPlayer \rangle$

$F4AMortgage \triangleq$
```

```
    $\land \lnot terminated$
    $\land phase = $ "free-4-all"
    $\land$ IF $free4AllOrder = \langle\rangle$
       THEN FALSE
       ELSE LET $player \triangleq Head(free4AllOrder)$
            IN   $MortgageProperty(player)$
    $\land$ UNCHANGED $\langle chanceCards, communityChestCards, debt, doublesCount,$
                  $free4AllOrder, goojfCcOwner, goojfChOwner, inJail, isBankrupt,$
                  $jailIndex, jailTime, phase, positions, turnPlayer\rangle$

$F4AUpgrade \triangleq$
    $\land \lnot terminated$
    $\land phase = $ "free-4-all"
    $\land$ IF $free4AllOrder = \langle\rangle$
       THEN FALSE
       ELSE LET $player \triangleq Head(free4AllOrder)$
            IN   $UpgradeStreet(player)$
    $\land$ UNCHANGED $\langle chanceCards, communityChestCards, debt, doublesCount,$
                  $free4AllOrder, goojfCcOwner, goojfChOwner, inJail, isBankrupt,$
                  $jailIndex, jailTime, phase, positions, turnPlayer\rangle$

$F4ADowngrade \triangleq$
    $\land \lnot terminated$
    $\land phase = $ "free-4-all"
    $\land$ IF $free4AllOrder = \langle\rangle$
       THEN FALSE
       ELSE LET $player \triangleq Head(free4AllOrder)$
            IN   $DowngradeStreet(player)$
    $\land$ UNCHANGED $\langle chanceCards, communityChestCards, debt, doublesCount,$
                  $free4AllOrder, goojfCcOwner, goojfChOwner, inJail, isBankrupt,$
                  $jailIndex, jailTime, phase, positions, turnPlayer\rangle$

$EndTurn \triangleq$
    $\land \lnot terminated$
    $\land phase = $ "free-4-all"
    $\land free4AllOrder = \langle\rangle$
    $\land GiveTurnToNextLivePlayer(turnPlayer)$
    $\land$ UNCHANGED $\langle bankMoney, board, chanceCards, communityChestCards,$
                  $debt, goojfCcOwner, goojfChOwner, inJail, isBankrupt,$
                  $jailIndex, jailTime, money, positions\rangle$

$TakeFree4AllAction \triangleq$
    $\lor ConcludeFree4AllActions$
    $\lor F4AUnmortgage$
    $\lor F4AMortgage$
    $\lor F4AUpgrade$
```

```
        ∨ F4ADowngrade
        ∨ EndTurn

Init ≜ ∧ turnPlayer = 1
       ∧ positions = [i ∈ 1 .. NumPlayers ↦ 1]
       ∧ money = [i ∈ 1 .. NumPlayers ↦ StartingMoney]
       ∧ inJail = [i ∈ 1 .. NumPlayers ↦ FALSE]
       ∧ jailTime = [i ∈ 1 .. NumPlayers ↦ 0]
       ∧ isBankrupt = [i ∈ 1 .. NumPlayers ↦ FALSE]
       ∧ phase = "pre-roll"
       ∧ bankMoney = TotalMoney − (NumPlayers ∗ StartingMoney)
       ∧ goojfChOwner = NULL
       ∧ goojfCcOwner = NULL
       ∧ doublesCount  = 0
       ∧ board = ⟨
           [type ↦ "go"],
           [type ↦ "street", value ↦ 20, owner ↦ NULL, set ↦ 1, level ↦ 1,
                rent ↦ ⟨1, 4, 10⟩, houseCost ↦ 10, mortgaged ↦ FALSE],
           [type ↦ "street", value ↦ 22, owner ↦ NULL, set ↦ 1, level ↦ 1,
                rent ↦ ⟨2, 8, 20⟩, houseCost ↦ 12, mortgaged ↦ FALSE],
           [type ↦ "community-chest"],
           [type ↦ "chance"],
           [type ↦ "tax", value ↦ 20],
           [type ↦ "rail", value ↦ 25, owner ↦ NULL, mortgaged ↦ FALSE],
           [type ↦ "jail"],
           [type ↦ "rail", value ↦ 25, owner ↦ NULL, mortgaged ↦ FALSE],
           [type ↦ "free-parking"],
           [type ↦ "util", value ↦ 21, owner ↦ NULL, mortgaged ↦ FALSE],
           [type ↦ "util", value ↦ 21, owner ↦ NULL, mortgaged ↦ FALSE],
           [type ↦ "go-to-jail"]⟩
        ∧ jailIndex = 8
        ∧ free4AllOrder = NULL
        ∧ debt = NULL
        ∧ chanceCards = ⟨
            [type ↦ "collect", amount ↦ 10],
            [type ↦ "pay", amount ↦ 30],
            [type ↦ "advance", square ↦ 7],
            [type ↦ "go-to-jail"],
            [type ↦ "goojf-ch"]⟩
        ∧ communityChestCards = ⟨
            [type ↦ "collect", amount ↦ 20],
            [type ↦ "pay", amount ↦ 20],
            [type ↦ "advance", square ↦ 1],
            [type ↦ "go-to-jail"],
            [type ↦ "goojf-cc"]⟩
```

```
$Next \triangleq \lor TakePreRollAction$
           $\lor TakeRollAction$
           $\lor TakePostRollAction$
           $\lor DoublesCheck$
           $\lor TakeBankruptcyPreventionAction$
           $\lor TakeFree4AllAction$

$FairSpec \triangleq$
    $\land Init$
    $\land \Box[Next]_{vars}$
    $\land \mathrm{WF}_{vars}(Next)$

$TypeOK \triangleq \land turnPlayer \in 1 \,..\, NumPlayers$
             $\land \forall p \in 1 \,..\, NumPlayers :$
                 $\land positions[p] \in 1 \,..\, Len(board)$
                 $\land money[p] \in 0 \,..\, TotalMoney$
                 $\land inJail[p] \in \text{BOOLEAN}$
                 $\land isBankrupt[p] \in \text{BOOLEAN}$
                 $\land jailTime[p] \in 0 \,..\, 2$
             $\land phase \in \{$“pre-roll”, “roll”, “post-roll”, “bankruptcy-prevention”,
                          “doubles-check”, “free-4-all”$\}$
             $\land bankMoney \in 0 \,..\, TotalMoney$
             $\land \forall i \in \text{DOMAIN } board :$
                 $\land board[i].type \in \{$“go”, “street”, “community-chest”,
                                          “chance”, “tax”, “rail”, “jail”,
                                          “free-parking”, “util”, “go-to-jail”$\}$
                 $\land isProperty(board[i]) \Rightarrow \land board[i].value \in Nat$
                                              $\land board[i].owner \in 1 \,..\, NumPlayers \cup \{NULL\}$
                                              $\land board[i].mortgaged \in \text{BOOLEAN}$
                 $\land board[i].type =$ “street” $\Rightarrow \land board[i].set \in Nat$
                                                 $\land \forall j \in \text{DOMAIN } board[i].rent : j \in Nat$
                                                 $\land board[i].level \in \text{DOMAIN } board[i].rent$
                                                 $\land board[i].houseCost \in Nat$
                 $\land board[i].type =$ “tax” $\Rightarrow board[i].value \in Nat$
             $\land goojfChOwner \in 1 \,..\, NumPlayers \cup \{NULL\}$
             $\land goojfCcOwner \in 1 \,..\, NumPlayers \cup \{NULL\}$
             $\land doublesCount \in 0 \,..\, 2$
             $\land free4AllOrder \in \{NULL\} \cup Seq(1 \,..\, NumPlayers)$
             $\land free4AllOrder \neq NULL \Rightarrow$
                    $\forall i1, i2 \in \text{DOMAIN } free4AllOrder :$
                        $\lor i1 = i2$
                        $\lor free4AllOrder[i1] \neq free4AllOrder[i2]$
             $\land debt \in \{NULL\} \cup [creditor : \{NULL\} \cup 1 \,..\, NumPlayers,$
                                  $amount : Nat,$
                                  $nextPhase : \{$“pre-roll”, “roll”, “post-roll”,
                                    "doubles-check", "free-4-all"}]
        $\land \forall i \in \text{DOMAIN } chanceCards :$
            $\land chanceCards[i].type \in \{$"collect", "pay", "advance",
                                            "go-to-jail", "goojf-ch"$\}$
            $\land chanceCards[i].type \in \{$"collect", "pay"$\}$
                $\Rightarrow chanceCards[i].amount \in Nat$
            $\land chanceCards[i].type =$ "advance" $\Rightarrow chanceCards[i].square \in \text{DOMAIN } board$
        $\land \forall i \in \text{DOMAIN } communityChestCards :$
            $\land communityChestCards[i].type \in \{$"collect", "pay", "advance",
                                                    "go-to-jail", "goojf-cc"$\}$
            $\land communityChestCards[i].type \in \{$"collect", "pay"$\}$
                $\Rightarrow communityChestCards[i].amount \in Nat$
            $\land communityChestCards[i].type =$ "advance"
                $\Rightarrow chanceCards[i].square \in \text{DOMAIN } board$
        $\land jailIndex \in \text{DOMAIN } board$

$InvNoPossessionsIfBankrupt \triangleq$
    $\forall p \in 1 \mathrel{..} NumPlayers :$
        $isBankrupt[p] \Rightarrow \land Cardinality(ownedPropertyIdxs(p)) = 0$
                               $\land money[p] = 0$
                               $\land goojfChOwner \neq p$
                               $\land goojfCcOwner \neq p$

$InvNoActionsPossibleIfBankrupt \triangleq$
    $\forall p \in 1 \mathrel{..} NumPlayers :$
        $isBankrupt[p] \Rightarrow \land turnPlayer \neq p$
                               $\land free4AllOrder \neq NULL$
                                   $\Rightarrow \forall i \in \text{DOMAIN } free4AllOrder : free4AllOrder[i] \neq p$

$InvNoDebtToBankruptPlayer \triangleq$
    IF $debt = NULL$
     THEN TRUE
     ELSE $debt.creditor \neq NULL \Rightarrow \neg isBankrupt[debt.creditor]$

$InvConservationOfMoney \triangleq$
    $bankMoney + SeqSum(money) = TotalMoney$

$InvStreetLevelRange \triangleq$
    $\forall i1, i2 \in \text{DOMAIN } board :$
        IF $board[i1].type \neq$ "street" $\lor board[i2].type \neq$ "street"
         THEN TRUE
         ELSE $board[i1].set = board[i2].set$
            $\Rightarrow abs(board[i1].level - board[i2].level) \leq 1$
```

B
Auction1 TLA+ Specification

Starts on next page.

```
┌──────────────────────── MODULE Auction1 ────────────────────────┐
EXTENDS Naturals

CONSTANTS UNKNOWN, NONE, Participants, MaxAmount

VARIABLES initialMoney, lastBid, winner

A1vars ≜ ⟨initialMoney, lastBid, winner⟩

A1Bid ≜
  ∧ ∀ p ∈ Participants : winner[p] = UNKNOWN
  ∧ ∃ p ∈ Participants :
      ∃ newBid ∈ (lastBid[p] + 1) .. initialMoney[p] :
        lastBid′ = [lastBid EXCEPT ![p] = newBid]
  ∧ UNCHANGED ⟨winner, initialMoney⟩

A1FirstChooseWinner ≜
  ∧ ∀ p ∈ Participants : winner[p] = UNKNOWN
  ∧  ∨ ∃ p, p2 ∈ Participants :
          ∧ ∀ p3 ∈ Participants \ {p} : lastBid[p] > lastBid[p3]
          ∧ winner′ = [winner EXCEPT ![p2] = p]
     ∨ ∃ p ∈ Participants : winner′ = [winner EXCEPT ![p] = NONE]
  ∧ UNCHANGED ⟨lastBid, initialMoney⟩

A1OthersChooseWinner ≜
  ∧ ∃ p, p2 ∈ Participants :
    ∧ winner[p]  ≠ UNKNOWN
    ∧ winner[p2] = UNKNOWN
    ∧ winner′ = [winner EXCEPT ![p2] = winner[p]]
  ∧ UNCHANGED ⟨lastBid, initialMoney⟩

A1Init ≜
  ∧ initialMoney ∈ [Participants → 0 .. MaxAmount]
  ∧ lastBid = [p ∈ Participants ↦ 0]
  ∧ winner = [p ∈ Participants ↦ UNKNOWN]

A1Next ≜
  ∨ A1Bid
  ∨ A1FirstChooseWinner
  ∨ A1OthersChooseWinner

A1TypeOK ≜
  ∧ initialMoney ∈ [Participants → 0 .. MaxAmount]
  ∧ lastBid ∈ [Participants → Nat]
  ∧ winner ∈ [Participants →
                Participants ∪ {UNKNOWN, NONE}]

terminated ≜ ∀ p ∈ Participants : winner[p] ≠ UNKNOWN
```

$$\begin{aligned}
&\mathit{agreed} \triangleq \forall\, p,\ p2 \in \mathit{Participants} : \\
&\qquad\qquad \lor\ \mathit{winner}[p] = \mathit{UNKNOWN} \\
&\qquad\qquad \lor\ \mathit{winner}[p2] = \mathit{UNKNOWN} \\
&\qquad\qquad \lor\ \mathit{winner}[p] = \mathit{winner}[p2]
\end{aligned}$$

$$\begin{aligned}
&\mathit{solvable} \triangleq \forall\, p \in \mathit{Participants} : \\
&\quad \mathit{lastBid}[p] \in 0 \,..\, (\mathit{initialMoney}[p])
\end{aligned}$$

$$\begin{aligned}
&\mathit{winWithHigherBid} \triangleq \forall\, p,\ p2 \in \mathit{Participants} : \\
&\quad \mathit{winner}[p2] = p \Rightarrow \mathit{lastBid}[p] > \mathit{lastBid}[p2] \lor p = p2
\end{aligned}$$

$$\mathit{valid} \triangleq \mathit{solvable} \land \mathit{winWithHigherBid}$$

$$\begin{aligned}
&\mathit{winnerStaysSame} \triangleq \forall\, p \in \mathit{Participants} : \\
&\quad \mathit{winner}[p] \neq \mathit{UNKNOWN} \Rightarrow \mathit{winner}'[p] = \mathit{winner}[p]
\end{aligned}$$

$\mathit{termination} \triangleq \Diamond\Box\mathit{terminated}$ liveness
$\mathit{agreement} \triangleq \Box\mathit{agreed}$ safety
$\mathit{validity} \triangleq \Box\mathit{valid}$ safety
$\mathit{integrity} \triangleq \Box[\mathit{winnerStaysSame}]_{A1vars}$ safety

$$\begin{aligned}
&A1\mathit{FairSpec} \triangleq \\
&\quad \land\ A1\mathit{Init} \\
&\quad \land\ \Box[A1\mathit{Next}]_{A1vars} \\
&\quad \land\ \mathrm{WF}_{A1vars}(A1\mathit{FirstChooseWinner}) \\
&\quad \land\ \mathrm{WF}_{A1vars}(A1\mathit{OthersChooseWinner})
\end{aligned}$$

C Auction2 TLA+ Specification

Starts on next page.

```
──────────────────────── MODULE Auction2 ────────────────────────
EXTENDS Naturals, FiniteSets

CONSTANTS NULL, Participants, MaxAmount, UNKNOWN, NONE

VARIABLES initialMoney, lastBid, bid, round, passed, winner

A2vars ≜ ⟨initialMoney, lastBid, bid, round, passed, winner⟩

 p is ready to act once all others have passed or caught up.
readyForAction(p) ≜
    ∀ p2 ∈ Participants :
       ∨ passed[p2]
       ∨ round[p] = round[p2]

─────────────────────────────────────────────────────────────────

A2Init ≜
    ∧ lastBid = [p ∈ Participants ↦ 0]
    ∧ bid = [p ∈ Participants ↦ NULL]
    ∧ round = [p ∈ Participants ↦ 1]
    ∧ passed = [p   ∈ Participants ↦ FALSE]
    ∧ initialMoney ∈ [Participants → 0 .. MaxAmount]
    ∧ winner = [p  ∈ Participants ↦ UNKNOWN]

A2Bid ≜ ∃ p ∈ Participants :
    ∧ winner[p] = UNKNOWN
    ∧ ¬passed[p]
    ∧ bid[p] = NULL
    ∧ ∃ p2 ∈ Participants \ {p} : round[p2] = round[p]
    ∧ readyForAction(p)
    ∧ ∃ newBid ∈ (lastBid[p] + 1) .. initialMoney[p] :
        ∧ ∀ p2   ∈ Participants : newBid > lastBid[p2]
        ∧ bid′ = [bid EXCEPT ![p] = newBid]
    ∧ UNCHANGED ⟨lastBid, round, passed, initialMoney, winner⟩

A2Stand ≜ ∃ p ∈ Participants :
    ∧ winner[p] = UNKNOWN
    ∧ ¬passed[p]
    ∧ bid[p] = NULL
    ∧ ∃ p2 ∈ Participants \ {p} : round[p2] = round[p]
    ∧ ∀ p2 ∈ Participants \ {p} : lastBid[p2] < lastBid[p]
    ∧ readyForAction(p)
    ∧ bid′ = [bid EXCEPT ![p] = lastBid[p]]
    ∧ UNCHANGED ⟨lastBid, round, passed, initialMoney, winner⟩

A2Pass ≜ ∃ p ∈ Participants :
    ∧  winner[p] = UNKNOWN
    ∧  $\neg passed[p]$
    ∧  $bid[p] = NULL$
    ∧  $\exists\, p2 \in Participants \setminus \{p\} : round[p2] = round[p]$
    ∧  $readyForAction(p)$
    ∧  $passed' = [passed$ EXCEPT $![p] =$ TRUE$]$
    ∧  UNCHANGED $\langle bid, lastBid, round, initialMoney, winner\rangle$

$A2NextRound \triangleq \exists\, p \in Participants :$
    ∧ $winner[p] = UNKNOWN$
    ∧ $Cardinality(\{p2 \in Participants : \neg passed[p2]\}) \neq 0$
    ∧ $bid[p] \neq NULL$
    ∧ $\forall\, p2 \in Participants :$
         ∨ $passed[p2]$
         ∨ IF $round[p] = round[p2]$
              THEN $bid[p2] \neq NULL$
              ELSE $round[p2] > round[p]$
    ∧ $lastBid' = [lastBid$ EXCEPT $![p] = bid[p]]$
    ∧ $bid' = [bid$ EXCEPT $![p] = NULL]$
    ∧ $round' = [round$ EXCEPT $![p] = @ + 1]$
    ∧ UNCHANGED $\langle passed, initialMoney, winner\rangle$

$A2ChooseWinner \triangleq \exists\, p \in Participants :$
    ∧ $winner[p] = UNKNOWN$
    ∧ ∨ ∧ $\forall\, p2 \in Participants : passed[p2]$
          ∧ $winner' = [winner$ EXCEPT $![p] = NONE]$
      ∨ $\exists\, p2 \in Participants :$
          ∧ $\neg passed[p2]$
          ∧ $\forall\, p3 \in (Participants \setminus \{p2\}) :$
               ∧ $passed[p3]$
               ∧ $lastBid[p2] > lastBid[p3]$
               ∧ $round[p2] > round[p3]$
          ∧ $winner' = [winner$ EXCEPT $![p] = p2]$
    ∧ UNCHANGED $\langle bid, lastBid, passed, round, initialMoney\rangle$

$A2Next \triangleq$
    ∨  $A2Bid$
    ∨  $A2Stand$
    ∨  $A2Pass$
    ∨  $A2NextRound$
    ∨  $A2ChooseWinner$

$A2TypeOK \triangleq$
    ∧ $lastBid \in [Participants \rightarrow Nat]$
    ∧ $bid \in [Participants \rightarrow Nat \cup \{NULL\}]$
    ∧ $round \in [Participants \rightarrow Nat \setminus \{0\}]$
    ∧ $passed \in [Participants \rightarrow$ BOOLEAN $]$
    $\land\ winner \in [Participants \rightarrow \{UNKNOWN,\ NONE\} \cup Participants]$
    $\land\ initialMoney \in [Participants \rightarrow 0 \mathrel{..} MaxAmount]$

$InvIncreasingBids \triangleq \forall\, p \in Participants :$
    $\lor\ bid[p] = NULL$
    $\lor\ \land \forall\, p2 \in Participants \setminus \{p\} :$
            $round[p] = round[p2] \Rightarrow bid[p] > lastBid[p2]$
      $\land\ bid[p] \geq lastBid[p]$

$A2FairSpec \triangleq$
    $\land\ A2Init$
    $\land\ \Box[A2Next]_{A2vars}$
    $\land\ \mathrm{WF}_{A2vars}(A2Pass)$
    $\land\ \mathrm{WF}_{A2vars}(A2NextRound)$
    $\land\ \mathrm{WF}_{A2vars}(A2ChooseWinner)$

INSTANCE $Auction1$

THEOREM $A2FairSpec \Rightarrow A1FairSpec$
```

D
Auction3 TLA+ Specification

Starts on next page.

```
──────────────────────── MODULE Auction3 ────────────────────────
EXTENDS Naturals, FiniteSets, Sequences, TLC

CONSTANTS NULL, Participants, MaxAmount, UNKNOWN, NONE, PASS, CHANGE

VARIABLES msgs, frontiers, initialMoney, lastBid, bid, round, passed, winner

A3vars ≜ ⟨msgs, frontiers, initialMoney, lastBid, bid, round, passed, winner⟩

├──────────────────────────────────────────────────────────────────┤
max(n1, n2) ≜ IF n1 > n2 THEN n1 ELSE n2

knowsHasPassed(p, p2) ≜
    ∧ LET lastMsgIdx  ≜ frontiers[p][p2]
      IN  IF lastMsgIdx = 0 THEN FALSE
             ELSE msgs[p2][lastMsgIdx] = PASS

knownLastBid(p, p2) ≜
    LET frontier ≜ frontiers[p]
    IN  IF frontier[p2] = 0 THEN 0   p knows nothing about p2
           ELSE LET lastKnownMsg ≜ msgs[p2][frontier[p2]]
                IN  IF lastKnownMsg = PASS
                     THEN IF frontier[p2] = 1 THEN 0   p2 passed immediately
                            ELSE msgs[p2][frontier[p2] − 2]   PASS must be preceeded by CHANGE
                     ELSE IF lastKnownMsg ∈ Nat
                            THEN lastKnownMsg
                            ELSE msgs[p2][frontier[p2] − 1]   lastKnownMsg = CHANGE

count(el, seq) ≜
    LET RECURSIVE helper(_)
        helper(s) ≜
            IF s = ⟨⟩ THEN 0
             ELSE IF Head(s) = el
                   THEN 1 + helper(Tail(s))
                   ELSE helper(Tail(s))
    IN  helper(seq)

knownRound(p, p2) ≜
    LET lastMsgIdx ≜ frontiers[p][p2]
    IN  IF lastMsgIdx = 0 THEN 1
           ELSE count(CHANGE, SubSeq(msgs[p2], 1, lastMsgIdx)) + 1

A3readyForAction(p) ≜
    ∀ p2 ∈ Participants :
        ∨ knowsHasPassed(p, p2)
        ∨ round[p] = knownRound(p, p2)

addMsg(p, msg) ≜
```

```
    $\land$ $frontiers' = [frontiers$ EXCEPT $![p][p] = @ + 1]$
    $\land$ $msgs' = [msgs$ EXCEPT $![p] = @ \circ \langle msg \rangle]$
```

```
$A3Init \triangleq$
    $\land$ $msgs = [p \in Participants \mapsto \langle\rangle]$
    $\land$ $frontiers = [p \in Participants \mapsto [pa \in Participants \mapsto 0]]$
    $\land$ $lastBid = [p \in Participants \mapsto 0]$
    $\land$ $bid = [p \in Participants \mapsto NULL]$
    $\land$ $round = [p \in Participants \mapsto 1]$
    $\land$ $passed = [p \in Participants \mapsto$ FALSE$]$
    $\land$ $initialMoney \in [Participants \to 0 \mathrel{..} MaxAmount]$
    $\land$ $winner = [p \in Participants \mapsto UNKNOWN]$

$A3Bid \triangleq \exists p \in Participants :$
    $\land$ $winner[p] = UNKNOWN$
    $\land$ $\neg passed[p]$
    $\land$ $bid[p] = NULL$
    $\land$ $\exists p2 \in Participants \setminus \{p\} : knownRound(p, p2) = round[p]$
    $\land$ $A3readyForAction(p)$
    $\land$ $\exists newBid \in (lastBid[p] + 1) \mathrel{..} initialMoney[p] :$
        $\land$ $\forall p2 \in Participants : newBid > knownLastBid(p, p2)$
        $\land$ $bid' = [bid$ EXCEPT $![p] = newBid]$
        $\land$ $addMsg(p, newBid)$
    $\land$ UNCHANGED $\langle lastBid, round, passed, initialMoney, winner \rangle$

$A3Stand \triangleq \exists p \in Participants :$
    $\land$ $winner[p] = UNKNOWN$
    $\land$ $\neg passed[p]$
    $\land$ $bid[p] = NULL$
    $\land$ $\exists p2 \in Participants \setminus \{p\} : knownRound(p, p2) = round[p]$
    $\land$ $\forall p2 \in Participants \setminus \{p\} : knownLastBid(p, p2) < lastBid[p]$
    $\land$ $A3readyForAction(p)$
    $\land$ $bid' = [bid$ EXCEPT $![p] = lastBid[p]]$
    $\land$ $addMsg(p, lastBid[p])$
    $\land$ UNCHANGED $\langle lastBid, round, passed, initialMoney, winner \rangle$

$A3Pass \triangleq \exists p \in Participants :$
    $\land$ $winner[p] = UNKNOWN$
    $\land$ $\neg passed[p]$
    $\land$ $bid[p] = NULL$
    $\land$ $\exists p2 \in Participants \setminus \{p\} : knownRound(p, p2) = round[p]$
    $\land$ $A3readyForAction(p)$
    $\land$ $passed' = [passed$ EXCEPT $![p] =$ TRUE$]$
    $\land$ $addMsg(p, PASS)$
    $\land$ UNCHANGED $\langle bid, lastBid, round, initialMoney, winner \rangle$

$A3NextRound \triangleq \exists p \in Participants :$
    $\land winner[p] = UNKNOWN$
    $\land Cardinality(\{p2 \in Participants : \neg knowsHasPassed(p, p2)\}) \neq 0$
    $\land bid[p] \neq NULL$
    $\land \forall p2 \in Participants :$
        $\lor knowsHasPassed(p, p2)$
        $\lor$ IF $round[p] = knownRound(p, p2)$
            THEN IF $frontiers[p][p2] = 0$ THEN FALSE
                    ELSE $msgs[p2][frontiers[p][p2]] \in Nat$
            ELSE $knownRound(p, p2) > round[p]$
    $\land lastBid' = [lastBid$ EXCEPT $![p] = bid[p]]$
    $\land bid' = [bid$ EXCEPT $![p] = NULL]$
    $\land round' = [round$ EXCEPT $![p] = @ + 1]$
    $\land addMsg(p, CHANGE)$
    $\land$ UNCHANGED $\langle passed, initialMoney, winner \rangle$

$A3Merge \triangleq \exists sender, receiver \in Participants :$
    $\land$ LET $sFrontier \triangleq frontiers[sender]$
            $rFrontier \triangleq frontiers[receiver]$
            $newRFrontier \triangleq [p \in Participants \mapsto max(sFrontier[p], rFrontier[p])]$
       IN   $frontiers' = [frontiers$ EXCEPT $![receiver] = newRFrontier]$
    $\land$ UNCHANGED $\langle initialMoney, msgs, bid, lastBid, passed, round, winner \rangle$

$A3ChooseWinner \triangleq \exists p \in Participants :$
    $\land winner[p] = UNKNOWN$
    $\land \lor \land \forall p2 \in Participants : knowsHasPassed(p, p2)$
          $\land winner' = [winner$ EXCEPT $![p] = NONE]$
       $\lor \exists p2 \in Participants :$
          $\land \neg knowsHasPassed(p, p2)$
          $\land \forall p3 \in (Participants \setminus \{p2\}) :$
              $\land knowsHasPassed(p, p3)$
              $\land knownLastBid(p, p2) > knownLastBid(p, p3)$
              $\land knownRound(p, p2) > knownRound(p, p3)$
          $\land winner' = [winner$ EXCEPT $![p] = p2]$
    $\land$ UNCHANGED $\langle msgs, frontiers, bid, lastBid, passed, round, initialMoney \rangle$

$A3Next \triangleq$
    $\lor$ $A3Bid$
    $\lor$ $A3Stand$
    $\lor$ $A3Pass$
    $\lor$ $A3NextRound$
    $\lor$ $A3Merge$
    $\lor$ $A3ChooseWinner$
```

$A3FairSpec \triangleq$
$\quad \land A3Init$
$\quad \land \Box[A3Next]_{A3vars}$
$\quad \land \mathrm{WF}_{A3vars}(A3Pass)$
$\quad \land \mathrm{WF}_{A3vars}(A3NextRound)$
$\quad \land \mathrm{WF}_{A3vars}(A3Merge)$
$\quad \land \mathrm{WF}_{A3vars}(A3ChooseWinner)$

INSTANCE $Auction2$

THEOREM $A3FairSpec \Rightarrow A2FairSpec$

University of Basel

Faculty of Science

Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis:	Formal Specification and Decentralized Implementation of Monopoly Using TLA+ and Git
Name Assessor:	Prof. Dr. Christian Tschudin
Name Student:	Luca Gloor
Matriculation No.:	2022-051-189

I attest with my signature that I have written this work independently and without outside help. I also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly.

Additionally, I affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used. This paper may be checked for plagiarism and use of AI-supported technology using the appropriate software. I understand that unethical conduct may lead to a grade of 1 or "fail" or expulsion from the study program.

Place, Date: Brugg, 17.06.2025 Student: L. Gloor

Will this work, or parts of it, be published?

☐ No

☑ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: 17.06.2025

Place, Date: Brugg, 17.06.2025 Student: L. Gloor

Place, Date: ______________ Assessor: ______________

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis.