UppaalTD: a Formal Tower Defense Game

Formal Methods for Concurrent and Real-Time Systems

Andrea Torti
Lorenza D'Amario

April 2025

Contents

Chapter 1

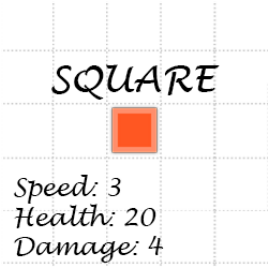
Introduction

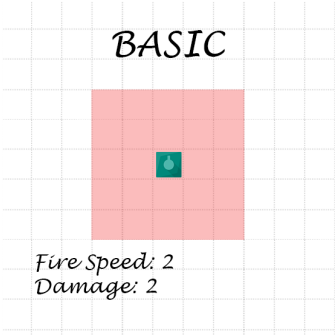
The objective of this project is to define and verify a formal model of a tower defense game using the UPPAAL modeling tool. UPPAAL is designed to represent systems as a collection of non-deterministic processes governed by real-valued clocks, synchronized through channels or shared variables. Its model checking engine is specifically built to verify a subset of TCTL (Timed Computation Tree Logic) formulas, enabling the formal analysis of time-critical behaviors and properties within the modeled system.

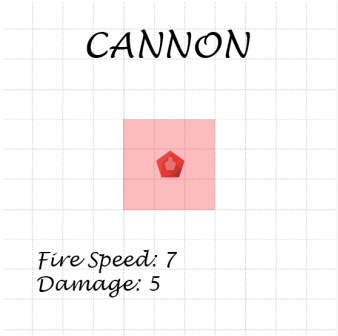
The TD game in question is modeled in two distinct ways: a version where enemy movement speed and turret firing speed are fixed to constant values, referred to as the *Vanilla version*, and a *Stochastic version*, where these two aspects are modeled using exponential distributions by leveraging UPPAAL's Statistical Model Checking (SMC) features.

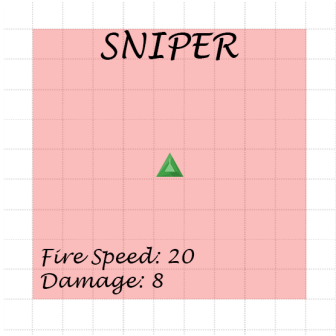
The game to be modeled features two types of enemies—*circle* and *square*—and three types of turrets—*basic*, *cannon*, and *sniper*—whose respective statistics are shown in the images below.

For a more detailed description of the game, you may take a look at the homework assignment pdf document.









Chapter 2

Model Overview

2.1 Templates List

- **GameTime:** This template is responsible for ending the game after the specified timeout and ensuring that the simulation continues afterward to prevent the UPPAAL engine from deadlocking.
- **Turret:** This template models the behavior of a single turret, including shooting, reloading, and scanning for enemies.
- **Enemy & EnemyP:** These two templates jointly represent a single enemy. `EnemyP` acts as the controller, managing the enemy's lifecycle and coordinating actions, while `Enemy` handles its movement through the map.

2.2 End Game Triggers

The end of the game is initiated by one of the following three distinct conditions:

1. No enemy remains *in game* — *win*.
2. The *timeout* expires — *win*.
3. The main tower's *life points* reach zero or below — *loss*.

Every event capable of fulfilling one of these conditions has been explicitly modeled to set the global boolean variable `end_game` to `true`:

- A turret successfully shoots an enemy whose demise leaves no enemies in play or pending spawn.
- The `GameTime` template detects that the timeout has elapsed.
- An enemy attacks the `main_tower` reducing its life points to zero or less.
- An enemy exits the map boundary when no other enemies remain alive or waiting to spawn.

2.3 Configurations

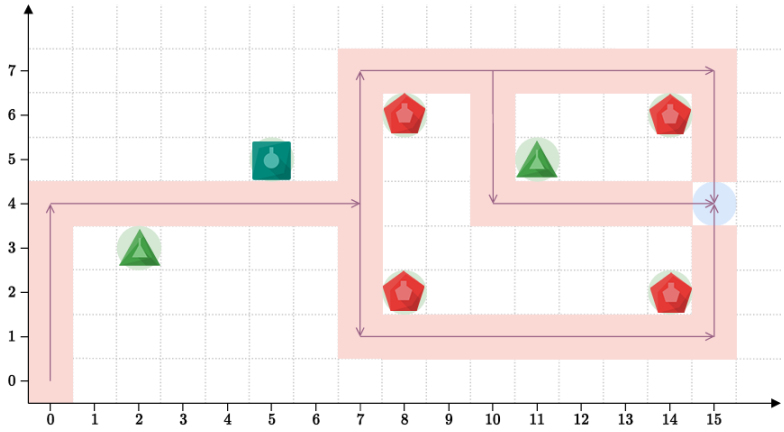


Figure 2.1: Configuration 1 – Given Example

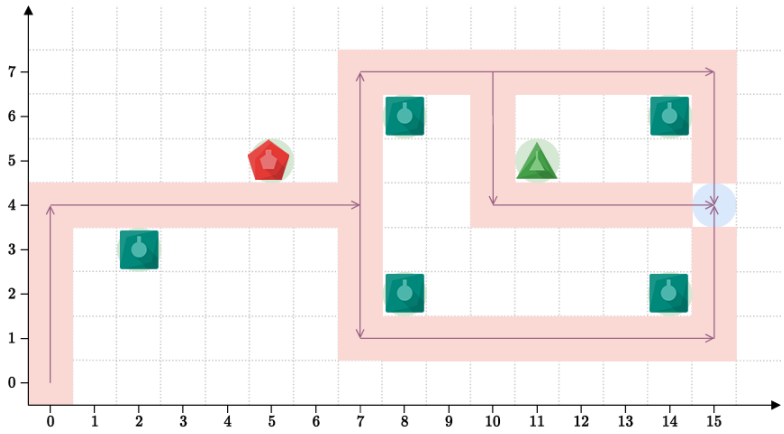


Figure 2.2: Configuration 2 – Better Turrets Placement

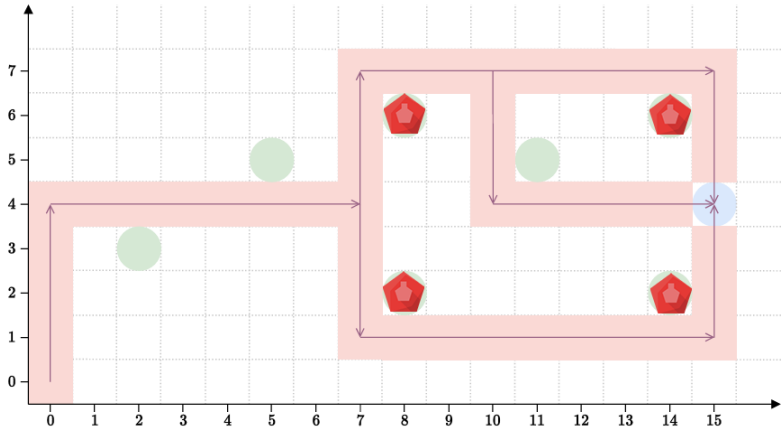
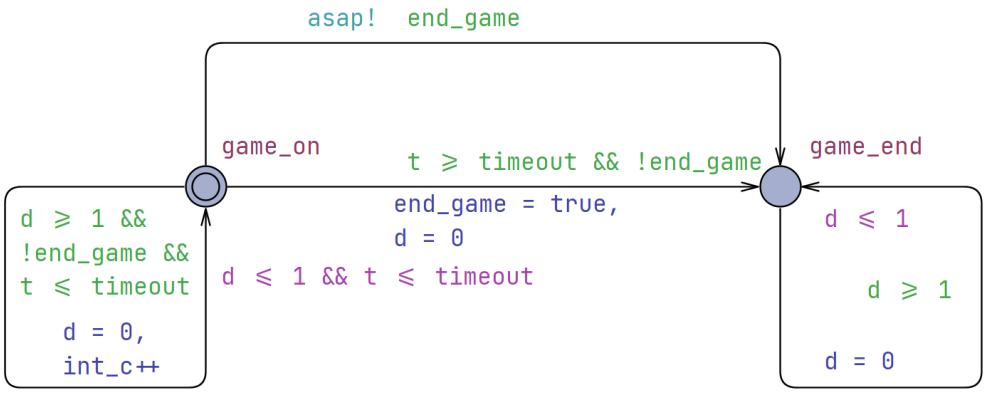


Figure 2.3: Configuration 3 – Worse Turrets Placement

Chapter 3

Templates

3.1 GameTime



The `GameTime` template always has a single instance and is used to end the game when a timeout expires, as well as to prevent UPPAAL from deadlocking once the end of the game is reached. Additionally, while the game is active, it maintains a global integer clock, incrementing its value by one every simulation second.

Its only variable is its own clock, which resets every second, and its only constant is the timeout. The timeout is set to allow the last enemy to reach the main tower in the Vanilla version of the game, while in the Stochastic version, it is set to 200 to ensure that both winning and losing configurations are possible.

This template will either wait for the `end_game` signal to be triggered externally in order to immediately transition to the `game_end` location, or it will transition on its own and set `end_game` to `true` if the timeout has expired.

3.2 Turret

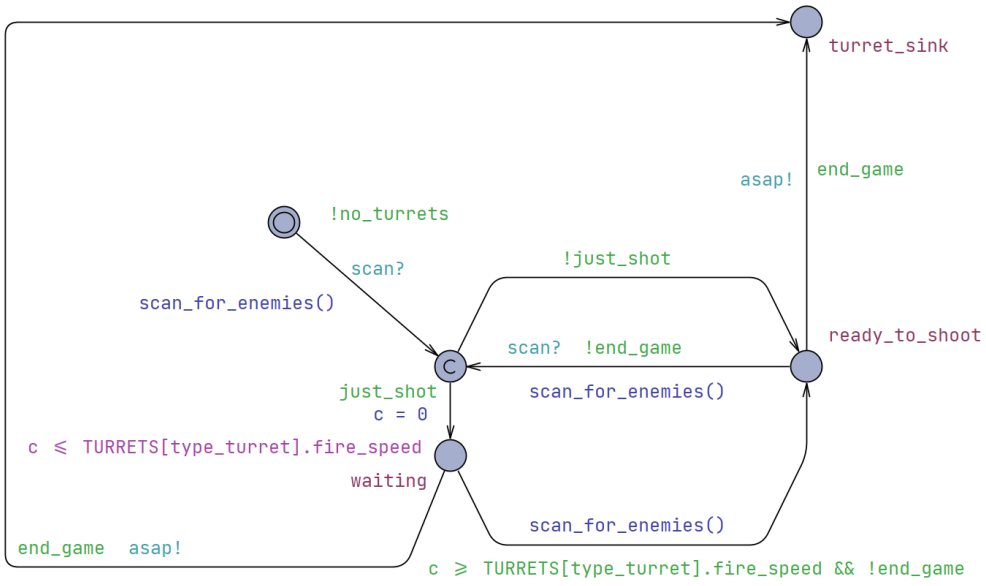


Figure 3.1: Vanilla Turret Template

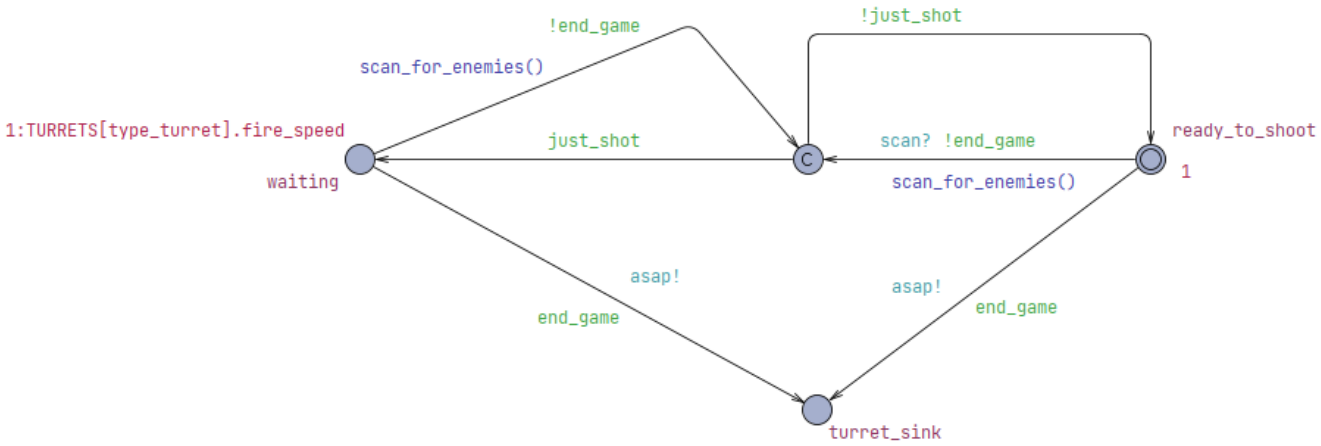


Figure 3.2: Stochastic Turret Template

The turret template represents a single turret in the game and handles its shooting and reloading timer.

It takes three input parameters: type (represented as an integer), and x and y coordinates. The turret uses its type parameter to retrieve its constant characteristics (reload speed, damage, and range) from a constant struct.

The boolean `just_shot` is set to `true` when the turret successfully shoots an enemy, and is reset after its cooldown timer (when the turret reloads its next shot).

The function `scan_for_enemies` is called when a scan is triggered (see next paragraph), while the function `bang` is called within it if a valid target is found and is used to shoot it, taking the enemy

ID to shoot as input. These functions handle shooting enemies, calculating their updated health values, and selecting which enemy to target based on the specified priority rules.

Its scan broadcast channel is triggered from the `EnemyP` template when an enemy either spawns or moves. The turret also autonomously scans as soon as it finishes reloading.

The turret template may trigger the end of the game (by setting the `end_game` variable to `true`) in the `bang` function if the enemy it shoots dies and is also the last enemy in the game (i.e., no more enemies are alive or able to spawn).

3.3 Enemy & EnemyP

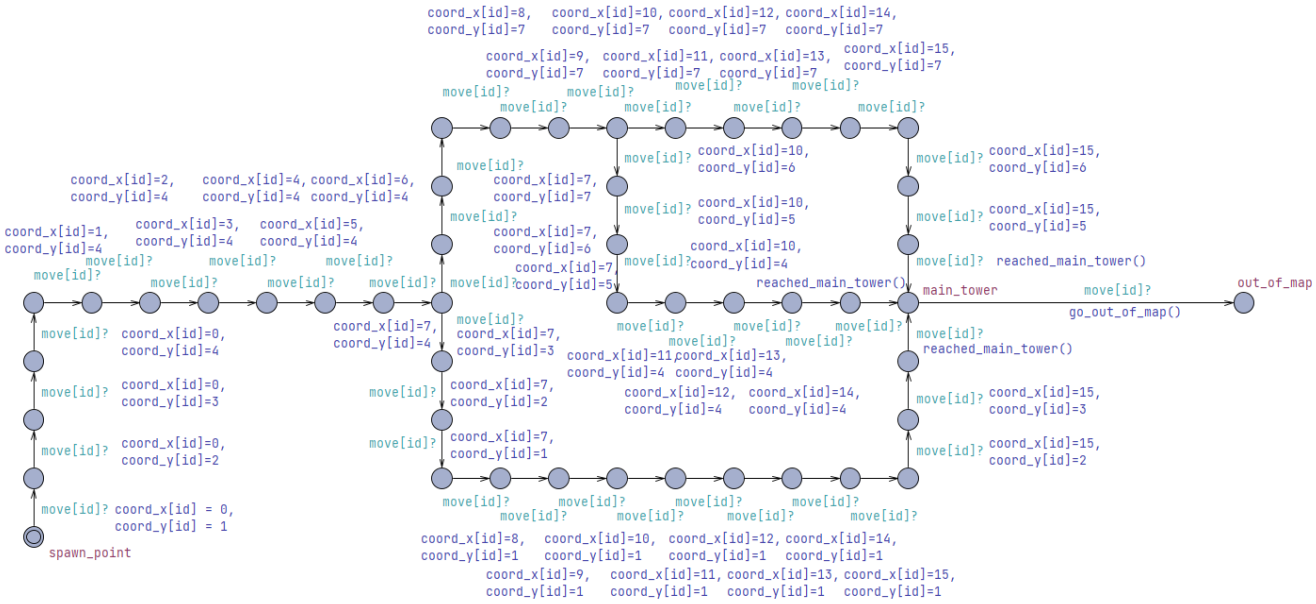


Figure 3.3: Vanilla Enemy Template

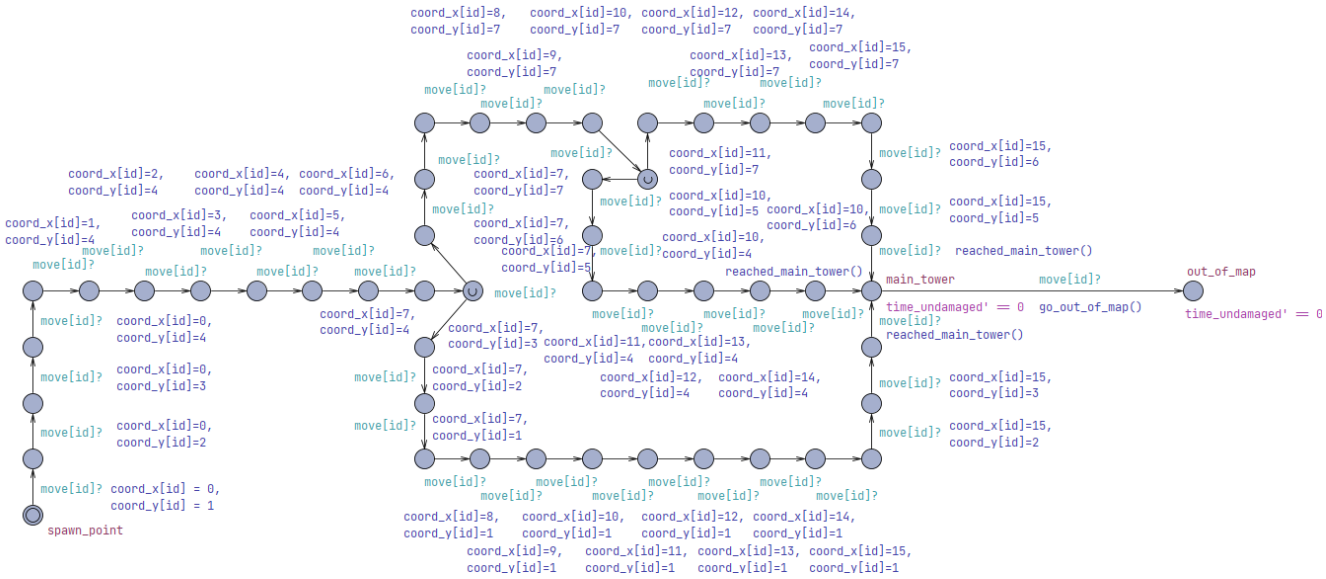


Figure 3.4: Stochastic Enemy Template

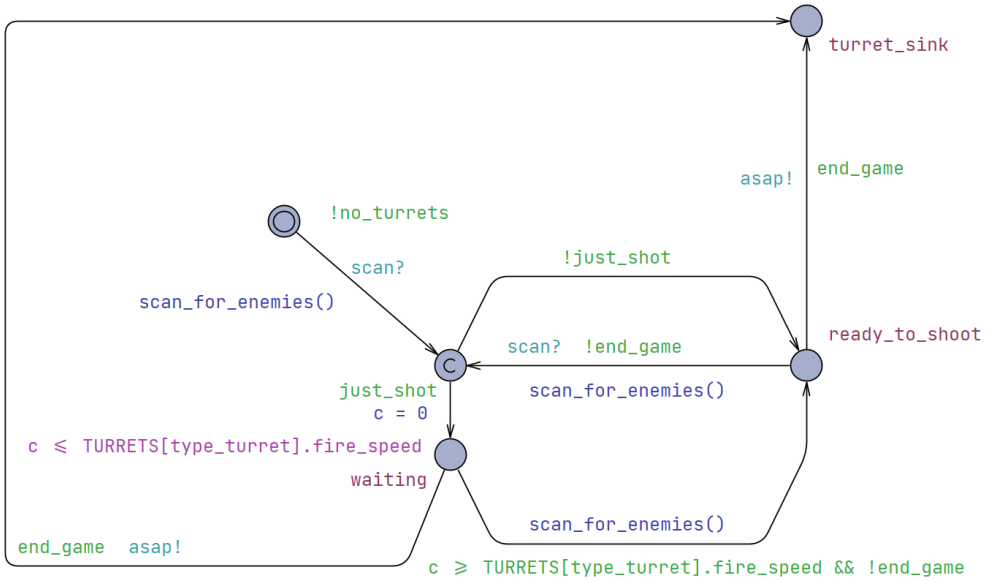


Figure 3.5: Vanilla EnemyP Template

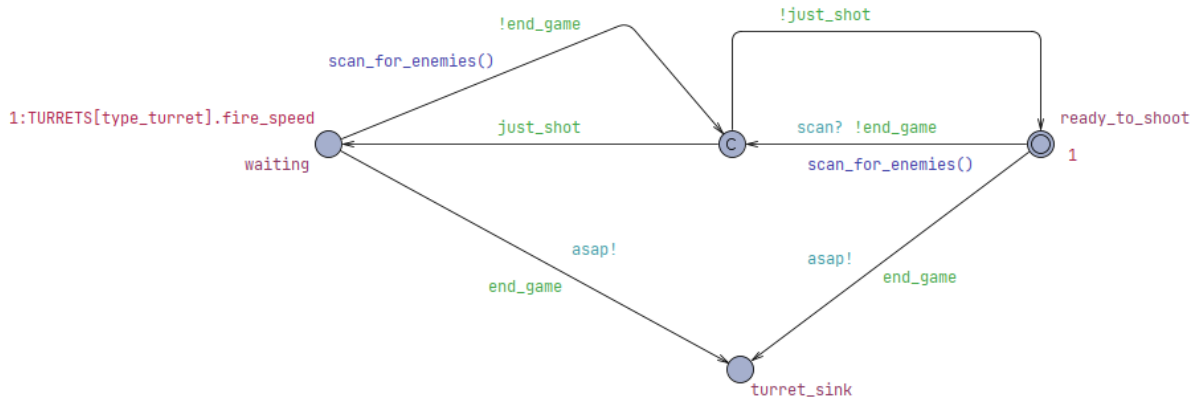


Figure 3.6: Stochastic EnemyP Template

The `Enemy` and `EnemyP` templates define the movement logic of the enemies. Each pair of instances represents a single enemy. Synchronization between the two is achieved through the channel `move[id]`. The ID of each enemy is known to both templates, as it is passed as a parameter.

EnemyP: The `EnemyP` template manages the life cycle of each enemy.

It takes three input parameters: `id`, `type` (represented as an integer), and `serial_number`. The `EnemyP` template uses the `id` for synchronization; the `type` is used to retrieve the enemy's constant characteristics (map coordinates, health, spawn delay) from a constant struct; the `serial_number` is used to determine the birth time as `serial_number*spawn_delay`.

A local clock `c` is used within the template to control enemy movement in the Vanilla version of the game.

The `EnemyP` template triggers the `scan` broadcast channel when an enemy becomes alive and whenever it moves (triggering `move[id]`). The `scan` channel informs the turrets that there has been a change in enemy distribution on the map.

When an enemy dies (i.e., its health drops to or below zero) or reaches the `out_of_map` location, `alive[id]` is set to `false` and the enemy can no longer move. The only transition available then leads to the location `outside_or_dead_or_game_end`.

`EnemyP` reads the `end_game` variable but never updates it.

Enemy: The `Enemy` template handles the actual movement of an enemy on the map. Movement is linked with the update of the enemy's coordinates (`x` and `y`), and it also manages path selection at branching points. The path choice is non-deterministic, in accordance with the specification: in Uppaal, if multiple transitions are enabled at a given instant, only one is taken, selected non-deterministically. In the Stochastic Version, the input channel `move[id]` triggers a transition to an `urgent` location, which then proceeds non-deterministically toward one of the available paths with equal probability. This modeling choice is necessary because the SMC simulator in UPPAAL does not support non-deterministic input directly.

It takes two input parameters: `id` and `type` (represented as an integer). The `Enemy` template uses `id` for synchronization, and the `type` (square or circle, 0 or 1) to retrieve the constant `damage`.

Two functions are implemented in the template:

- `reached_main_tower`: Triggered when the enemy enters the `main_tower` location. It reduces

the main tower's life points by the amount of damage inflicted by that enemy. If this causes `main_tower_life_points` to drop to or below zero, it sets the global variable `end_game` to `true`.

- `go_out_of_map`: Called when the enemy exits the map. It effectively removes the enemy from the game. If no enemies remain active (i.e., `in_game[id]` is `false` for all `id`s), this function also sets `end_game` to `true`.

Chapter 4

Design Choices

- To force transitions to occur as soon as their guards are satisfied, two approaches are employed interchangeably: the use of an urgent broadcast channel named `asap`, and the application of tight timing constraints on both the source location and the transition itself. The latter approach is sometimes infeasible, particularly when the transition depends on external changes to boolean variables.
- Templates that require specific variable initializations perform them by invoking an `initialize()` function during their first transition. This approach allows all relevant variables—especially those that can be derived from the enemy or turret type—to be set internally, thereby reducing the number of parameters that need to be explicitly passed to the template.
- To prevent the UPPAAL engine from entering a deadlock state upon reaching the end of the game, an infinite self-loop has been added to the `game_end` location in the `GameTime` template, as it is guaranteed to have only one instance.
- The `in_game` and `alive` variables are maintained separately for each enemy. `in_game` is set to `true` at initialization and is used to determine game completion in the "all enemies killed" condition, while `alive` becomes `true` only upon the enemy's spawn and governs whether it can be targeted by turrets.
- The `committed` keyword is occasionally preferred over `urgent` for locations to reduce the number of possible execution paths, as it eliminates unnecessary interleavings.
- A boolean variable, `no_turrets`, is introduced in the Vanilla version to disable turret behavior entirely, preventing them from initializing.
- A Python script is employed to automatically generate the system declaration for the 700 enemies present in the Stochastic version of the model.
- **Note:** In the Vanilla version, the order in which enemies move at a given time instant is nondeterministic. Consequently, even if turret targeting priorities are designed to favor square enemies or those with lower serial numbers, a circle enemy that moves first may trigger a scan and be targeted immediately—consuming the turret's shot before a higher-priority enemy is detected. A straightforward solution would be to model turret scans to self-trigger every

second, but with a slight delay (e.g., 0.01) relative to enemy movement. This would ensure compliance with the targeting priority requirement. However, it would violate the constraint that turrets must fire instantly as soon as an enemy enters their range. This compromise has been adopted to maintain closer alignment and greater consistency between the Vanilla and Stochastic versions of the model, thereby improving overall legibility.

Chapter 5

Verification Results

5.1 Vanilla Version

Two main classes of verifications have been performed in the Vanilla version: with turrets enabled and with turrets disabled. The properties verified in each case are outlined below.

5.1.1 Without Turrets

- **Absence of Deadlock:** The system is verified to never reach a deadlock state during all executions without turrets.
- **All Enemies Reach the Main Tower in All Executions:** Each enemy is confirmed to reach the Main Tower. This is verified by checking that there exists at least a state where for every enemy i the variable `main_tower_reached_time[i]` is greater than 0. Since turrets are disabled no enemy is killed. Moreover, the timeout is set long enough in order for all enemies to have time to reach the Main Tower.
- **Enemies Reach the Main Tower Within Expected Time Bounds:** It is verified that each enemy reaches the Main Tower within a maximum expected time. For every enemy, the variable `main_tower_reached_time[i]` is required to be less than or equal to:

  $$n \cdot s \quad \text{for square enemies,} \quad n \cdot c \quad \text{for circle enemies}$$

  where n is the length of the longest path from the start of the Map to the Main Tower, and s and c are the movement speeds of square and circle enemies, respectively.
- **Enemies Never Leave the Red Path:** It is verified that, for every execution and at every simulation step, every enemy's coordinates remain within the set of positions that define the red path. This ensures that no enemy ever deviates from the intended route.

5.1.2 With Turrets

- **Determine Whether the Configurations are Winning or Losing:** A configuration is considered *winning* if, in all executions, the variable `main_tower_life_points` remains

strictly positive until the end of the game. Conversely, a *losing* configuration is one in which this variable drops to zero or below in all executions.

While configurations 1 and 2 consistently result in a win across all executions, configuration 3 is non-deterministic in outcome: it has been verified that there exists at least one execution in which the main tower survives (a win), and at least one in which it is destroyed (a loss).

- **Absence of Deadlock:** The system is verified to never reach a deadlock state during all executions under configuration 1 with turrets enabled.

5.2 Stochastic Version

In the Stochastic version, additional utility variables are introduced solely for verification purposes. The queries use UPPAAL SMC's statistical model checking capabilities, which are tailored to the stochastic interpretation of timed automata.

- **Estimate How Long the Main Tower Remains Undamaged:** The `E[]` evaluation is used to estimate the expected time until the main tower first takes damage. The variable `time_undamaged` is set to a non-zero value the moment the first enemy reaches and damages the tower. The simulation is run with a timeout of 200 time units under configuration 1.
- **Determine the Probability That the Main Tower Survives:** The `Pr[]` probability query is used to compute the chance that the tower survives (i.e., its life points remain positive) until the 200 time unit timeout. Two variables are used in the query: `main_tower_life_points` ≥ 0 and $t \geq 200$. Since the tower's life points are initially positive, the time condition is necessary to ensure the property is only evaluated at the end of the simulation. The result obtained is that in about ten percent of the executions, the Main Tower remains undamaged within 200 seconds.
- **Simulate and Analyze Different Configurations:** A set of simulations is run for three configurations, each for 220 time units and a fixed number of iterations. Five variables are tracked during simulation:
  - `num_enemies_alive`: the number of enemies moving on the map at each time unit.
  - `killed_by_turrets`: the number of enemies eliminated by turrets.
  - `attacked_main_tower`: the number of enemies that reached and attacked the Main Tower.
  - `exited_map`: the number of enemies that, after attacking the Main Tower, survived and exited the Map.
  - `main_tower_life_points`: tracks the remaining life points of the main tower.

  Although the simulation runs for 220 time units, all variable values stop changing after 200 due to the game's timeout.

  The results from the three configurations show:

  - The *example* configuration leads to a mix of wins and losses.

– The *improved turret placement* configuration results in a higher probability of winning.

– The *weaker placement* configuration results in a higher number of losses.

Across all simulations, the Main Tower's life points show a steady decline—supporting the idea that the timeout of 200 time units is well-calibrated for a fair challenge.

The number of alive enemies tends to increase over time, suggesting that turrets struggle to keep up with enemy spawning. However, the `killed_by_turrets` metric also steadily increases, indicating that the turrets are working, just not fast enough to prevent build-up.

Interestingly, the value of `attacked_main_tower` sometimes exceeds that of `exited_map`, indicating that some enemies are killed after damaging the tower but before exiting the map—highlighting that turret intervention can occur post-attack.

Finally, the estimated value from the query[Figure 5.4] measuring how long the tower remains undamaged corresponds well with the moment damage first occurs in the simulation [Figure 5.1], demonstrating coherence between theoretical and observed behaviors.

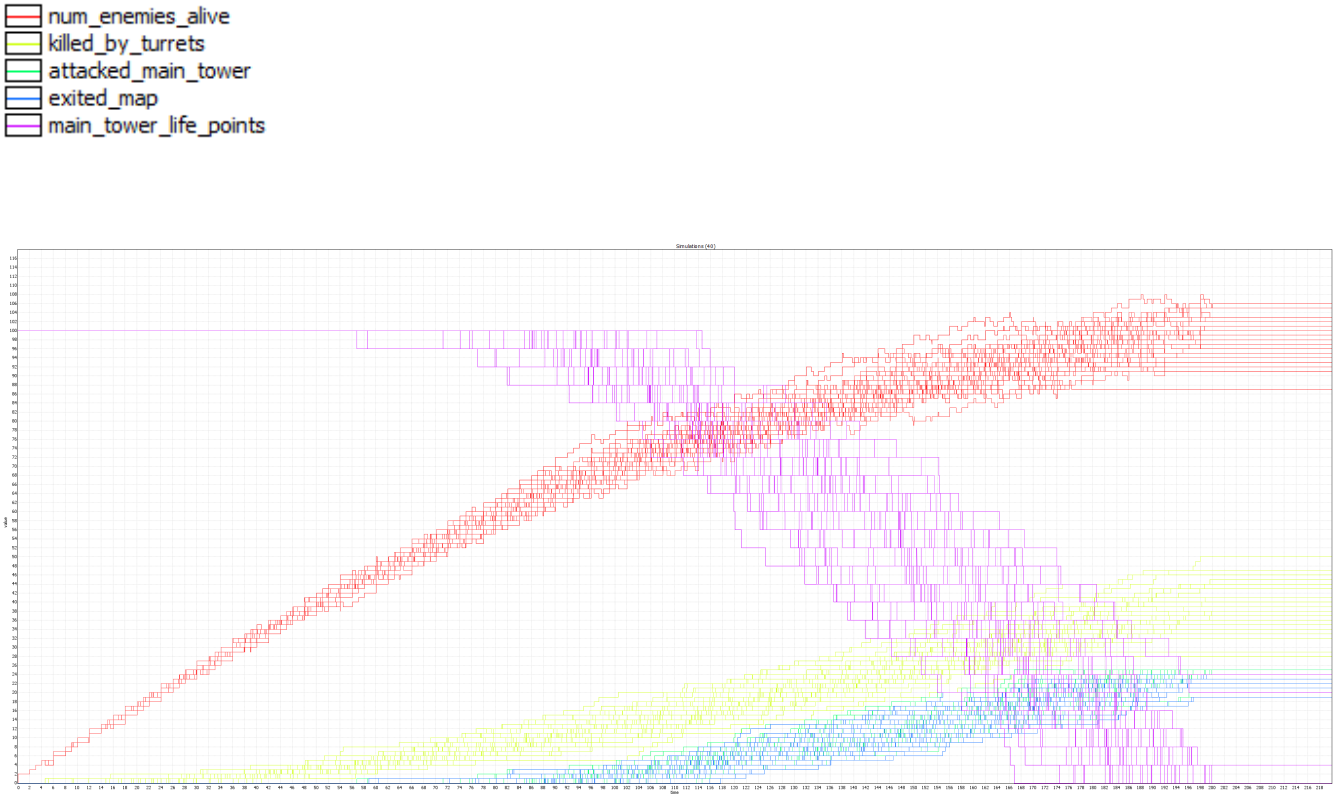


Figure 5.1: Example 1 - Given Example

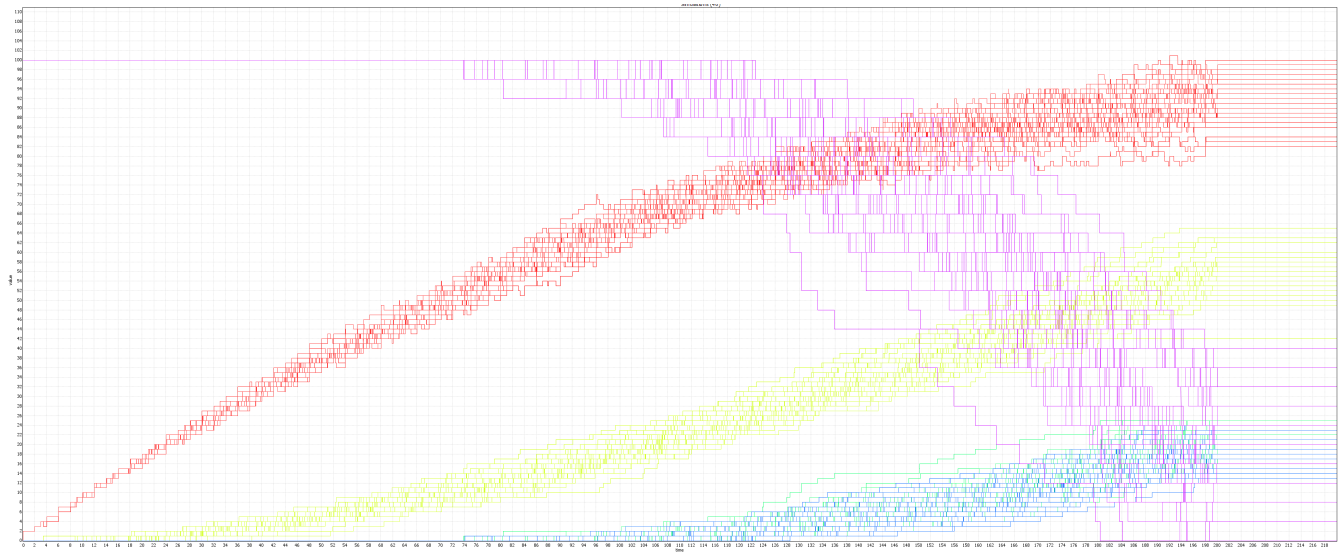


Figure 5.2: Example 2 - Better Turrets Placement

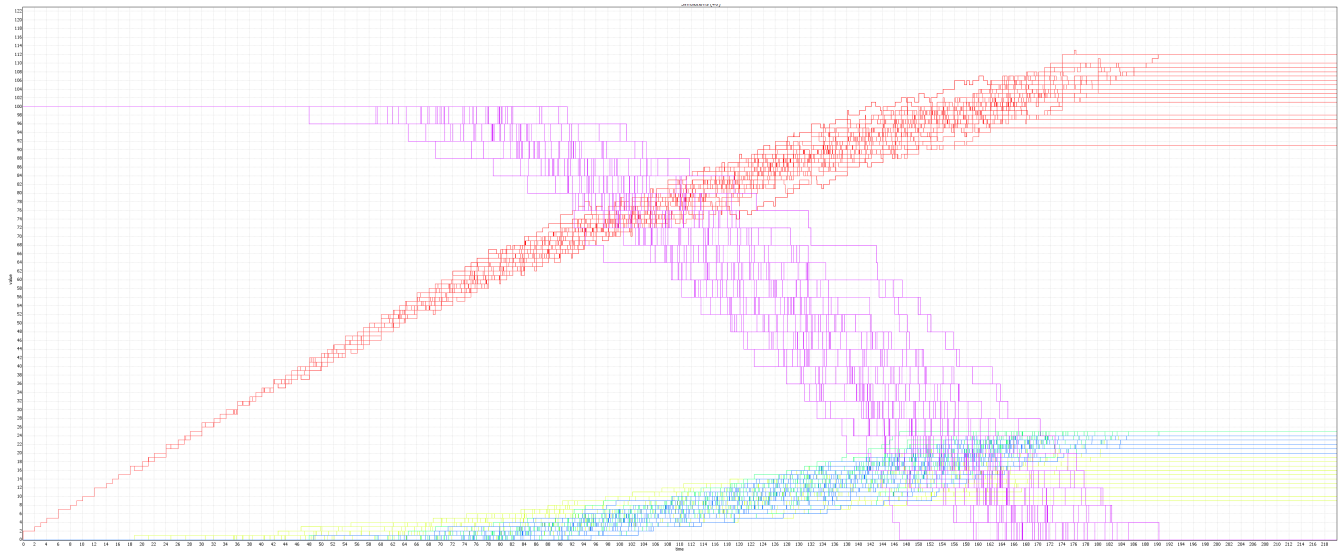


Figure 5.3: Example 3 - Worse Turrets Placement

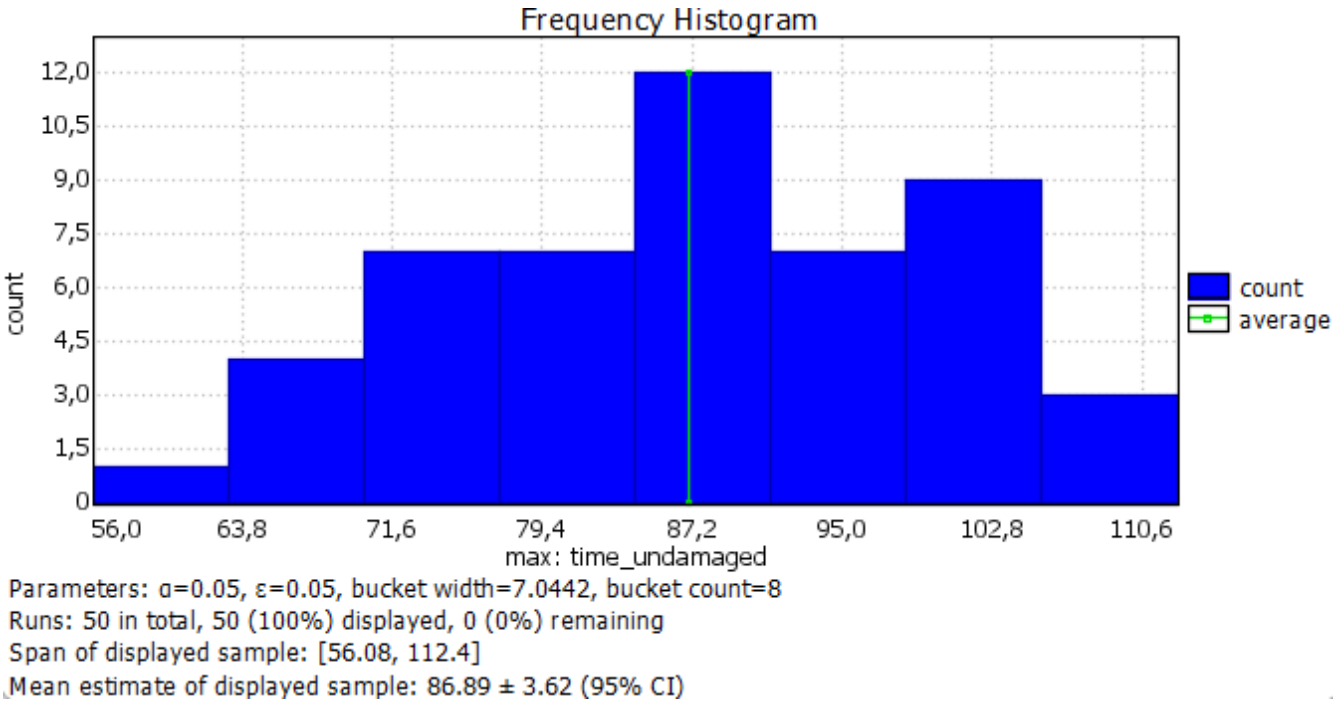


Figure 5.4: Frequency Histogram - configuration 1