# **BATTLESHIP**

**PROJECT SUMMARY**

Battleship is a two player game where each player tries to sink all five of the enemy’s hidden ships on a grid. The first player to sink all of the enemy’s ships wins. However, we introduced a unique twist by modifying the traditional ruleset.

In this version, we are not telling the model whether it has sunk a ship or not, meaning that it has to figure out whether a ship is complete on its own. This introduces new problems that need to be solved with interesting constraints. The user has the option of a pre-made board or to make their own board, allowing for plenty of scenarios to solve. This model only works with submarines and destroyers and works with any sized board.

# **PROPOSITIONS**

* Checked(location, status): A group of propositions that all signify a location is for sure some status. These statuses are *Hit*, *Miss*, *Complete*, *Sunk*, *Cornered Hit*.
  + *Hit, Miss,* and *Sunk* are all self explanatory.
  + *Complete* means that the location is surrounded by no unchecked locations and is *Hit*.
  + *Cornered Hit* means that the location is surrounded by one hit and the rest misses or boundaries.
* Unchecked(location, status): A group of propositions that all signify a location is possibly some status. These are *Unchecked, Unlikely Segment, Possible Segment, Highly Possible Segment, Possible Sunk.*
  + *Unchecked* means that the model doesn’t know anything about that location.
  + Unlikely, possible, and highly possible segments are the order in which the model will attempt to shoot at with unlikely being the last priority.
  + *Possible Sunk* means that it is next to a *Cornered Hit*
* Boundary(location): Specifies that a particular location is a boundary (nothing can be on a boundary).
* Ship(location, stype, possible): Specifies that a ship of a particular type (e.g., a destroyer or submarine) is certainly or uncertainly at a given location.

# **CONSTRAINTS**

There are four main types of constraints that we have used for this model, ones that we’ve labeled as “basic constraints”, finding segments of different possibilities, finding sunk ships, and finding ship types.

# Basic Constraints

* Mutual Exclusivity of Statuses at a Location: There are many statuses that should not be on the same location as each other. These include:
  + You cannot have a checked status on an *Unchecked* proposition.
    - StatusCxy(Unchecked(x, y) → ¬Checked(x, y, StatusC))
  + You cannot have a checked or unchecked status on a boundary.
    - StatusUxy(Boundary(x, y) → ¬Unchecked(x, y, StatusU))
    - StatusCxy(Boundary(x, y) → ¬Checked(x, y, StatusC))
  + Other miscellaneous ones are you cannot have a miss and a hit at the same spot and you cannot have unlikely, possible, and highly possible segments on a hit or miss.
    - xy(Hit(x, y) → ¬Miss(x,y))
    - etc…

Finding Segments of Different Possibilities

* Finding Unlikely Segments: In the code, this constraint is not grouped with the two constraints below, this is because unlikely segments rely on propositions that are in a different function so reading the code flows better when it’s kept there. Unlikely segments are the last thing that the model will attempt to hit and they are located right next to a *Possible Sunk* location.
  + xy(UnlikelySegment(x, y) → PossibleSunk(x+1, y) PossibleSunk(x-1, y) PossibleSunk(x, y+1) PossibleSunk(x, y-1))
* Finding Possible Segments: A possible segment is next to a hit location
  + xy(PossibleSegment(x, y) → Hit(x+1, y) Hit(x-1, y) Hit(x, y+1) Hit(x, y-1))
* Finding Highly Possible Segments: Highly possible segments are whenever there are two hits in a row, meaning that it is likely that there is a third in that row.
  + xy(HPossibleSegment(x, y) → (Hit(x+1, y) Hit(x+2, y)) (Hit(x-1, y) Hit(x-2, y)) (Hit(x, y+1) Hit(x, y+2)) (Hit(x, y-1) Hit(x, y-2))

Finding Sunk Ships

* Complete Location: Surrounded by hits, misses, and boundaries. In the code we say that:
  + rightSpot ⊢ xy(Hit(x+1, y) Miss(x+1, y) Boundary(x+1, y)), etc. which allows for the simpler constraint:
    - xy(Complete(x, y) → rightSpot leftSpot upSpot downSpot Hit(x,y)

These next two constraints are very long if written formally, so I will describe them instead with an example. Please view the findSunkShips() function to see the full constraints written out.

* Certain Sunk: There is a *Possible Sunk* and a *Sunk* so I’ve named this constraint “certain sunk” to signify that the model is certain of a ship being sunk here. A *Complete* location with a *Complete* location in any direction, with a *Complete*, *Miss*, or *Boundary* two spaces away is considered a *Sunk* location.
* Cornered Hit: A *Cornered Hit* is a *Hit* location surrounded by exactly 1 *Hit* and 3 of either a *Miss* or *Boundary*. One example of this would be:
  + xy(CorneredHit(x, y) → Hit(x, y) Boundary(x-1, y) Boundary(x, y-1) Miss(x, y+1) Hit(x+1, y))
* Possible Sunk: Some location that is *Hit* and is either adjacent to some *Cornered Hit* or is a *Cornered Hit*:
  + xy(PossibleSunk(x, y) → Hit(x, y) (adjacentCorner CorneredHit(x, y)))
  + Where: adjacentCorner ⊢ xy(CorneredHit(x+1, y) CorneredHit(x-1, y) CorneredHit(x, y+1) CorneredHit(x, y-1))

Finding Ship Types

These constraints are also very long, so I will just be giving one example of each

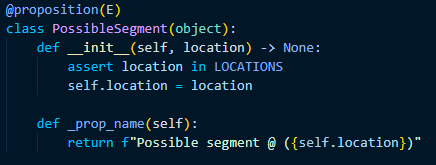
* Finding Subs: A *Sunk* location with a *Sunk* location next to it followed by not a *Sunk* location is a submarine. Ex:
  + xy(Ship(x, y, sub) → Sunk(x, y) Sunk(x+1, y) ¬Sunk(x+2, y)
* Finding Destroyers: A *Sunk* location followed by 2 *Sunk* locations is a destroyer. Ex:
  + xy(Ship(x, y, des) → Sunk(x, y) Sunk(x+1, y) Sunk(x+2, y)

# **MODEL EXPLORATION**

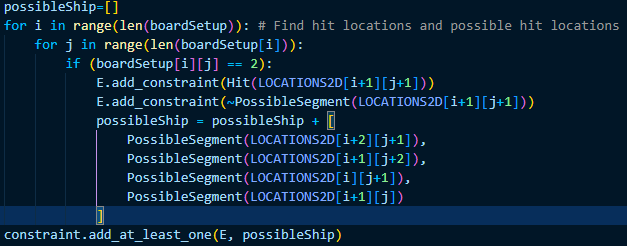
Since the draft, we have realised that we changed what we previously wrote for model exploration. Instead of removing them we decided that it might be more interesting to look at where things have changed since the draft. Items that were originally in the draft document will be followed by (DRAFT).

## Addition of the PossibleSegment proposition (DRAFT)

We were originally just going to use hit/miss/ship statuses for each location on the board but we realised that PossibleSegment was necessary for the computer to “decide” where ship segments are located.

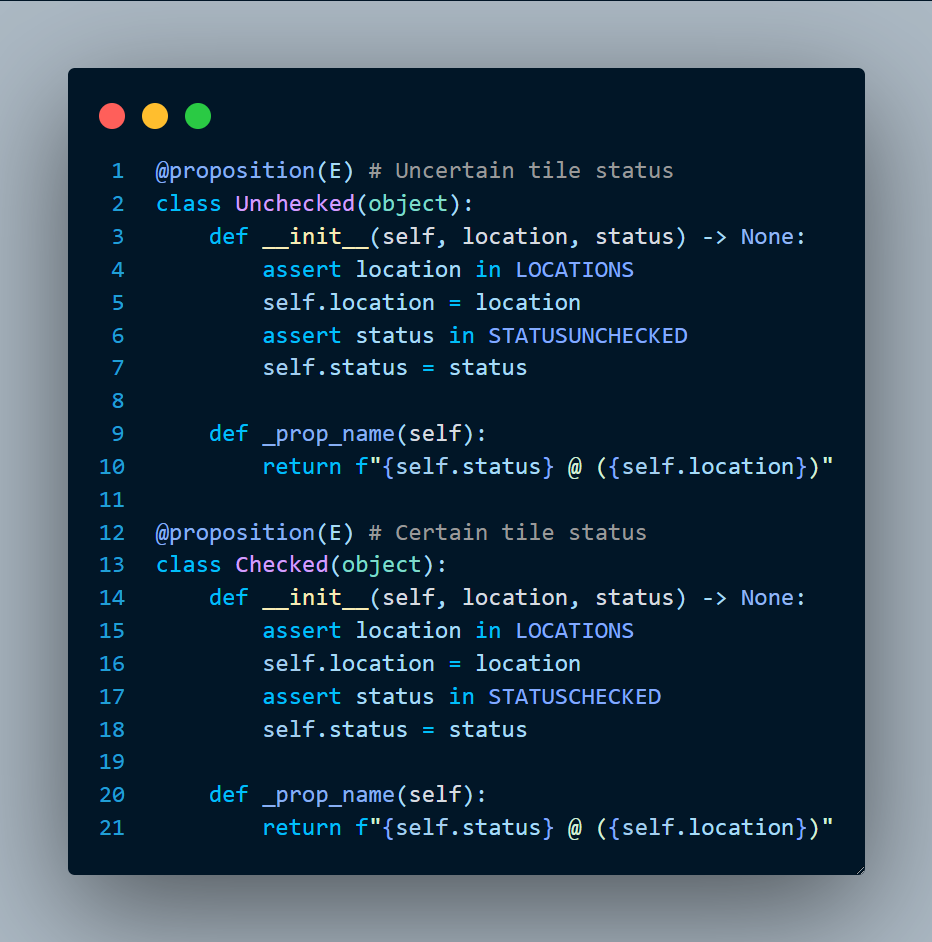


While Hit is used for showing the current state on the board, PossibleSegment is different in that it’s the way that the model decides what to do next.

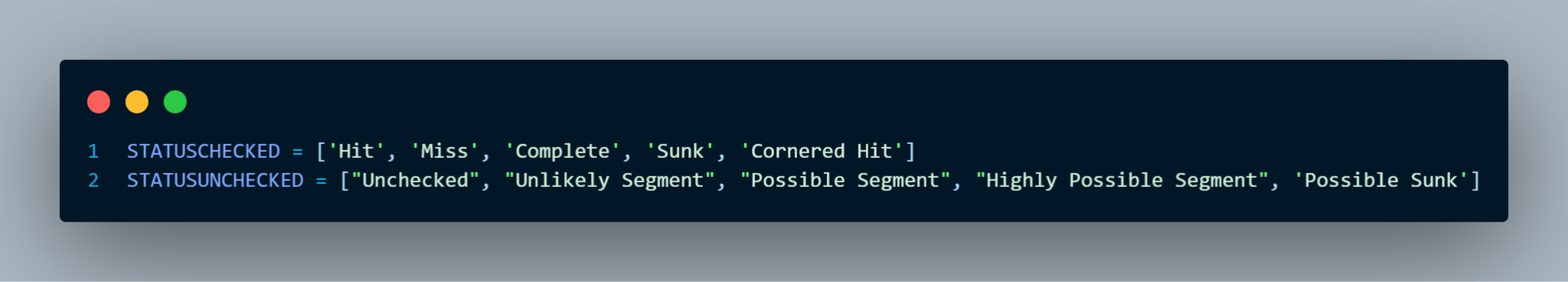


Rework of most propositions (FINAL)

As we worked through implementing constraints we realised the need for more and more propositions. Instead of writing out a proposition class for each individual proposition, we found that it was much more organised to group up propositions that were similar to each other.

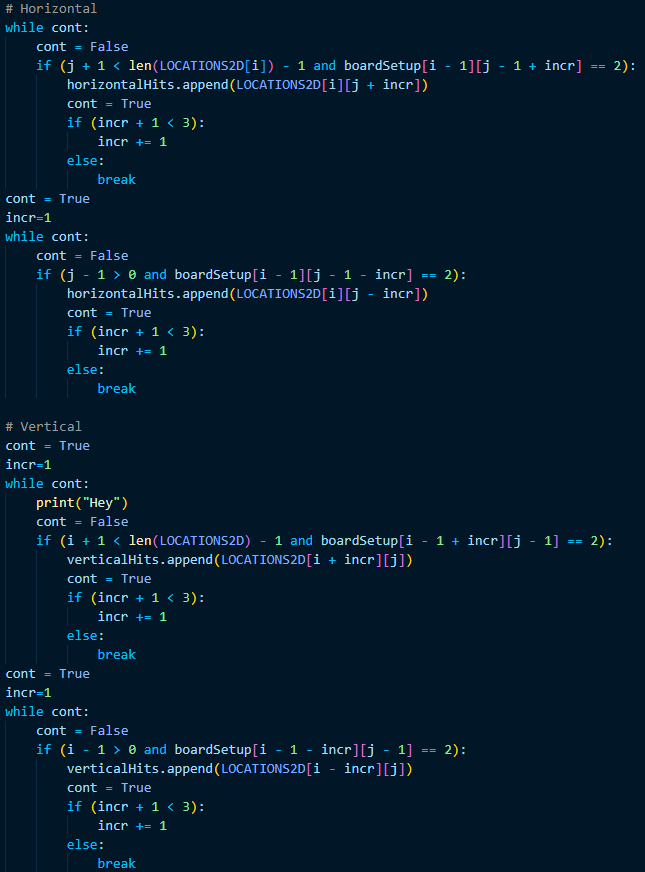


These two proposition classes had a status variable which differentiated each unique proposition, this led to the propositions part of the code being more organised and made it easy to add new propositions.



Implementing the FindShipType() function (DRAFT)

As we worked through implementing the model, we realised that we needed some way to distinguish between ships on the board. This led to us creating the findShipType function. This quickly became much more complicated than we expected which led us to using many while loops and if statements to decide the type of ship instead of logic methods.

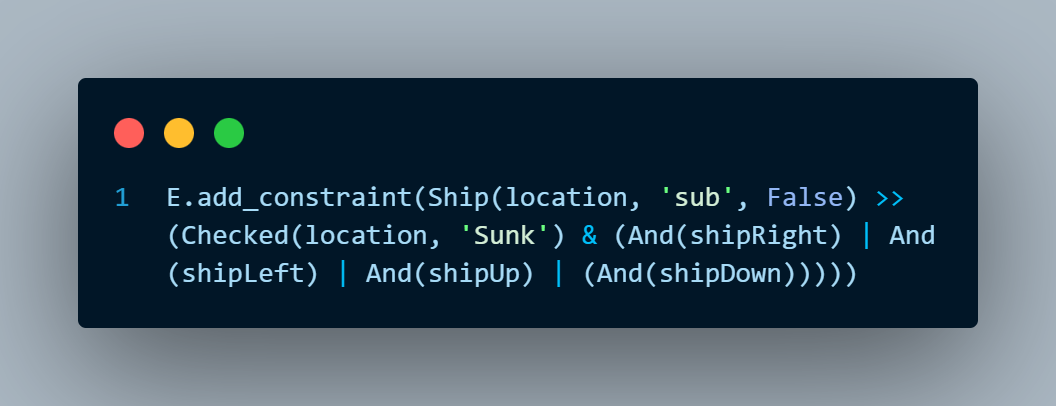


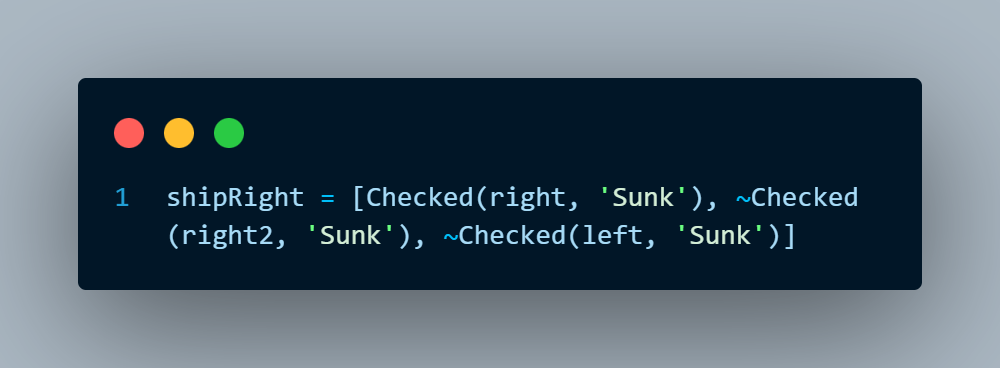
At the end of the function a simple constraint is set including the location and type. The function works correctly for simple cases, but for something more complex like ships in a “T” shape, it fails to differentiate between them. We also realised just how inefficient this code is, and it will be improved later.

## 

Changing the FindShipType() function (FINAL)

We noticed after a little while that the FindShipType() function wasn’t making full use of bauhaus and was relying on plain python for the majority of it. We changed this so that constraints were used instead to solve what type of ship was sunk.

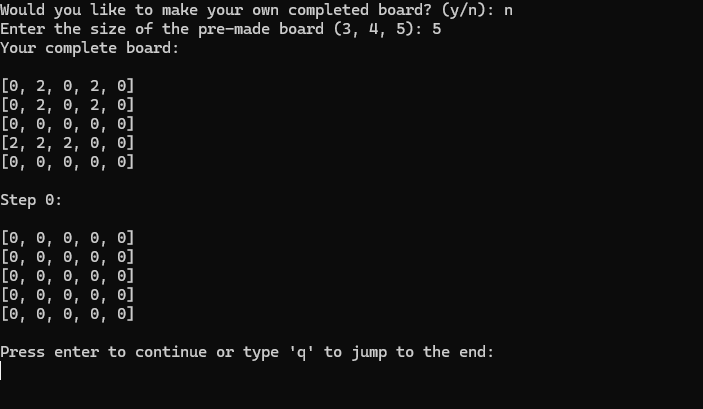




This is just an example of a constraint in FindShipType(), there are many more for different situations.

In-depth Model Testing

Since our model was working with so many possibilities, we knew that we needed some way to test a bunch of different battleship scenarios. This is where the CompleteBoards.py file and GetUserBoard function came in handy. Since at the start of the program we made the model read off of a battleship board (2D list), it was easy to make some features that made it really easy to test.

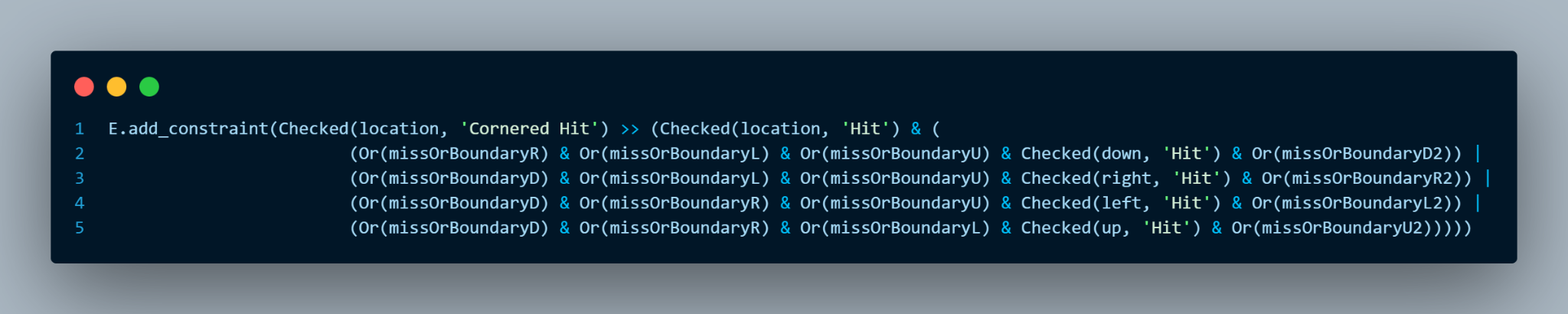


The user can say whether they want a pre-made board, and if that’s the case it will randomly grab one from the CompleteBoards.py file. We asked ChatGPT to create 10 boards sized 3x3, 4x4, and 5x5 for this file and could easily ask for more if it’s needed. In the case where the user wants a specific setup to test something, they can create their own which helped us in solving many issues.



Model Efficiency Issues

One of the largest issues we faced was that the model would not think similarly to a human and thus be very bad at figuring out where a ship would be. The biggest one that we decided to solve was when a ship was clearly sunk the model would continue to hit around the hit markers. In the image to the right, the model would have shot at the bottom right corner, even though there is clearly no ship there. The three propositions that made this possible were the “Cornered Hit”, “Possible Sunk”, and “Unlikely Segment”. The biggest constraint to implement was finding where a cornered hit was. In the above scenario it would be the hit in the middle of the board since this hit is “cornered” between a hit and misses.



A cornered hit basically meant that it and any hit adjacent to it would be a “Possible Sunk” which meant that any spot adjacent to a “Possible Sunk” would be an “Unlikely Segment”. Since there is still a possibility that there is a ship there, the spot isn’t completely removed from the choices that the model has, but it has a very low priority (similar to how we would think). With this the model is much smarter than before and can solve the game at a comparable level to a human.

# **JAPE PROOF IDEAS**

1. Adjacency and Ship Type Identification

A submarine is a ship with a size of 2. To confirm that a sequence of two consecutive hits form a submarine, the model verifies the following:

* The hits are adjacent (either horizontally or vertically).
* There are no misses at the same locations.
* The locations follow ship type exclusivity rules, ensuring that a submarine cannot overlap with any other ship.

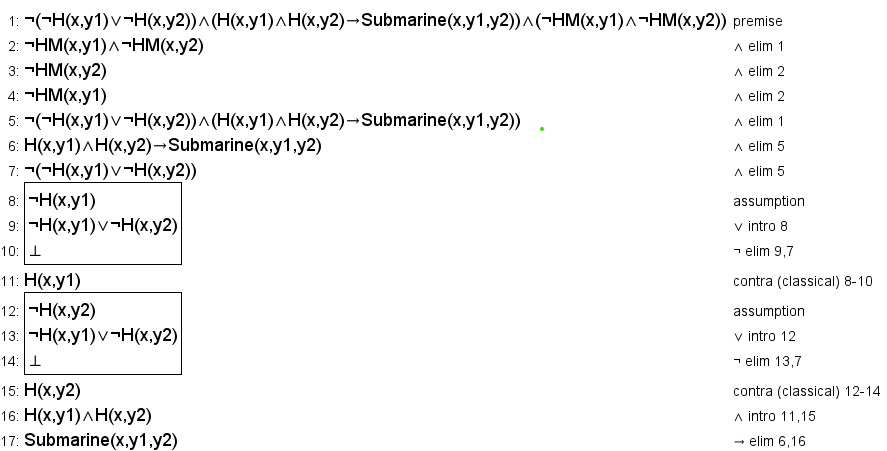
The proof aims to demonstrate that if there are two hits at adjacent locations (H(x,y1) and H(x,y2), they fulfill the conditions to be identified as a submarine.

#### Propositions:

* H(x,y): There is a "hit" at position (x,y).
* HM(x,y): There is a "miss" at position (x,y).
* Submarine(x,y1,y2): The two locations form a submarine.

#### Variables:

* x,y1,y2​: Coordinates on the game board.
* H(x,y1): A hit at position (x,y1).
* HM(x,y): A miss at position (x,y).
* Submarine(x,y1,y2): Indicates a submarine is identified at adjacent positions.



1. Boundary and Hit Constraints

The model enforces constraints that boundary locations on the game board (B(x,y)) cannot also be hit (H(x,y)). This rule ensures that players cannot target invalid positions on the board. This proof aims to validate that if a position is marked as a boundary, it cannot simultaneously be a hit.

This proof confirms this by:

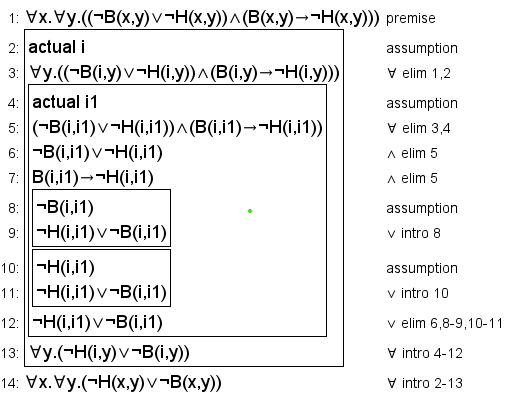
* Defining a boundary location (B(x,y)).
* Ensuring that the boundary location and hit are mutually exclusive.
* Showing consistency with the game's logical constraints.

**Propositions:**

* B(x,y): The position (x,y) is a boundary.
* H(x,y): There is a hit at position (x,y).

**Variables:**

* x,y: Coordinates on the game board.
* B(x,y): The position (x,y) is a boundary.
* H(x,y): There is a hit at position (x,y).



1. Destroyer Formation Validation

A destroyer is a ship with a size of 3. This proof validates that three consecutive hits in a horizontal row (H(x1,y), H(x2,y), H(x3,y)) form a destroyer. The validation includes:

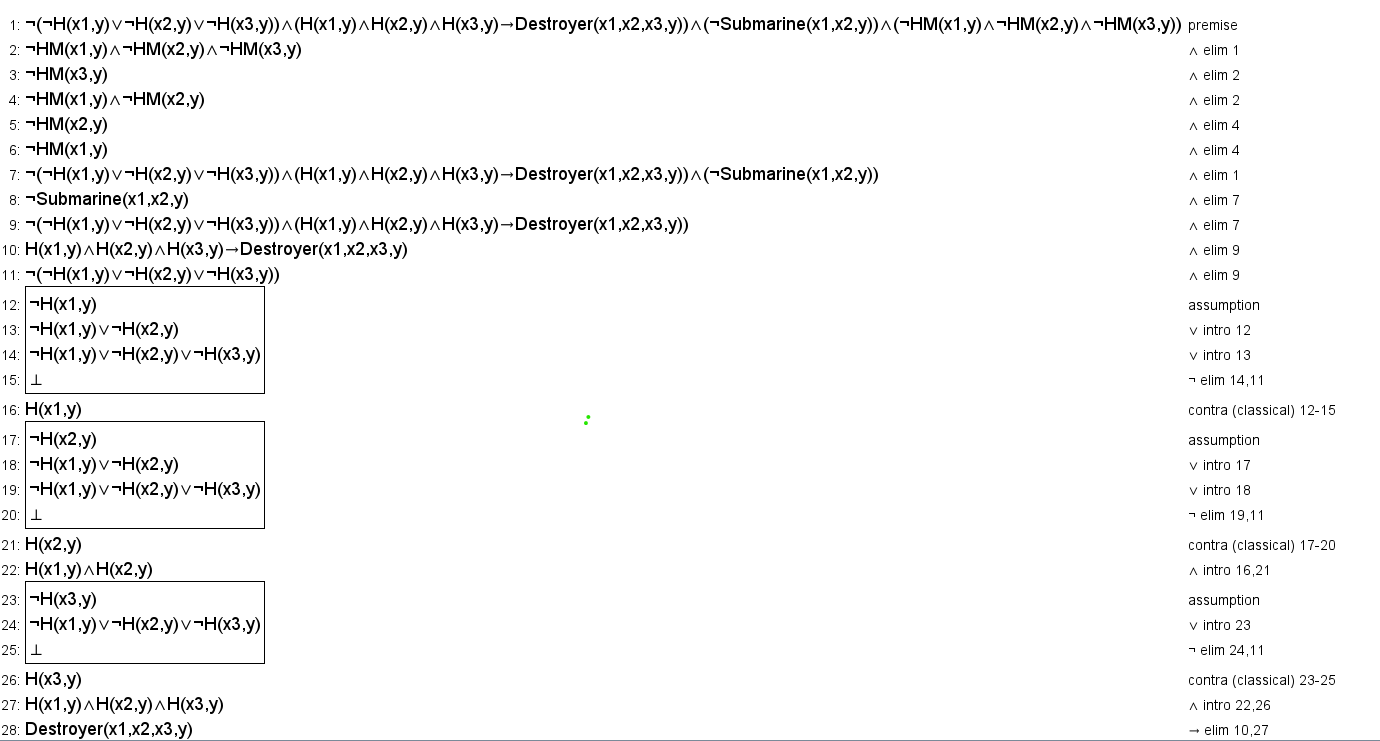
* Checking adjacency of the three hits.
* Ensuring the three hits cannot belong to other ship types (e.g., submarine or carrier).
* Verifying that no misses (M) occur at the same locations.

**Propositions:**

* H(x,y): There is a hit at position (x,y).
* HM(x,y): A miss at position (x,y).
* Destroyer(x1,x2,x3,y): The three locations form a destroyer.
* Submarine(x1,x2,y): Indicates a submarine at specific positions.

**Variables:**

* x1,x2,x3,y: Coordinates on the game board.
* H(x,y): There is a hit at position (x,y).
* HM(x,y): A miss at position (x,y).
* Destroyer(x1,x2,x3,y): The three locations form a destroyer.
* Submarine(x1,x2,y): Indicates a submarine at specific positions.



# FIRST-ORDER EXTENSION

## **Predicates and Types**

First, we'll define the predicates and types that correspond to the elements in our project:

### Predicates:

1. **Checked(l, s):** Location l has a certain status s (e.g., 'Hit', 'Miss', 'Complete', 'Sunk').
2. **Unchecked(l, s):** Location l has an uncertain status s (e.g., 'Unchecked', 'Possible Segment', 'Highly Possible Segment').
3. **Boundary(l):** Location l is a boundary of the board.
4. **Ship(l, t):** A ship of type t is at location l.
5. **Adjacent(l1, l2):** Locations l1 and l2 are adjacent (up, down, left, or right).
6. **Direction(l1, l2, d):** Location l2 is in direction d from l1 ('up', 'down', 'left', 'right').
7. **TypeSize(t, n):** Ship type t has size n.

### Types:

* **Loc(x):** x is a location.
* **StatusC(s):** s is a certain status (belongs to STATUSCHECKED).
* **StatusU(s):** s is an uncertain status (belongs to STATUSUNCHECKED).
* **SType(t):** t is a ship type (e.g., 'des', 'sub').
* **Direction(d):** d is a direction ('up', 'down', 'left', 'right').
* **Num(n):** n is a natural number.

### Constants:

* **STATUSCHECKED:** {'Hit', 'Miss', 'Complete', 'Sunk'}
* **STATUSUNCHECKED:** {'Unchecked', 'Possible Segment', 'Highly Possible Segment'}
* **STYPES:** {'des', 'sub'}
* **TypeSizes:** TypeSize('des', 3), TypeSize('sub', 2)

### Functions:

* **Neighbour(l, d):** The location adjacent to l in direction d.
* **Opposite(d):** The opposite direction of d (e.g., Opposite('up') = 'down').

In what remains, we go through some of the constraints that can now be represented in first-order logic.

Mutual Exclusivity of Statuses at a Location

In our project, we will ensure that a location cannot have conflicting statuses. For example, a location cannot be both 'Hit' and 'Miss'.

**First-Order Logic Representation:**

For all locations l:

1. **Hit vs. Miss:**
2. **Hit vs. Unchecked Statuses:**
3. **Miss vs. Unchecked Statuses:**
4. **Boundary Locations Cannot Have Checked or Unchecked Statuses:**
   1. For certain statuses:
   2. For uncertain statuses:
5. **Unchecked Locations Cannot Have Certain Statuses:**

**Explanation:**

* These constraints ensure that each location has a consistent status and that boundaries are not misclassified.
* They directly correspond to the constraints added in the code to prevent conflicting statuses.

Determining 'Complete' and 'Sunk' Ships

Determine if a 'Hit' location is 'Complete' or 'Sunk' based on the status of its surrounding locations.

**First-Order Logic Representation:**

1. **Complete Ship Segment:**

1. **Sunk Ship Segment:**

The determination of a 'Sunk' ship segment involves more complex conditions, checking for adjacent 'Complete' segments and ensuring that the ship is fully identified. For simplicity, we'll abstract this:

**Explanation:**

* **Complete:** A 'Hit' location is 'Complete' if all its adjacent locations are either 'Miss', 'Hit', or 'Boundary'.
* **Sunk:** A 'Complete' location is part of a 'Sunk' ship if it's connected to other 'Complete' or 'Sunk' segments, forming a ship of the appropriate size.

Adjacency and Board Structure

**Code Implementation:**

The code checks adjacent locations when determining possible segments, sunk ships, and ship types.

**First-Order Logic Representation:**

1. **Defining Adjacency:**
2. **Directions and Neighbors:**
   * For each direction d, the Neighbor function gives the adjacent location in that direction.

**Explanation:**

* Adjacency is a fundamental concept for modelling the Battleship game, as ships occupy consecutive locations.
* By defining adjacency and directions, we can express constraints involving neighbouring locations.

Possible Segments and Highly Possible Segments

**Possible Segment：**

**Possible Segment：**

**Explanation:**

* A 'Possible Segment' is a location adjacent to a 'Hit'.
* A 'Highly Possible Segment' is a location adjacent to two consecutive 'Hits' in the same direction.

'Cornered Hit' and 'Possible Sunk’

**Definition of 'Cornered Hit':**

A location is a 'Cornered Hit' if it is a 'Hit' and is surrounded by 'Miss' or 'Boundary' locations in all but one direction, and in that remaining direction, there's a 'Hit'.  
Constants:

**Definition of 'Possible Sunk':**

A location is a 'Possible Sunk' if it is a 'Hit' and is adjacent to a 'Cornered Hit' or is itself a 'Cornered Hit'.

Constants:

A location is an 'Unlikely Segment' if it is adjacent to a 'Possible Sunk' location

**Explanation:**

* **'Cornered Hit':** Represents a 'Hit' that is surrounded by 'Miss' or 'Boundary' locations except in one direction, suggesting the end of a ship.
* **'Possible Sunk':** Indicates that a ship may have been sunk, but it's not confirmed.
* **'Unlikely Segment':** Locations near a 'Possible Sunk' are less likely to contain a ship segment.

Identifying Ship Types

**Ship Type Identification for Certain Ships:**

**Ship Type Identification for Possible Ships:**

**Explanation:**

* **ShipPattern(l, t):** A predicate that is true if the arrangement of 'Sunk' locations around l matches the pattern of a ship type t (e.g., 'sub' or 'des').
* **PossibleShipPattern(l, t):** Similar to ShipPattern but for 'Possible Sunk' locations.
* Modeling ShipPattern and PossibleShipPattern requires expressing the conditions that define the shapes and sizes of ships based on adjacent statuses. This involves complex combinations of adjacent locations and can be quite intricate.
* For instance, a 'sub' (size 2) might be identified if there's a sequence of two 'Sunk' locations in a straight line without additional 'Sunk' locations extending beyond them.

Board Initialization Constraints

**Initial Statuses:**

For all locations l:

* If l is marked as 'Hit' in the initial setup:

If l is marked as 'Miss' in the initial setup:

* If l is not marked (i.e., 'Unchecked'):

**Boundary Locations:**

For all boundary locations l: