# Project Summary

Our project aims to predetermine whether the player will lose a life in the retro video game of Jezzball if they were to construct a line. We will be utilizing propositional logic and SAT solvers to be the backbone of our reasoning and justification. This will be achieved by breaking down the large and inherent aspects that we easily identify into an atomic degree, which then can be solved by propositional logic bit by bit. Our model aims to figure this out based on the current cursor positions and cursor orientation (vertical or horizontal for building lines), as well as with a list of the current ball positions and velocities. It will also need a map of which cells have been captured already. We will be doing all of this while being confined within the restrictions of our constraints. Ultimately, this model will give us an indicator of whether we have lost a life in the game.

# Propositions

Some aspects of the game that can be modeled as propositions:

**Horizontal** – The horizontal orientation for building a line is selected. If the **Horizontal** proposition is true, this indicates that we will be building a line horizontally. While **Horizontal** is true, **Vertical** can’t be true.

**Vertical** – The vertical orientation for building a line is selected. If the **Vertical** proposition is true, this indicates that we will be building a line vertically. While **Vertical** is true, Horizontal can't be true.

**Orientation (o)** – Nests the two orientations into one single proposition. Could be either **Horizontal** or **Vertical**.

**Mouse(x, y, o)** – The cursor/mouse is located at an **x position** and a **y position** with an **Orientation o**.

**Captured(x, y)** – A cell within the true game domain located at **x** and **y** is already captured. In the game, it is visually a black cell. This indicates that no game ball or line being built can exist within this captured cell. Once **Captured** is true for an (x,y), it will remain that way for the remainder of the game.

**Position(i, x, y, t)** – A game ball denoted i, is currently found at **x** and **y** during the time of **t**. Because this is a component of the game ball, it does not indicate the velocity the game ball is travelling at.

**Build(D, x, y, t)** – A builder is building at the location (**x**, **y**) in the direction specified by D, at time t. When **Build** proposition is true, no additional **Build** propositions for any other (**x,y**) or any D (self-inclusive) can be true. In other words, there can be only ONE active builder building at one given moment in time.

**Building Cell (D, x, y, t)** – A **BC** (**building cell**) at (x,y) will occupy a cell in the game board. When **BC** is true, the cell at (x,y) has been traversed by **Build** and is awaiting for the builder to finish. When a build attempt is successful, **BC** at (x,y) will remain false for the remainder of the game, and **Captured** for that (x,y) will evaluate to true.. When a game ball reaches (x,y) of this **BC**, the player loses a life.

**Building Finish (D, t)** – A **BF** (**building finish**) for the **Build** going in the direction **D** is true when that specific builder has reached a captured cell at time t.

**LoseLife(t)** – The player has lost a life while building a line during time t. This means that for the **Lost** proposition to be true, **Build** must be true. At the very moment the player loses a life, the game ball has collided with the builder or a building cell. The outcome for this proposition is the outcome of our model. So, for every conclusion, a value for **Lost** must be returned. **Lost** Boolean is a variable and can be either true or false.

To handle the velocities of a ball, we will need to use the following predicates, that also follow the physics of the real game:

**VelocityX(i, t)** – An expression that indicates whether a **ball i** is moving in the positive or negative x direction at time t. If **VelocityX** is true, then for the **ball i**, it is moving in the positive **x direction**. The contrapositive is true.

**VelocityY(i, t)** – An expression that indicates whether a ball i is moving in the positive or negative y direction at time t. If **VelocityY** is true, then for the **ball i**, it is moving in the positive y **direction**. The contrapositive is true.

# Constraints

1. **Build Direction**: *A constraint to prevent the mouse from pointing both vertical and horizontal. The cursor's orientation may only be either vertical or horizontal, but never both*.

∀o. ¬ ((Horizontal ∨ Vertical) → (Horizontal ∧ Vertical))

1. **Boundaries for Game Ball:** *Constraint to ensure a ball may never be inside of a captured cell. A game ball may never share its (x,y) position with a captured cell. In other words, both a game ball and a captured cell can never have the same (x,y).*

*-* ∀x. ∀y. ¬(Position(i, x, y, t) → Captured(x, y))

1. **Amount of Active Builders:** *A constraint to regulate the number of active builders building at one give moment in time. At a* *time t, there will only be one active builder building either horizontally or vertically.*

*-* ∀Build.((Build(x, y, N, t) ∧ Build(x, y, S, t) ) ∨ (Build(x, y, E, t) ∧ Build(x, y, W, t))) → ∀x. ∀y. (¬∃B. (Build(x, y, D t) ))

1. **Losing Condition:** *A constraint to determine whether the player has lost a life. This occurs when a game ball collides with a builder.*

- ∀Position.(∃x.∃y.(Position(i, x, y, t ∧ Build(x, y, N, t)) → Lost)

# Model Exploration

# Outline

Our current model will output True or False depending on whether the player will lose a life or not after building a line. We had initially debated between this or whether the output was going to be the current number of lives the player has left. But we chose to go with the former because first, providing a binary output allows the player to understand their result with clarity. Secondly, a binary output is better suited to be modelled using propositional and predicate logic. It’s important to note that we still keep track of the lives the user has left for game-functionality, however this information is not included in the output.

# Figuring the physics of the game

The next challenge we faced was figuring out the mechanics of how the program was going to determine if a ball was going to collide with a line being built. In JezzBall, the balls only move in 4 diagonals, meaning if the ball is travelling in the North-East direction, the x and y direction would both be positive. This allowed us to create these 2 propositions:

**VelocityX(i)** – An expression that’s True if a ball ‘i’ is moving is the positive x direction and False if o it’s moving in the negative x direction.

**VelocityY(i)** –An expression that’s True if a ball ‘i’ is moving is the positive y direction and False if it’s moving in the negative y direction.

A black screen with colorful text

Description automatically generatedBecause we have had this process figured out with the necessary propositions and definitions for them, the constraints on this section were not difficult to sort out.

# Exploring the Builder

One of the most crucial features of Jezzball is the builder. This is so much more relevant because our model focuses on whether we have lost a life or not based on the deployment of a single builder. We have reasoned that the builder, will be separated into two subcomponents when deployed. This is because in the classic game, we can distinctively distinguish two builders going in 180 degree opposite directions, labeled differently by colors.

So, to keep up with consistencies, we have done the same as well. With this, our approach to creating a builder is slightly modified. Instead of one single unit we call the “builder”, a builder is comprised of two smaller individual builders that go in opposite directions; even though they are individual, they both operate under an alias of a single “builder”.

What this means for our model is that when one of the two components of the builder collides with the ball, the player has lost a life, and that builder is destroyed. Because our model focuses on the loss of a life, it does not matter how many lives you can lose upon a deployment of a builder. We will default it to max 1 life can be lost per deployment of a builder, for simplicity.

A computer screen with text

Description automatically generatedWe have then assigned the cursor orientation, which could be either Horizontal or Vertical to assume two builders with opposite cardinal directions under one single alias.

After this, we explored different ways for which we can assign building cells to the cells traversed by a working builder. This next objective unpacked a lot of rationalization as now we have to create our logic for the builder (what happens if it reaches a captured cell, BuilderFinished prop and laying down builder cells). Here are the interactions based on all four edges of the game domain:

A screen shot of a computer program

Description automatically generated

# Jape Proof Ideas

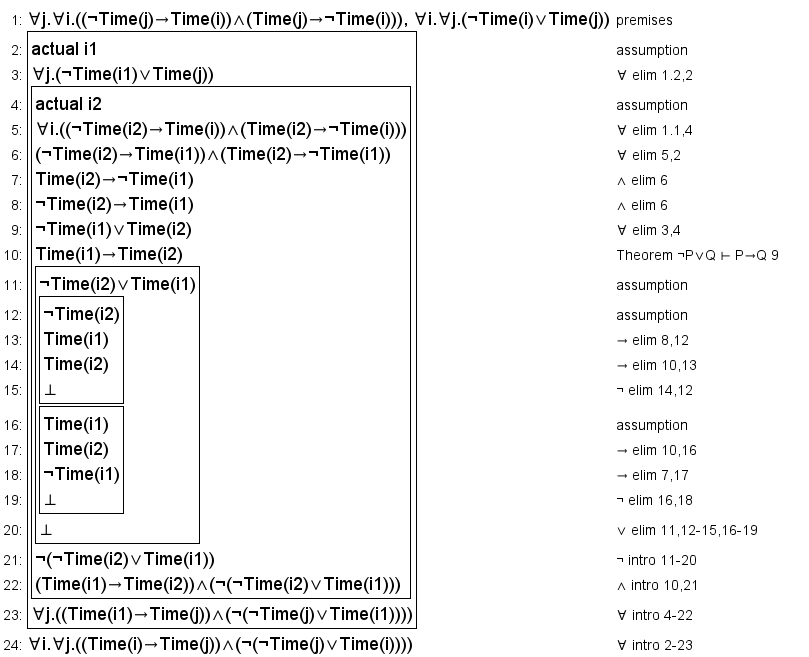
Time ordering: Ensures that time could be only one entity at one moment.

* Time progresses sequentially and cannot be multiple states at once. Time can only move forwards as expressed in propositional logic.

∀j. ∀i.((¬Time(j)→ Time(i))∧ (Time(j)→ ¬Time(i))) - shows that time can never be going backwards

∀j. ∀i.(¬Time(i)∨ Time(j)) *-* shows the logical flow in which time can only go forwards

∀i. ∀j.((Time(i) →Time(j))∧ (¬(¬Time(j)∨Time(i)) - shows that time is one singular instance (there can never be multiple placeholders for time)



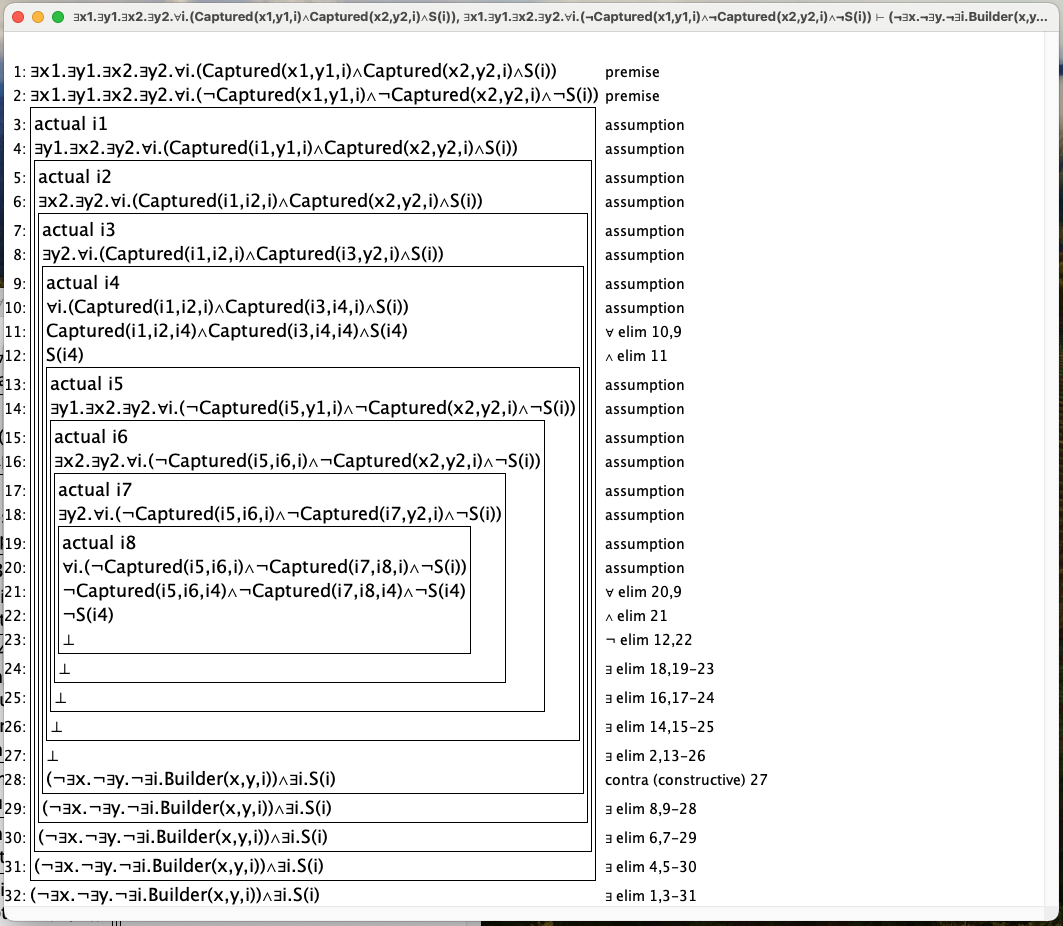
Understanding the win condition by Cell Adjacency

* The game is in a winning state if and only if cells (x1,y1) and (x2,y2) have been captured at time I. Conversely, the game remains in a non-winning state if cells (x1,y1) and (x2,y2) have not been captured at time i. With these two premises, we can deduce that if the cells have been captured, the game has been won and there are no builders building any lines at time i.

∃x1.∃y1.∃x2.∃y2.∀i.(Captured(x1,y1,i)∧ Captured(x2,y2,i)∧ S(i)) - definition of cell adjacency (winning state)

∃x1.∃y1.∃x2.∃y2.∀i.(¬Captured(x1,y1,i)∧ ¬Captured(x2,y2,i)∧ S(i)) - a non winning state infered by cell adjacency not being present

(¬∃x.¬∃y.¬∃i.Builder(x,y,i))∧ ∃i.S(i) - as a result, no builders are left right after either both a winning or non winning state is determined.

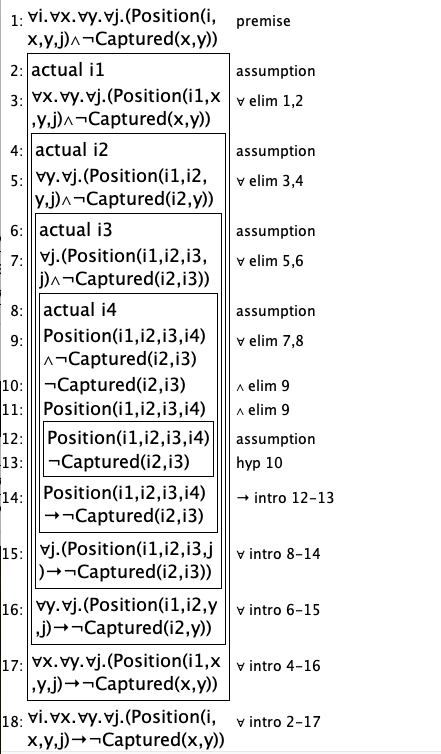


Physics: A ball and a captured cell cannot share the same coordinates.

* The position of a ball 'i' cannot coincide with the position of a captured cell at (x,y) at time j. This premise infers that if a ball is at position (x,y) then the cell at that position has not been captured.

∀i.∀x.∀y.∀j.(Position(I,x,y,j)∧¬Captured(x,y,)) - shows the interaction between a ball and captured cell, they cannot share a cell

∀i.∀x.∀y.∀j.(Position(I,x,y,j) → ¬Captured(x,y,)) - we conclude if a ball is at a cell, that cell cannot be a captured cell that that instance in time; the converse is true aswell.



# First-Order Extension

We can use first-order extension to introduce additional predicates and quantifiers to handle more complex relationships and interactions. Here are some examples:

**Proposition Extensions:**

* **Time (T)**: Introduce a predicate to represent time, allowing you to model changes and interactions over time. For example, you can add "Time(t)" to indicate the current time step.
* **Collision (C)**: Create a predicate to handle collisions between balls and builders. For instance, "Collision(i, j, t)" can represent that ball i collides with builder j at time t.
* **Win (W)**: Add a predicate to denote when the player wins the game. "Win" can be true when certain conditions are met, such as capturing a specific number of cells.
* **Ball Deactivation (BAd)**: "BallDeactivation(i, t)" indicates that ball i is no longer in play at time t.
* **Builder Deactivation (BDe)**: "BuilderDeactivation(b, t)" denotes that builder b is no longer active at time t.
* **Builder Direction (BD)**: "BuilderDirection(b, t, D)" specifies that builder b is constructing a line in direction D at time t.
* **Cell Adjacency (CA**): "CellAdjacency(x1, y1, x2, y2)" expresses that cells (x1, y1) and (x2, y2) are adjacent.

**Constraint Extensions:**

* **Time Ordering**: To ensure that time steps are ordered correctly, you can use a universal quantifier. For example, "∀t. Time(t) → Time(t+1)" enforces that time progresses sequentially for all time steps.
* **Mutual Exclusivity**: To extend the constraint for cursor orientation while adding quantifiers, you can use a combination of universal and existential quantifiers. For example, "∀t. ∃o. (Horizontal(t) ∨ Vertical(t)) → (Horizontal(t) ∧ Vertical(t))" ensures that for all time steps, either the horizontal or vertical orientation is selected, but not both.
* **Ball Movements**: To enforce constraints related to ball movements, you can use a universal quantifier. For instance, "∀i. ∀t. VelocityX(i, t) → (Position(i, x, y, t) ∧ (Position(i, x+1, y, t) ∨ Position(i, x-1, y, t)))" ensures that for all balls and time steps, the ball's X velocity corresponds to its X position at the next time step.
* **Winning Condition**: For creating constraints related to winning conditions, you can use an existential quantifier. For example, "∃x1. ∃y1. ∃x2. ∃y2. ∀t. Win → (Captured(x1, y1, t) ∧ Captured(x2, y2, t))" specifies that there exist positions (x1, y1) and (x2, y2) such that for all time steps, winning conditions are met when specific cells are captured.
* **Ball-Ball Collisions**: To extend collision constraints to handle interactions between balls while adding quantifiers, you can use a combination of universal and existential quantifiers. For example, "∀i. ∀j. ∃t. Collision(i, j, t) → ¬(∃x. ∃y. Position(i, x, y, t) ∧ Position(j, x, y, t))" specifies that for all pairs of balls i and j, there exists a time t when a collision occurs, and at that time, they cannot occupy the same position.