

## Master Thesis

# **Modelling and Implementing the "Catan" Boardgame as a Replicated State Machine for Peer-to-Peer Systems**

**30th May 2025**



# Acknowledgements

For the opportunity to work on this thesis, I would like to thank Prof. Dr. Christian Tschudin and Dr. Erick Lavoie. I want to thank Dr. Erick Lavoie in particular, his expertise and support helped me tackle this thesis. He provided insights, constructive feedback throughout this project and motivated me to keep working.

In the creation of this thesis, ChatGPT was only used as a search engine, to look up papers, code examples, and definitions. All the text was written by hand and no graphics or images were generated by AI.

# Abstract

Creating peer-to-peer applications is challenging, since all peers replicate the application's whole state and must provide eventual convergence of the state, despite transmission delays and network partitions. This challenge encourages programmers to clearly specify their algorithm's properties. However, peer-to-peer platforms today only provide informal English specifications regarding their distributed behaviour, which requires programmers to have a deep understanding of the specific platform to predict their behaviour for specific applications. New developments over the last decades introduced new tools to help writing formal specifications. One such example is TLA+ and its model checker TLC, which provide an accessible approach to formal modelling.

In this thesis, we showcase this formal approach by modelling and implementing the multi-player board game "Catan". We develop the first TLA+ peer-to-peer specification for "Catan" and present it in detail. We then use the specification as a blueprint to implement a peer-to-peer version of "Catan" in Python. To replicate the state between peers, we use single-writer append-only logs, implemented using Git commits and references (branches).

We discuss the advantages of creating TLA+ specifications and using Git as the underlying tool for state replication. We additionally provide performance metrics of our implementation, that may serve as comparison for implementations of "Catan" on other combinations of programming language and peer-to-peer platforms in the future.

Our experience highlighted that modelling facilitated communication about and debugging of the expected behaviour of the application during the prototyping stage. We also noticed we could better focus on aspects not covered by the model during the implementation. Given this experience, we believe explicitly modelling the behaviour of peer-to-peer applications is quite promising for improving the reliability of peer-to-peer applications and improving the productivity of peer-to-peer developers in the future.

# Contents

<b>Acknowledgements</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>3</b>
2.1 Core Mechanics of Settlers of Catan . . . . .	3
2.1.1 Resources and Building . . . . .	4
2.1.2 Trade . . . . .	6
2.1.3 Game Setup . . . . .	6
2.2 Hexagonal Efficient Coordinate System . . . . .	6
2.3 Append-only logs . . . . .	7
2.4 State machines . . . . .	7
2.5 TLA+ and TLC . . . . .	8
2.5.1 TLA+ Notation . . . . .	8
2.5.2 Example . . . . .	9
<b>3 Model</b>	<b>11</b>
3.1 Physical Game State . . . . .	11
3.1.1 Overview . . . . .	11
3.1.2 Player Buildings and Hands . . . . .	12
3.2 TLA+ Model . . . . .	13
3.2.1 Running Example . . . . .	13
3.2.2 Immutable Definitions . . . . .	13
3.2.3 State Machine Overview . . . . .	17
3.2.4 Game State . . . . .	17
3.2.5 Invariants . . . . .	22
3.2.6 Initial State . . . . .	23
3.2.7 Actions . . . . .	23
<b>4 Implementation</b>	<b>30</b>
4.1 Libraries . . . . .	30
4.2 Game State . . . . .	30
4.2.1 Representation . . . . .	30
4.2.2 Git Branches and Commits Structure . . . . .	30
4.2.3 Modifying and Synchronizing . . . . .	31
4.2.4 Interface Files/Code . . . . .	32

<b>5</b>	<b>Evaluation</b>	<b>33</b>
5.1	Metrics . . . . .	33
5.1.1	Methodology . . . . .	33
5.1.2	Simulation . . . . .	34
5.1.3	Measurements . . . . .	34
5.1.4	Results . . . . .	35
5.2	Correctness . . . . .	38
5.2.1	Methodology . . . . .	38
5.2.2	TLA+ . . . . .	38
5.2.3	Simulations . . . . .	38
<b>6</b>	<b>Related Work</b>	<b>40</b>
6.1	TLA+ . . . . .	40
6.2	Catan implementations . . . . .	40
6.3	Formalized Games . . . . .	41
6.4	Git-based Applications using Append-Only Logs . . . . .	41
<b>7</b>	<b>Conclusion</b>	<b>42</b>
7.1	Insights of the Development Methodology . . . . .	42
7.1.1	Working with TLA+ . . . . .	42
7.1.2	Git . . . . .	43
7.2	Future Work . . . . .	43
7.3	Perspectives . . . . .	44
	<b>Bibliography</b>	<b>44</b>
<b>A</b>	<b>TLA+ Specification and TLC settings</b>	<b>48</b>
<b>B</b>	<b>Game Rules</b>	<b>70</b>

# List of Figures

2.1	Default Map of Catan . . . . .	4
2.2	Game Pieces of one Player . . . . .	4
2.3	Resource generation . . . . .	5
2.4	Building rules . . . . .	5
2.5	Coordinate example . . . . .	6
2.6	Small state machine example . . . . .	7
2.7	TLA+ State Graph . . . . .	10
3.1	Representation of a two-player game . . . . .	11
3.2	Game pieces and cards . . . . .	12
3.3	Cut-out of the default map . . . . .	13
3.4	Map tiles Encoding Example . . . . .	15
3.5	State machine of Catan . . . . .	17
3.6	Settlement points (red) and road points (orange) . . . . .	20
3.7	Settlement Points Example . . . . .	20
3.8	Road points example . . . . .	21
4.1	Folder structure representing complete game state . . . . .	31
4.2	Modified Files . . . . .	32
5.1	Machine specifications . . . . .	33
5.2	Needed Storage Space of "Catan" simulations . . . . .	35
5.3	Histograms of Git commit deltas . . . . .	36
5.4	Elapsed time of two- and four-player simulations . . . . .	37
5.5	Local synchronization impact on elapsed time per commit . . . . .	37

# Chapter 1

## Introduction

Developing applications in peer-to-peer systems poses multiple challenges to developers. Each peer replicates the whole state of the program and all peers must eventually converge the same state, even if there are transmission delays between peers, or the network partitions due to machine failures or outages [2]. To ensure that the state converges, it is useful for programmers to have a clear specification of what the peer-to-peer system provides and what behaviour is expected. Especially when replicas can be out of sync temporarily.

Nowadays, there exist many peer-to-peer platforms, for example Secure-Scuttlebutt (SSB) [26], TinySSB [32], and P2Panda [20]. These platforms have been created to ease the burden on developers and make peer-to-peer application development more approachable. However, they usually have specifications stated in English [25, 29, 21]. This leaves room for interpretation about the platforms behaviours, requirements and assumptions, which are needed to create applications on these platforms. This makes it hard to compare these systems and can lead to surprises during development, since it is hard to see in advance, if a specific peer-to-peer platform is the most suitable for a task or not.

In contrast to these English specifications, the software verification community [14, 15] and Leslie Lamport in particular [8, 7, 31] have been developing tools to help reason about the correctness of concurrent programs. They do this by modelling them as state machines and describing those with discrete mathematics and first-order logic. Usually, to be able to use tools of that category [1, 18, 19], developers need a deep understanding of the tools themselves and mathematics. Developments over the past decades brought forth TLA+ [9] and an accompanying model checker TLC [35], which provide a lower barrier to entry into formal modelling.

In this thesis, we report on the development of a peer-to-peer application, that combines a mathematical specification and a corresponding implementation built on an eventually-consistent peer-to-peer replication platform. As a case study, we model the application logic of the multi-player board game "Catan", and later implement it using the Git version control system as our replication platform, exploiting its in-built primitives and efficient protocols.

Stating the program logic with the help of TLA+ has several advantages. First it improves upon an English specification by clearly specifying properties with mathematics, abstracting complexity and system specific traits. Second, the TLA+ model allows for a quick verification of peer-to-peer platform requirements to check if an application is compatible or not. Third, it is a high-level guideline for the application's implementation. This helps developers concentrate on the chosen platform and language quirks instead of needing to think about the application logic.

We represent the execution of the state machine of "Catan" using replicated single-writer append-only logs [5]. We chose them because they are eventually consistent, en-

sure sequential ordering of messages, and can take any type of payload. The append-only logs are implemented as Git branches. Each player has a dedicated branch and each commit represents an action by a player. The parent relation between commits represents the causality between actions. The state of the application after each action is represented as a directory of files saved in the corresponding commit.

Using Git to implement the application has several advantages: Git has been in constant development for several decades and has been used by millions of developers, which made it reliable and efficient. Git provides many tools for state recovery and inquiries on the commit histories. Most integrated development editors (IDE) provide tools for improved debugging and there are multiple sophisticated libraries for different programming languages that can interact with Git repositories.

Our original contributions with this thesis are three-fold. First, we provide the first formal specification of the "Catan" board game with TLA+. Second, our implementation of "Catan" is the first to use Git as a replication platform, and is the only one that we know of that is peer-to-peer. Third, we provide performance metrics related to storage, size of updates during replication, and the incurred latency of synchronization, to serve as comparisons for future implementations on other peer-to-peer and centralized platforms.

The thesis is organized as follows: We introduce important concepts in Chapter 2, the physical game state of "Catan" and our TLA+ model in Chapter 3, the implementation of that model in Python in Chapter 4, the evaluation of our model and implementation in Chapter 5, work related to our thesis in Chapter 6, and finally our conclusion of this project in Chapter 7. In the appendix are the full TLA+ model A and the rulebook for "Catan" B.

## Chapter 2

# Background

In this chapter, we introduce the basics of "Catan", the hexagonal efficient coordinate system, append-only logs, state machines and how we use TLA+ and TLC to represent them.

### 2.1 Core Mechanics of Settlers of Catan

In this section, we first give an overview of what "Catan" is about, then we introduce its two main mechanics.

The winning condition of the board game is to reach ten victory points. These can be earned by building villages or cities, buying victory point cards, and building the longest road or having the biggest army. To achieve that, a player must spend resource cards, which are earned by villages and cities, or procured via trade.

All of this happens on a board that is made out of hexagonal pieces, arranged as shown in Figure 2.1. Each tile has one of five resource types, or is the desert centrepiece which yields no resource. Edge tiles may have a port for better trading opportunities.

Players take turns acting upon the board, we call the player that currently takes action the active player. The active player first rolls the die, trades, and then may build, as long as resources are available. The active player's turn ends after the building phase. Now, we present in deeper detail how the resource and trading mechanics work.

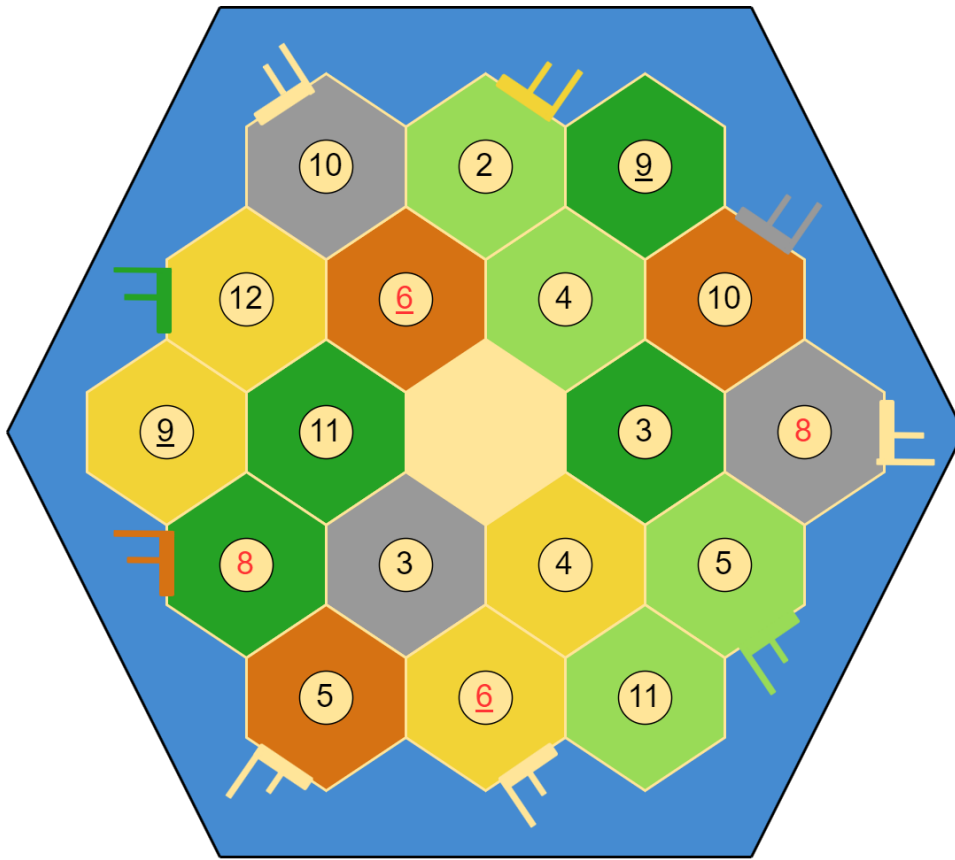


Figure 2.1: Default Map of Catan

### 2.1.1 Resources and Building

As mentioned above, gathering resources and building villages and cities is one of the cornerstones of "Catan", since the winning condition can only be reached by building. A player has a limited amount of buildings, as shown in Figure 2.2: each player has five villages, four cities, and 15 streets. A village earns the player one point, a city two points and streets are needed to expand to build more settlements. These buildings can be built using the five resources of the game, namely, lumber (dark green), brick (brown), wool (light green), grain (yellow) and ore (grey).

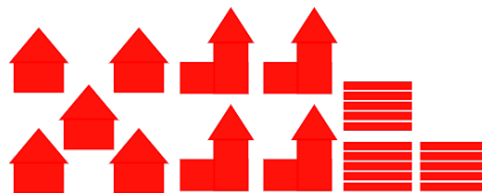


Figure 2.2: Game Pieces of one Player

Each turn, the current player rolls two dice and their sum determines which fields produce resources that turn, namely all that have a matching number to the dice sum. The amount produced for each player depends on the number of villages or cities they have adjacent to a resource tile with the rolled number.

Let us take a look at the example in Figure 2.3 and assume a ten was rolled. There

is one field with a ten and it has a village of the red player adjacent to it. This would earn the red player one ore card. If a six were to be rolled, the red player would gain one brick from the village and two bricks from the city. This implies that the placement of settlements is paramount to optimize resource gain.

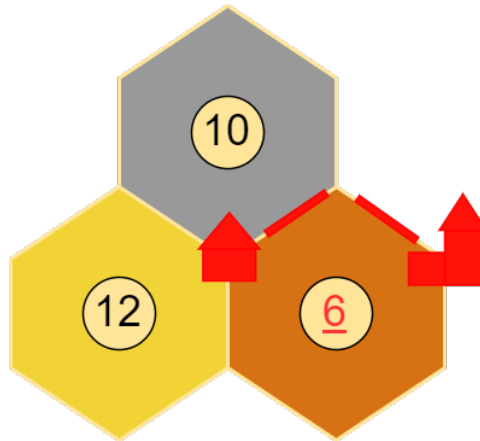


Figure 2.3: Resource generation

There are certain rules to building. First, each village has at least one street attached to it. Second, to further build villages, an edge distance of two between player villages or cities needs to be fulfilled. In Figure 2.4 (left, at the top), we see an example of a viable building spot marked with a circle, because it is connected by roads and is two roads away from an existing settlement. In the same figure (left, on the right), we see an example of a settlement point blocked by the green player, marked by a cross, because the green player has a settlement that is only one edge away. Third, to build cities, a player already needs a village, the city then replaces the village. Fourth, streets can be built up to an enemy player's buildings and in case of gaps in between enemy streets, as long as the two-edge-gap-rule is adhered to, allows a player to place a village in between them as shown in Figure 2.4 on the right. This means that a player can deny another player from building by placing a settlement in the same spot or one edge away.

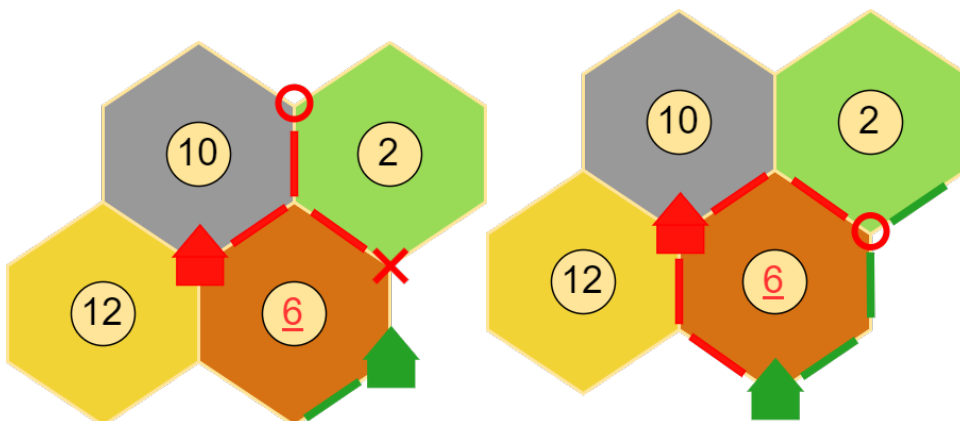


Figure 2.4: Building rules

### 2.1.2 Trade

Another core component of the board game is trading. There are two different ways to trade.

The first is to trade with the bank at a fixed exchange rate. The higher number of resources in the exchange must be of the same type. The default rate to trade with the bank four to one, unless the player has a settlement adjacent to a port. A Port enables the player to trade three to one with the bank or two to one with a specific resource type.

The second is to ask other players to trade. The active player can initiate trade with everyone. The other players however can only initiate trade with the active player. The rate at which the players trade is open but it is prohibited to gift resources.

### 2.1.3 Game Setup

After a party of players is established, the game enters its setup phase. The setup phase has two rounds.

In the first round, each player throws the two dice and the player with the highest sum wins, in case of a draw the players in question roll again. The winner begins by placing a village on any vertex of the board and a street adjacent to it. Then in counter-clockwise fashion, the other players do the same up to the last player.

Now round two begins with the last player. This means the last player can place two villages and two streets at once, balancing the bad luck of the dice roll. The direction turns to clockwise and the other players finish by placing their second village until we are back at the player that initiated the first round. Every player gets resources from all fields adjacent to their second village. From here on out, the game is set up and we enter the turn cycles. A turn consists of a dice roll, trading and building.

## 2.2 Hexagonal Efficient Coordinate System

The Hexagonal Efficient Coordinate System (HECS) [24] is a coordinate system for hexagonal grids. We use it to give our map tiles their coordinates. A HECS coordinate is a tuple  $(a, r, c) \in \{0,1\} \times \mathbb{Z} \times \mathbb{Z}$ . This gives us an array, a row and a column per tuple. Rows that vertically align are in the same array. In Figure 2.5 we can see an example of HECS being used. The top and bottom row are in array 0, the middle row is in array 1. Within the same array, the row and column work as in a standard Cartesian coordinate system.

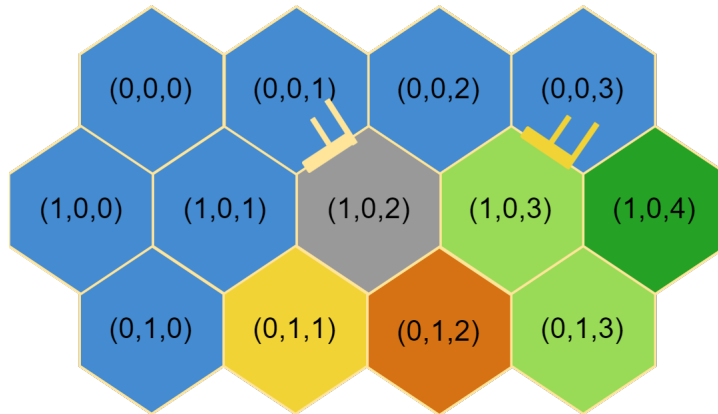


Figure 2.5: Coordinate example

## 2.3 Append-only logs

Append-only logs are a data structure that are often used in decentralized systems due to the following properties [10, 32]: 1) New data can only be appended, i.e. added at the end of the data structure, giving it a chronological ordering of messages added to its structure; 2) Once data is appended, it cannot be deleted or altered in any way, it becomes immutable. These two characteristics make append-only logs very helpful for replication and reconstruction with other peers. Since the data cannot be altered after being added, both peers need to have the same sequence of the same log. If one sequence is longer than the other, the shorter sequence is in need of an update.

Single-writer append-only logs are logs where only one author has access to writing rights for a log. Others can still help replicate that log. An example of this is the communication algorithm of Secure Scuttlebutt (SSB) [5], where single-writer append-only logs are used in a gossip algorithm to facilitate message exchange between members of a social network. To prevent an author from tampering with the messages of another author, all messages are signed with a private key and they include the cryptographic hash and signature of the previous message. This allows authors to check if the message is indeed from the correct user.

## 2.4 State machines

State machines are a common tool in computer science to represent programs and to make it easier to reason about their properties. We give a simple example of a state machine in Figure 2.6. For all the following formalisms we will use the notation of Lamport [8] to explain the example shown in Figure 2.6.

A state machine consists of: a set of states  $\mathcal{S} = \{s_0, s_1, s_2\}$ ; a set of initial states  $\mathcal{I} = \{s_0\}$  that is a subset of the possible states, i.e.  $\mathcal{I} \subseteq \mathcal{S}$ ; a set of actions  $\mathcal{A} = \{start(), end(), repeat()\}$ ; and a next-state relation  $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ . In the example of Figure 2.6, the next-states are:  $\mathcal{N} = \{\langle s_0, start(), s_1 \rangle, \langle s_1, repeat(), s_1 \rangle, \langle s_1, end(), s_2 \rangle\}$ .

We only work with deterministic automata, which means each state  $s$  has at most one element per action  $\alpha$  in the next-state relation  $\mathcal{N}$  where  $s$  is the current state in the relation  $\langle s, \alpha, s' \rangle \in \mathcal{N}$ . Deterministic state machines terminate or deadlock if they reach a state that does not have a next-state relation to another state. If this end-state is unexpected, this is considered a deadlock, otherwise this is considered a normal termination.

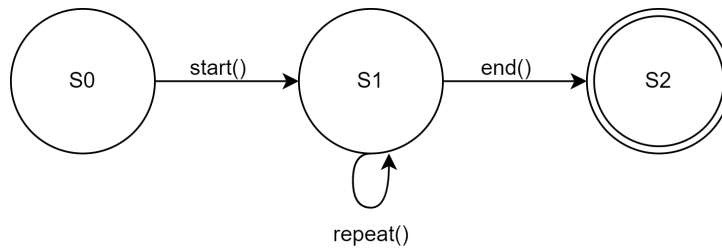


Figure 2.6: Small state machine example

## 2.5 TLA+ and TLC

TLA+ is a language created by Leslie Lamport to formally describe complex concurrent systems [9] as state machines, using discrete mathematics and first-order logic. Each TLA+ specification should have at least two things: 1) An initial state and 2) a disjunction of all possible next steps. The TLA toolbox [6] helps creating these specifications by providing an editor that uses the TLC model checker [35] to quickly verify their behaviour during development. Additionally, TLC can check if safety and liveness properties hold for a model by either traversing the whole state space or doing more depth oriented simulation runs if the state space is too large.

### 2.5.1 TLA+ Notation

We introduce the basic TLA+ notations we will use in the rest of the thesis.

**Notation 1 (Definition):**

TLA+ uses " $\triangleq$ " to introduce definitions, it means "equal by definition".

**Notation 2 (Range):**

An integer range is written as  $s..n$ , where  $s < n$  and  $s..n \triangleq \{i \mid s \leq i \leq n\}$ .

**Notation 3 (Tuples):**

Tuples are written as  $\langle\langle f_1, \dots, f_n \rangle\rangle$  and accessed with an index starting at one. A Tuple  $T \triangleq \langle\langle f_1, \dots, f_n \rangle\rangle$  will return  $f_i$  if we access  $T[i]$ ,  $i \in 1..n$ .

**Notation 4 (Cardinality):**

We retrieve the cardinality of a set  $S$  and a tuple  $T$  using  $\text{Cardinality}(S)$  and  $\text{Len}(T)$  respectively.

**Notation 5 (Records):**

A TLA+ record is similar to a dictionary in Python: We define field names and through those names, we can access the values of the record. To assign a value we use the "maps-to" operator " $\mapsto$ ", e.g.:  $[\text{arr} \mapsto 0, \text{row} \mapsto 1, \text{col} \mapsto 2]$ . To access a field, we use the name of the record, in this case "coords" and write  $\text{coords.arr}$ , the dot denotes an access to a local field of the record.

**Notation 6 (DOMAIN):**

The DOMAIN operator returns all the field keys of a record as a set. Given a record  $R \triangleq [\text{arr} \mapsto 0, \text{row} \mapsto 1, \text{col} \mapsto 2]$ ,  $\text{DOMAIN } R \triangleq \{\text{"arr"}, \text{"row"}, \text{"col"}\}$ .

**Notation 7 (TLA+ Constants):**

TLA+ constants are either declared with the CONSTANTS keyword and assigned with an unspecified value, guaranteed to be unique, or are operators without parameters with a definition (e.g.  $\text{OP} \triangleq \dots$ ). Constants are the immutable parameters of the system.

**Notation 8 (TLA+ Variables):**

TLA+ variables are introduced using the VARIABLES keyword and are the mutable parameters of the system. Variables are untyped but the values they may take is constrained by writing an invariant that asserts the set of possible values.

**Notation 9 (Primed Variables):**

When writing an action (as an operator), we define the new state of a variable with a prime. For example, to assert that the next state of a variable *var* will be equal to 1, we write  $\text{var}' = 1$ .

**Notation 10 (EXCEPT):**

The TLA+ EXCEPT operator is syntactic sugar to more concisely define a new record in terms of an existing record by only specifying which fields and values are different. If we take the record *R* from Notation 6 and want to change the *row* to 4 and *col* to 0 we write:

$$R' = R \text{ EXCEPT } \begin{array}{l} \text{!.row} = 4 \\ \text{!.col} = 0 \end{array}$$

To shorten formulas like incrementation, syntactic sugar using @ is introduced:  $\text{!.row} = R.\text{row} + 1 \triangleq \text{!.row} = @ + 1$ . The @ replaces the access formula to the current field we are modifying.

**2.5.2 Example**

We now show an example of a TLA+ model that represents the state machine of Figure 2.6. We declare  $s_0, s_1, s_2$  as constants, add a variable state with the constraint that it is one of the three states. We create the three actions *start*, *repeat*, and *end*, which have the precondition, respectively, that state is equal to  $s_0, s_1$ , or  $s_2$ . *Start* changes state from  $s_0$  to  $s_1$ , *repeat* leaves everything as is, and *end* changes state from  $s_1$  to  $s_2$ . We define the initial state as  $s_0$ , and the next step function as the disjunction of the three actions. This leads to the code in Definition 1. By using TLC to run the specification with visualization enabled, we get Figure 2.7.

**Definition 1 (Small State Machine):**

```

CONSTANTS  s0,
            s1,
            s2

VARIABLES state
TypeOK  $\triangleq$ 
  state  $\in$  s0, s1, s2
start  $\triangleq$ 
   $\wedge$  state = s0
   $\wedge$  state' = s1
repeat  $\triangleq$ 
   $\wedge$  state = s1
   $\wedge$  UNCHANGED(state)
end  $\triangleq$ 
   $\wedge$  state = s1
   $\wedge$  state' = s2
Init  $\triangleq$  state = s0
Next  $\triangleq$ 
   $\vee$  start
   $\vee$  repeat
   $\vee$  end

```

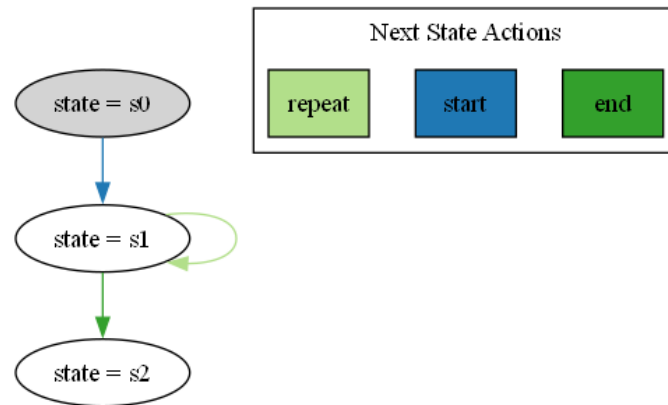


Figure 2.7: TLA+ State Graph

## Chapter 3

# Model

In this chapter, we specify a state machine that reflects the behaviour of a physical game of "Catan". First, we give an informal description based on the state of the physical board game when playing, as described in the rule set (appendix B). Then, we show a mathematical model, that specifies all possible states of the game and valid actions that players may perform based on any given state.

### 3.1 Physical Game State

#### 3.1.1 Overview

Figure 3.1 shows an example of the state of a two-player game of "Catan". The game board is set-up to reflect the standard map of "Catan" previously shown in Figure 2.1. The black game piece in the middle is the bandit, which prevents tiles from producing resources, and players from building around the hexagon it is currently on. Each player has their own unique colour for all game pieces. The pieces that were not used yet are on the side of the board and already used pieces are on the board. Villages and cities are placed on vertexes of the map, roads on edges.

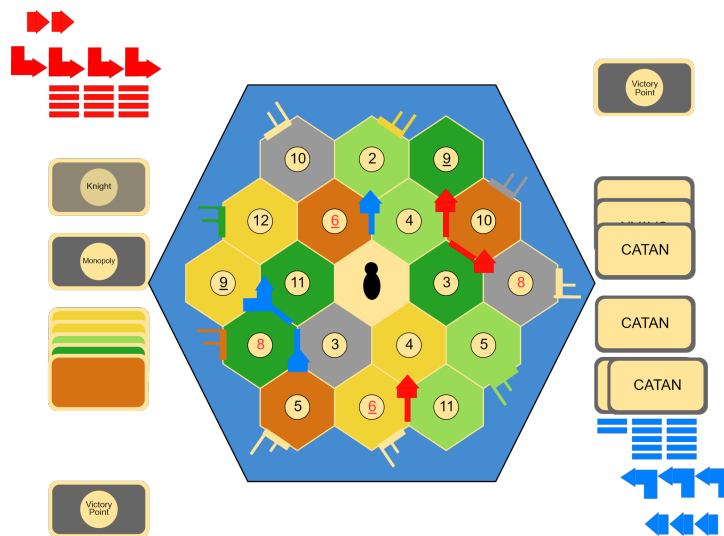


Figure 3.1: Representation of a two-player game

### 3.1.2 Player Buildings and Hands

We now take a closer look at the player pieces on the left side of the board, represented in Figure 3.2. In terms of buildings, there are two villages, four cities and 12 roads left. There are also several categories of cards. We have the resource cards, which are used to buy buildings or development cards. The colour of the card represents the resource type: Green is lumber, brown is brick, light green is wool, yellow is grain, and grey is ore. The second category is the development cards, that we further split into three different categories: 1) Cards that were bought in the current turn and cannot be played yet (light gray). 2) Cards that were bought previously and are available to be played (dark gray), and 3) Cards that have been played, which are either knights or victory points. 1) and 2), as well as resource cards, are hidden from other players, 3) is shown to all players, since they provide victory points. We do not consider privacy in the rest of the thesis but still use the previous categories because they simplify the specification of actions.

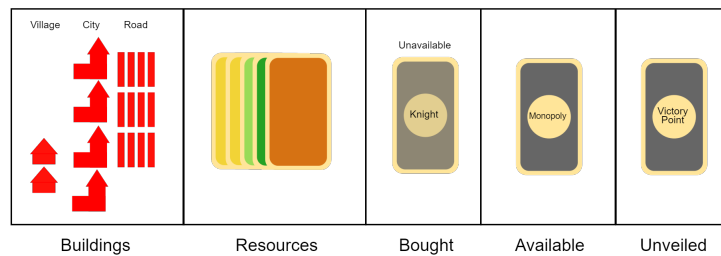


Figure 3.2: Game pieces and cards

## 3.2 TLA+ Model

In this section, we use TLA+ [9] to provide a mathematical model of the game as a state machine. To create a state machine we need an initial state and a set of actions, which can be taken, given some preconditions. To describe the actions and the initial state, we need to encode all the information of the board game, this includes the map layout, the game pieces, the cards, and all actions.

To make our model more approachable, we split our description into immutable and mutable definitions. For example, the map layout never changes during the game, so we consider it immutable, but a player may gain and lose cards, so we consider the player hand mutable. We then describe the actions a player may take in any given state of the game, based on both immutable and mutable definitions. The initial state contains all state that is mutable and describes the board game before any action has been taken.

In the next sections, we introduce the key concepts of our specification by highlighting the most important parts. We omit the other parts, which follow easily from what has been explained. The reader may refer to appendix A for the complete specification.

### 3.2.1 Running Example

Figure 3.3 shows the top left corner of the "Catan" default map layout, previously shown in Figure 2.1 that we will use throughout the chapter to provide examples. Figure 3.3 shows all elements we need to encode if we exclude the actions. This section of the map covers all major elements of the game, including map tiles with different resources and numbers, wharfs, and locations where players may place settlements and roads.

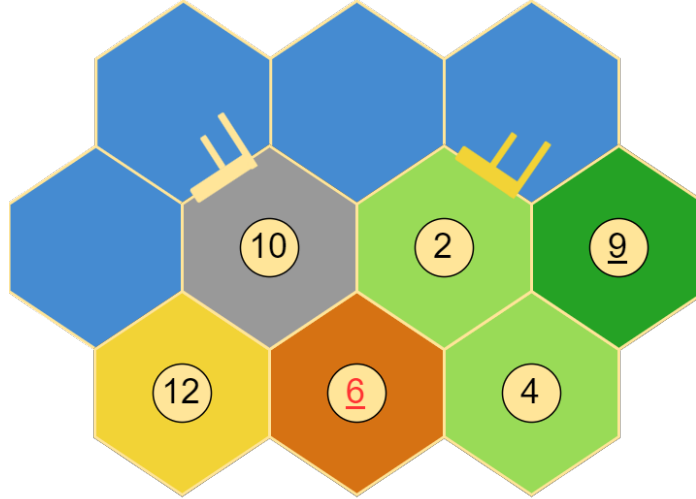


Figure 3.3: Cut-out of the default map

### 3.2.2 Immutable Definitions

We begin our presentation with immutable definitions (lines 4 to 248 in Appendix A). All immutable definitions are either constants, introduced with the `CONSTANT` keyword, or operators taking no argument, introduced with a definition, e.g. `"OP  $\triangleq$  ..."`. All constants defined with the keyword, are assigned a model value when model-checking, i.e. an unspecified value guaranteed to be unique.

## Cards

We begin with all types of cards that are available in the game. There are two main types of cards: Resource and development cards. We group card types in sets to make actions easier to specify later. We also define pools that determine the total amount of each type available during a game.

Resource cards have five types, one for each of the five resources in "Catan": *Lumber*, *Brick*, *Wool*, *Grain*, and *Ore*. We represent them as constants.

### Definition 2 (Resource Card Types):

$RCT \triangleq \{\text{Lumber}, \text{Brick}, \text{Wool}, \text{Grain}, \text{Ore}\}$

Progress card types are defined as follows: *Monopoly*, *Year-of-Plenty*, and *Road-Building*. Similar to resource card types, development card types are represented as constants.

### Definition 3 (Progress Card Types):

$PCT \triangleq \{\text{Monopoly}, \text{YearOfPlenty}, \text{RoadBuilding}\}$

Development cards types consist of all progress cards types, to which we add *Knight*, and *VictoryPoint*.

### Definition 4 (Development Card Types):

$DCT \triangleq PCT \cup \{\text{Knight}, \text{VictoryPoint}\}$

There are 19 resource cards of each type available in a game of "Catan". We use a tuple of counters to reflect the total of each card type available in the game. These counters are later used in Invariant 1, to make sure all cards are persistently in the game.

### Definition 5 (Resource Card Pool):

$RCP \triangleq \langle\langle 19, 19, 19, 19, 19 \rangle\rangle$

The available development cards are two *Monopoly*, two *YearOfPlenty*, two *Road-Building*, 14 *Knight*, and five *VictoryPoint* cards. Similar to resource cards, the pool is represented as an immutable tuple and serves as a reference for Invariant 2.

### Definition 6 (Development Card Pool):

$DCP \triangleq \langle\langle 2, 2, 2, 14, 5 \rangle\rangle$

We now define how many cards of each type can be bought and kept by a player. We give a range from zero to the maximum amount of cards specified in the Development Card Pool (Definition 6). We assume *VictoryPoint* cards are always unveiled immediately upon buying, therefore we do not track them in the bought cards.

### Definition 7 (Bought Cards Type):

$BCT \triangleq [\text{Monopoly}: 0..2, \text{YearOfPlenty}: 0..2, \text{RoadBuilding}: 0..2, \text{Knight}: 0..14]$

Unveiled Cards are defined analogously to Definition 7 but only have the *Knight* and *VictoryPoint* as fields.

### Definition 8 (Unveiled Cards Type):

$UCT \triangleq [\text{Knight}: 0..14, \text{VictoryPoint}: 0..5]$

## The Map

To represent the map we encode the hexagonal tiles. They have six resource types:  $\perp$ , *Lumber*, *Brick*, *Wool*, *Grain*, and *Ore*. The bottom ( $\perp$ ) represents tiles that do not generate resources, like water and desert tiles. Each tile has an integer  $n$  assigned to it from two to 12, excluding seven. This number is used for resource generation (see Section 2.1.1). To reflect the location of a tile, we use the Hexagonal Efficient

Coordinate System (see Section 2.2). The complete information for each tile is stored in a TLA+ record, which provides a convenient syntax to define the sets of records:

**Notation 11 (Set of Records):**

TLA+ allows us to define a set of records, by using the notation:  $[card: S]$ . This returns the set of distinct records, where the value associated to the label "card" is an element of  $S$ . For example, let  $S$  be the Resource Card Type Set (Definition 2). This would result in:  $[card: S] \triangleq \{[card \mapsto Lumber], [card \mapsto Brick], [card \mapsto Wool], [card \mapsto Grain], [card \mapsto Ore]\}$ .

The following defines the set of all valid records representing map tiles for Catan:

**Definition 9 (Map Tiles):**

$$MT \triangleq [$$

$$\text{coords: [arr: } 0..1, \text{ row: Int, col: Int],}$$

$$\text{res: } RCT \cup \{\perp\},$$

$$\text{n: } 2..6 \cup 8..12 \cup \{\perp\}$$

$$]$$

A valid map is a subset of  $MT$  in which each tile has unique coordinates.

**Definition 10 (Map):**

$$M \subseteq MT \wedge (\forall t_1, t_2 \in M \wedge t_1.\text{coords} = t_2.\text{coords} \iff t_1 = t_2)$$

Figure 3.4 shows examples of tiles. The grey tile is represented as follows: It has the coordinates  $\{1, 0, 2\}$ , meaning it is in tile array one, in the zeroth row and first column. Its resource type is Ore and its number is 10. It is represented as the following record:  $[coords \mapsto \{1, 0, 2\}, res \mapsto Ore, n \mapsto 10]$ , which for illustration purposes we simplify as a tuple  $\langle\langle\{1, 0, 1\}, Ore, 10\rangle\rangle$  in the figure. Water tiles are a special case, the water tile to the left of our ore tile is represented with the tuple:  $\langle\langle\{1, 0, 0\}, \perp, \perp\rangle\rangle$ . Both the type as well as the number are  $\perp$ , since it does not produce resources.

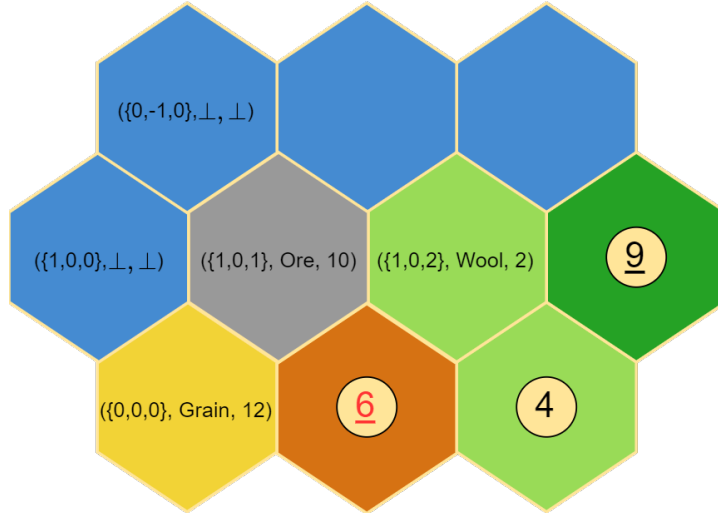


Figure 3.4: Map tiles Encoding Example

To distinguish all the port types on the map, we define *ThreeToOne* as a port usable for all resources, which allows a player to trade three of the same resource type for another resource card. For each resource type we add a wharf, that can trade only the specific resource two to one.

**Definition 11 (Wharf Types):**

$$WT \triangleq \{\text{ThreeToOne}\} \cup \text{RCT}$$

We define  $W$  to be the set of all wharfs displayed on the default map shown in Figure 2.1. Each element has coordinates that are part of the map, respectively the two tiles that are adjacent and part of the coastline.

**Definition 12 (Wharf):**

$$W \triangleq \{[\text{coords} \mapsto \{t_1, t_5\}, \text{wt} \mapsto \text{ThreeToOne}], \dots\}$$

$$\forall w \in W: w.\text{wt} \in WT \wedge \forall t \in w.\text{coords}: t \in M$$

$$\forall w \in W: \exists t_1, t_2 \in w.\text{coords}: \text{isAdjacent}(t_1, t_2) \wedge \text{isCoast}(t_1, t_2)$$
**Board Game Pieces**

We now define helpers for the game pieces that can be placed on the board. There are two settlement types: *Village* and *City*, each represented as a constant.

**Definition 13 (Settlement Types):**

$$ST \triangleq \{\text{Village}, \text{City}\}$$

Throughout the game, each player has access to a pool of 15 roads, five villages and four cities. This pool is used later to verify Invariant 3.

**Definition 14 (Player Building Pool):**

$$BP \triangleq \langle \langle 15, 5, 4 \rangle \rangle$$
**Phases**

At the beginning of "Catan" there is a two-phase setup that takes place. Each phase is represented as a constant.

**Definition 15 (Setup Phases):**

$$SUP \triangleq \{\text{PhaseOne}, \text{PhaseTwo}\}$$

Each turn of the game, post-setup, is decomposed into phases. In addition, we use  $\perp$  to represent that the game is in the setup phase and  $\top$  to represent that a player has won the game. All are represented as constants.

**Definition 16 (Turn Phases):**

$$TP \triangleq \{\perp, \text{DiceRoll}, \text{Trading}, \text{Building}, \top\}$$

### 3.2.3 State Machine Overview

We now present the mutable parts of the specification. This part is specified as a state machine, for which we first provide an overview.

Our state machine is initialized with the state  $s_0$  later given in Definition 20. As shown in Figure 3.5, each player goes through setup phase one and two, later presented in Actions 1, 2. After the setup, each step of the state machine follows the order of the turn phases: *DiceRoll*, *Trading* and *Building*. Since a player may trade multiple times in a turn, we write the state as  $S_{r,i}$ , where  $r$  is the round and  $i$  the states index. With a fixed  $r$  over all states,  $i$  is smaller than  $j$  and  $j$  is smaller than  $k$ . Previous rounds always have lower state indexes than the later rounds and successor rounds always have higher indexes than the previous ones. A player escapes the trading or building loop by taking the empty trade or empty build action respectively. When a player fulfils the victory condition while ending the building phase, the state machine terminates, otherwise another round starts.

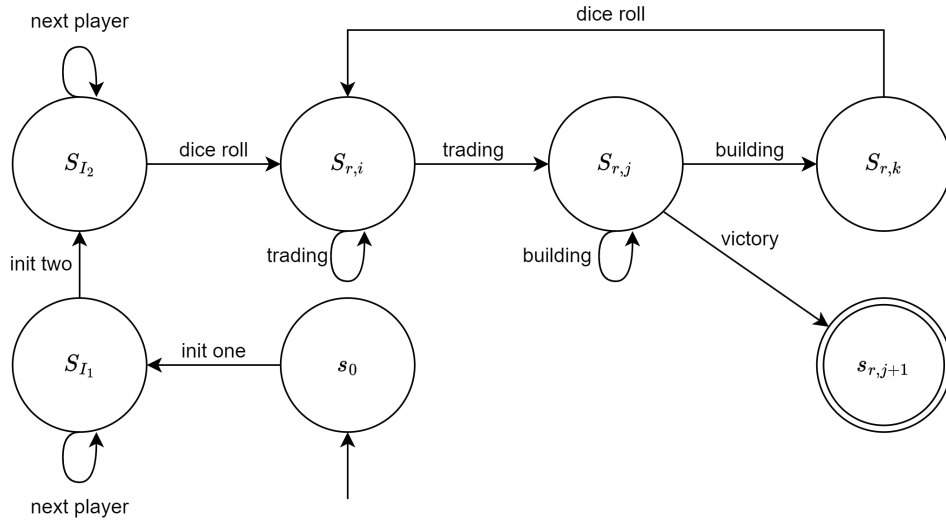


Figure 3.5: State machine of Catan

The next sections details the specifics of the states and actions.

### 3.2.4 Game State

In this section, we define all possible states as TLA+ variables. Unlike constants, variables may be changed by actions. We define what values are considered valid for each variable, which is then checked by TLC and reported, should a value be outside the specification. All the variable definitions can also be found in the appendix section A from line 302 to line 338.

#### Setup

Before a game of "Catan" begins, we need to establish who the players are and in which order they play. A random order of all possible players is chosen and defined as two global constants, one being a set and one being a tuple. The conditions on the players are: 1) There must be at least two players and 2) there must not be more than four players. The set *Players* will be used to check for membership and the tuple *P* will

be used to determine the order and ensure the correct choice of the next active player.

**Definition 17 (Players):**

PossiblePlayers  $\triangleq \{p_1, p_2, p_3, p_4\}$

Players  $\subseteq \text{PossiblePlayers} \wedge \text{Cardinality}(\text{Players}) \geq 2$

Let  $P$  be an arbitrary sequence of Players.

The setup state contains the current setup phase (one of the phases of Definition 15) and the active player, who is always one of the *Players*. In all later definitions for actions (Section 3.2.7), only the active player is given permission to alter the state of variables.

**Definition 18 (Setup state):**

SU  $\in [\text{sup: SUP, ap: Players}]$

**Game State (Proper)**

The state of the game is a record, containing the active player, the state of: the bank, the discard pile, the player hands, the types and number of unplaced player buildings, the settlement and road points, the bandit, and the current turn phase. Possible values are defined below, where the set of all possible values for each component is an operator taking no argument. By convention, we write each with capital letters.

**Definition 19 (Game state):**

G  $\in [\text{ap: Players, bnk: BNK, dp: DP, h: H, b: B, s: S, r: R, ba: M, tp: TP}]$

We now describe in turn the possible states for the bank (*BNK*), the discard pile (*DP*), the players' hands (*H*), the unplaced player buildings (*B*), the settlement points (*S*), the road points (*R*), the bandit (the set of all map tiles *M*), and the turn phase (*TP*).

The bank is a neutral entity that holds all the cards that are not in a player's hand. We represent that with a record of resources and development cards. At the beginning of the game, the bank holds 19 resource cards of each type. The development cards held are: two *Monopoly*, two *YearOfPlenty*, two *RoadBuilding*, 14 *Knight*, and five *VictoryPoint* cards. As the game progresses and cards are obtained or bought by players, the amount for each card type may decrease but can never be below zero.

**Definition 19.1 (Bank):**

BNK  $\triangleq [$   
 RC: [Lumber: 0..19, Brick: 0..19, Wool: 0..19, Grain: 0..19, Ore: 0..19],  
 DC: [Monopoly: 0..2, YearOfPlenty: 0..2, RoadBuilding: 0..2,  
 Knight: 0..14, VictoryPoint: 0..5]  
 $]$

The discard pile holds the played and discarded progress cards. It is represented as a record with three fields, which can hold between zero and the maximum card amount available in the Development Card Pool (Definition 6).

**Definition 19.2 (Discard Pile):**

DP  $\triangleq [\text{Monopoly: 0..2, YearOfPlenty: 0..2, RoadBuilding: 0..2}]$

The hand of a player consists of a record, which includes the resource cards (*RC*) and the three development card sets that the player currently owns (*DC*): bought cards (*BC*), available cards (*AC*), and unveiled cards (*UC*). *BC* and *AC* are members of the set of records *BCT* (Definition 7) and *UC* is a member of *UCT* (Definition 8). All types of cards follow the rule that a player must never have less than zero cards for each type and no more than the total number of cards available in the game.

**Definition 19.3** (Single Player Hand):
$$PH \triangleq [$$

$$\quad RC: [Lumber: 0..19, Brick: 0..19, Wool: 0..19, Grain: 0..19, Ore: 0..19],$$

$$\quad DC: [$$

$$\quad \quad BC: BCT,$$

$$\quad \quad AC: BCT,$$

$$\quad \quad UC: UCT$$

$$\quad ]$$

$$]$$

TLA+ offers convenient syntax to define the set of all possible functions that map values from one set to those of another set:

**Notation 12** (Set of Functions):

The set of functions, written as  $[S \rightarrow T]$  is similar to the set of records (Notation 11), but instead of having a single field, we now map a whole set of fields. Concretely, the set of functions computes the Cartesian product of  $S$  and  $T$ , but instead of a just sets, we get a set of sets of records. For example, let us assume  $S \triangleq \{p1, p2\}$  and  $T \triangleq \{0,1\}$ . Then we would get:

$$[S \rightarrow T] \triangleq \{ \{ [p1 \mapsto 0], [p2 \mapsto 0] \},$$

$$\quad \{ [p1 \mapsto 0], [p2 \mapsto 1] \},$$

$$\quad \{ [p1 \mapsto 1], [p2 \mapsto 0] \},$$

$$\quad \{ [p1 \mapsto 1], [p2 \mapsto 1] \} \}$$

We use the set of functions notation to represent the state of all possible player hands by mapping each player to every possible Player Hand ( $PH$ ). Note that  $PH$  (Definition 19.3), is the hand for a single player and should not be confused with Player Hands  $H$ , which holds all possible hands for all players, specifically the Cartesian product of *Players* and  $PH$ .

**Definition 19.4** (Player Hands):
$$H \triangleq [Players \rightarrow PH]$$

Player Buildings is a record containing all buildings that a player has not placed yet. Their minimal amount is zero and their maximal amount is constrained by the player building pool ( $PB$ , Definition 14).

**Definition 19.5** (Player Buildings):
$$PB \triangleq [Road: 0..15, Village: 0..5, City: 0..4]$$

The set of all possible player buildings is represented as the Cartesian product of *Players* and player buildings ( $PB$ ).

**Definition 19.6** (Buildings):
$$B \triangleq [Players \rightarrow PB]$$
**The Map**

To represent pieces on the board, we use the points a player can build on. Since settlements (villages and cities) cannot be built on the same points as roads, we differentiate between settlement points and road points. In Figure 3.6 the settlement points are shown in red and road points in orange.

As can be seen, settlement points are on the intersection of three hexagonal tiles, that are all adjacent to each other and at least one of the three tiles is not water. These coordinate triplets are unique and defined in the set  $CT$  (see Appendix A). Initially, settlement points have no owner and no building, which we represent as  $\perp$  for both. During the game, at most one player may place a village on a settlement point. Later the same player may upgrade the village to a city. The settlement type ( $st$ ) of a settlement

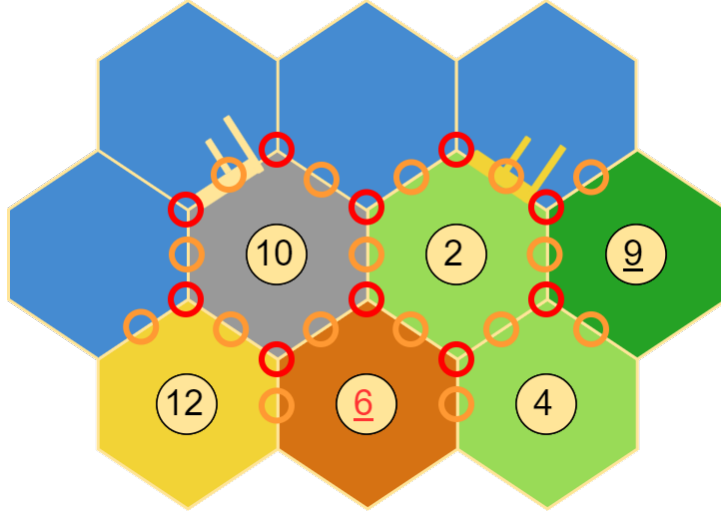


Figure 3.6: Settlement points (red) and road points (orange)

point is therefore either one of the settlement types ( $ST$ , Definition 13), or  $\perp$ , and the owner is either one of the players ( $Players$ , Definition 17) or  $\perp$ .

**Definition 19.7** (Settlement Points):

$$S \triangleq [CT \rightarrow [\text{own}: Players \cup \{\perp\}, \text{st}: ST \cup \{\perp\}]]$$

Figure 3.7, illustrates three different examples of settlement points. The first example is at the shore with coordinate tiles  $t1$ ,  $t2$ , and  $t5$ , belongs to player one (red) and is a village. The second example is at the bottom, close to the centre, belongs to player two (green) and is a city. The third example, at the bottom left, is an empty settlement point, in which both the owner and settlement type fields are  $\perp$ .

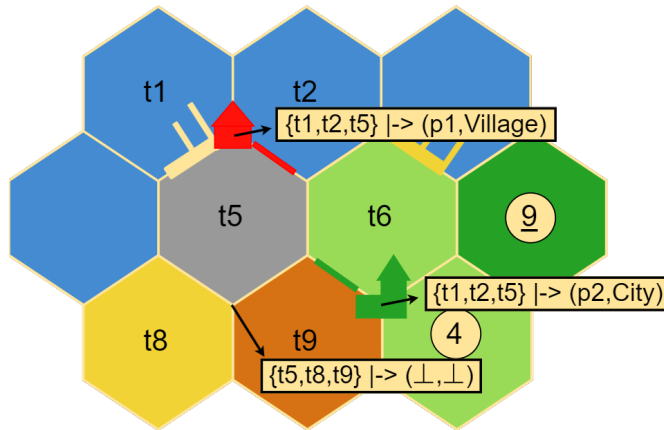


Figure 3.7: Settlement Points Example

Road points are similar to settlement points, but have coordinates using only two hexagonal tiles. Tiles used as coordinates must be adjacent and at least one of the two is not a water tile. These coordinate pairs are unique and defined in the set  $CD$  (Appendix A). Similar to settlement points, we map every coordinate pair in  $CD$  to records indicating ownership. Unlike the settlement points, there is only a single type of road, so we do not need a type distinction. If there is no road, the owner is  $\perp$ .

**Definition 19.8** (Road Points):

$$R \triangleq [CD \rightarrow [\text{own: Players} \cup \{\perp\}]]$$

In Figure 3.8 we see three examples of road points: the red player owns a road with coordinates  $r2$  and  $r5$ , the green player owns the road on  $\{r6, r9\}$ , and the road point with coordinates  $r8$  and  $r9$  is not yet built.

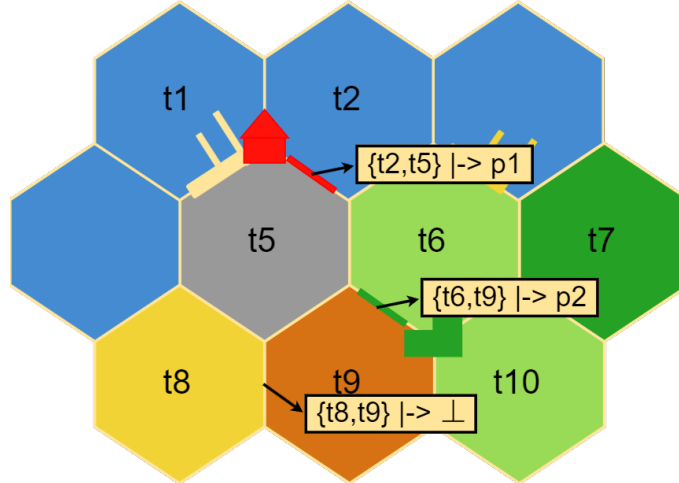


Figure 3.8: Road points example

The bandit is a special piece that sits on a single map tile and prevents building and resource production. It has no state other than which tile it is currently on, so the set of all possible states for the bandit is the set of map tiles ( $M$ ), excluding water tiles.

### 3.2.5 Invariants

We introduce the invariants that need to hold throughout the board game to ensure correctness.

We begin with the invariant that constrains the defined variables  $SU$  and  $G$ . Its name is `TypeOK` and it is equal to the conjunction of the two formulas defining the type of the setup state (Definition 18) and the game state (Definition 19). It forces TLC to check that the state of  $SU$  and  $G$  are valid as defined after each taken action.

The next invariant verifies that the amount of cards in the resource card pool is equal to the sum of cards in player hands and in the bank. The function `SumResourceAllPlayers(res)` sums up the resource cards of all player hands, that are of the type specified by `res`. The function `getIndex(e1, tup)` returns the position of the element `e1` in the tuple `tup`.

**Invariant 1** (ConservationOfResourceCards):

$$\begin{aligned} \text{ConservationOfResourceCards} \triangleq \\ \forall \text{rct} \in \text{DOMAIN } G.\text{bnk.RC}: \\ \text{SumResourceAllPlayers}(\text{rct}) + G.\text{bnk.RC}[\text{rct}] = \text{RCP}[\text{getIndex}(\text{rct}, \text{RCTST})] \end{aligned}$$

TLA+ provides a convenient syntax to make conjunctions with multiple terms and large expressions easier to read:

**Notation 13** (TLA Conjunctions):

TLA+ allows conjunctions to be used like bullet points. For example, if we have predicates  $A$  and  $B$ , these two notations are equal:  $A \wedge B \triangleq \begin{array}{l} \wedge A \\ \wedge B \end{array}$

We use the conjunction syntax to represent the invariant that there is conservation of development cards. We gather all the development cards of all players, the discard pile, and the bank and check if they are equal to the development card pool. The function `SumDevelopmentAllPlayers(dct)` sums up the development cards of all the player hands with type `dct`.

**Invariant 2** (ConservationOfDevelopmentCards):

$$\begin{aligned} \text{ConservationOfDevelopmentCards} \triangleq \\ \wedge \quad \forall \text{dct} \in \text{DOMAIN } G.\text{dp}: \text{SumDevelopmentAllPlayers}(\text{dct}) + \\ \quad G.\text{dp}[\text{dct}] + \\ \quad G.\text{bnk.DC}[\text{dct}] = \\ \quad \text{DCP}[\text{getIndex}(\text{PCTMAP}[\text{dct}], \text{PCTST})] \\ \wedge \quad \text{SumDevelopmentAllPlayers}(\text{"Knight"}) + G.\text{bnk.DC.Knight} = \text{DCP}[4] \\ \wedge \quad \text{SumDevelopmentAllPlayers}(\text{"VictoryPoint"}) + G.\text{bnk.DC.VictoryPoint} = \text{DCP}[5] \end{aligned}$$

Our last invariant is that each player should have all board game pieces either in hand or on the board. We write this by asserting that the sum of unused and placed buildings of that player must always be equivalent to the player's total building pool ( $PB$ , Definition 14). We define a function `NrPlayerRoadsOnMap(p)` that returns the number of roads on the map, that are owned by player  $p$ . Analogously to the road function, we define `NrPlayerSettlementOnMap(p, st)` to return all settlements on the map with the type `st` and owner  $p$ .

**Invariant 3** (ConservationOfBuildings):

$$\begin{aligned} \text{ConservationOfBuildings} \triangleq \\ \forall p \in \text{Players}: \\ \wedge \quad G.b[p].\text{Road} + \text{NrPlayerRoadsOnMap}(p) = \text{BP}[1] \\ \wedge \quad G.b[p].\text{Village} + \text{NrPlayerSettlementOnMap}(p, \text{Village}) = \text{BP}[2] \\ \wedge \quad G.b[p].\text{City} + \text{NrPlayerSettlementOnMap}(p, \text{City}) = \text{BP}[3] \end{aligned}$$

### 3.2.6 Initial State

In this section, we define the initial state of our state machine. We define the setup and game state to reflect the empty default map in Figure 2.1. We begin with the player at index one in  $P$ , in setup phase one. The bank holds all the cards, the discard pile is empty, each player starts with an empty hand and all buildings. Moreover, all settlement and road points are empty, the bandit is placed on the centre desert tile, and the turn phase is  $\perp$ :

**Definition 20 (Init):**

$$\begin{aligned} \text{Init} \triangleq & \\ \wedge \quad & \text{SU} = [\text{sup} \mapsto \text{PhaseOne}, \text{ap} \mapsto P[1]] \\ \wedge \quad & \text{G} = [ \\ & \quad \text{ap} \mapsto P[1], \\ & \quad \text{bnk} \mapsto [ \\ & \quad \quad \text{RC} \mapsto [\text{Lumber} \mapsto 19, \text{Brick} \mapsto 19, \text{Wool} \mapsto 19, \text{Grain} \mapsto 19, \text{Ore} \mapsto 19], \\ & \quad \quad \text{DC} \mapsto [\text{Monopoly} \mapsto 2, \text{YearOfPlenty} \mapsto 2, \text{RoadBuilding} \mapsto 2, \\ & \quad \quad \quad \text{Knight} \mapsto 14, \text{VictoryPoint} \mapsto 5] \\ & \quad ], \\ & \quad \text{dp} \mapsto [\text{Monopoly} \mapsto 0, \text{YearOfPlenty} \mapsto 0, \text{RoadBuilding} \mapsto 0], \\ & \quad \text{h} \mapsto [p \in \text{Players} \mapsto [ \\ & \quad \quad \text{RC} \mapsto [\text{Lumber} \mapsto 0, \text{Brick} \mapsto 0, \text{Wool} \mapsto 0, \text{Grain} \mapsto 0, \text{Ore} \mapsto 0], \\ & \quad \quad \text{DC} \mapsto [ \\ & \quad \quad \quad \text{BC} \mapsto [\text{Monopoly} \mapsto 0, \text{YearOfPlenty} \mapsto 0, \text{RoadBuilding} \mapsto 0, \text{Knight} \mapsto 0], \\ & \quad \quad \quad \text{AC} \mapsto [\text{Monopoly} \mapsto 0, \text{YearOfPlenty} \mapsto 0, \text{RoadBuilding} \mapsto 0, \text{Knight} \mapsto 0], \\ & \quad \quad \quad \text{UC} \mapsto [\text{Knight} \mapsto 0, \text{VictoryPoint} \mapsto 0] \\ & \quad \quad ] \\ & \quad ], \\ & \quad \text{b} \mapsto [p \in \text{Players} \mapsto [\text{Road} \mapsto 15, \text{Village} \mapsto 5, \text{City} \mapsto 4]], \\ & \quad \text{s} \mapsto [c \in \text{CT} \mapsto [\text{own} \mapsto \perp, \text{st} \mapsto \perp]], \\ & \quad \text{r} \mapsto [c \in \text{CD} \mapsto [\text{own} \mapsto \perp]], \\ & \quad \text{ba} \mapsto [\text{coords} \mapsto [\text{arr} \mapsto 1, \text{row} \mapsto 1, \text{col} \mapsto 3], \text{res} \mapsto \perp, \text{n} \mapsto \perp], \\ & \quad \text{tp} \mapsto \perp \\ & \quad ] \end{aligned}$$

### 3.2.7 Actions

In this section, we introduce the actions of our model. An action is a transition from one state to another state. This transition can only be taken if the action as a whole evaluates to true. An action consists of two parts: 1) The enabling conditions, which must be true for the action to be taken, and 2) the primed variables, that mark which variables are changed by this action and how. We begin by introducing the setup steps prior to the game properly starting. We then follow with trading and building. The full TLA+ definitions can be found in the appendix section A from line 341 to line 815.

#### Setup Steps

In phase one of the game's setup, each player must choose a settlement point that is viable according to the rules and place a road on one of the three adjacent road points. The enabling conditions are: 1) the turn phase is bottom, 2) the state is in *PhaseOne* and 3) the active player is a member of *Players*.

The exists ( $\exists$ ) show choices, that can be made by a player, since the predicate following can be true for multiple values. The first choice is the settlement point where we want to place our village. The predicate `buildable(sp)` guarantees that the settlement point is viable according to the rules (Section 2.1.1). Next, the player chooses

a road point, to place a road. The predicate  $\text{isAdjacentRoadToSettlement}(sp, rp)$  ensures the settlement and the road are adjacent to each other and the predicate  $\text{roadHasNoBandit}(rp)$  ensures the bandit is not on a tile bordering the road point.

If these conjunctions can be fulfilled, we change the state appropriately by changing the owner of the settlement and road point, removing a road and a village from the player buildings, and change the active player. In case this action is taken by the last player, we go to *SetupPhaseTwo*. For this phase, we need the reverse order of the players, which can be computed using the function  $\text{reverse}(\text{tup})$ , that reverse a tuple. This sets the next active player to be the one in the first position of the reversed player tuple  $P$ . Otherwise, if we stay in *SetupPhaseOne*, the next player in  $P$  becomes the active player.

**Action 1 (SetupPhaseOne):**

$$\begin{aligned} \text{SetupPhaseOne} \triangleq & \\ & \wedge G.\text{tp} = \text{bot} \\ & \wedge \text{SU}.\text{sup} = \text{PhaseOne} \\ & \wedge \text{SU}.\text{ap} \in \text{Players} \\ & \wedge \exists sp \in \text{DOMAIN } G.s: \\ & \quad \wedge G.s[sp].\text{own} = \perp \\ & \quad \wedge \text{buildable}(sp) \\ & \quad \wedge \exists rp \in \text{DOMAIN } G.r: \\ & \quad \quad \wedge G.r[rp].\text{own} = \perp \\ & \quad \quad \wedge \text{isAdjacentRoadToSettlement}(sp, rp) \\ & \quad \quad \wedge \text{roadHasNoBandit}(rp) \\ & \quad \quad \wedge G' = [G \text{ except } \quad !.b[\text{SU}.\text{ap}].\text{Road} = @ - 1, \\ & \quad \quad \quad !.b[\text{SU}.\text{ap}].\text{Village} = @ - 1, \\ & \quad \quad \quad !.s[sp] = [\text{own} \mapsto \text{SU}.\text{ap}, \text{st} \mapsto \text{Village}], \\ & \quad \quad \quad !.r[rp] = [\text{own} \mapsto \text{SU}.\text{ap}]] \\ & \quad \wedge \text{IF } \text{getIndex}(\text{SU}.\text{ap}, P) = \text{Cardinality}(\text{Players}) \\ & \quad \quad \text{THEN } \text{SU}' = [\text{SU EXCEPT } \quad !.ap = \text{reverse}(P)[1], \\ & \quad \quad \quad !.sup = \text{PhaseTwo}] \\ & \quad \quad \text{ELSE } \text{SU}' = [\text{SU EXCEPT } !.ap = P[(\text{getIndex}(\text{SU}.\text{ap}, P) + 1)]] \end{aligned}$$

TLA+ provides a convenient syntax to define sets of elements satisfying a predicate:

**Notation 14 (Set of Elements satisfying a Predicate):**

In TLA+ we can specify a set with a predicate as a filter. Only elements that fulfil the predicate will be part of the set. For example, if we have the set  $N \triangleq \{2,4,8\}$  and we want only numbers above 5, we write:  $\{n \in N: n > 5\} \triangleq \{8\}$ .

We use this notation in the next phase to filter resource types.

In the second setup phase, in reverse order, we repeat what we did in *SetupPhaseOne*. However, each player receives resources from the three coordinate tiles of the placed village. We take a closer look at the local variable *gain*. The innermost set returns all coordinates of the chosen settlement point ( $sp$ ) that have a resource type. Each coordinate is mapped to a one (villages produce one resource) and then  $\text{SumResFunc}$  creates a record with the amount of resources gained. This record has the same fields as the  $RC$  field in the player hand. We use  $\text{AddRec}(\text{rec1}, \text{rec2})$  and  $\text{SubRec}(\text{rec1}, \text{rec2})$  to add or subtract records with the same length and domain. In this case the player hand is empty and we overwrite it with the gain record. The bank subtracts *gain* from its current record using  $\text{SubRec}$ . If the last player of the reverse order takes this action, we move the turn phase to *DiceRoll*.

**Action 2 (SetupPhaseTwo):**

$$\text{SetupPhaseTwo} \triangleq$$

```

 $\wedge$  G.tp = bot
 $\wedge$  SU.sup = PhaseTwo
 $\wedge$   $\exists$  sp  $\in$  DOMAIN G.s:
   $\wedge$  G.s[sp].own =  $\perp$ 
   $\wedge$  buildable(sp)
   $\wedge$   $\exists$  rp  $\in$  DOMAIN G.r:
     $\wedge$  G.r[rp].own =  $\perp$ 
     $\wedge$  isAdjacentRoadToSettlement(sp, rp)
     $\wedge$  roadHasNoBandit(rp)
     $\wedge$  SU' = [SU EXCEPT
      !.ap = IF getIndex(SU.ap, reverse(P)) = Cardinality(Players)
      THEN @,
      ELSE reverse(P)[(getIndex(SU.ap, reverse(P)))]
    ]
   $\wedge$  LET gain  $\triangleq$  SumResFunc([c  $\in$  {s  $\in$  sp: s.res  $\neq$  bot}  $\mapsto$  1])
  IN G' = [G EXCEPT
    !.h[SU.ap].RC = gain,
    !.bnk.RC = SubRec(@, gain),
    !.b[SU.ap].Road = @ - 1,
    !.b[SU.ap].Village = @ - 1,
    !.s[sp] = [own  $\mapsto$  SU.ap, st  $\mapsto$  Village],
    !.r[rp] = [own  $\mapsto$  SU.ap]
    !.tp = IF getIndex(SU.ap, reverse(P)) = Cardinality(Players)
    THEN DiceRoll
    ELSE  $\perp$ ]

```

With this the setup phases have been completed. Each player has two villages with an adjacent road and earned resources from the second village. From this point on, the setup state *SU* will no longer change. The game enters the standard turn cycle and the first player begins with the dice roll phase.

### Dice Roll

The turn of each player begins with a dice roll. If a seven is rolled, the active player must move the bandit and can steal a resource from another player. If any other number is rolled, each player may receive resources. The most interesting part in this phase is the computation of the cards lost, in case of a seven and the cards gained otherwise. We use *ResourceGainPlayer(p, d)* to compute the resource gain of all players. For each player *p*, we gather the map tiles, which are 1) part of the coordinates of a settlement owned by *p* and 2) have a number that matches the rolled die *d*. We split these map tiles into two sets, one with settlement points that have villages and one with cities. In the end we sum up the resources of all the gathered settlement points. We gain two resources for a tile with a city and one for a tile with a village. *SumResFunc* returns a record with all the resource gains, that can be added to the current state of the player and subtracted from the bank.

#### Function 1 (ResourceGainPlayer(p,d)):

```

ResourceGainPlayer(p,d)  $\triangleq$ 
  LET
    V  $\triangleq$  s  $\in$  UNION {sp  $\in$  DOMAIN G.s: G.s[sp].own = p  $\wedge$  G.s[sp].st = Village}:
      s.n = d
    C  $\triangleq$  s  $\in$  UNION {sp  $\in$  DOMAIN G.s: G.s[sp].own = p  $\wedge$  G.s[sp].st = City}:
      s.n = d
  IN
    SumResFunc([c  $\in$  (V  $\cup$  C)  $\mapsto$  IF c  $\in$  C THEN 2 ELSE 1])

```

The action *DiceRoll* is the most complicated one in the game, thus we split it in two parts: The first part is the case where a seven is rolled (*DiceRoll1A*), and the second

case is for all other dice results (DiceRollB).

**Action 3 (DiceRollP):**

```

DiceRollPhase  $\triangleq$ 
 $\wedge$  G.tp = DiceRoll
 $\wedge$  SU.sup = PhaseTwo
 $\wedge$   $\exists d \in 1..12$ :
    IF d = 7
    THEN DiceRollA
    ELSE DiceRollB

```

Players only lose cards if the sum of their resource cards is higher than seven. We use the function `SumResourcesSinglePlayer(p)` to sum up the resources of player  $p$ . If a player does not lose resources, we define the loss as the record with zero resources of each type. Should a player have more than seven cards, the player needs to transfer half (rounded down) of the cards to the bank. We achieve this by taking the Cartesian product of the *Players* and *ResourceRecsWithSpecificSum*, which constructs the set of records, that have a sum of resources between four and seven. We then filter these records to adhere to the properties we need for each player. First, the sum of all fields in the record (computed using `SumFunc(func)` and `IterateRec(rec)`) must be equal to half the sum of the player's current resource cards. Second, the records must have an amount of resources from zero to the player's amount of resource cards. From the remaining records one is chosen using the leading exists operator.

When rolling a seven, the active player must move the bandit. *BanditMoves* contains all valid tiles to move the bandit to. The only conditions that the bandit has, is that the tile the player moves it to: 1) is a part of the map, 2) is not the current tile that the bandit is on, 3) is not water. In our specification, the player always tries to steal from another player, hence we choose a tile that has a settlement of an adversary adjacent to it. Then we check if the adversary has a resource card we can steal.

Now we need to adjust all player hands according to their losses. We construct new player hands and replace  $G.h$  completely. We map a new player hand to each player and need to cover several cases for *RC*: First we check that if we deduct the losses from the adversary's account, there is still a resource of type *res* that we can steal. If no, we only deduct the loss. If yes, the active player subtracts the loss and adds the stolen resource to the resource record. To add a single element to a record, we use the `AddRecEl(rec, f, n)` function. It adds  $n$  to the field  $f$  in record  $rec$ . The adversary loses the resources in the loss record and additionally the stolen good. If the player is neither the active player nor the adversary, that is stolen from, then we subtract the loss record. If no, all players just lose the loss record. The bank in turn receives the sum of all the losses, the function `SumResRecs(recs)` accomplishes this.

The development cards change state as well. All the cards that are in *BC* are moved into *AC*, because bought cards can be played in the next turn of a player. The unveiled cards stay the same.

At last we set the new tile for the bandit and go to the trading phase.

**Action 4 (DiceRollA):**

```

DiceRollA  $\triangleq$ 
 $\exists$  loss  $\in$  [Players  $\rightarrow$  ResourceRecsWithSpecificSum]:

```

$$\begin{aligned}
& \wedge \quad \forall p \in \text{Players}: \\
& \quad \text{IF } \text{SumResourcesSinglePlayer}(p) \leq 7 \\
& \quad \text{THEN } [\text{Lumber} \mapsto 0, \text{Brick} \mapsto 0, \text{Wool} \mapsto 0, \text{Grain} \mapsto 0, \text{Ore} \mapsto 0] \\
& \quad \text{ELSE} \\
& \quad \quad \wedge \quad \text{SumFunc}(\text{IterateRec}(\text{loss}[p])) = \text{SumResourcesSinglePlayer}(p) \div 2 \\
& \quad \quad \wedge \quad \forall f \in \text{DOMAIN } G.h[p].\text{RC}: \\
& \quad \quad \quad \text{loss}[p][f] \in 0..G.h[p].\text{RC}[f] \\
& \wedge \quad \exists t \in \text{BanditMoves}: \\
& \quad \exists q \in \text{Players}: \\
& \quad \quad \wedge \quad q \neq G.ap \\
& \quad \quad \wedge \quad \text{hasSettlementOnTile}(t, q) \\
& \quad \quad \wedge \quad \exists \text{res} \in \text{DOMAIN } G.bnk.\text{RC}: \\
& \quad \quad \quad \wedge \quad G.h[q].\text{RC}[\text{res}] \geq 0 \\
& \quad \quad \quad \wedge \quad G' = [G \text{ EXCEPT } \begin{aligned} & !.bnk.\text{RC} = \text{AddRec}(@, \text{SumResRecs}(\text{loss})), \\ & !.h = [p \in \text{Players} \mapsto \\ & \quad [\text{RC} \mapsto \\ & \quad \quad \text{IF } G.h[q].\text{RC}[\text{res}] - \text{loss}[q][\text{res}] > 0 \\ & \quad \quad \text{THEN IF } p = G.ap \\ & \quad \quad \quad \text{THEN AddRecEl(} \\ & \quad \quad \quad \quad \text{SubRec}(G.h[p].\text{RC}, \text{loss}[p]), \\ & \quad \quad \quad \quad \text{res}, 1) \\ & \quad \quad \quad \text{ELSE IF } p = q \\ & \quad \quad \quad \quad \text{THEN AddRecEl(} \\ & \quad \quad \quad \quad \quad \text{SubRec}(G.h[p].\text{RC}, \text{loss}[p]), \\ & \quad \quad \quad \quad \quad \text{res}, -1) \\ & \quad \quad \quad \quad \text{ELSE SubRec}(G.h[p].\text{RC}, \text{loss}[p]) \\ & \quad \quad \quad \text{ELSE SubRec}(G.h[p].\text{RC}, \text{loss}[p]), \\ & \quad \quad \text{DC} \mapsto \\ & \quad \quad \quad [\text{BC} \mapsto [\text{Monopoly} \mapsto 0, \text{YearOfPlenty} \mapsto 0, \\ & \quad \quad \quad \quad \text{RoadBuilding} \mapsto 0, \text{Knight} \mapsto 0], \\ & \quad \quad \quad \text{AC} \mapsto \text{AddRec}(G.h[p].\text{DC.AC}, G.h[p].\text{DC.BC}), \\ & \quad \quad \quad \text{UC} \mapsto G.h[p].\text{DC.UC}] \\ & \quad \quad \quad ], \\ & \quad \quad \quad !.ba = t, \\ & \quad \quad \quad !.tp = \text{Trading}] \end{aligned} \\
& \quad \quad \quad \wedge \quad \text{UNCHANGED}(I)
\end{aligned}$$

If we do not roll a seven, things are simpler. We compute all the gains for each player, add the gains to the player hands and remove the sum of all gains from the bank, move all bought cards to the available cards, and move the turn phase to *Trading*. In the special case where the bank does not have enough resources to cover the gains of all players, no player gains anything.

**Action 5 (DiceRollB):**

DiceRollB  $\triangleq$   
 LET  
   gain  $\triangleq [p \in \text{Players} \mapsto \text{ResourceGainPlayer}(p, d)]$   
 IN

```

 $\wedge$  IF
  THEN
     $\wedge$   $G' = [G \text{ EXCEPT } \begin{array}{l} \text{!.bnk.RC} = \text{SubRec}(@, \text{SumResRecs}(\text{gain})), \\ \text{!.h} = [p \in \text{Players} \mapsto \\ \quad [\text{RC} \mapsto \text{AddRec}(\text{G.h}[p].\text{RC}, \text{gain}[p]), \\ \quad \text{DC} \mapsto [\text{BC} \mapsto [\text{Monopoly} \mapsto 0, \\ \quad \quad \text{YearOfPlenty} \mapsto 0, \text{RoadBuilding} \mapsto 0, \\ \quad \quad \text{Knight} \mapsto 0] \\ \quad ], \\ \quad \text{AC} \mapsto \text{AddRec}(\text{G.h}[p].\text{DC.AC}, \text{G.h}[p].\text{DC.BC}), \\ \quad \text{UC} \mapsto \text{G.h}[p].\text{DC.UC}], \\ \quad ], \\ \text{!.tp} = \text{Trading}] \end{array}$ 
     $\wedge$  UNCHANGED(I)
  ELSE UNCHANGED( $\langle\langle I, G \rangle\rangle$ )

```

### Trade

We have simplified trading, so that it only occurs between a single player and the bank during their turn (i.e. trading between players is not supported). The rate at which a player can trade, depends on the ports the player has access to. The most complex conditions for a trade are for the two-to-one rates. A player needs a resource with an amount of at least two and there must exist a wharf that has the same resource type and its coordinates are a subset of the coordinates of a player owned settlement. If these conditions are met, a player can exchange two resource cards for another and the state of the bank and the player's hand are modified accordingly.

#### Action 6 (TradeTwoToOne):

```

 $\wedge$   $G.\text{tp} = \text{Trading}$ 
 $\wedge$   $\exists \text{give} \in \{\text{rct} \in \text{DOMAIN } G.\text{bnk.RC} : G.\text{h}[G.\text{ap}].\text{RC}[\text{rct}] \geq 2 \wedge$ 
 $\quad \exists w \in \text{PlayerPorts}(G.\text{ap}) : w.\text{wt} \in \text{RCT} \wedge w.\text{wt} = \text{RCTMAP}[\text{rct}]\}$ 
 $\quad \exists \text{receive} \in \{\text{res} \in \text{DOMAIN } G.\text{bnk.RC} : G.\text{bnk.RC}[\text{res}] > 0\}$ 
 $\quad \wedge G' = [G \text{ except } \begin{array}{l} \text{!.bnk.RC} = \text{AddRecEl}(\text{AddRecEl}(@, \text{give}, 2), \text{receive}, -1), \\ \text{!.h}[G.\text{ap}].\text{RC} = \text{AddRecEl}(\text{AddRecEl}(@, \text{give}, -2), \text{receive}, 1) \end{array}]$ 
 $\wedge$  UNCHANGED(I)

```

If a player cannot or does not want to trade, the action taken is the *EmptyTrade*, which only changes the turn phase and leaves the rest of the state unchanged.

#### Action 7 (EmptyTrade):

```

 $\wedge$   $G.\text{tp} = \text{Trading}$ 
 $\wedge$   $G' = [G \text{ EXCEPT } \text{!.tp} = \text{Building}]$ 
 $\wedge$  UNCHANGED(I)

```

### Building

In the building phase, a player may build, or buy development cards. We show only one example, since the placement of villages and its conditions were already shown in the setup phases and the cities simply replace existing villages. Our example is the action *RoadBuilding*. First, we check if the player has the necessary resources to build a road. All costs are defined as records with the respective amount of resources needed (e.g.  $\text{RoadCost} \triangleq [\text{Lumber} \mapsto 1, \text{Brick} \mapsto 1, \text{Wool} \mapsto 0, \text{Grain} \mapsto 0, \text{Ore} \mapsto 0]$ ). Another thing we need to validate is, if there is still a road piece in the player's building pool left.

**Predicate 1 (CanBuildRoad):**

$$\wedge \quad \forall \text{ res} \in \text{DOMAIN RoadCost}: G.h[G.ap].RC[\text{res}] \geq \text{RoadCost}[\text{res}]$$

$$\wedge \quad G.b[G.ap].\text{Road} > 0$$

If that is the case then we choose a road point, that is either adjacent to a settlement the player owns or to another player-owned road point. The chosen road point is adjusted to be owned by the player and the cost of the road is transferred from the player to the bank. A road piece is deducted from the player's buildings. The turn phase is not advanced yet, because the player may still build other buildings or buy a development card.

**Action 8 (BuildRoad):**

$$\text{BuildRoad} \triangleq$$

$$\begin{aligned} &\wedge \quad G.tp = \text{Building} \\ &\wedge \quad \text{CanBuildRoad} \\ &\wedge \quad \exists rp \in \text{DOMAIN } G.r: \\ &\quad \wedge \quad G.r[rp].\text{own} = \text{bot} \\ &\quad \wedge \quad \text{roadHasNoBandit}(rp) \\ &\quad \wedge \quad \exists rpt \in \text{DOMAIN } G.r: \\ &\quad \quad \wedge \quad \vee \quad \wedge \quad \text{isAdjacentRoad}(rp, rpt) \\ &\quad \quad \quad \wedge \quad G.r[rpt].\text{own} = G.ap \\ &\quad \quad \vee \quad \wedge \quad \exists sp \in \text{DOMAIN } G.s: \\ &\quad \quad \quad \wedge \quad G.s[sp].\text{own} = G.ap \\ &\quad \quad \quad \wedge \quad \text{isAdjacentRoadToSettlement}(sp, rp) \\ &\quad \wedge \quad G' = [G \text{ except } \quad !.bnk.RC = \text{AddRec}(@, \text{RoadCost}), \\ &\quad \quad \quad !.h[G.ap].RC = \text{SubRec}(@, \text{RoadCost}), \\ &\quad \quad \quad !.b[G.ap].\text{Road} = @ - 1, \\ &\quad \quad \quad !.r[rp] = [\text{own} \mapsto G.ap]] \\ &\quad \wedge \quad \text{UNCHANGED}(I) \end{aligned}$$

Like in the trading phase, if a player cannot or does not want to build, there is an empty build action, that changes only the turn phase. If we end the building phase, however, we additionally check the victory condition. If a player has ten or more victory points that are gained by villages (one point), cities (two points), the longest road (two points), the mightiest army (two points), and victory point cards (one point), then the game ends and the turn phase is changed to  $\top$ . Once the game reaches the  $\top$  state, no further action can be taken and the state machine terminates.

**Action 9 (EmptyBuild):**

$$\begin{aligned} &\wedge \quad G.tp = \text{Building} \\ &\wedge \quad G' = [G \text{ EXCEPT } !.tp = \text{IF PlayerPoints}(G.ap) \geq 10 \text{ THEN } \top \text{ ELSE DiceRoll}] \\ &\wedge \quad \text{UNCHANGED}(I) \end{aligned}$$

## Chapter 4

# Implementation

In this chapter we discuss how we implemented the model mentioned in Chapter 3 in python. We mention all the essential libraries we used and how we integrated Git to handle the synchronization of the game state. To stay faithful to the model and avoid introducing errors due to inconsistencies, the implementation follows the structure of the specification. The full implementation can be found on GitHub [12].

### 4.1 Libraries

The most important libraries we used are: 1) GitPython, which allows us to use Git commands at a high level in python, while still enabling us to use low-level commands. 2) Pygame, that allows us to draw a visualization of the game state and provides a simple way to create user interfaces and helps handling player events, like hovering the mouse over objects, clicking objects, and pressing keys.

### 4.2 Game State

#### 4.2.1 Representation

Each player has an append-only log which is implemented as a Git branch. Since Git is usually used as version control, which operates with folders and files, we represent the state of our game accordingly. We do that in a manner, such that the folder structure mirrors the model definition of the entire game state, seen in the initial state of the model 20. This led us to the structure in Figure 4.1, where each file contains the part of the state which corresponds to the file's name.

#### 4.2.2 Git Branches and Commits Structure

We introduce how we handle all branches and who has access to them. We create a branch for each player in a match, named after the player's colour and the simulation number (e.g. "Red.0"). We assume that only player "Red" commits to the Red-branch and the other players only commit to their branch. Each player can, however, synchronize their branch state with the other player's branches to obtain newer commits they are not yet replicating.

We have a special case after a seven is rolled because player concurrently choose which cards to loose. To do so, each player creates a temporary branch with a unique and deterministic reference name. The temporary branches are named after the id of



Figure 4.1: Folder structure representing complete game state

the initial dice roll commit and then we add "loss" and the standard naming convention to it (e.g. "{commit\_id}\_loss\_Red.0").

The Git commit fields we use in our implementation are as follows: The commit message contains the name of the action (e.g. "trading\_4\_to\_1") and the player number. The parent is usually the head reference of the branch. In case we have a rolled seven, the merge commit has multiple parents to include all concurrent commits. The author and committer are the player's colour designation paired with the simulation number (e.g. "Red.5").

### 4.2.3 Modifying and Synchronizing

The most important part of the implementation, is how we keep the game state up-to-date and synchronized with all peers.

To change the state following a player's action, we modify the respective files, add them to a Git commit, name the commit after the action that was taken, and commit it to the player's git branch. The history of commits not only serves for synchronizing with other peers, we also use it to obtain a summary of the sequence of players' actions for debugging.

To retrieve other players' actions and the latest state of the game, we track them as remote branches and update periodically. For most actions, since only the active player may modify the state and all players perform their own actions sequentially, there are no concurrent updates. We therefore simply use the built-in Git merge command: updates

result in "fast-forward" merges and cannot create conflicts.

The only exception is when a seven is rolled and all players with more than seven cards have to choose which cards they want to discard(see Section 3.2.7). In that case, we use the special naming convention mentioned in Section 4.2.2. The active player actively looks for these branches, awaits all choices and then merges the results using a custom merge, modifies all files and commits the result. The result is then propagated normally to all other players. Each player can delete the temporary branch after having replicated the merged result.

In the usual non concurrent case, an example of an action changing the state could look like this: We assume that we play a two-player match and the initialization phases are over. Player one rolls the dice and each player receives resources from the bank, after that the turn phase changes. This means we modify the bank, both player hands, and the turn phase. All changed files can be seen in Figure 4.2. We add these files to the "dice roll"-commit and append it to Player one's branch. Player 2 can then merge with Player one's branch and is up to date. The update could happen instantly after the phase of player one is finished or player two could stay in an unsynchronized state until player two eventually decides to synchronize with player one again.



Figure 4.2: Modified Files

#### 4.2.4 Interface Files/Code

To make the state in the files usable in our code, we created an input/output interface, which handles the conversion from python class objects into strings and vice versa. So to use the road points for example, we read the "road\_points"-file line by line and convert each line into a RoadPoint class object, which we use to represent all roads in the game. If we build a road, the respective RoadPoint object is modified and the corresponding line the "road\_points"-file is overwritten with the string representation of the object.

## Chapter 5

# Evaluation

In this chapter we evaluate the resource consumption of our implementation (Chapter 4). We also argue why the model we used is correct. We will first introduce the metrics we have taken in Section 5.1 and then discuss the correctness of our model in Section 5.2.

### 5.1 Metrics

The metrics we evaluate are: 1) the storage space required per game, 2) the average growth per action and 3) the overhead in latency introduced by using a distributed system instead of a single central server. We present our Methodology in Section 5.1.1 first. Then we talk about the results in Section 5.1.4.

#### 5.1.1 Methodology

All measurements were taken on a single computer. Its specifications can be found in Figure 5.1. We expect all measurements, except the computation, to yield similar results even with a different computer.

OS	Windows 11
CPU	i9-12900K 3.19GHz 16 Cores
MEM	32GB DDR4
GPU	RTX 2070 Super 8GB GDDR6

Figure 5.1: Machine specifications

### 5.1.2 Simulation

As explained in Chapter 4, the state of a game of "Catan" is represented by a Git tree inside a commit. Every commit reflects a player's action and contains the new state resulting from that action. To mimic a distributed game on multiple machines, we ensure that the history generated on one machine is equal to the history that would have been generated by multiple peers.

To do this, we create a separate branch for each player. This branch is only manipulated by the player it represents.

The whole simulation is set up to run a loop until the winning condition is reached or an invariant is violated. In each simulation run, a player is chosen in a round-robin manner and merges states with the other branches. After synchronizing, the simulator follows the standard turn phase of "Catan" (see Section 2.1): The chosen player takes an action if possible, given the previous state of the game. This ensures that any player can try to invoke an action, even if another player is the current active player, in order to possibly catch bugs in enabling conditions. When the victory condition is reached, a new simulation starts.

To distinguish different simulation runs, each run creates its own directory with a unique repository name. The gathering of data is done by traversing the Git commit tree after the execution, except for the elapsed time, which is taken during the simulations.

### 5.1.3 Measurements

We selected metrics in order to provide a reasonable baseline for developers that may re-implement our design on other combination of platforms and languages, or design new optimizations for our current implementation. We measured: 1) the storage space of a full game of "Catan", 2) the average size of a state change, and 3) the latency of actions as well as the time spent on managing the state. We now explain the methodology of how we measure these three metrics in more detail.

#### Storage

After the simulations have all been finished, we use the unix "`du -sh`" command on the `./git` directory. This returns a human readable estimate of the disk usage of all files in the specified directory. With this we get a measure of how much disk space the git repository needs. The amount of disk space includes all actions for all players and all intermediary states of the game, as would be the case in a peer-to-peer scenario since all peers replicate the full history.

#### Git delta sizes

As explained in Section 5.1.2, when the active player changes, the first action is to synchronize the state with all other peers. This means that our deltas are only between parent-child commits. Because of this, we compute the differences between all references that are parent and child using the "`git diff ref~x ref~(x-1) | wc -c`". It computes the difference between the two references and then counts the bytes for the changed files.

#### Latency of Actions

We calculate the computation time by measuring the time spent on performing an action, i.e. computing the new state and committing it, and synchronizing between the Git branches. To simulate the latency between two peers, we measure the time it takes to

pull from a GitHub repository in two cases: 1) The state is the same and we just confirm that the current state is up-to-date. 2) The state differs and needs to be updated.

We do not measure throughput, since we only deal with small peer-to-peer groups of up to four members and for all practical purposes it is negligible compared to the amount available on commonly used WiFi networks and home internet connections.

### 5.1.4 Results

In this section, we present the data we measured and present our interpretation.

#### Storage

We simulated 100 two-player matches and 100 four-player matches of "Catan" and measured the disk space needed and the number of commits until the game terminated. The number of commits corresponds one-to-one with the number of actions all players have taken.

We begin with two-player simulations, shown in Figure 5.2a. We can observe that most games need at least 500 to 1250 actions to reach a state with a winner. They need around 1.75 to 2.5MB in total for storage. The red circle shows the average of 2.12MB with a standard deviation of 0.41MB. The average number of commits is 880 with a standard deviation of 360 commits. If we normalize the repository size with the number of commits, the normalized average is 2.77KB per commit with a standard deviation of 0.99KB.

We also report on four-players simulations, as shown in Figure 5.2b. We see that the disk space is 2.3MB on average with a standard deviation of 0.62MB and an average number of commits of 1168 with a standard deviation of 369 commits. The normalized values have an average of 2.18KB per commit with a standard deviation of 0.81KB, showing less variation than the two-player data.

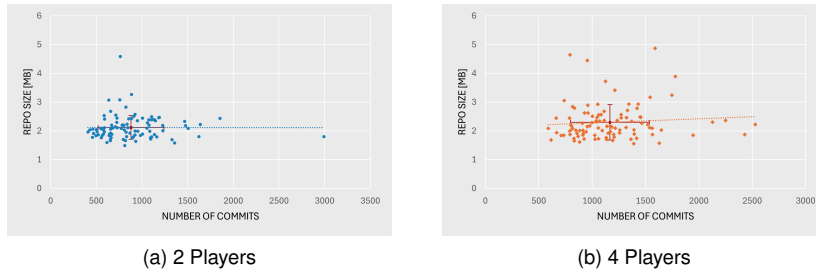


Figure 5.2: Needed Storage Space of "Catan" simulations

#### Discussion

It is clear that a minimum of states is needed for a player to amass resources and build up points in order to reach ten victory points. This can be further delayed if the dice rolls are unfavourable for the players and they do not receive resources. Furthermore, since our simulator selects actions randomly, it can happen that simulated players take actions that are not optimal for the state they are in. Games with human players are likely to be shorter and therefore require less total space, so our results represent an upper bound.

Comparing the two-player and four-player games, we can clearly see, that the two-player matches need on average 50% less commits to crown a winner compared to a four-player game. Having more players means the space on the map, the resources,

and the victory points available are spread more due to the increased number of players, which leads to more actions being taken.

The two new players adding four records (two Player Building (Definition 19.5) and two Player Hand (Definition 19.3 records)), explains the difference in average repository size. These records give further state that can be modified and incur further size costs.

We see big variances in the repository sizes, these can be explained by the empty actions a player must take if they have a resource shortage. These actions only change the turn phase, and have minimal disk space cost. These droughts of resources happen more often in two player-games, since the turn frequency is higher than in a four-player game and players have less time to gather resources from their adversaries' dice rolls. This could explain the slightly lower variance in disk size for four-player matches.

### Git delta sizes

We gathered the commit delta sizes of a two- and a four-player game into the histograms in Figure 5.3. Both matches provided 750 commit deltas. We see that we often have deltas with zero impact on the size, for this two-player match 24.4% are empty updates. The four-player match has 22.9% empty updates. The average size of an update is 0.49KB, with a standard deviation of 0.32KB in two-player matches and for four-player games the average update has a size of 0.63KB with a deviation of 0.45KB.

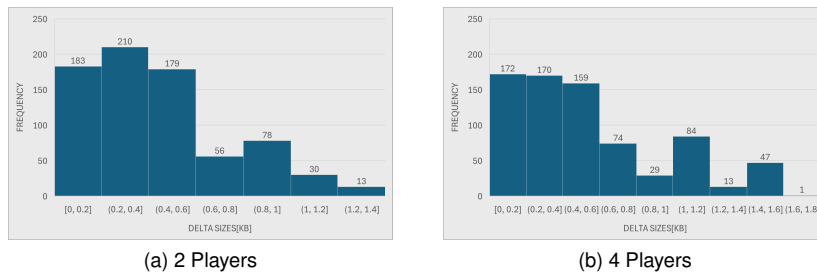


Figure 5.3: Histograms of Git commit deltas

### Discussion

We have a similar amount of empty updates and overall the four-player match has a higher update cost, which can be explained by the additional records of the two player difference. The bigger state incurs higher update costs. We expected to see significantly more empty actions in the two-player game than in the four-player game.

### Latency Of actions

We simulated 100 two-player simulations and 100 four-player simulations. In Figure 5.4 we can see, that two-player matches need an average of 113s to finish, with a standard deviation of 45.6s. Four-player matches finish on average in 406s, with a standard deviation of 130.5s.

Figure 5.5 shows the elapsed time normalized with the number of commits and broken up into synchronization and computation. We can see that the synchronization process needs a significant amount of the total time spent. For two-player games, the average time spent on synchronization is 64.4ms with a standard deviation of 4.9ms. Computation on the other hand needs an average of 64.1ms with a standard deviation

of 5.6ms. This means 50% of the elapsed time can be attributed to the synchronization alone. For four-player matches synchronization takes 249.4ms on average with a standard deviation of 25.6ms and the elapsed time for the computation is 99.7ms with a standard deviation of 9.9ms. This results in a synchronization percentage of around 70%.

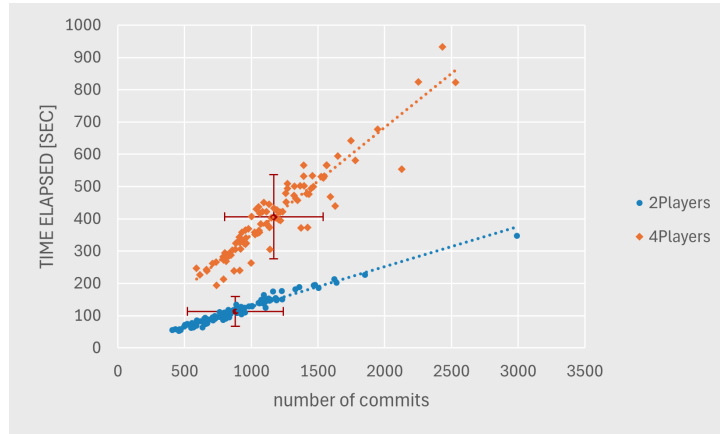


Figure 5.4: Elapsed time of two- and four-player simulations

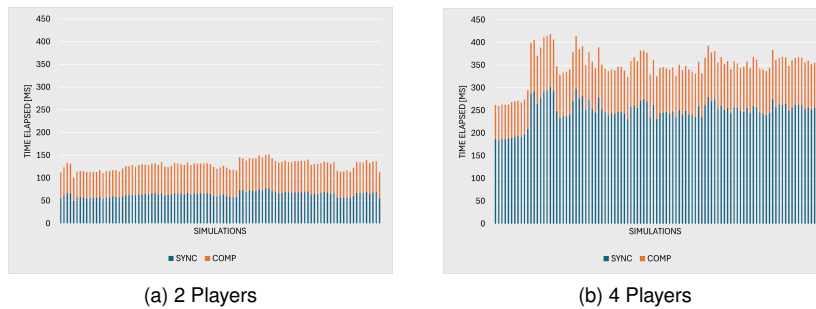


Figure 5.5: Local synchronization impact on elapsed time per commit

The latency of pulling from GitHub, rather than synchronizing locally, without differing state is 1s and 1.06s with differing state. The standard deviation for both is around 0.05s.

## Discussion

It is easy to see in Figure 5.4, that four-player matches need more time to be simulated than two-player games. The reason can be seen in Figure 5.5, the computation amount did not significantly increase for more players, but the synchronization cost rose with a factor of four. This could be explained by the simulations eagerness to always replicate the state, when the player changes. This could potentially be improved.

We already discussed the difference in the amount of commits in the discussion section of the storage evaluation above, hence we will continue with the steep latency cost incurred when we would synchronize data with peers. One second dwarfs the 250ms needed to synchronize the state locally. In a scenario where a human actively plays the game, a second is negligible.

## 5.2 Correctness

In this section, we outline the methodology we used to show that our model is correct. We also discuss our TLA+ specification and the python implementation.

### 5.2.1 Methodology

We use our TLA+ specification introduced in Model 3 to check safety properties. They are checked after each action and the model checker stops should a property be violated. Instead of letting the model checker go through all possible states, we use simulation-mode, which chooses random actions and reaches higher depths faster. We do this because the exhaustive search got to a depth of 9 from an average game depth of 1100. It ran for 3 hours with 550'000 states per minute and searched 115 million states and still had actions that could not be reached yet. For this reason we decided to use sampling of runs instead of doing an exhaustive state space search.

The python implementation that we used to gather data, simulates the game as well. We implemented it to reflect the TLA+ model and check the invariants of the game after each action. When an invariant is violated, the error is written into a log and the state of the game is reset to the last action and the simulation tries again. Additionally, we have a visualization that can be used to check if the game board and all its pieces are in the right place.

### 5.2.2 TLA+

While doing simulations with TLA+, we already reached terminal states without violating safety properties, which means these properties hold. However we had to minimize some of the state spaces for that to be possible. The loss computation in particular took considerable time to compute valid values.

The safety properties consist of the invariants that checks if the available number of game pieces and cards is consistent. For example that no village piece goes missing or that no card is removed from a player without being added to another player or the bank. Other properties are the type correctness of all variables in the state. We check that each variable is in the set of possible states as defined by the model. This is rechecked after every action, even if it does not modify the variable.

Since this was our first time working with TLA+, the invariants helped getting rid of many small mistakes and the simulations helped spotting issues by mentioning which states were not entered in the whole run. This method of running simulations and repairing issues was repeated until all states were reachable and no safety property was violated during a run. This guarantees that these properties hold.

We did not check the liveness properties using TLC, but we did with the python simulations. Every match eventually terminated with a winner and the turn phase equal to T.

### 5.2.3 Simulations

The python implementation used the TLA+ model as a blueprint and should have no bugs that were already found with the specification. After repairing all the bugs revealed by the invariant checks and correcting the logic for roads, after seeing weird road placement in the visualization, we ran the python implementation for at least 200 two-player matches to gather data for our evaluation, in which all reached the termination state without issue. After running four-player matches an issue with the bank going into negative resources popped up, which we repaired as well and ran another 100 simulations for two- and four-player matches each. No issue was found in these

200 runs. We sampled the finished games by running them with the visualization to see the end state and confirm that all placements of the game pieces are legitimate.

## Chapter 6

# Related Work

In this chapter we dive into specifications made with TLA+, discuss different implementations of "Catan", and existing research around formalizing games and using formal languages to design and test them.

### 6.1 TLA+

With the publication of the TLA+ Toolbox [6], which uses the TLC model checker [35] to help users write models and the accompanying book for specifying systems [9], it has become more approachable for many programmers to write formal specifications for programs. The use of formal methods like TLA+ can help uncover unlikely bugs and improve the design of a distributed system and is actively being used by giants like Amazon Web Services [17]. Developers at Amazon mention that the TLA+ specification not only helps with the design and discovery of bugs but it can double as a high level documentation of a complex system as well. However, real systems have points of failure outside the logic that can be captured by TLA+ and thus it does not cover everything. After creating a model ourselves, we can only agree that a specification helps with unearthing bugs. Thinking about states on a higher level invites redesigning parts of the specification before a single line of code is written.

Most publicly available examples of TLA+ specifications are focused on logic puzzles, small algorithms, and protocols [30]. In regards to games, we have found a TLA+ model for tic-tac-toe [28] and some single-player puzzle games [27, 30] but nothing that reaches the complexity of a board game like "Catan". To our knowledge, we provide the first specification of TLA+ for a complex board game.

### 6.2 Catan implementations

There are many open source implementations for "Catan". Some implement all aspects of the board game, including a server and host application [3, 16], or only a local version, where all players take turns on the same machine [22]. The second type of implementation uses the game to train AI adversaries using reinforcement learning [13, 33]. To our knowledge, there are no implementations of "Catan" that utilise a peer-to-peer approach. Neither are there any that use Git in particular to synchronize state. Further search for games that use Git as a synchronization tool were without result, which means that to the best of our knowledge, our work is the first to use Git to synchronize game state.

## 6.3 Formalized Games

There are other formal languages that have been used to describe games, for example the Planning Domain Definition Language (PDDL), which is an attempt to standardize AI planning languages. The viability of PDDL as a chess puzzle solver was explored in a recent thesis [23] and found wanting for examples with bigger board size than 3x3. SAS is another formal language used to formalize actions in a game. It was used to train AI to find the optimal building order in Starcraft: Brood Wars [34]. There are many examples that focus on AI training but there is less work focusing on the design and correctness aspects of games using formal methods.

## 6.4 Git-based Applications using Append-Only Logs

The GOC-Ledger [11] and its modification, the  $\delta$ -GOC-Ledger are both implemented using Git and compared [4]. The GOC-Ledger relaxes the total ordering used by conventional blockchains and uses state based conflict-free data types instead of consensus algorithms to reconcile states between peers. Its delta version further improves by reducing message sizes significantly. Another application is the 2-phase single-author append-only log [10], which can eventually detect forks from malicious participants and exclude them from the system. This allows for detection of malicious behaviour instead of needing to prevent it.

## Chapter 7

# Conclusion

In this thesis we introduced a formal approach to implement the board game "Catan". We captured all the characteristics of the game with a TLA+ model, that we used to verify the correctness of the model itself and to create a Python implementation. The model helped us see redundant definitions and improved the way we defined settlement and road points. It helped us communicate about the game without having to define platform specific details and it made us think about what the core invariants of our application are. We used these invariants in our TLA+ model and the implementation. This and the implemented visualization of the game state, helped unearthing many bugs and accelerated the development of the application.

We presented a Python application that can simulate matches without player input and uses Git to implement append-only logs. It provides the full history of a game in the Git repository and was useful for debugging and gathering statistics. To our knowledge, this thesis provided the first TLA+ model for a complex multi-player board game, as well as the first peer-to-peer implementation of "Catan" using Git. We additionally presented performance metrics that may serve as a point of comparison for future implementations on different centralized or peer-to-peer systems. Results were: First, that a game of "Catan" with four players and a full git history needs on average less than 3MB of storage space. Second, a fifth of all commits were empty, meaning they had a negligible difference to their previous state. Third, in a four-player match, the time spent to compute an action and push the new state is dominated by the synchronization. When we synchronized a four-player match with a peer, we spent 100ms on the action and ten times the amount on the synchronization.

With this thesis we wanted to give programmers insight into the usefulness of prototyping with a model first, to check for logic issues and implement programs based on a model blueprint. In addition, we showed the usefulness of Git in a peer-to-peer setting as a mechanism to handle synchronization in an efficient manner.

## 7.1 Insights of the Development Methodology

### 7.1.1 Working with TLA+

Many programmers, myself included, are wary of formal methods when programming. However, working with TLA+ feels more like programming than cold, hard mathematics. The editor and compiler help mitigate errors and the testing environment feels approachable. Creating a model before implementing an idea makes it necessary to think about the whole application at a high-level, e.g. what are the necessary constants and variables, what actions can be taken in what state, and which states are considered valid states. This process can lead to optimized data structures or changes in

the base approach in general. This happens without incurring too much programming debt, since we don't need to rewrite a complete implementation, we simply adjust our specification. It does, however, need a shift from thinking about classes, functions and programming data structures, into a more set, predicate, and definition oriented universe. It may take some time to accomplish this shift, but after it is done, thinking in that fashion becomes second nature. The TLA+ specification is a good tool to discuss programs with other people, since the code of the model is more compact than the implementation. In our example, the TLA+ model is 700 lines long, 100 of these lines being the hard-coded parts of the map. Our python implementation on the other hand is 2'200 lines without the visualization and 2'700 with UI. This is a difference of a factor of three to four, which is quite significant. The fact that the model specification of the program is shorter, more high-level, and without specific platform details, makes it valuable as documentation. A new person on a project may have an easier time reading a specification than having to study thousands of lines of code, that are bloated with platform specific details.

Thinking about an application on a high-level, and writing the specification as a documentation supplement (aspects not related to distributed behaviour, e.g. security, still need to be documented, and TLA+ does not cover them...), are just the initial benefits. While writing the model, we can check it for errors using type definitions, safety and liveness properties. This helps us find mistakes in the code and in the logic of the model. After the model is complete, we can check if each action is reachable, and if it always terminates. If that is the case, we can use the model to implement our application on a chosen system.

In our case, we found half the bugs in our implementation, just by checking the invariants of the model after each action. This included a city being built on another city instead of replacing a village, development cards vanishing or the bank running out of cards to provide resources to the players. Having a visualization of what is happening is helpful as well, in our implementation there was a logic error, that allowed wrong road placements, because the definition of an adjacent road was wrong. Errors such as these, are hard to detect in the model itself, since the IDE cannot know what the correct definition of an adjacent road is. So, like with any method to verify that a program actually does, what we intend it to do, it is always helpful to have multiple ways to verify the result. All in all TLA+ was quite helpful with defining what data is immutable or mutable, what data can be computed from the state, finding bugs, and what the invariants are, that need to hold throughout a game.

### 7.1.2 Git

Git provides interesting information to further ensure everything is in order. We used Git to record the complete game state in a Git tree as well as the entire game history using a directed acyclic graph of commits with each commit containing the name of the action taken, the player number, and further information (e.g. the dice roll number). This allowed us to gather statistics on all games and check if every action is indeed taken. In our case we could see that the distribution of dice rolls was uniform, which does not reflect two 6-sided dice.

The history is incredibly useful for reproducing bugs. In case we encounter an error, we can just rebase to the last correct commit and rerun the program to re-encounter the bug.

## 7.2 Future Work

Since we only implemented a simplified version of "Catan", an extension would be to allow for trade between players during the trading phase. Even more freedom of

play could be allowed by merging the trading and building phases into a single phase, according to the advanced player rules of "Catan". There are also many add-ons to the basic "Catan" game that could be added to the base game, which would include new buildings, new maps and new mechanics to earn victory points.

Our implementation cannot be played with human players because it is lacking an interface to perform actions and a mechanism to create and join a game. There could be interesting projects in implementing those with augmented reality or projector-based interfaces.

Our TLA+ model could be used to implement "Catan" in another peer-to-peer system and compare it to our baseline implementation. It could also be used as an inspiration to create models for other board games or programs.

## 7.3 Perspectives

TLA+ has the potential to become a very powerful tool in the future. With a few quality of life improvements for the editor and model checker, it could be elevated to the standard of current IDEs and make the mathematics feel more like programming. If we reach the point where programmers think of TLA+ as just another programming language, the barrier to entry would be significantly lower. Furthermore, if there exists a toolbox that is convenient to use and provides clarity for concurrent programs to the users, more people might be willing to try and create concurrent applications or systems, which would further incentivize improvements to TLA+ or even motivate others to create something similar, creating competition in this field. Especially in today's times, where privacy and control over data are becoming more relevant, programmers are incentivized to create applications that protect these values and TLA+ might provide a good entry point for people interested.

Multi-core processors are ubiquitous in current computers and server clusters used for computation are more common than ever. Being able to write error-free applications that can leverage the computation power of multiple machines or cores is hard. Since they need to be concurrent and reasoning about all states that can occur is challenging. However, if we have tools that make it easier to do these challenging tasks without a programmer needing to get a mathematics degree can only improve the landscape of applications that get created in the future or might just help to improve an existing system by refining and validating its correctness.

# Bibliography

- [1] Yves Bertot and Pierre Castéran. “Interactive Theorem Proving and Program Development: Coq’Art: The Calculus of Inductive Constructions”. In: *Textbook*. Springer, 2004. ISBN: 978-3-540-20854-9.
- [2] Eric Brewer. “Towards robust distributed systems”. In: July 2000, p. 7. doi: 10.1145/343477.343502.
- [3] Hemil Desai. *Imperials*. Accessed: May 2025. URL: <https://github.com/hemildesai/imperial>.
- [4] Jannick Heisch. “Delta-GOC-Ledger: Incremental Checkpointing and Lower Message Sizes for Grow-Only Counters Ledgers with Delta-CRDTs”. Archive url: (vpn protected) [https://central.dmi.unibas.ch/data/storage/theses/2024/heisch\\_jannick/Delta-GOC-Ledger - Incremental Checkpointing and Lower Message Sizes for Grow-Only Counters Ledgers with Delta-CRDTs.pdf](https://central.dmi.unibas.ch/data/storage/theses/2024/heisch_jannick/Delta-GOC-Ledger - Incremental Checkpointing and Lower Message Sizes for Grow-Only Counters Ledgers with Delta-CRDTs.pdf). MA thesis. Universität Basel, 2024. URL: [https://web.archive.org/web/20250227032428/https://cn.dmi.unibas.ch/fileadmin/user\\_upload/redesign-cn-dmi/pubs/theses/master/Heisch-Delta-GOC-Ledger.pdf](https://web.archive.org/web/20250227032428/https://cn.dmi.unibas.ch/fileadmin/user_upload/redesign-cn-dmi/pubs/theses/master/Heisch-Delta-GOC-Ledger.pdf).
- [5] 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. Delft, Netherlands: Association for Computing Machinery, 2021, pp. 19–24. ISBN: 9781450381970. doi: 10.1145/3428662.3428794. URL: <https://doi.org/10.1145/3428662.3428794>.
- [6] Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. “The TLA+ Toolbox”. In: *Electronic Proceedings in Theoretical Computer Science* 310 (Dec. 2019), pp. 50–62. ISSN: 2075-2180. doi: 10.4204/eptcs.310.6. URL: <http://dx.doi.org/10.4204/EPTCS.310.6>.
- [7] Leslie Lamport. *A Science of Concurrent Programs*. Final draft available at <https://lamport.azurewebsites.net/tla/science-book.html>. Cambridge University Press, 2024.
- [8] Leslie Lamport. *Computation and State Machines*. <https://www.microsoft.com/en-us/research/publication/computation-state-machines/>. Microsoft Research. 2008.
- [9] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 032114306X.
- [10] Erick Lavoie. *2P-BFT-Log: 2-Phase Single-Author Append-Only Log for Adversarial Environments*. 2023. arXiv: 2307.08381 [cs.DC]. URL: <https://arxiv.org/abs/2307.08381>.
- [11] Erick Lavoie. *GOC-Ledger: State-based Conflict-Free Replicated Ledger from Grow-Only Counters*. 2023. arXiv: 2305.16976 [cs.DC]. URL: <https://arxiv.org/abs/2305.16976>.

- [12] Tim Matter. *P2P Catan Simulation*. 2025. URL: <https://github.com/Matter95/CatanDS>.
- [13] Peter McAughan et al. *QSettlers: Deep Reinforcement Learning for Settlers of Catan*. Accessed: May 2025. URL: <https://akrishna77.github.io/QSettlers/>.
- [14] Stephan Merz. “On the Logic of TLA<sup>+</sup>”. In: *Computers and Informatics* 22 (2003), pp. 351–379.
- [15] Stephan Merz. “The Specification Language TLA<sup>+</sup>”. In: *Logics of Specification Languages*. Ed. by Dines Bjørner and Martin C. Henson. Monographs in Theoretical Computer Science. Berlin-Heidelberg: Springer, 2008, pp. 401–451.
- [16] Jeremy Monin. *Full Java Catan Implementation*. Accessed: May 2025. URL: <https://nand.net/jsettlers/>.
- [17] Chris Newcombe et al. “How Amazon web services uses formal methods”. In: *Commun. ACM* 58.4 (Mar. 2015), pp. 66–73. ISSN: 0001-0782. DOI: 10.1145/2699417. URL: <https://doi.org/10.1145/2699417>.
- [18] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002. ISBN: 978-3-540-43376-7. DOI: 10.1007/3-540-45949-9. URL: <https://link.springer.com/book/10.1007/3-540-45949-9>.
- [19] Ulf Norell. “Dependently Typed Programming in Agda”. In: *Proceedings of the 6th International Conference on Advanced Functional Programming (AFP 2008)*. Vol. 5832. LNCS. Springer, 2009, pp. 230–266. DOI: 10.1007/978-3-642-04652-0\_5. URL: [https://doi.org/10.1007/978-3-642-04652-0\\_5](https://doi.org/10.1007/978-3-642-04652-0_5).
- [20] *p2panda GitHub*. Accessed: May 2025. URL: <https://github.com/p2panda>.
- [21] *P2Panda Specification*. Accessed: May 2025. URL: <https://aquadoggo.p2panda.org/specifications/>.
- [22] Clifford Ressel. *Local Catan Implementation*. Accessed: May 2025. URL: <https://github.com/CSRessel/catan>.
- [23] Ken Rotaris. *Single-Player Chess as a Planning Problem*. Accessed: May 2025. 2022. URL: <https://ai.dmi.unibas.ch/papers/theses/rotaris-bachelor-22.pdf>.
- [24] Nicholas I. Rummelt and Joseph N. Wilson. “Array set addressing: enabling technology for the efficient processing of hexagonally sampled imagery”. In: *Journal of Electronic Imaging* 20.2 (2011), p. 023012. DOI: 10.1117/1.3589306.
- [25] *SSB Specification*. Accessed: May 2025. URL: <https://spec.scuttlebutt.nz/introduction.html>.
- [26] Dominic Tarr et al. “Secure Scuttlebutt: An Identity-Centric Protocol for Subjective and Decentralized Applications”. In: *Proceedings of the 6th ACM Conference on Information-Centric Networking*. ICN '19. Macao, China: Association for Computing Machinery, 2019, pp. 1–11. ISBN: 9781450369701. DOI: 10.1145/3357150.3357396. URL: <https://doi.org/10.1145/3357150.3357396>.
- [27] *The Peg Game TLA<sup>+</sup> Specification*. Accessed: May 2025. URL: <https://identity.pub/2019/05/01/peg-game.html>.
- [28] *Tic Tac Toe TLA<sup>+</sup> Specification*. Accessed: May 2025. URL: [https://www.monkeynut.org/tic-tac-toe/?utm\\_source=chatgpt.com](https://www.monkeynut.org/tic-tac-toe/?utm_source=chatgpt.com).
- [29] *TinySSB Specification*. Accessed: May 2025. URL: <https://github.com/tinySSB/tiny-ssb-spec>.
- [30] *TLA<sup>+</sup> Specification Examples*. Accessed: May 2025. URL: <https://github.com/tlaplus/Examples>.

- [31] *TLA+ Video Series*. Accessed: May 2025. URL: <https://lamport.azurewebsites.net/video/videos.html>.
- [32] Christian Tschudin. *TinySSB GitHub*. Accessed: May 2025. URL: <https://github.com/ssbc/tinySSB>.
- [33] Karan Vombatkere. *Catan AI*. Accessed: May 2025. URL: <https://github.com/kvombatkere/Catan-AI>.
- [34] Severin Wyss. *A Formalism for Build Order Search in StarCraft Brood War*. Accessed: May 2025. 2016. URL: <https://ai.dmi.unibas.ch/papers/theses/wyss-bachelor-16.pdf>.
- [35] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model Checking TLA+ Specifications". In: *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. CHARME '99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 54–66. ISBN: 3540665595.

## Appendix A

# TLA+ Specification and TLC settings

The next pages provide the complete TLA+ specification. To verify the specification with TLC, use the following model properties:

1. **Constants:** *PossiblePlayers*  $\triangleq \{p1, p2, p3, p4\}$  is a set of model values and all other parameters defined with **CONSTANTS** are model values.
2. **Invariants:** *TypeOK*, *ConservationOfResourceCards*, *ConservationOfDevelopmentCards*, *ConservationOfBuildings*
3. **Options:** Simulation mode with a maximum length of 2500

```

1  |----- MODULE Catan -----|
2  EXTENDS Integers, FiniteSets, Sequences, TLC

4  CONSTANT PossiblePlayers,
5          bot,
6          Lumber, Brick, Wool, Grain, Ore,
7          Monopoly, YearOfPlenty, RoadBuilding, Knight, VictoryPoint,
8          ThreeToOne,
9          Road, Village, City,
10         DevCard,
11         PhaseOne, PhaseTwo,
12         DiceRoll, Trading, Building,
13         top

15  Choose an arbitrary order of players
16  chooseSeq(P)  $\triangleq$ 
17    LET RECURSIVE helper(-)
18      helper(S)  $\triangleq$ 
19        IF  $S = \{\}$  THEN  $\langle \rangle$ 
20        ELSE LET  $p \triangleq$  CHOOSE  $s \in S : \text{TRUE}$ 
21              IN
22                 $\langle p \rangle \circ \text{helper}(S \setminus \{p\})$ 
23    IN helper(P)

25  Players of the game
26  Players  $\triangleq$  CHOOSE
27       $s \in \text{SUBSET } \textit{PossiblePlayers} :$ 
28       $\text{Cardinality}(s) \geq 2$ 

29  Player Order
30   $P \triangleq \text{chooseSeq}(\textit{Players})$ 

31  Resource Card Types
32   $RCT \triangleq \{\textit{Lumber, Brick, Wool, Grain, Ore}\}$ 
33   $RCTMAP \triangleq [\textit{Lumber} \mapsto \textit{Lumber}, \textit{Brick} \mapsto \textit{Brick}, \textit{Wool} \mapsto \textit{Wool},$ 
34       $\textit{Grain} \mapsto \textit{Grain}, \textit{Ore} \mapsto \textit{Ore}]$ 
35   $RCTST \triangleq \langle \text{"Lumber", "Brick", "Wool", "Grain", "Ore"} \rangle$ 

36  Progress Card Types
37   $PCT \triangleq \{\textit{Monopoly, YearOfPlenty, RoadBuilding}\}$ 
38   $PCTMAP \triangleq [\textit{Monopoly} \mapsto \text{"Monopoly"}, \textit{YearOfPlenty} \mapsto \text{"YearOfPlenty"},$ 
39       $\textit{RoadBuilding} \mapsto \text{"RoadBuilding"}]$ 
40   $PCTST \triangleq \langle \text{"Monopoly", "YearOfPlenty", "RoadBuilding"} \rangle$ 

41  Development Card Types
42   $DCT \triangleq PCT \cup \{\textit{Knight, VictoryPoint}\}$ 

43  Resource Card Pool
44   $RCP \triangleq \langle 19, 19, 19, 19, 19 \rangle$ 

45  Development Card Pool
46   $DCP \triangleq \langle 2, 2, 2, 14, 5 \rangle$ 

```

The set of players  
 bottom  
 Resource Types  
 Development Cards  
 Wharf Type  
 Building Type  
 Purchasable Item  
 Setup Phases  
 Turn Phases  
 top

```

47   Bought Card Types
48    $BCT \triangleq [Monopoly : 0 \dots 2, YearOfPlenty : 0 \dots 2, RoadBuilding : 0 \dots 2, Knight : 0 \dots 14]$ 
49   Unveiled Card Types
50    $UCT \triangleq [Knight : 0 \dots 14, VictoryPoint : 0 \dots 5]$ 
51   Wharf Types
52    $WT \triangleq \{ThreeToOne\} \cup RCT$ 
53   Map Tiles
54    $MT \triangleq [$ 
55        $coords : [arr : 0 \dots 1, row : Int, col : Int],$ 
56        $res : RCT \cup \{bot\},$ 
57        $n : 2 \dots 6 \cup 8 \dots 12 \cup \{bot\}$ 
58   ]

60   Wharfs
61    $W \triangleq \{$ 
62        $[coords \mapsto \{$ 
63            $[coords \mapsto [arr \mapsto 0, row \mapsto 0, col \mapsto 1],$ 
64            $res \mapsto bot, n \mapsto bot],$ 
65            $[coords \mapsto [arr \mapsto 1, row \mapsto 0, col \mapsto 2],$ 
66            $res \mapsto Ore, n \mapsto 10]$ 
67        $\},$ 
68        $wt \mapsto ThreeToOne$ 
69    $],$ 
70    $[coords \mapsto \{$ 
71        $[coords \mapsto [arr \mapsto 0, row \mapsto 0, col \mapsto 3],$ 
72        $res \mapsto bot, n \mapsto bot],$ 
73        $[coords \mapsto [arr \mapsto 1, row \mapsto 0, col \mapsto 3],$ 
74        $res \mapsto Wool, n \mapsto 2]$ 
75    $\},$ 
76    $wt \mapsto Grain$ 
77    $],$ 
78    $[coords \mapsto \{$ 
79        $[coords \mapsto [arr \mapsto 1, row \mapsto 0, col \mapsto 5],$ 
80        $res \mapsto bot, n \mapsto bot],$ 
81        $[coords \mapsto [arr \mapsto 0, row \mapsto 1, col \mapsto 4],$ 
82        $res \mapsto Brick, n \mapsto 10]$ 
83    $\},$ 
84    $wt \mapsto Ore$ 
85    $],$ 
86    $[coords \mapsto \{$ 
87        $[coords \mapsto [arr \mapsto 0, row \mapsto 1, col \mapsto 0],$ 
88        $res \mapsto bot, n \mapsto bot],$ 
89        $[coords \mapsto [arr \mapsto 0, row \mapsto 1, col \mapsto 1],$ 
90        $res \mapsto Grain, n \mapsto 12]$ 
91    $\},$ 

```

```

92         wt ↦ Lumber
93     ],
94     [coords ↦ {
95         [coords ↦ [arr ↦ 1, row ↦ 1, col ↦ 5],
96         res ↦ Ore, n ↦ 8],
97         [coords ↦ [arr ↦ 1, row ↦ 1, col ↦ 6],
98         res ↦ bot, n ↦ bot]
99     },
100     wt ↦ ThreeToOne
101 ],
102 [coords ↦ {
103     [coords ↦ [arr ↦ 0, row ↦ 2, col ↦ 0],
104     res ↦ bot , n ↦ bot],
105     [coords ↦ [arr ↦ 0, row ↦ 2, col ↦ 1],
106     res ↦ Lumber, n ↦ 8]
107 },
108 wt ↦ Brick
109 ],
110 [coords ↦ {
111     [coords ↦ [arr ↦ 0, row ↦ 2, col ↦ 4],
112     res ↦ Wool, n ↦ 5],
113     [coords ↦ [arr ↦ 1, row ↦ 2, col ↦ 5],
114     res ↦ bot , n ↦ bot]
115 },
116 wt ↦ Wool
117 ],
118 [coords ↦ {
119     [coords ↦ [arr ↦ 1, row ↦ 2, col ↦ 2],
120     res ↦ Brick, n ↦ 5],
121     [coords ↦ [arr ↦ 0, row ↦ 3, col ↦ 1],
122     res ↦ bot , n ↦ bot]
123 },
124 wt ↦ ThreeToOne
125 ],
126 [coords ↦ {
127     [coords ↦ [arr ↦ 1, row ↦ 2, col ↦ 3],
128     res ↦ Grain, n ↦ 6],
129     [coords ↦ [arr ↦ 0, row ↦ 3, col ↦ 3],
130     res ↦ bot , n ↦ bot]
131 },
132 wt ↦ ThreeToOne
133 ]
134 }
135 ST ≜ {Village, City}
136 BP ≜ ⟨15, 5, 4⟩

```

Settlement Types  
Building Pool

137  $SUP \triangleq \{PhaseOne, PhaseTwo\}$  Setup Phases  
138  $TP \triangleq \{bot, DiceRoll, Trading, Building, top\}$  Turn Phases  
139  $M \triangleq \{$  Map  
140     Top Row of Island  
141      $[coords \mapsto [arr \mapsto 0, row \mapsto 0, col \mapsto 1], res \mapsto bot, n \mapsto bot],$   
142      $[coords \mapsto [arr \mapsto 0, row \mapsto 0, col \mapsto 2], res \mapsto bot, n \mapsto bot],$   
143      $[coords \mapsto [arr \mapsto 0, row \mapsto 0, col \mapsto 3], res \mapsto bot, n \mapsto bot],$   
144      $[coords \mapsto [arr \mapsto 0, row \mapsto 0, col \mapsto 4], res \mapsto bot, n \mapsto bot],$   
146     First Row of Island  
147      $[coords \mapsto [arr \mapsto 1, row \mapsto 0, col \mapsto 1], res \mapsto bot, n \mapsto bot],$   
148      $[coords \mapsto [arr \mapsto 1, row \mapsto 0, col \mapsto 2], res \mapsto Ore, n \mapsto 10],$   
149      $[coords \mapsto [arr \mapsto 1, row \mapsto 0, col \mapsto 3], res \mapsto Wool, n \mapsto 2],$   
150      $[coords \mapsto [arr \mapsto 1, row \mapsto 0, col \mapsto 4], res \mapsto Lumber, n \mapsto 9],$   
151      $[coords \mapsto [arr \mapsto 1, row \mapsto 0, col \mapsto 5], res \mapsto bot, n \mapsto bot],$   
153     Second Row of Island  
154      $[coords \mapsto [arr \mapsto 0, row \mapsto 1, col \mapsto 0], res \mapsto bot, n \mapsto bot],$   
155      $[coords \mapsto [arr \mapsto 0, row \mapsto 1, col \mapsto 1], res \mapsto Grain, n \mapsto 12],$   
156      $[coords \mapsto [arr \mapsto 0, row \mapsto 1, col \mapsto 2], res \mapsto Brick, n \mapsto 6],$   
157      $[coords \mapsto [arr \mapsto 0, row \mapsto 1, col \mapsto 3], res \mapsto Wool, n \mapsto 4],$   
158      $[coords \mapsto [arr \mapsto 0, row \mapsto 1, col \mapsto 4], res \mapsto Brick, n \mapsto 10],$   
159      $[coords \mapsto [arr \mapsto 0, row \mapsto 1, col \mapsto 5], res \mapsto bot, n \mapsto bot],$   
161     Third Row of Island  
162      $[coords \mapsto [arr \mapsto 1, row \mapsto 1, col \mapsto 0], res \mapsto bot, n \mapsto bot],$   
163      $[coords \mapsto [arr \mapsto 1, row \mapsto 1, col \mapsto 1], res \mapsto Grain, n \mapsto 9],$   
164      $[coords \mapsto [arr \mapsto 1, row \mapsto 1, col \mapsto 2], res \mapsto Lumber, n \mapsto 11],$   
165      $[coords \mapsto [arr \mapsto 1, row \mapsto 1, col \mapsto 3], res \mapsto bot, n \mapsto bot],$   
166      $[coords \mapsto [arr \mapsto 1, row \mapsto 1, col \mapsto 4], res \mapsto Lumber, n \mapsto 3],$   
167      $[coords \mapsto [arr \mapsto 1, row \mapsto 1, col \mapsto 5], res \mapsto Ore, n \mapsto 8],$   
168      $[coords \mapsto [arr \mapsto 1, row \mapsto 1, col \mapsto 6], res \mapsto bot, n \mapsto bot],$   
170     Fourth Row of Island  
171      $[coords \mapsto [arr \mapsto 0, row \mapsto 2, col \mapsto 0], res \mapsto bot, n \mapsto bot],$   
172      $[coords \mapsto [arr \mapsto 0, row \mapsto 2, col \mapsto 1], res \mapsto Lumber, n \mapsto 8],$   
173      $[coords \mapsto [arr \mapsto 0, row \mapsto 2, col \mapsto 2], res \mapsto Ore, n \mapsto 3],$   
174      $[coords \mapsto [arr \mapsto 0, row \mapsto 2, col \mapsto 3], res \mapsto Grain, n \mapsto 4],$   
175      $[coords \mapsto [arr \mapsto 0, row \mapsto 2, col \mapsto 4], res \mapsto Wool, n \mapsto 5],$   
176      $[coords \mapsto [arr \mapsto 0, row \mapsto 2, col \mapsto 5], res \mapsto bot, n \mapsto bot],$   
178     Fifth Row of Island  
179      $[coords \mapsto [arr \mapsto 1, row \mapsto 2, col \mapsto 1], res \mapsto bot, n \mapsto bot],$   
180      $[coords \mapsto [arr \mapsto 1, row \mapsto 2, col \mapsto 2], res \mapsto Brick, n \mapsto 5],$   
181      $[coords \mapsto [arr \mapsto 1, row \mapsto 2, col \mapsto 3], res \mapsto Grain, n \mapsto 6],$   
182      $[coords \mapsto [arr \mapsto 1, row \mapsto 2, col \mapsto 4], res \mapsto Wool, n \mapsto 11],$

```

183   [coords ↦ [arr ↦ 1, row ↦ 2, col ↦ 5], res ↦ bot, n ↦ bot],
185   Bottom Row of Island
186   [coords ↦ [arr ↦ 0, row ↦ 3, col ↦ 1], res ↦ bot, n ↦ bot],
187   [coords ↦ [arr ↦ 0, row ↦ 3, col ↦ 2], res ↦ bot, n ↦ bot],
188   [coords ↦ [arr ↦ 0, row ↦ 3, col ↦ 3], res ↦ bot, n ↦ bot],
189   [coords ↦ [arr ↦ 0, row ↦ 3, col ↦ 4], res ↦ bot, n ↦ bot],
190   [coords ↦ [arr ↦ 0, row ↦ 3, col ↦ 5], res ↦ bot, n ↦ bot]
191 }

193 |-----|

195 isAdjacent(t1, t2)  $\triangleq$ 
196    $\wedge t1 \in M \wedge t2 \in M$ 
197    $\wedge$  LET
198     arr1  $\triangleq$  t1.coords.arr
199     row1  $\triangleq$  t1.coords.row
200     col1  $\triangleq$  t1.coords.col
201     arr2  $\triangleq$  t2.coords.arr
202     row2  $\triangleq$  t2.coords.row
203     col2  $\triangleq$  t2.coords.col
204     deltas  $\triangleq$ 
205     IF arr1 = 0
206     THEN
207       Top-left, Top-right, Left, Right, Bottom-left, Bottom-right
208       {⟨1, -1, 0⟩, ⟨1, -1, 1⟩, ⟨0, 0, -1⟩, ⟨0, 0, 1⟩, ⟨1, 0, 0⟩, ⟨1, 0, 1⟩}
209     ELSE
210       Top-left, Top-right, Left, Right, Bottom-left, Bottom-right
211       {⟨-1, 0, -1⟩, ⟨-1, 0, 0⟩, ⟨0, 0, -1⟩, ⟨0, 0, 1⟩, ⟨-1, 1, -1⟩, ⟨-1, 1, 0⟩}
212     IN
213      $\exists d \in \text{deltas} : \text{arr2} = \text{arr1} + d[1] \wedge \text{row2} = \text{row1} + d[2] \wedge \text{col2} = \text{col1} + d[3]$ 

215 allAdjacent(t1, t2, t3)  $\triangleq$ 
216    $\wedge t1 \in M \wedge t2 \in M \wedge t3 \in M$ 
217    $\wedge \text{isAdjacent}(t1, t2)$ 
218    $\wedge \text{isAdjacent}(t1, t3)$ 
219    $\wedge \text{isAdjacent}(t2, t3)$ 

221 isCenterTile(t)  $\triangleq$ 
222    $\wedge t \in M$ 
223    $\wedge t.\text{coords} = [\text{arr} \mapsto 1, \text{row} \mapsto 1, \text{col} \mapsto 1]$ 

225 isCoast(t1, t2)  $\triangleq$ 
226    $\wedge t1 \in M \wedge t2 \in M$ 
227    $\wedge \neg \text{isCenterTile}(t1) \wedge \neg \text{isCenterTile}(t2)$ 
228    $\wedge \vee t1.\text{res} = \text{bot} \wedge t2.\text{res} \neq \text{bot}$ 
229    $\vee t1.\text{res} \neq \text{bot} \wedge t2.\text{res} = \text{bot}$ 

```

```

231  All viable coordinate triplets
232   $CT \triangleq \{\{k[1], k[2], k[3]\} : k \in \{t \in M \times M \times M :$ 
233     $allAdjacent(t[1], t[2], t[3]) \wedge (t[1].res \neq bot \vee t[2].res \neq bot \vee t[3].res \neq bot)\}\}$ 
234  All viable coordinate pairs
235   $CD \triangleq \{\{k[1], k[2]\} : k \in \{t \in M \times M :$ 
236     $isAdjacent(t[1], t[2]) \wedge (t[1].res \neq bot \vee t[2].res \neq bot)\}\}$ 

238  ASSUME
239    Map is a subset of all possible map tiles
240     $\wedge M \subseteq MT$ 
241    Map has no duplicate tiles
242     $\wedge \forall t1, t2 \in M : t1.coords = t2.coords \equiv t1 = t2$ 
243    all Wharf tiles are part of the map
244     $\wedge \forall w \in W : w.wt \in WT \wedge \forall t \in w.coords : t \in M$ 
245    All Wharfs are on the coast
246     $\wedge \forall w \in W : \exists t1, t2 \in w.coords : isAdjacent(t1, t2) \wedge isCoast(t1, t2)$ 

248  |-----|

250   $reverse(seq) \triangleq$ 
251    LET RECURSIVE  $helper(-)$ 
252     $helper(s) \triangleq$ 
253      IF  $s = \langle \rangle$  THEN  $s$ 
254      ELSE  $Append(helper(Tail(s)), Head(s))$ 
255    IN  $helper(seq)$ 

257   $getIndex(el, seq) \triangleq$ 
258    LET RECURSIVE  $helper(-)$ 
259     $helper(s) \triangleq$ 
260      IF  $s = el$  THEN 0
261      ELSE IF  $Head(s) = el$ 
262        THEN 1
263      ELSE  $1 + helper(Tail(s))$ 
264    IN  $helper(seq)$ 

266   $SumFunc(func) \triangleq$ 
267    LET RECURSIVE  $helper(-, -)$ 
268     $helper(f, D) \triangleq$ 
269      IF  $D = \{\}$  THEN 0
270      ELSE LET  $d \triangleq$  CHOOSE  $e \in D : TRUE$ 
271        IN  $f[d] + helper(f, D \setminus \{d\})$ 
272    IN  $helper(func, DOMAIN func)$ 

274   $AddRec(rec1, rec2) \triangleq [i \in DOMAIN rec1 \mapsto rec1[i] + rec2[i]]$ 
275   $SubRec(rec1, rec2) \triangleq [i \in DOMAIN rec1 \mapsto rec1[i] - rec2[i]]$ 
276   $AddRecEl(rec1, d, n) \triangleq [i \in DOMAIN rec1 \mapsto IF i = d THEN rec1[i] + n ELSE rec1[i]]$ 

```

```

278 SumResFunc(func)  $\triangleq$ 
279   LET RECURSIVE helper(-, -)
280   helper(f, D)  $\triangleq$ 
281     IF D = {} THEN [Lumber  $\mapsto$  0, Brick  $\mapsto$  0, Wool  $\mapsto$  0, Grain  $\mapsto$  0, Ore  $\mapsto$  0]
282     ELSE LET c  $\triangleq$  CHOOSE c  $\in$  D : TRUE
283     IN
284       IF c.res = Lumber THEN AddRecEl(helper(f, D \ {c}), "Lumber", f[c])
285       ELSE IF c.res = Brick THEN AddRecEl(helper(f, D \ {c}), "Brick", f[c])
286       ELSE IF c.res = Wool THEN AddRecEl(helper(f, D \ {c}), "Wool", f[c])
287       ELSE IF c.res = Grain THEN AddRecEl(helper(f, D \ {c}), "Grain", f[c])
288       ELSE IF c.res = Ore THEN AddRecEl(helper(f, D \ {c}), "Ore", f[c])
289       ELSE AddRecEl(helper(f, D \ {c}), Lumber, 0)
290   IN helper(func, DOMAIN func)

292 SumResRecs(func)  $\triangleq$ 
293   LET RECURSIVE helper(-, -)
294   helper(f, D)  $\triangleq$ 
295     IF D = {} THEN [Lumber  $\mapsto$  0, Brick  $\mapsto$  0, Wool  $\mapsto$  0, Grain  $\mapsto$  0, Ore  $\mapsto$  0]
296     ELSE LET d  $\triangleq$  CHOOSE e  $\in$  D : TRUE
297     IN AddRec(f[d], helper(f, D \ {d}))
298   IN helper(func, DOMAIN func)

300 IterateRec(rec)  $\triangleq$  [d  $\in$  DOMAIN rec  $\mapsto$  (rec)[d]]

302 |-----|

304 VARIABLES SU, Setup State
305           G      Game State
306   Bank
307   BNK  $\triangleq$ 
308   [
309     RC : [Lumber : 0 .. 19, Brick : 0 .. 19, Wool : 0 .. 19,
310           Grain : 0 .. 19, Ore : 0 .. 19],
311     DC : [Monopoly : 0 .. 2, YearOfPlenty : 0 .. 2, RoadBuilding : 0 .. 2,
312           Knight : 0 .. 14, VictoryPoint : 0 .. 5]
313   ]
314   Discard Pile
315   DP  $\triangleq$  [Monopoly : 0 .. 2, YearOfPlenty : 0 .. 2, RoadBuilding : 0 .. 2]
316   Player Hand
317   PH  $\triangleq$  [
318     RC : [Lumber : 0 .. 19, Brick : 0 .. 19, Wool : 0 .. 19, Grain : 0 .. 19, Ore : 0 .. 19],
319     DC : [BC : BCT,
320           AC : BCT,
321           UC : UCT
322     ]
323   ]

```

324 **Hands**  
 325  $H \triangleq [Players \rightarrow PH]$   
 326 **Player Buildings**  
 327  $PB \triangleq [Road : 0 \dots 15, Village : 0 \dots 5, City : 0 \dots 4]$   
 328 **Buildings**  
 329  $B \triangleq [Players \rightarrow PB]$   
 330 **Settlement Points**  
 331  $S \triangleq [CT \rightarrow [own : Players \cup \{bot\}, st : ST \cup \{bot\}]]$   
 332 **Road Points**  
 333  $R \triangleq [CD \rightarrow [own : Players \cup \{bot\}]]$   
 335  $TypeOK \triangleq$   
 336 **Setup State**  
 337  $\wedge SU \in [sup : SUP, ap : Players]$   
 338 **Game State**  
 339  $\wedge G \in [ap : Players, bnk : BNK, dp : DP, h : H, b : B, s : S, r : R, ba : M, tp : TP]$   
 341 |-----|  
 343  $isAdjacentSettlement(s1, s2) \triangleq$   
 344  $\wedge s1 \in \text{DOMAIN } G.s$   
 345  $\wedge s2 \in \text{DOMAIN } G.s$   
 346  $\wedge \exists t1, t2 \in s1 : t1 \neq t2 \wedge t1 \in s2 \wedge t2 \in s2$   
 348  $isAdjacentRoadToSettlement(s, r) \triangleq$   
 349  $\wedge s \in \text{DOMAIN } G.s$   
 350  $\wedge r \in \text{DOMAIN } G.r$   
 351  $\wedge r \subseteq s$   
 353  $isAdjacentRoad(r1, r2) \triangleq$   
 354  $\wedge r1 \in \text{DOMAIN } G.r$   
 355  $\wedge r2 \in \text{DOMAIN } G.r$   
 356  $\wedge \exists s \in \text{DOMAIN } G.s :$   
 357  $\wedge isAdjacentRoadToSettlement(s, r1)$   
 358  $\wedge isAdjacentRoadToSettlement(s, r2)$   
 360  $settlementHasNoBandit(s) \triangleq$   
 361  $\wedge s \in \text{DOMAIN } G.s$   
 362  $\wedge \neg G.ba \in s$   
 364  $roadHasNoBandit(r) \triangleq$   
 365  $\wedge r \in \text{DOMAIN } G.r$   
 366  $\wedge \neg G.ba \in r$   
 368  $tileHasNoBandit(t) \triangleq$   
 369  $\wedge t \in M$   
 370  $\wedge \neg G.ba \neq t$

```

372  $buildable(s) \triangleq$ 
373    $\wedge s \in \text{DOMAIN } G.s$ 
374    $\wedge settlementHasNoBandit(s)$ 
375    $\wedge \forall as \in \text{DOMAIN } G.s :$ 
376      $isAdjacentSettlement(s, as) \Rightarrow G.s[as].own = bot$ 
378 |-----|
380  $SetupPhaseOne \triangleq$ 
381    $\wedge G.tp = bot$ 
382    $\wedge SU.sup = PhaseOne$ 
383   choose where to settle with a village
384    $\wedge \exists sp \in \text{DOMAIN } G.s : G.s[sp].own = bot \wedge buildable(sp)$ 
385   choose a road adjacent to the chosen settlement
386    $\wedge \exists rp \in \text{DOMAIN } G.r :$ 
387      $\wedge G.r[rp].own = bot$ 
388      $\wedge isAdjacentRoadToSettlement(sp, rp)$ 
389      $\wedge roadHasNoBandit(rp)$ 
391    $\wedge G' = [G \text{ EXCEPT } !.b[ SU.ap ].Road = @ - 1,$ 
392      $!.b[ SU.ap ].Village = @ - 1,$ 
393      $!.s[sp] = [own \mapsto SU.ap, st \mapsto Village],$ 
394      $!.r[rp] = [own \mapsto SU.ap]$ 
395   ]
396    $\wedge \text{IF } getIndex(SU.ap, P) = Cardinality(Players)$ 
397     THEN
398        $SU' = [SU \text{ EXCEPT } !.ap = reverse(P)[1],$ 
399          $!.sup = PhaseTwo$ 
400     ]
401     ELSE
402        $SU' = [SU \text{ EXCEPT } !.ap = P[(getIndex(SU.ap, P) + 1)]]$ 
404  $SetupPhaseTwo \triangleq$ 
405    $\wedge G.tp = bot$ 
406    $\wedge SU.sup = PhaseTwo$ 
407   choose where to settle with the second village
408    $\wedge \exists sp \in \text{DOMAIN } G.s :$ 
409      $\wedge G.s[sp].own = bot$ 
410      $\wedge buildable(sp)$ 
411   choose a road adjacent to the chosen settlement
412    $\wedge \exists rp \in \text{DOMAIN } G.r :$ 
413      $\wedge G.r[rp].own = bot$ 
414      $\wedge isAdjacentRoadToSettlement(sp, rp)$ 
415      $\wedge roadHasNoBandit(rp)$ 
416    $\wedge SU' = [SU \text{ EXCEPT } !.ap = \text{IF } getIndex(SU.ap, reverse(P)) = Cardinality(Players) \text{ THEN } @$ 
417     ELSE  $reverse(P)[(getIndex(SU.ap, reverse(P)) + 1)]]$ 

```

```

418 receive resources from adjacent tiles to chosen settlement
419  $\wedge$  LET  $gain \triangleq SumResFunc([c \in \{s \in sp : s.res \neq bot\} \mapsto 1])$ 
420 IN  $G' = [G$  EXCEPT  $!.h[SU.ap].RC = gain,$ 
421  $!.bnk.RC = SubRec(@, gain),$ 
422  $!.b[SU.ap].Road = @ - 1,$ 
423  $!.b[SU.ap].Village = @ - 1,$ 
424  $!.s[sp] = [own \mapsto SU.ap, st \mapsto Village],$ 
425  $!.r[rp] = [own \mapsto SU.ap],$ 
426  $!.tp = \text{IF } getIndex(SU.ap, reverse(P)) = Cardinality(Players)$ 
427  $\text{ THEN } DiceRoll$ 
428  $\text{ ELSE } bot$ 
429 ]

431 |-----|

433  $hasSettlementOnTile(t, p) \triangleq$ 
434  $\wedge \exists s \in \text{DOMAIN } G.s : G.s[s].own = p$ 
435  $\wedge t \in s$ 

437  $BanditMoves \triangleq \{t \in M : t \neq G.ba \wedge t.res = bot \Rightarrow isCenterTile(t)\}$ 

439  $SumResourcesSinglePlayer(p) \triangleq SumFunc(IterateRec(G.h[p].RC))$ 

441  $SumResourceAllPlayers(rct) \triangleq SumFunc([p \in Players \mapsto (G.h[p].RC)[rct]])$ 

443  $ResourceGainPlayer(p, d) \triangleq$ 
444 LET
445  $V \triangleq \{s \in \text{UNION } \{sp \in \text{DOMAIN } G.s : G.s[sp].own = p \wedge G.s[sp].st = Village\} : s.n = d\}$ 
446  $C \triangleq \{s \in \text{UNION } \{sp \in \text{DOMAIN } G.s : G.s[sp].own = p \wedge G.s[sp].st = City\} : s.n = d\}$ 
447 IN
448  $SumResFunc([c \in (V \cup C) \mapsto \text{IF } c \in C \text{ THEN } 2 \text{ ELSE } 1])$ 

450 gather all resource records with a sum between 4 and 7
451  $ResourceRecsWithSpecificSum \triangleq \{rec \in [Lumber : 0 \dots 6, Brick : 0 \dots 6, Wool : 0 \dots 6,$ 
452  $Grain : 0 \dots 6, Ore : 0 \dots 6] :$ 
453  $\wedge SumFunc(IterateRec(rec)) \geq 4$ 
454  $\wedge SumFunc(IterateRec(rec)) \leq 7$ 
455  $\}$ 

457 |-----|

459  $DiceRollPhase \triangleq$ 
460  $\wedge G.tp = DiceRoll$ 
461  $\wedge \exists d \in 2 \dots 12 :$ 
462 IF  $d = 7$ 
463 THEN  $\exists loss \in [Players \rightarrow ResourceRecsWithSpecificSum] :$ 
464  $\wedge \forall p \in Players :$ 
465 IF  $SumResourcesSinglePlayer(p) \leq 7$ 

```

```

466 THEN  $loss[p] = [Lumber \mapsto 0, Brick \mapsto 0, Wool \mapsto 0,$ 
467  $Grain \mapsto 0, Ore \mapsto 0]$ 
468 ELSE
469  $\wedge SumFunc(IterateRec(loss[p])) = SumResourcesSinglePlayer(p) \div 2$ 
470  $\wedge \forall f \in DOMAIN\ G.h[p].RC :$ 
471  $loss[p][f] \in 0 \dots G.h[p].RC[f]$ 
472 choose a tile to move the bandit to
473  $\wedge \exists t \in BanditMoves : \exists q \in Players : q \neq G.ap \wedge hasSettlementOnTile(t, q)$ 
474 random resource from adjacent player of  $t$ 
475  $\wedge \exists res \in DOMAIN\ G.bnk.RC : G.h[q].RC[res] \geq 0$ 
476  $\wedge G' = [G\ EXCEPT\ !.bnk.RC = AddRec(@, SumResRecs(loss)),$ 
477  $!.h = [p \in Players \mapsto$ 
478  $Can\ steal\ resource\ from\ player\ (even\ after\ losing\ resources)$ 
479  $[RC \mapsto IF\ G.h[q].RC[res] - loss[q][res] > 0$ 
480  $THEN\ IF\ p = G.ap\ THEN\ AddRecEl($ 
481  $SubRec(G.h[p].RC, loss[p]),$ 
482  $res,$ 
483  $1)$ 
484  $ELSE\ IF\ p = q\ THEN\ AddRecEl($ 
485  $SubRec(G.h[p].RC, loss[p]),$ 
486  $res,$ 
487  $- 1)$ 
488  $ELSE\ SubRec(G.h[p].RC, loss[p])$ 
489  $ELSE\ SubRec(G.h[p].RC, loss[p]),$ 
490  $Move\ bought\ cards\ to\ available\ cards$ 
491  $DC \mapsto [BC \mapsto [Monopoly \mapsto 0, YearOfPlenty \mapsto 0,$ 
492  $RoadBuilding \mapsto 0, Knight \mapsto 0],$ 
493  $AC \mapsto AddRec(G.h[p].DC.AC, G.h[p].DC.BC),$ 
494  $UC \mapsto G.h[p].DC.UC]$ 
495  $]],$ 
496  $!.ba = t,$ 
497  $!.tp = Trading$ 
498  $]$ 
499  $\wedge UNCHANGED\ (SU)$ 
500 ELSE
501 LET
502  $resource\ gain\ for\ all\ players\ given\ the\ current\ dice\ roll$ 
503  $gain \triangleq [p \in Players \mapsto ResourceGainPlayer(p, d)]$ 
504 IN
505  $if\ bank\ does\ not\ have\ all\ the\ resources,\ do\ not\ change\ state\ and\ continue\ (otherwise\ will\ violate\ typeOK)$ 
506  $IF\ \forall res \in DOMAIN\ G.bnk.RC : G.bnk.RC[res] \geq SumResRecs(gain)[res]$ 
507 THEN
508  $\wedge G' = [G\ EXCEPT\ !.bnk.RC = SubRec(@, SumResRecs(gain)),$ 
509  $!.h = [p \in Players \mapsto$ 
510  $[RC \mapsto AddRec(G.h[p].RC, gain[p]),$ 

```

```

511                                     Move bought cards to available cards
512                                      $DC \mapsto [BC \mapsto [Monopoly \mapsto 0, YearOfPlenty \mapsto 0,$ 
513                                          $RoadBuilding \mapsto 0, Knight \mapsto 0],$ 
514                                          $AC \mapsto AddRec(G.h[p].DC.AC, G.h[p].DC.BC),$ 
515                                          $UC \mapsto G.h[p].DC.UC$ 
516                                     ]
517                                     ]
518                                     ],
519                                      $!.tp = Trading$ 
520                                 ]
521                                 $\wedge \text{UNCHANGED } (SU)$ 
522                                 $\text{ELSE UNCHANGED } \langle SU, G \rangle$ 

524 |-----|

526  $PlayerPorts(p) \triangleq \{w \in W : \exists sp \in \text{DOMAIN } G.s : w.coords \subseteq sp \wedge G.s[sp].own = p\}$ 

528 |-----|

530  $EmptyTrade \triangleq$ 
531    $\wedge G.tp = Trading$ 
532    $\wedge G' = [G \text{ EXCEPT } !.tp = Building]$ 
533    $\wedge \text{UNCHANGED } (SU)$ 

535  $TradeFourToOne \triangleq$ 
536    $\wedge G.tp = Trading$ 
537   choose a resource to trade with
538    $\wedge \exists give \in \{res \in \text{DOMAIN } G.bnk.RC : G.h[G.ap].RC[res] \geq 4\} :$ 
539   choose a resource to trade for
540    $\exists receive \in \{res \in \text{DOMAIN } G.bnk.RC : G.bnk.RC[res] > 0\} :$ 
541    $\wedge G' = [G \text{ EXCEPT } !.bnk.RC = AddRecEl(AddRecEl(@, give, 4), receive, -1),$ 
542    $!.h[G.ap].RC = AddRecEl(AddRecEl(@, give, -4), receive, 1)$ 
543   ]
544    $\wedge \text{UNCHANGED } (SU)$ 

546  $TradeThreeToOne \triangleq$ 
547    $\wedge G.tp = Trading$ 
548   choose a resource to trade with (needs 3:1 port)
549    $\wedge \exists give \in \{res \in \text{DOMAIN } G.bnk.RC :$ 
550    $\wedge G.h[G.ap].RC[res] \geq 3$ 
551    $\wedge \exists w \in PlayerPorts(G.ap) : w.wt = ThreeToOne\} :$ 
552   choose a resource to trade for
553    $\wedge \exists receive \in \{res \in \text{DOMAIN } G.bnk.RC : G.bnk.RC[res] > 0\} :$ 
554    $\wedge G' = [G \text{ EXCEPT } !.bnk.RC = AddRecEl(AddRecEl(@, give, 3), receive, -1),$ 
555    $!.h[G.ap].RC = AddRecEl(AddRecEl(@, give, -3), receive, 1)$ 
556   ]
557    $\wedge \text{UNCHANGED } (SU)$ 

```

558  $\text{TradeTwoToOne} \triangleq$   
559  $\wedge G.tp = \text{Trading}$   
560  $\text{choose a resource to trade with (needs specific 2:1 port)}$   
561  $\wedge \exists \text{give} \in \{rct \in \text{DOMAIN } G.bnk.RC :$   
562  $\wedge G.h[G.ap].RC[rct] \geq 2$   
563  $\wedge \exists w \in \text{PlayerPorts}(G.ap) :$   
564  $\wedge w.wt \in RCT$   
565  $\wedge w.wt = RCTMAP[rct]\} :$   
566  $\text{choose a resource to trade for}$   
567  $\wedge \exists \text{receive} \in \{rct \in \text{DOMAIN } G.bnk.RC : G.bnk.RC[rct] > 0\} :$   
568  $\wedge G' = [G \text{ EXCEPT } !.bnk.RC = \text{AddRecEl}(\text{AddRecEl}(@, \text{give}, 2), \text{receive}, -1),$   
569  $\quad \quad \quad !.h[G.ap].RC = \text{AddRecEl}(\text{AddRecEl}(@, \text{give}, -2), \text{receive}, 1)$   
570  $\quad \quad \quad ]$   
571  $\wedge \text{UNCHANGED } (SU)$

---

573 |

575  $\text{RoadCost} \triangleq [Lumber \mapsto 1, Brick \mapsto 1, Wool \mapsto 0, Grain \mapsto 0, Ore \mapsto 0]$   
576  $\text{VillageCost} \triangleq [Lumber \mapsto 1, Brick \mapsto 1, Wool \mapsto 1, Grain \mapsto 1, Ore \mapsto 0]$   
577  $\text{CityCost} \triangleq [Lumber \mapsto 0, Brick \mapsto 0, Wool \mapsto 0, Grain \mapsto 2, Ore \mapsto 3]$   
578  $\text{DevCardCost} \triangleq [Lumber \mapsto 0, Brick \mapsto 0, Wool \mapsto 1, Grain \mapsto 1, Ore \mapsto 1]$

580  $\text{CanBuildRoad} \triangleq$   
581  $\text{Player has the needed resources and game pieces}$   
582  $\wedge \forall res \in \text{DOMAIN } \text{RoadCost} : G.h[G.ap].RC[res] \geq \text{RoadCost}[res]$   
583  $\wedge G.b[G.ap].Road > 0$

585  $\text{CanBuildVillage} \triangleq$   
586  $\text{Player has the needed resources and game pieces}$   
587  $\wedge \forall res \in \text{DOMAIN } \text{VillageCost} : G.h[G.ap].RC[res] \geq \text{VillageCost}[res]$   
588  $\wedge G.b[G.ap].Village > 0$

590  $\text{CanBuildCity} \triangleq$   
591  $\text{Player has the needed resources and game pieces}$   
592  $\wedge \forall res \in \text{DOMAIN } \text{CityCost} : G.h[G.ap].RC[res] \geq \text{CityCost}[res]$   
593  $\wedge G.b[G.ap].City > 0$

595  $\text{CanBuyDevCard} \triangleq$   
596  $\text{Player has the needed resources and bank still has cards}$   
597  $\wedge \forall res \in \text{DOMAIN } \text{DevCardCost} : G.h[G.ap].RC[res] \geq \text{DevCardCost}[res]$   
598  $\wedge \text{SumFunc}(\text{IterateRec}(G.bnk.DC)) > 0$

600  $\text{RoadsOnMap} \triangleq \{rp \in \text{DOMAIN } G.r : G.r[rp].own \neq bot\}$   
601  $\text{PlayerRoadsOnMap}(p) \triangleq \{rp \in \text{DOMAIN } G.r : G.r[rp].own = p\}$   
602  $\text{AllRoadSets} \triangleq [p \in \text{Players} \mapsto \{s \in \text{SUBSET } \text{PlayerRoadsOnMap}(p) : \text{Cardinality}(s) \geq 5\}]$   
603  $\text{isPath}(seq) \triangleq$   
604  $\wedge \forall i \in 1 \dots (\text{Len}(seq) - 1) :$

```

605      $\wedge seq[i] \in RoadsOnMap$ 
606      $\wedge seq[i + 1] \in RoadsOnMap$ 
607      $\wedge isAdjacentRoad(seq[i], seq[i + 1])$ 
608      $\wedge \forall i, j \in DOMAIN\ seq : i \neq j \Rightarrow seq[i] \neq seq[j] \wedge G.r[seq[i]].own = G.r[seq[j]].own$ 

610  $Max(set) \triangleq \text{CHOOSE } el \in set : \forall n \in set : el \geq n$ 

612  $AllPaths \triangleq [p \in Players \mapsto \{Cardinality(r) :$ 
613      $r \in \{rt \in AllRoadSets[p] :$ 
614      $rt \neq \{\} \wedge isPath(chooseSeq(rt))\}]$ 

616  $PlayerPoints(p) \triangleq$ 
617     LET
618         All Villages and Cities owned by this player
619          $V \triangleq \{sp \in DOMAIN\ G.s : G.s[sp].own = p \wedge G.s[sp].st = Village\}$ 
620          $C \triangleq \{sp \in DOMAIN\ G.s : G.s[sp].own = p \wedge G.s[sp].st = City\}$ 
621         Player with the most Knights (threshold at least 3) gets 2 points
622          $MightiestArmy \triangleq$  IF
623              $\wedge G.h[p].DC.UC.Knight \geq 3$ 
624              $\wedge \forall q \in Players :$ 
625                  $\wedge q \neq p$ 
626                  $\wedge G.h[p].DC.UC.Knight \geq G.h[q].DC.UC.Knight$ 
627             THEN 2
628             ELSE 0
629         Player with the longest road (threshold at least 5) gets 2 points
630          $LongestRoad \triangleq$  IF
631              $\wedge Cardinality(AllPaths[G.ap]) > 0$ 
632              $\wedge Max(AllPaths[G.ap]) > 5$ 
633              $\wedge \forall q \in Players :$ 
634                  $\wedge q \neq p$ 
635                  $\wedge Max(AllPaths[G.ap]) \geq Max(AllPaths[q])$ 
636             THEN 2
637             ELSE 0
638     IN
639      $SumFunc([c \in (V \cup C) \mapsto \text{IF } c \in C \text{ THEN } 2 \text{ ELSE } 1]) +$ 
640      $G.h[p].DC.UC.VictoryPoint +$ 
641      $MightiestArmy +$ 
642      $LongestRoad$ 

644 |-----|

646  $EmptyBuild \triangleq$ 
647      $\wedge G.tp = Building$ 
648     check if a player has enough points to win the game, if not continue
649      $\wedge G' = [G \text{ EXCEPT } !.tp = \text{IF } PlayerPoints(G.ap) \geq 10 \text{ THEN } top \text{ ELSE } DiceRoll]$ 
650      $\wedge \text{UNCHANGED } (SU)$ 

```

```

652 BuildRoad  $\triangleq$ 
653    $\wedge G.tp = \textit{Building}$ 
654    $\wedge \textit{CanBuildRoad}$ 
655   choose a road
656    $\wedge \exists rp \in \text{DOMAIN } G.r : G.r[rp].\textit{own} = \textit{bot} \wedge \textit{roadHasNoBandit}(rp) \wedge$ 
657     either there is an adjacent road or settlement, owned by the player
658      $(\exists rpt \in \text{DOMAIN } G.r : (\textit{isAdjacentRoad}(rp, rpt) \wedge G.r[rpt].\textit{own} = G.ap) \vee$ 
659        $\exists sp \in \text{DOMAIN } G.s : G.s[sp].\textit{own} = G.ap \wedge \textit{isAdjacentRoadToSettlement}(sp, rp))$ 
660      $\wedge G' = [G \text{ EXCEPT } !.bnk.RC = \textit{AddRec}(@, \textit{RoadCost}),$ 
661        $!.h[G.ap].RC = \textit{SubRec}(@, \textit{RoadCost}),$ 
662        $!.b[G.ap].\textit{Road} = @ - 1,$ 
663        $!.r[rp] = [\textit{own} \mapsto G.ap]$ 
664     ]
665      $\wedge \text{UNCHANGED } (SU)$ 

667 BuildVillage  $\triangleq$ 
668    $\wedge G.tp = \textit{Building}$ 
669    $\wedge \textit{CanBuildVillage}$ 
670   choose a space for a village
671    $\wedge \exists sp \in \text{DOMAIN } G.s : G.s[sp].\textit{own} = \textit{bot} \wedge \textit{buildable}(sp) \wedge$ 
672     there is an adjacent road owned by the player
673      $\exists rp \in \text{DOMAIN } G.r : G.r[rp].\textit{own} = G.ap \wedge \textit{isAdjacentRoadToSettlement}(sp, rp)$ 
674      $\wedge G' = [G \text{ EXCEPT } !.bnk.RC = \textit{AddRec}(@, \textit{VillageCost}),$ 
675        $!.h[G.ap].RC = \textit{SubRec}(@, \textit{VillageCost}),$ 
676        $!.b[G.ap].\textit{Village} = @ - 1,$ 
677        $!.s[sp] = [\textit{own} \mapsto G.ap, \textit{st} \mapsto \textit{Village}]$ 
678     ]
679      $\wedge \text{UNCHANGED } (SU)$ 

681 BuildCity  $\triangleq$ 
682    $\wedge G.tp = \textit{Building}$ 
683    $\wedge \textit{CanBuildCity}$ 
684   choose a village to upgrade
685    $\wedge \exists sp \in \text{DOMAIN } G.s : G.s[sp].\textit{own} = G.ap \wedge G.s[sp].\textit{st} = \textit{Village} \wedge$ 
686      $\textit{settlementHasNoBandit}(sp)$ 
687      $\wedge G' = [G \text{ EXCEPT } !.bnk.RC = \textit{AddRec}(@, \textit{CityCost}),$ 
688        $!.h[G.ap].RC = \textit{SubRec}(@, \textit{CityCost}),$ 
689        $!.b[G.ap].\textit{Village} = @ + 1,$ 
690        $!.b[G.ap].\textit{City} = @ - 1,$ 
691        $!.s[sp] = [\textit{own} \mapsto G.ap, \textit{st} \mapsto \textit{City}]$ 
692     ]
693      $\wedge \text{UNCHANGED } (SU)$ 

695 BuyDevCard  $\triangleq$ 
696    $\wedge G.tp = \textit{Building}$ 
697    $\wedge \textit{CanBuyDevCard}$ 

```

```

698 randomly get a development card, Victory Points are instantly revealed
699  $\wedge \exists dc \in \text{DOMAIN } G.bnk.DC : G.bnk.DC[dc] > 0$ 
700  $\wedge G' = [G \text{ EXCEPT } !.bnk.RC = \text{AddRec}(@, \text{DevCardCost}),$ 
701  $!.h[G.ap].RC = \text{SubRec}(@, \text{DevCardCost}),$ 
702  $!.bnk.DC = \text{AddRecEl}(@, dc, -1),$ 
703  $!.h[G.ap].DC = \text{IF } dc = \text{"VictoryPoint"}$ 
704  $\text{ THEN } [BC \mapsto @.BC,$ 
705  $AC \mapsto @.AC,$ 
706  $UC \mapsto \text{AddRecEl}(@.UC, dc, 1)$ 
707  $]$ 
708  $\text{ ELSE } [BC \mapsto \text{AddRecEl}(@.BC, dc, 1),$ 
709  $AC \mapsto @.AC,$ 
710  $UC \mapsto @.UC$ 
711  $]$ 
712  $]$ 
713  $\wedge \text{UNCHANGED } (SU)$ 

```

---

```

717 Monopoly card steals all cards of a chosen resource in the hand of other players.
718 The card is put into the discard pile after playing.
719  $\text{PlayMonopoly} \triangleq$ 
720  $\wedge \exists res \in \text{DOMAIN } G.bnk.RC :$ 
721  $\text{ LET } gain \triangleq \text{SumResourceAllPlayers}(res)$ 
722  $\text{ IN}$ 
723  $G' = [G \text{ EXCEPT } !.dp.Monopoly = @ + 1,$ 
724  $!.h = [p \in \text{Players} \mapsto$ 
725  $\text{ Monopolize one resource from all players}$ 
726  $[RC \mapsto \text{IF } p = G.ap \text{ THEN } \text{AddRecEl}(\$ 
727  $G.h[p].RC,$ 
728  $res,$ 
729  $(gain - G.h[p].RC[res]))$ 
730  $\text{ ELSE } \text{AddRecEl}(\$ 
731  $G.h[p].RC,$ 
732  $res,$ 
733  $- G.h[p].RC[res]),$ 
734  $\text{ Move bought cards to available cards}$ 
735  $DC \mapsto \text{IF } p = G.ap$ 
736  $\text{ THEN } [BC \mapsto G.h[p].DC.BC,$ 
737  $AC \mapsto \text{AddRecEl}(\$ 
738  $G.h[p].DC.AC,$ 
739  $\text{"Monopoly"},$ 
740  $-1),$ 
741  $UC \mapsto G.h[p].DC.UC$ 
742  $]$ 

```

743 ELSE  $G.h[p].DC$   
 744 ]  
 745 ]  
 746 ]  
 747  $\wedge \text{UNCHANGED } (SU)$

749 Year of plenty lets the player choose two resource cards to receive. Put onto the discard pile after playing.  
 750  $\text{PlayYearOfPlenty} \triangleq$   
 751  $\exists res1 \in \text{DOMAIN } G.bnk.RC : G.bnk.RC[res1] > 0 \wedge$   
 752  $\exists res2 \in \text{DOMAIN } G.bnk.RC : G.bnk.RC[res2] > 0$   
 753  $\wedge G' = [G \text{ EXCEPT } !.bnk.RC = \text{AddRecEl}(\text{AddRecEl}(@, res1, -1), res2, -1),$   
 754  $!.h[G.ap].RC = \text{AddRecEl}(\text{AddRecEl}(@, res1, 1), res2, 1),$   
 755  $!.h[G.ap].DC.AC.YearOfPlenty = @ - 1,$   
 756  $!.dp.YearOfPlenty = @ + 1$   
 757 ]  
 758  $\wedge \text{UNCHANGED } (SU)$

760 The player can place two roads onto the map without paying. Put onto the discard pile after playing.  
 761  $\text{PlayRoadBuilding} \triangleq$   
 762  $\wedge G.b[G.ap].Road \geq 2$   
 763  $\wedge \exists rp1 \in \text{DOMAIN } G.r : G.r[rp1].own = bot \wedge \text{roadHasNoBandit}(rp1) \wedge$   
 764  $\text{Adjacent to a road the player owns}$   
 765  $(\exists rpt \in \text{DOMAIN } G.r : (\text{isAdjacentRoad}(rp1, rpt) \wedge G.r[rpt].own = G.ap) \vee$   
 766  $\text{Adjacent to a settlement the player owns}$   
 767  $\exists sp \in \text{DOMAIN } G.s : G.s[sp].own = G.ap \wedge \text{isAdjacentRoadToSettlement}(sp, rp1))$   
 768  $\wedge \exists rp2 \in \text{DOMAIN } G.r : G.r[rp2].own = bot \wedge rp1 \neq rp2 \wedge \text{roadHasNoBandit}(rp2) \wedge$   
 769  $\text{Adjacent to the newly built road}$   
 770  $(\text{isAdjacentRoad}(rp1, rp2) \vee$   
 771  $\text{Adjacent to a road the player owns}$   
 772  $\exists rpt \in \text{DOMAIN } G.r : (\text{isAdjacentRoad}(rp2, rpt) \wedge G.r[rpt].own = G.ap) \vee$   
 773  $\text{Adjacent to a settlement the player owns}$   
 774  $\exists sp \in \text{DOMAIN } G.s : G.s[sp].own = G.ap \wedge \text{isAdjacentRoadToSettlement}(sp, rp2))$   
 775  $\wedge G' = [G \text{ EXCEPT } !.h[G.ap].DC.AC.RoadBuilding = @ - 1,$   
 776  $!.dp.RoadBuilding = @ + 1,$   
 777  $!.b[G.ap].Road = @ - 2,$   
 778  $!.r[rp1] = [own \mapsto G.ap],$   
 779  $!.r[rp2] = [own \mapsto G.ap]$   
 780 ]  
 781  $\wedge \text{UNCHANGED } (SU)$

783 Move the bandit and steal a resource if another player is adjacent to the robbers field.  
 784 Knight cards are unveiled after activation and work towards the mightiest army points.  
 785  $\text{PlayKnight} \triangleq$   
 786  $\exists t \in \text{BanditMoves} : \exists q \in \text{Players} : q \neq G.ap \wedge \text{hasSettlementOnTile}(t, q) \wedge$   
 787  $\text{random resource from adjacent player of } t$   
 788  $\exists res \in \text{DOMAIN } G.bnk.RC : G.h[q].RC[res] \geq 0 \wedge$

```

789   Can steal resource from player
790   IF  $G.h[q].RC[res] > 0$ 
791   THEN
792      $\wedge G' = [G \text{ EXCEPT } !.h[G.ap].RC = \text{AddRecEl}(G.h[G.ap].RC, res, 1),$ 
793      $!.h[q].RC = \text{AddRecEl}(G.h[q].RC, res, -1),$ 
794      $!.h[G.ap].DC.AC.Knight = @ - 1,$ 
795      $!.h[G.ap].DC.UC.Knight = @ + 1,$ 
796      $!.ba = t$ 
797   ]
798    $\wedge \text{UNCHANGED}(SU)$ 
799   ELSE
800      $\wedge G' = [G \text{ EXCEPT } !.h[G.ap].DC.AC.Knight = @ - 1,$ 
801      $!.h[G.ap].DC.UC.Knight = @ + 1,$ 
802      $!.ba = t$ 
803   ]
804    $\wedge \text{UNCHANGED}(SU)$ 

806   Play a card in the Available Card set of a players hand
807    $\text{PlayDevCard} \triangleq$ 
808    $\wedge G.tp \in \{\text{Trading}, \text{Building}\}$ 
809    $\wedge \exists dc \in \text{DOMAIN } G.h[G.ap].DC.AC : G.h[G.ap].DC.AC[dc] > 0 \wedge$ 
810   IF  $dc = \text{"Monopoly"}$  THEN  $\text{UNCHANGED} \langle G, SU \rangle \text{ PlayMonopoly}$ 
811   ELSE IF  $dc = \text{"YearOfPlenty"}$  THEN  $\text{UNCHANGED} \langle G, SU \rangle \text{ PlayYearOfPlenty}$ 
812   ELSE IF  $dc = \text{"RoadBuilding"}$  THEN  $\text{UNCHANGED} \langle G, SU \rangle \text{ PlayRoadBuilding}$ 
813   ELSE  $\text{UNCHANGED} \langle G, SU \rangle \text{ PlayKnight}$ 

815 |-----|

817    $\text{NrPlayerRoadsOnMap}(p) \triangleq$ 
818    $\text{SumFunc}([d \in \text{DOMAIN } G.r \mapsto \text{IF } G.r[d].\text{own} = p \text{ THEN } 1 \text{ ELSE } 0])$ 

820    $\text{NrPlayerSettlementOnMap}(p, st) \triangleq$ 
821    $\text{SumFunc}([d \in \text{DOMAIN } G.s \mapsto \text{IF } G.s[d].\text{own} = p \wedge G.s[d].st = st \text{ THEN } 1 \text{ ELSE } 0])$ 

823    $\text{SumDevelopmentAllPlayers}(dct) \triangleq$ 
824   IF  $dct \in \text{DOMAIN } G.dp$ 
825   THEN  $\text{SumFunc}([p \in \text{Players} \mapsto (G.h[p].DC.BC)[dct]]) +$ 
826    $\text{SumFunc}([p \in \text{Players} \mapsto (G.h[p].DC.AC)[dct]])$ 
827   ELSE IF  $dct = \text{"Knight"}$ 
828   THEN  $\text{SumFunc}([p \in \text{Players} \mapsto (G.h[p].DC.BC)[dct]]) +$ 
829    $\text{SumFunc}([p \in \text{Players} \mapsto (G.h[p].DC.AC)[dct]]) +$ 
830    $\text{SumFunc}([p \in \text{Players} \mapsto (G.h[p].DC.UC)[dct]])$ 
831   ELSE  $\text{SumFunc}([p \in \text{Players} \mapsto (G.h[p].DC.UC)[dct]])$ 

833 |-----|

835    $\text{ConservationOfResourceCards} \triangleq$ 

```



```

882          $r \mapsto [c \in CD \mapsto [own \mapsto bot]],$ 
883          $ba \mapsto [coords \mapsto [arr \mapsto 1, row \mapsto 1, col \mapsto 3],$ 
884              $res \mapsto bot, n \mapsto bot],$ 
885          $tp \mapsto bot$ 
886     ]

888  $Next \triangleq$ 
889      $\vee SetupPhaseOne$ 
890      $\vee SetupPhaseTwo$ 
891      $\vee DiceRollPhase$ 
892      $\vee EmptyTrade$ 
893      $\vee TradeFourToOne$ 
894      $\vee TradeThreeToOne$ 
895      $\vee TradeTwoToOne$ 
896      $\vee EmptyBuild$ 
897      $\vee BuildRoad$ 
898      $\vee BuildVillage$ 
899      $\vee BuildCity$ 
900      $\vee BuyDevCard$ 
901      $\vee PlayDevCard$ 

903  $Spec \triangleq Init \wedge \Box[Next]_{\langle G, SU \rangle}$ 

905  $WeakFairness \triangleq$ 
906      $\wedge WF_{\langle G, SU \rangle}(SetupPhaseOne)$ 
907      $\wedge WF_{\langle G, SU \rangle}(SetupPhaseTwo)$ 
908      $\wedge WF_{\langle G, SU \rangle}(DiceRollPhase)$ 
909      $\wedge WF_{\langle G, SU \rangle}(EmptyTrade)$ 
910      $\wedge WF_{\langle G, SU \rangle}(TradeFourToOne)$ 
911      $\wedge WF_{\langle G, SU \rangle}(TradeThreeToOne)$ 
912      $\wedge WF_{\langle G, SU \rangle}(TradeTwoToOne)$ 
913      $\wedge WF_{\langle G, SU \rangle}(EmptyBuild)$ 
914      $\wedge WF_{\langle G, SU \rangle}(BuildRoad)$ 
915      $\wedge WF_{\langle G, SU \rangle}(BuildVillage)$ 
916      $\wedge WF_{\langle G, SU \rangle}(BuildCity)$ 
917      $\wedge WF_{\langle G, SU \rangle}(BuyDevCard)$ 
918      $\wedge WF_{\langle G, SU \rangle}(PlayDevCard)$ 

920  $FairSpec \triangleq$ 
921      $\wedge Spec$ 
922      $\wedge WeakFairness$ 

924  $GameEnded \triangleq G.tp = top$ 
925  $EventuallyAlwaysGameEnded \triangleq \Diamond \Box GameEnded$ 

927 THEOREM  $Liveness \triangleq FairSpec \Rightarrow EventuallyAlwaysGameEnded$ 
928 THEOREM  $Safety \triangleq Spec \Rightarrow \wedge TypeOK$ 

```

929                                     $\wedge$  *ConservationOfResourceCards*  
930                                     $\wedge$  *ConservationOfDevelopmentCards*  
931                                     $\wedge$  *ConservationOfBuildings*

933 |  
    \ \* Modification *Histor*  
    \ \* Last modified *Wed Apr 23 17:46:25 CEST 2025* by *tim\_m*  
    \ \* Created *Tue Apr 22 17:19:51 CEST 2025* by *tim\_m*

## **Appendix B**

# **Game Rules**

The next pages provide a complete copy of the Game Rules for reference. To the best of our knowledge, this is permitted under "fair use" for academic purposes.

KLAUS TEUBER

# CATAN<sup>®</sup>

## GAME RULES & ALMANAC



# CATAN

*Dear Settlers,*

*To make it as easy as possible for you to start playing **Catan**®, we use an award-winning rules system, which consists of 3 parts. First, if you do not know how to play **Catan**, please read the Game Overview on page 16 (the back cover). Next, read the Game Rules on pages 2-6 (red borders) and start to play the game. If you have questions during the game, consult the Catan Almanac on pages 6-15 (gold borders).*

*Now you are ready for your first adventure on Catan. Have fun settling this new land together!*

**— Klaus Teuber**

You can find further information at:

[catan.com](http://catan.com)

[catanstudio.com](http://catanstudio.com)

[klausteuber.com](http://klausteuber.com)

## GAME RULES

These 4-page rules (pages 2-5) contain all the important information that you need to play!

If you need more information during the game, you can look up keywords (marked ✱) in the “Almanac,” which follows these rules.

### GAME COMPONENTS

- 19 terrain hexes (tiles)
- 6 sea frame pieces
- 9 harbor pieces
- 18 circular number tokens
- 95 resource cards (bearing the symbols for the brick, grain, lumber, ore, and wool resources)
- 25 development cards (14 knight cards, 6 progress cards, 5 victory point cards)
- 4 “Building Costs” cards
- 2 special cards: “Longest Road” & “Largest Army”
- 16 cities (4 of each color, shaped like churches)
- 20 settlements (5 of each color, shaped like houses)
- 60 roads (15 of each color, shaped like bars)
- 2 dice (1 yellow, 1 red)
- 1 robber
- 1 “Game Rules & Almanac” booklet

### CONSTRUCTING THE ISLAND

The frame pieces hold the board together and prevent the pieces from moving after the board is in place. Before building the island, assemble the frame by matching the numbers at the ends of the frame pieces together (i.e., 1-1, 2-2, etc.).

You can then construct the island of Catan using the 19 terrain hexes as shown on page 3.

### Starting Set-up for Beginners

You can play the game *Catan* on a variable game board. For your first game, however, we suggest that you use the “Starting Set-up for Beginners ✱.” (See Illus. A on page 3.) This set-up is well-balanced for all players.

Before your first game, you must remove the die-cut components from the cardboard holders. Carefully punch out and separate the pieces. When punching tiles out of the die-cut sheets, always push the tiles through from the front, “cut” side (pushing from the back may cause the tiles to rip).

Lay out the map as specified in Illustration A (or on the back of this booklet).

First, assemble the frame as shown. Second, create Catan by placing the 19 terrain hexes on the table—again as shown. Third, place the circular number tokens on top of the designated terrain hexes. Finally, place your settlements and roads.

# RULES

Illustration A

## STARTING MAP FOR BEGINNERS

To make it as easy as possible for you to get started with *Catan*, we use an award-winning rules system, which consists of 3 parts—the *Overview*, the *Game Rules*, and the *Almanac*.

If you've never played *Catan*, please read the game *Overview* first—it's on the back cover of this booklet. Next, read the *Game Rules* and start to play. And finally, if you have questions during the game, please consult the *Almanac* (it begins on page 6).

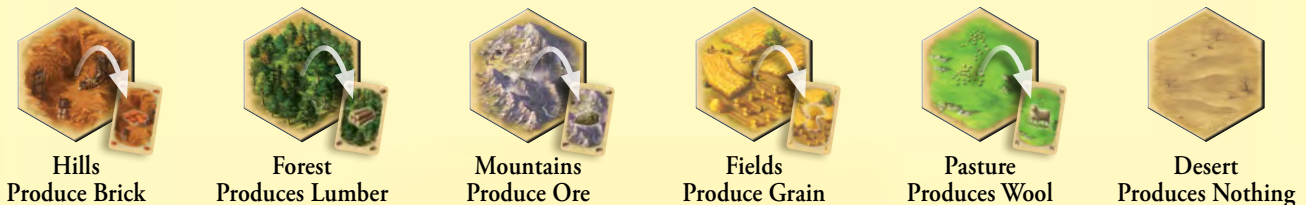


Begin the game with the resource cards produced by the settlements marked with white stars. See ☆

## ODDS FOR DICE ROLLS

2 & 12	= 3%
3 & 11	= 6%
4 & 10	= 8%
5 & 9	= 11%
6 & 8	= 14%
7	= 17%

## RESOURCE PRODUCTION



## Starting Set-up for Experienced Players

It is more fun to play with a variable game board—with the game board laid out randomly. The board changes each game. If you would like to use the variable set-up, you can find the guidelines in the Almanac under Set-up, Variable ☆. Also look for useful tips under Set-up Phase ☆ and Tactics ☆.

## SETTING UP THE GAME

Select a color and take your 5 settlements, 4 cities, and 15 roads (no more and no less!). Place your 2 roads and your 2 settlements on the game board. Place your remaining settlements, roads, and cities down in front of you.

**Note:** If you are playing a 3-player game, nobody plays the red position indicated on the starting map.

Take your color's building costs card. (See Illustration B.)

Place the special cards "Longest Road" and "Largest Army" beside the game board along with the 2 dice.

Sort the resource cards into 5 stacks and put them face up next to the game board. Shuffle the development cards and place them face down by the board.

You receive resources ☆ for each terrain hex around your starting settlement marked with a white star ☆ (see Illustration A). Take the appropriate resource cards from their stacks.

Illustration B

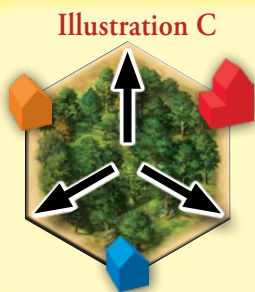


# RULES

**Example:** See Illustration A. Blue receives 1 brick card, 1 lumber card, and 1 ore card for his leftmost settlement (i.e., his settlement marked with a star ☆).

Each player keeps their resource cards hidden in their hand.

**Important:** Settlements and cities may only be placed at the corners of the terrain hexes—never along the edges (see Illustration C). Roads may only be placed at the edges of the terrain hexes—1 road per edge (see Illustration D). The Distance Rule ★ means many intersections along roads will remain unoccupied.



## TURN OVERVIEW

Unless you're using the Starting Set-Up for Experienced Players, the oldest player goes first. On your turn, you can do the following in the order listed:

- You must roll for resource production ★ (the result applies to all players).
- You may trade ★ resource cards with other players and/or use maritime trade ★.
- You may build ★ roads ★, settlements ★ or cities ★ and/or buy development cards ★. You may also play one development card ★ at any time during your turn.

After you're done, pass the dice to the player to your left, who then continues the game with step 1.

**Tip:** For advanced players, we recommend combining the second and third steps. You can find more details in the Almanac under "Combined Trade/Build Phase ★."

## The Turn in Detail

### 1. Resource Production

You begin your turn by rolling both dice. The sum of the dice determines which terrain hexes produce resources.



Each player who has a settlement on an intersection ★ that borders a terrain hex marked with the number rolled receives 1 resource card of the hex's type. For an example see resource production ★. If you have 2 or 3 settlements bordering that hex, you receive 1 resource card for each settlement. You receive 2 resource cards for each city you own that borders that hex. If there are not enough of a given resource in the supply to fulfill everyone's production, then no one receives any of that resource during that turn (unless it only affects 1 player).

## 2. Trade ★

Afterwards, you may trade freely (using either or both types of trades below) to gain needed resource cards:

### a) Domestic Trade ★

On your turn, you can trade resource cards with any of the other players. You can announce which resources you need and what you are willing to trade for them. The other players can also make their own proposals and counteroffers.

**Important:** Players may only trade with the player whose turn it is. The other players may not trade among themselves.

### b) Maritime Trade ★

You can also trade without the other players! During your turn, you can always trade at 4:1 by putting 4 identical resource cards back in their stack and taking any 1 resource card of your choice for it. If you have a settlement or city on a harbor ★, you can trade with the bank more favorably: at either a 3:1 ratio or, in certain harbors, at 2:1 (trading the resource type shown).

**Important:** The 4:1 trade is always possible, even if you do not have a settlement on a harbor.

## 3. Build ★

Now you can build. Through building, you can increase your victory points ★, expand your road network, improve your resource production, and/or buy useful development cards. To build, you must pay specific combinations of resource cards (see the Building Costs Card ★). Take the appropriate number of roads, settlements, and/or cities from your supply and place them on the game board. Keep development cards hidden in your hand.

You cannot build more pieces than what is available in your pool—a maximum of 5 settlements, 4 cities, and 15 roads.

### a) Road ★ Requires: Brick & Lumber

A new road must always connect to 1 of your existing roads, settlements, or cities. Only 1 road can be built on any given path ★.

The first player to build a continuous road (not counting forks) of at least 5 road segments receives the special card "Longest Road ★." If another player succeeds in building a longer road than the one created by the current owner of the "Longest Road" card, they immediately take the special card (and its 2 victory points). **Tip:** This creates a 4 victory point swing!



## RULES

### b) Settlement \* Requires: Brick, Lumber, Wool, & Grain

Take special note of the “Distance Rule” \*: you may only build a settlement at an intersection if all 3 of the adjacent intersections are vacant (i.e., none are occupied by any settlements or cities—even yours).



Each of your settlements must connect to at least 1 of your own roads. Regardless of whose turn it is (i.e., during any production phase), when a terrain hex produces resources, you receive 1 resource card for each settlement you have adjacent to that terrain hex.

Each settlement is worth 1 victory point.

### c) City \* Requires: 3 Ore & 2 Grain

You may only establish a city by upgrading one of your settlements.

When you upgrade a settlement to a city, put the settlement (house) piece back in your supply and replace it with a city piece (church).



Cities produce twice as many resources as settlements. You acquire 2 resource cards for an adjacent terrain hex that produces resources.

Each city is worth 2 victory points.

### d) Buying a Development Card \* Requires: Ore, Wool, & Grain

When you buy a development card, draw the top card from the deck. There are 3 different kinds of these cards: knight \*, progress \*, and victory point \*. Each has a different effect (see 4.b, below).



Development cards never go back into the supply, and you cannot buy development cards if the supply is empty.

Keep your development cards hidden (in your hand) until you use them, so your opponents can't anticipate your play.

## 4. Special Cases

### a) Rolling a “7” and Activating the Robber \*

If you roll a “7,” no one receives any resources.

Instead, every player who has more than 7 resource cards must select half (rounded down) of their resource cards and return them to the bank.

Then you must move the robber \*. Proceed as follows:

- (1) You must move the robber \* immediately to the number token of any other terrain hex or to the desert \* hex.

- (2) Then you steal 1 (random) resource card from an opponent who has a settlement or city adjacent to the target terrain hex. The player who is robbed holds their resource cards face down. You then take 1 card at random. If the target hex is adjacent to 2 or more players' settlements or cities, you choose which one you want to steal from.

**Important:** If the production number for the hex containing the robber is rolled, the owners of adjacent settlements and cities do not receive resources. The robber prevents it.

### b) Playing Development Cards \*

At any time during your turn, you may play 1 development card (put it face up on the table). That card, however, may not be a card you bought during the same turn (except for a victory point card, as described below)!

### Knight Cards (Purple Frame) \*

If you play a knight card, you must immediately move the robber. See “Rolling a ‘7’ and Activating the Robber” above and follow steps 1 and 2.

Once played, knight cards remain face up in front of you. The first player to have 3 knight cards in front of themselves receives the special card “Largest Army,” which is worth 2 victory points. If another player has more knight cards in front of them than the current holder of the Largest Army card, they immediately take the special card and its 2 victory points.

### Progress Cards (Green Frame) \*

If you play a progress card, follow its instructions. Then the card is removed from the game (i.e., toss it in the box).

### Victory Point Cards (Orange Frame) \*

You must keep victory point cards hidden. You may only reveal them during your turn and when you are sure that you have 10 victory points—that is, to win the game. Of course, you can reveal them after the end of the game if someone else wins. You may play any number of victory point cards during your turn, even during the turn you purchase them.



## ENDING THE GAME

If you have **10 or more** victory points **during your turn**, the game ends and you are the winner! If you reach 10 points when it is not your turn, the game continues until any player (including you) has 10 points on their turn.

# ALMANAC

This “Catan Almanac” contains detailed, alphabetical entries and examples for *Catan*. These are not the “Game Rules.” You do not have to read this material prior to your first game. Instead, use the Game Rules. Then read this to enjoy the complete experience.

This almanac includes advanced rules and clarifications. You can also refer to it if any questions arise during a game.

## B

### BUILD (BUILDING)

You may build on your turn after you have rolled for resource production and finished trading. To build, you must turn in the specified combinations of resource cards (see the Building Costs Cards ♣). Return the resource cards to the supply stacks.

You can build as many items and buy as many cards as you desire—as long as you have enough resources to “pay” for them and they are still available in the supply. (See Settlements ♣, Cities ♣, Roads ♣, and Development Cards ♣.)

Each player has a supply of 15 roads, 5 settlements, and 4 cities. If you build a city, return the settlement to your supply. Roads and cities, however, remain on the board until the end of the game once they are built.

Your turn is over after “building,” and the player to your left continues the game.

New rule variant: see *Combined Trade/Build Phase* ♣.

### BUILDING COSTS CARDS

The building costs cards show what can be built and which resources are required. When you pay building costs, you must return the necessary resources to their supply stacks. You can build settlements ♣ and roads ♣, upgrade settlements to cities ♣, and buy development cards ♣.

## C

### CITIES

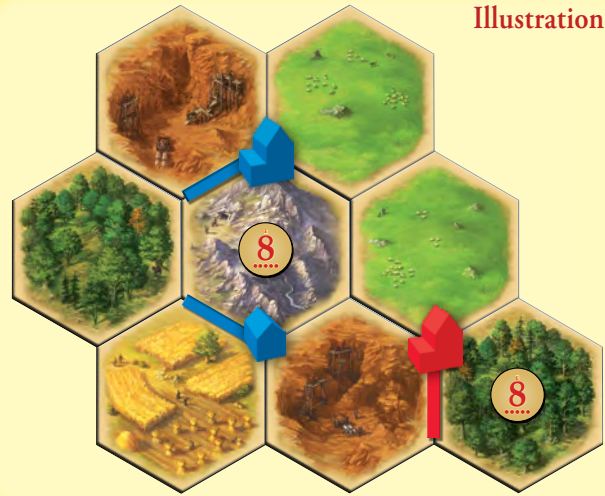
You cannot build a city directly. You can only upgrade an existing settlement to a city. You pay the required resources, return the settlement to your supply, and replace the settlement with a city on the same intersection ♣. Each city is worth 2 victory points. You receive double resource production (2 resource cards) from the adjacent terrain hexes whenever those numbers are rolled.



When you build a city, the upgraded settlement piece becomes available again. You can use it to build another settlement later.

**Example:** See Illustration E. Claudia, the blue player, rolls a resource production roll of “8.” She receives 3 ore cards: 1 ore for her settlement and 2 ore for her city. Benny, the red player, receives 2 lumber for his city.

Illustration E



**Hint:** It is extremely difficult to win the game without upgrading settlements to cities. Since you only have 5 settlements available, you can only reach 5 victory points by only building settlements.

### COAST

When a terrain hex borders on the sea (i.e., a frame piece), it is called a “coast.” You can build a road along a coast. You can build settlements and upgrade settlements to cities on intersections that border on the sea. However, since a site on the coast borders only 1 or 2 terrain hexes, coastal settlements generate smaller resource yields. Still, coastal sites often lie on harbors, which allow you to use maritime trade ♣ to trade resources at more favorable rates.

### COMBINED TRADE/BUILD PHASE

The separation of the trade and build phases was introduced to make the sequence easier to learn for beginners. We recommend experienced players ignore this separation.

After rolling for resource production, you can trade and build in any order (you can trade, build, trade again and build again, etc.). You can even use a harbor on the same turn you build a settlement there. Using this method speeds up the game a lot.

## D

### DESERT

The desert is the only terrain hex that does not produce resources. The robber ♣ starts the game there. A settlement or a city built adjacent to the desert yields fewer resources than those built next to one of the other terrain types.



### DEVELOPMENT CARDS

There are 3 different kinds of cards: Knight Cards ♣, Progress Cards ♣, and Victory Point Cards ♣.

When you buy a development card, take the top card of the draw pile into your hand. Keep your development cards hidden until you play them. This keeps the other players in the dark.

development



You cannot trade or give away development cards.

You may only play 1 development card during your turn—either 1 knight card or 1 progress card. You can play the card at any time, even before you roll the dice. You may not, however, play a card that you bought during the same turn.

**Exception:** If you buy a card and it is a victory point card ♣ that brings you to 10 points, you may immediately reveal this card (and all other VP cards) and win the game.

You only reveal victory point cards when the game is over—once you or an opponent reaches 10+ victory points and declares victory.

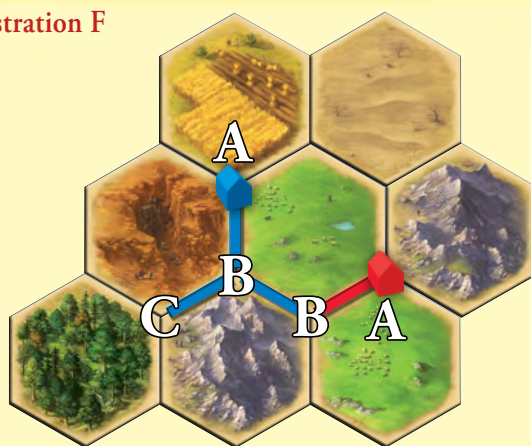


### DISTANCE RULE

You may only build a settlement on an unoccupied intersection ♣ and only if none of the 3 adjacent intersections contains a settlement or city.

**Example:** See Illustration F. Coleman, the blue player, wants to build a settlement. The settlements marked “A” are already in play. Coleman cannot build on the intersections marked “B.” He can only build at intersection “C.”

Illustration F



### DOMESTIC TRADE

On your turn, you may trade resources with the other players (after rolling for resource production). You and the other players negotiate the terms of your trades—such as which cards will be exchanged. You may trade as many times as you can, using single or multiple cards. However, you cannot give away cards, or trade matching resources (“trade” 3 ore for 1 ore, for example).

**Important:** While it is your turn, you must be a part of all trades, and the other players may not trade among themselves.

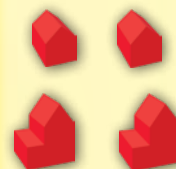
**Example:** It is Pete’s turn. He needs one brick to build a road. He has 2 lumber and 3 ore. Pete asks aloud, “Who will give me 1 brick for 1 ore?” Beth answers, “If you give me 3 ore, I’ll give you a brick.” Cooper interjects, “I’ll give you 1 brick if you give me 1 lumber and 1 ore.” Pete accepts Cooper’s offer and trades a lumber and an ore for a brick. Note Beth may not trade with Cooper, since it is Pete’s turn.

## E

### ENDING THE GAME

If you have—or reach—10 victory points on your turn, the game ends immediately and you win! You can only win during your turn. If somehow you find you have 10 victory points during another player’s turn, you must wait until your next turn to claim victory.

**Example:** Siobhán has 2 settlements (2 points), the Longest Road special card (2 points), 2 cities (4 points), and 2 victory point cards (2 points). She reveals her 2 victory point cards, giving her the 10 points needed to win. She surprises her opponents and claims victory!



## G

## GAME PLAY

Here is a summary of the the game sequence, plus some more specific entries where you can find details:

- (1) Lay out the game board: *Set-up, Variable* \*
- (2) Initial set-up: *Set-up Phase* \*
- (3) Play

The starting player begins the game. The other players follow in clockwise order.

On your turn, you complete these 3 phases in order:

- Roll for *Resource Production* \* (the roll applies to all players)
- *Trade* \*
- *Build* \*

You may play 1 development card any time during your turn.

Pass the dice to the player on your left at the end of your turn. That player then takes their turn using the same 3 phases.

## H

## HARBORS

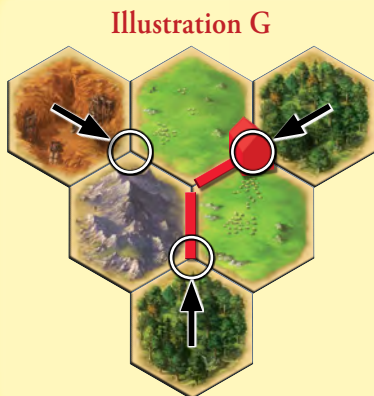
Harbors allow you to trade resources more favorably. In order to control a harbor, you must build a settlement on a coastal intersection \* which borders the harbor. See also “Maritime Trade” \*.



## I

## INTERSECTIONS

Intersections are the points where 3 hexes meet. See Illustration G. You may only build settlements on intersections. The influence (for resource yields) of settlements and cities extends into the 3 adjacent terrain hexes that form the intersection.



## K

## KNIGHT CARDS

When you play a “Knight” development card during your turn, you must immediately move the robber \*. Place the knight card face up in front of you.

You **must** move the robber away from its current spot and onto the number token of **any other** terrain hex or on the desert.

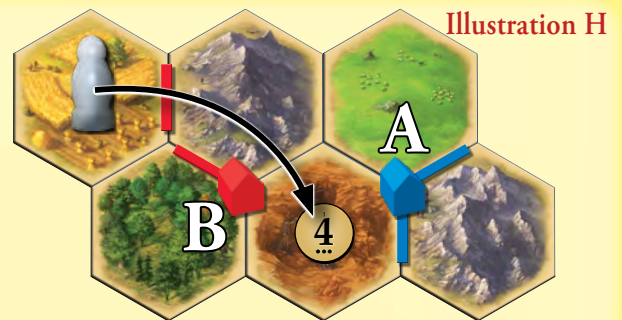
You then steal 1 resource card from a player who has a settlement or a city adjacent to the robber. If there are 2 or more such players, you may choose your victim.

The player you elect to rob keeps their cards face down while you take 1 of their cards at random. If that player has no cards, you get nothing! (However, you can always ask players about the **number** of cards they hold. They must answer truthfully.)

If you are the first player to have 3 knight cards face up in front of you, you take the “Largest Army” \* special card. This special card is worth 2 victory points.

If another player has more face-up knight cards than you, they take the special card and the 2 victory points that go with it.

**Example:** See Illustration H. On Niall's turn, he plays a knight card and moves the robber from the fields hex to the hills hex with the “4.” Niall may now steal a random resource card from player A or B.



## L

## LARGEST ARMY

If you are the first player to play 3 knight cards, you receive this special card, which is worth 2 victory points. You place the “Largest Army” card face up in front of you. If another player plays more knight cards than you have, they immediately take the special card. The 2 victory points likewise count for the new owner.



## LONGEST ROAD

If you are the first player to build a continuous road of at least 5 individual road pieces, you take this special card and place it face up in front of you. This card is worth 2 victory points.

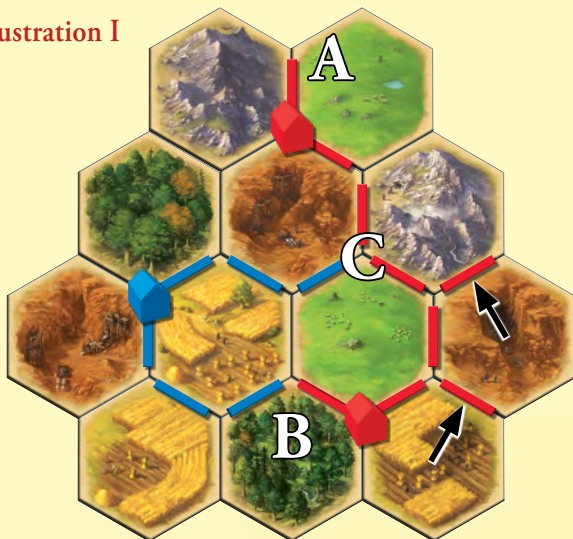


**Note:** If your road network branches, you may only count the single longest branch for purposes of the longest road.

If you hold the “Longest Road” card and another player builds a longer road, they immediately take your “Longest Road” card. They also acquire the 2 bonus victory points. (Since you also lose the 2 victory points, it is a 4 point swing!)

**Example:** See Illustration 1. Emily, the red player, builds a continuous road with 7 wooden pieces (A–B). The branch roads (marked with arrows) are not counted. Emily snags the “Longest Road” special card.

Illustration 1



You can break an opponent's road by building a settlement on an unoccupied intersection along that road!

**Example:** In Illustration 1, Coleman (the blue player) builds a settlement on intersection “C” (which is legal). This breaks Emily's road into 2 parts. Emily must give the special card to Coleman, who now has the Longest Road (and 2 more VPs).

**Special Case:** If your longest road is broken and you are tied for longest road, you still keep the “Longest Road” card. However, if you no longer have the longest road, but two or more players tie for the new longest road, set the “Longest Road” card aside. Do the same if no one has a 5+ segment road. The “Longest Road” card comes into play again when only 1 player has the longest road (of at least 5 road pieces).

## M

### MARITIME TRADE

On your turn, you can trade resources using maritime trade during the trade phase even without involving another player.

The most basic (and unfavorable) exchange rate is 4:1. You may trade 4 identical resource cards to the supply in exchange for 1 resource card of your choice. You do not need a harbor 🌊 (settlement at a harbor location) to trade at 4:1, so when nobody wants to trade...



**Example:** Benny returns 4 ore cards to the supply and takes 1 lumber card in exchange. Normally, he should first try a more favorable trade with the other players (domestic trade).

If you have built a settlement or city at a harbor 🌊 location, you can trade more effectively. There are 2 different kinds of harbor locations:

**Generic Harbor (3:1):** Here you may exchange 3 identical resource cards for any 1 other resource card during your trade phase.

**Example:** Brad has a settlement at a generic harbor. He can, for instance, exchange 3 lumber cards for 1 wool card.



**Special Harbor (2:1):** There is only 1 special harbor for each type of resource (with the same symbol). So, if you earn plenty of a certain type of resource, it can be useful to build on the special harbor for that resource type. The exchange rate of 2:1 only applies to the resource shown on the harbor location. A special harbor does not permit you to trade any other resource type at a more favorable rate (not even 3:1)!



**Example:** Ron built a settlement at the ore special harbor. He may exchange 2 ore cards for any 1 other resource card. He can also trade 4 ore cards for any 2 other cards. If he traded 4 wool instead of 4 ore, he would get only 1 card in return.

## N

## NUMBER TOKENS

The 18 number tokens are marked with the numerals “2” through “12.” There is only one “2” and one “12.” There is no “7.”



The more often a number is rolled, the more often each associated hex produces resources. Note the size of the numbers and the dots (pips) beneath the numbers on the tokens. The taller the number and the larger the quantity of dots, the more likely it is that number will be rolled. “6” and “8” (the red numbers) are the most frequently rolled numbers. They each have 5 dots, because there are 5 ways to roll these numbers on the 2 dice.

The letters on the back of the number tokens are only used during the setup phase (see Set-up, Variable \*).

## P

## PATHS

Paths are defined as the edges where 2 hexes and/or hexes and the frame meet. Paths run along the border of 2 terrain hexes or between a land hex and the frame. Only one road \* can be built on any path. At each end of a path is an intersection \*.



## PROGRESS CARDS

Progress cards are a type of development card. They have green frames. There are 2 each of 3 varieties:

**Road Building:** If you play this card, you may immediately place 2 free roads on the board (according to normal building rules).

**Year of Plenty:** If you play this card you may immediately take any 2 resource cards from the supply stacks. You may use these cards to build in the same turn.

**Monopoly:** If you play this card, you must name 1 type of resource. All the other players must give you *all* of the resource cards of this type that they have in their hands. If an opponent does not have a resource card of the specified type, they do not have to give you anything.

You may play only 1 development card \* during your turn.



## R

## RESOURCE CARDS (RESOURCES)

There are 5 different types of resources (see page 3): grain (from fields), brick (from hills), ore (from mountains), lumber (from forest), and wool (from pasture). These resources are represented by “resource cards.” You receive these cards as income from the resource production of these hexes. Resource production is determined by the dice roll at the beginning of each turn. You receive your income for each terrain hex adjacent to your settlements or cities every time the production number on the hex is rolled (exception: see Robber \*).



## RESOURCE PRODUCTION

On your turn, you must roll the dice for the turn’s resource production. The number rolled determines which hexes produce resources. Each number appears twice—except for “2” and “12,” which only appear once.

All players who have settlements \* or cities \* on the hexes indicated by the roll receive the yields (resource cards) of those hexes. Each settlement produces 1 resource card and each city produces 2 resource cards.

**Example:** See Illustration J. Loren, the blue player, rolls a “4.” Her settlement “A” borders a hills hex marked by the number “4,” so she takes a brick card. If settlement “A” had been a city, she would have received 2 brick cards. Bridget owns the red settlement “B” that borders on 2 hexes with the number “4”: hills and pasture. Bridget takes 1 brick card and 1 wool card from the supply stacks.

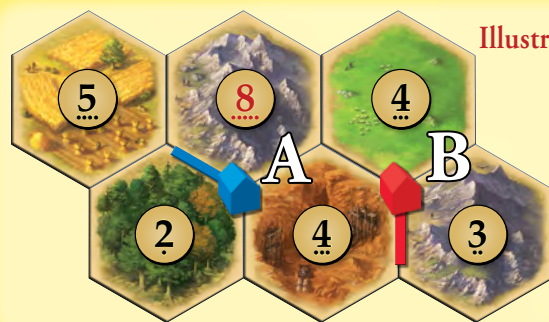


Illustration J

It is possible that during the game there will not be enough resources in the bank to supply all of the yields. If there are not enough resource cards to give every player all the production they earn, then no player receives any of that resource that turn.

**Exception:** If the shortage of resource cards only affects a single player, give that player as many of these resources as are left in the supply, and any extras are lost. In either case, production of other types of resources is not affected.

## RESOURCE TRADE

In the second phase of your turn, you may trade with the other players. The other players may not trade among themselves, only with the player whose turn it is. There are 2 different kinds of trade:

- Domestic trade ♣ and
- Maritime trade ♣.

## ROADS

The roads connect your settlements and cities. You build roads on paths ♣. You cannot build new settlements without also building roads. Roads provide victory points only if you hold the Longest Road ♣ special card. Only 1 road may be built on each path. You can build roads along the coast.

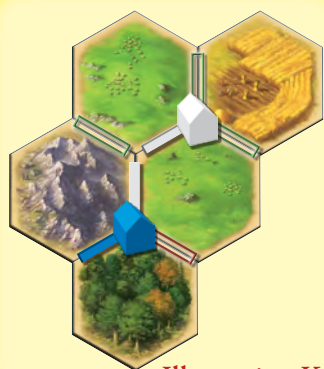


Illustration K

**Example:** See Illustration K. Liam, the white player, would like to build a road. He may build (place) his road on any of the paths outlined in green. Each of these paths connects to either Liam's road or his settlement, and is not blocked by the blue player's settlement (see the path outlined in red).

## ROBBER

The robber begins the game in the desert ♣. It is moved only by rolling a "7" ♣ or playing a Knight ♣ card.

If the robber is moved to any other terrain hex, it prevents that hex from producing resources. Players with settlements and/or cities adjacent to the target terrain hex receive no resources from this hex as long as the robber is in the hex.



Robber

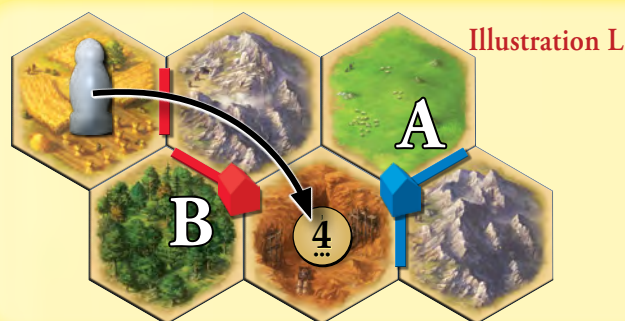


Illustration L

**Example:** See Illustration L. It is Xander's turn and he rolls a "7." He must move the robber. The robber was in a fields hex. Xander places it on the "4" number token of a hills hex. If a "4" is rolled in the coming turns, the owners of settlements

"A" and "B" do not receive a brick resource card. This lasts until the robber is moved again by another "7" or a knight card. Xander may also take 1 resource card at random from 1 of the players who own settlements "A" or "B".

## ROLLING A "7" AND ACTIVATING THE ROBBER

If you roll a "7" for resource production, none of the players receive resources. Instead:

- (1) Each player counts their resource cards. Any player with more than 7 resource cards (i.e., 8 or more) must choose and discard half of them. Return discards to the supply stacks. If you hold an odd number of cards, round down (e.g., if you have 9 resource cards, you discard 4).

**Example:** Alex rolls a "7." He has only 6 cards in his hand. Larry has 8 cards and Will has 11. Larry must discard 4 cards and Will 5 (rounding down).

- (2) Then you (the player who rolled the "7") must move the robber ♣ to the number token ♣ of any other terrain hex (or to the desert ♣ hex). This blocks the resource production of this hex, until the robber moves to another number token or the desert.

- (3) After discarding occurs, you also steal 1 resource card at random from a player who has a settlement or city adjacent to this new hex. If there are 2 or more players with buildings there, you may choose from which one to steal.

The robber must be moved. You may not choose to leave the robber on the same hex.

After moving the robber, your turn continues with the trade phase.

See also *Knights* ♣.

## S

## SETTLEMENTS

A settlement is worth 1 victory point. Settlements are built on intersections ♣ (where 3 hexes meet or 1 or 2 hexes meet the frame). You share in all of the resource production of each terrain hex adjacent to your settlements.

You must meet 2 conditions when building a settlement:

- (1) Your settlement must always connect to 1 or more of your own roads ♣.
- (2) You must observe the Distance Rule ♣. For an example of the distance rule, see Illustration M on page 12.



## ALMANAC

**Example:** See Illustration M. Olivia, the blue player, wants to build a new settlement. She may only do so at one of the intersections marked "B". She cannot build on "A" because of the Distance Rule, nor on "C" because no blue road leads to this intersection.

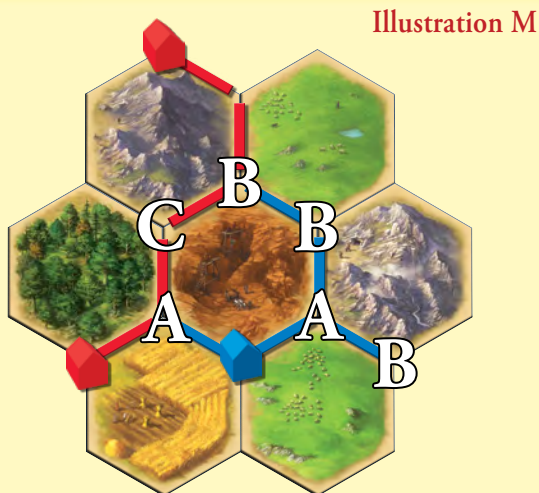


Illustration M

**Note:** If you have built all 5 of your settlements, you must upgrade 1 of your settlements to a city before you can build another settlement. You will then have the settlement in your supply, so you can build another settlement.

### SET-UP PHASE

Begin the "set-up phase" after you build the game map (see Illustration N and *Set-up, Variable* \*).

Everyone chooses a color and takes the corresponding game pieces:

- 5 settlements;
- 4 cities;
- 15 roads; and
- 1 building costs card.

Sort the resource cards into 5 stacks and place them face up beside the board.

Shuffle the development cards \* and place them face down beside the resource cards.

Place the 2 special cards and the dice beside the board.

Place the robber in the desert.

The set-up phase has 2 rounds. Each player builds 1 road and 1 settlement per round.



Illustration N

Suggested Beginners' Map Set-up

### Round One

Each player rolls both dice. The player who rolls highest is the starting player and begins.

The starting player places a settlement on an unoccupied intersection \* of their choice, then places a road adjacent to this settlement.

The other players then follow clockwise.

Everyone places 1 settlement and 1 adjoining road.

**Important:** When placing all other settlements, the Distance Rule \* (see page 7) always applies!

### Round Two

Once all players have built their first settlement, the player who went last in the first round begins round two. That player builds their second settlement and its adjoining road.

**Important:** After the starting player builds, the other players follow **counterclockwise**, so the starting player in round one places their second settlement last.

The second settlement can be placed on any unoccupied intersection, as long as the Distance Rule is observed. It doesn't have to connect to the first settlement. The second road must attach to the second settlement (pointing in any of the 3 directions).

Each player receives their starting resources immediately after building their second settlement. For each terrain hex adjacent to this second settlement, take a corresponding resource card from the supply.

**Note:** The starting player (the last to place their second settlement) begins the game: That player rolls both dice for resource production. You can find helpful tips about the set-up phase under "Tactics."

### SET-UP, VARIABLE

Assemble the frame as outlined on pages 2-3.

**Note:** If you want to vary relative harbor locations slightly, just shuffle the order of the frame pieces **and do not** place the random harbor pieces as outlined below in point 2.

Turn the terrain hexes face down. Shuffle the terrain hexes.

1. Randomly place the terrain tiles face up inside the frame arranged as shown in Illustration O.



Illustration O

2. Now take the 9 harbor pieces (the small 5-sided pieces with ships on them) and randomly place one on top of each harbor on the frame. See Illustration P.

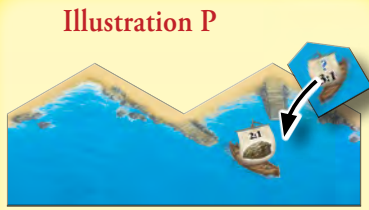


Illustration P

3. Place the 18 number tokens as shown in Illustration Q:

- Sort the number tokens beside the board, letter side face up.
- Place 1 token on each land hex. Start at a corner of the island. Place the number tokens on the terrain hexes in alphabetical order, proceeding *counter-clockwise* toward the center. Skip the desert.

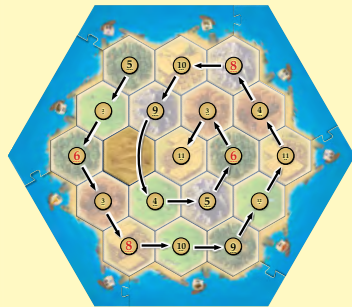


Illustration Q

**Important:** Alternatively, you can use a fully random set-up. Place 1 token on each land hex. Start at one corner of the island, and place the number tokens in random order. In such case, the tokens with the red numbers must not be next to each other. You may have to swap tokens to ensure that no red numbers are on adjacent hexes.

**Note:** The desert never gets a number token. It should be skipped.

More set-up instructions can be found in "Set-up Phase." ✱

## SOLDIER CARDS

Some earlier editions of *Catan* had soldier cards. These are now called knight cards. See Knight Cards ✱.

## STARTING SET-UP FOR BEGINNERS

If you want to use the starting set-up, lay out the board as shown in Illustration R (and the Game Overview):

- Assemble the frame exactly as shown in Illustration R.
- Place the terrain hexes exactly as shown in Illustration R.
- Place 2 settlements and 2 roads of each color as shown.

If only 3 are playing, remove the red pieces.

- Each player receives the 3 resources from the terrain hexes adjacent to their settlement marked by the white star.

The oldest player is the starting player. The oldest player takes the first turn and rolls for resource production.

## TACTICS



Since you play *Catan* with a variable map, the tactical considerations of each game are different. There are, nevertheless, some common points you should consider:

- Brick and lumber are the most important resources at the beginning of the game. You need both to build roads and settlements. You should try to place at least 1 of your first settlements on a good forest or hills hex.
- Do not underestimate the value of harbors. For instance, a player with settlements or cities on productive fields should try to build a settlement on the "grain" harbor.
- Leave enough room to expand when placing your first 2 settlements. Look at your opponents' sites and roads before making a placement. Beware of getting surrounded! If you plan to build toward a harbor, the middle of the island may be a tricky place for a starting settlement, for it can easily be cut off from the coast.
- The more you trade, the better your chances of victory. Even if it is not your turn, you should offer trades to the current player!

Illustration R



## TRADE

After you roll for resource production, you may trade with other players (domestic trade ) or with the bank (maritime trade ) .

- You may trade as long as you have resource cards.
- If you decide not to trade during your turn, no one can trade.
- You may trade with another player between your turns, but only if it is that player's turn and they elect to trade with you.
- You may **not** trade with the bank during another player's turn.
- You may **not** give away cards.
- You may **not** trade development cards.
- You may **not** trade like resources (e.g., 2 wool for 1 wool).

## V

## VICTORY POINT CARDS

Victory point cards are development cards , so they can be “bought.” These orange framed development cards represent important cultural achievements, represented by certain buildings.

Each victory point card is worth 1 victory point. You only reveal your victory point cards when you or someone else wins the game! Keep victory point cards hidden until you have 10 points during your turn and you can declare victory. (You should also reveal them if someone else wins.)



**Hint:** Again, keep your victory point cards hidden until the end of the game. Place them face down in front of you. Of course, if you have 1 or 2 unused cards face down in front of you for a long time, the other players will assume that they are victory point cards.

A general note on Catan rules. This is the 5th English-language edition of *Catan* (aka *Settlers of Catan*). Over the years, the rules have been clarified, refined, and occasionally updated. As of January 1, 2015, all of the rules in this version of *Catan* take precedence over any previously-published rules.

## VICTORY POINTS

The first player to reach (be at) 10 victory points (VPs) on their turn wins the game.

Players acquire victory points (VPs) for the following:

*1 settlement* = 1 VP

*1 city* = 2 VPs

*Longest Road special card* = 2 VPs

*Largest Army special card* = 2 VPs

*Victory point (VP) card* = 1 VP

Since each player begins with 2 settlements, each player begins the game with 2 victory points.

Therefore, you only need 8 more victory points to win the game!

**Hint:** In *Catan*, you will often see an icon of a rising sun (on VP cards and special VP cards, etc.). This is the VP symbol. It is a visual reminder for the things, other than settlements and cities, that earn you VPs. Each VP symbol you see is worth 1 VP.



## Victory Points in Catan



= 1 VP



= 2 VPs



= 2 VPs



= 2 VPs



= 1 VP



## CREDITS

**Designer:** Klaus Teuber (klausteuber.com)

**Original Development:** TM-Spiele GmbH

**5<sup>th</sup> Ed. Development Team:** Pete Fenlon, Arnd Fischer,  
Ron Magin, Benjamin Teuber, and Guido Teuber

**Art:** Michael Menzel, Pete Fenlon

**Art Direction & Graphic Design:** Pete Fenlon,  
Michaela Kienle, and Ron Magin

**Production:** Ron Magin & Pete Fenlon

**Special Thanks:** Robert T. Carty, Jr., Coleman Charlton,  
Morgan Dontanville, Alex Colón Hernández, Aud Ketilsdatter  
(LSKC/354-2), Stephanie Newman, Donna Prior and Kelli  
Schmitz.

**Original U.S. Product Development:** Coleman Charlton, Robert  
T. Carty Jr., Pete Fenlon, Nick Johnson, Will Niebling, William Niebling,  
Guido Teuber, Larry Roznai, and Alex Yeager

**With thanks to:** Bridget Roznai, Loren Roznai, Schar Niebling, Peter  
Bromley, Darwin Bromley, Trella Bromley, Bill Wordelmann, Elaine  
Wordelmann, Lou Rexing, Tom Smith, Keywood Cheeves, Mike Strack,  
Benny Teuber, Claudia Teuber, Liam Teuber, Leif Teuber, Emily Johnson,  
Olivia Johnston, Karl Roelofs, Arnd Beenen, Gero Zahn, and  
Scott Anderson

Copyright © 2020 Catan GmbH and Catan Studio. Catan,  
The Settlers of Catan, the “Catan Sun” logo, the “Glowing Yellow  
Sun” and “Catan Board” marks, and all marks herein are  
trademarks of Catan GmbH and are used under license by Catan  
Studio. Published by Catan Studio, 1995 W. County Rd. B2,  
Roseville, MN 55113. Phone +1.651.639.1905.

You have purchased a game of the highest quality.  
However, if you find any components missing  
or damaged, please visit:

[www.catanstudio.com/parts](http://www.catanstudio.com/parts)

For all other inquiries, contact us at:

[info@catanstudio.com](mailto:info@catanstudio.com)

If you would like to  
protect your cards in this  
game, we recommend  
Gamegenic card sleeves.

The badge here  
indicates what style of  
sleeves and the number  
of packs required to  
sleeve all of the cards in  
this CATAN expansion.



## ALMANAC INDEX

### Entries

### Page

Build (Building) . . . . .	6
Building Costs Cards . . . . .	6
Cities . . . . .	6
Coast . . . . .	6
Combined Trade/Build Phase . . . . .	6
Desert. . . . .	7
Development Cards . . . . .	7
Distance Rule . . . . .	7
Domestic Trade . . . . .	7
Ending the Game . . . . .	7
Game Play. . . . .	8
Harbors . . . . .	8
Intersections . . . . .	8
Knight Cards. . . . .	8
Largest Army . . . . .	8
Longest Road . . . . .	9
Maritime Trade . . . . .	9
Number Tokens . . . . .	10
Paths . . . . .	10
Progress Cards . . . . .	10
Resource Cards (Resources) . . . . .	10
Resource Production . . . . .	10
Resource Trade . . . . .	11
Roads . . . . .	11
Robber . . . . .	11
Rolling a “7” and Activating the Robber. . . . .	11
Settlements . . . . .	11
Set-up Phase . . . . .	12
Set-up, Variable . . . . .	12
Soldier Cards . . . . .	13
Starting Set-up for Beginners . . . . .	13
Tactics. . . . .	13
Trade . . . . .	14
Victory Point Cards . . . . .	14
Victory Points . . . . .	14

**CATAN**  
**STUDIO**

[catanstudio.com](http://catanstudio.com)

**CATAN**

[catan.com](http://catan.com)

# GAME OVERVIEW

**1** The island of Catan lies before you. The isle consists of 19 terrain tiles surrounded by ocean. Your goal is to settle on Catan, and expand your territory until it becomes the largest and most glorious in Catan.

**2** There are five productive terrain types and one desert on Catan. Each terrain type produces a different type of resource (The desert produces nothing). Each resource you receive is represented by a card. Here's what each terrain produces:



**Hills**  
Produce Brick



**Forest**  
Produces Lumber



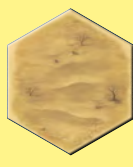
**Mountains**  
Produce Ore



**Fields**  
Produce Grain



**Pasture**  
Produces Wool



**Desert**  
Produces Nothing

**3** You begin the game with 2 settlements and 2 roads. Each settlement is worth 1 victory point. You therefore start the game with 2 victory points! The first player to acquire 10 victory points on their turn wins the game.

**4** To gain more victory points, you must build new roads and settlements and upgrade your settlements to cities. Each city is worth 2 victory points. To build or upgrade, you need to acquire resources.

**5** *How do you acquire resources? It's simple.* Each turn, you roll 2 dice to determine which terrain hexes produce resources. Each terrain hex is marked with a round number token. If, for example, a "10" is rolled, all terrain hexes with a "10" number token produce resources—in the illustration on the right, those terrain hexes are a mountains hex (ore) and a hills hex (brick).

**6** You only collect resources if you own a settlement or city bordering these terrain hexes. In the illustration, the red settlement [A] borders the "10" mountains and orange settlement [B] borders the "10" hills. If a "10" is rolled, the red player receives 1 ore card and the orange player receives 1 brick card.

**7** Since the settlements and cities usually border on 2-3 terrain types, they can "harvest" up to 3 different resources based on the dice roll. Here, the white settlement [C] borders on forest, mountains, and pasture. A settlement at [D] would only harvest the production from 2 terrain hexes (hills and mountains). Finally, a settlement at [E] would only harvest the production from 1 terrain hex (pasture). However [E] is also at a wool harbor.

**8** Since it's impossible for you to have settlements adjacent to all terrain hexes and number tokens, you may receive certain resources only at rare intervals—or never. This is tough, because building requires specific resource combinations.

**9** For this reason, you can trade with other players. Make them an offer! A successful trade might yield you a big build!

**10** You can only build a new settlement on an unoccupied intersection if you have a road leading to that intersection and the nearest settlement is at least two intersections away.

**11** Carefully consider where you build settlements. The numbers on the round tokens are depicted in varying sizes. They also have dots (pips) below the numbers. The taller the depicted number, and the more pips it has, the more

likely that number is to be rolled. The red numbers 6 and 8 are the tallest numbers with the most pips; they are likely to be rolled most frequently.



**Bottom line:** The more frequently a number is rolled, the more often the hexes with those numbers produce resources.

You should consider settling on hexes that have good potential for production (*i.e.* 6 and 8 vs. 2 and 12). However, these same high-producing hexes are often the primary target for the robber.

