UPPAALTD: A FORMAL TOWER DEFENSE GAME

Formal Methods for Concurrent and Real-Time Systems Homework

ANDREA BELLANI (andrea1.bellani@mail.polimi.it)

July 11, 2025

ABSTRACT Uppaal is a tool for modeling, simulating and verifying real-time system as networks of timed automatons. In this report, we present our modeling and verification of the game UppaalTD (both vanilla and stochastic versions) in Uppaal. In Addition, we also present the most interesting modeling choices we discarded or adjusted in the final version of the model.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Definitions

- **requirements** : the official specifications of UppaalTD's parameters and rules;
- **alive**, **targettable** or **shootable** enemy : the enemy has an health strictly greater than zero and is present on the map;
- **dismissed** enemy : the enemy is dead or it shot the MT and the following delay has expired;
- **configuration** : wave's composition and turrets configuration chosen (generalized definition from requirements one);
- **ending** of a wave : a wave is considered *ended* when each enemy is dismissed (i.e. it can't move or shoot anymore because it has already shoot or it was killed by turrets);
- **winning** configuration : in Vanilla version, a configuration is winning if, by setting it, a wave of 3 circles and 3 square can never defeat the MT. In addition, we define also this terminologies (also for Vanilla version with 3 circles and 3 squares):
  - **weakly-winning** configuration : a configuration that does not let MT to be defeat in at least one game execution;
  - **strongly-winning** configuration : a configuration that does not let MT to be damaged in any game execution;
  - **weakly-strongly-winning** configuration : a configuration that does not let MT to be damaged in at least one game execution;
- **location** : a location of an Uppaal template;
- **state** : the entire game's state in a certain time instant (i.e. each automaton's position and each variable's or clock's value);
- *Chebyshev distance* : given two generic n-dimensional points $x = (x_1 \dots x_n)$ and $y = (y_1 \dots y_n)$, their Chebyshev distance can be computed as $\max_{1 \leqslant i \leqslant n} \{|x_i - y_i|\}$.

1.2 Project development timeline

1.3 Document structure

Main sections:

1. **Introduction** : sum up of our definitions for the terminologies used in the document and version history of our model;
2. **Model description** : description of the models and focus on the most critical modeling choices;
3. **Verification results** : detailed analysis of the queries verified;
4. **Analysis of selected configurations** : presentation and deeper analysis of the models behaviors for some interesting configurations;
5. **Conclusions** : resume of the conclusions obtained.

Appendixes:

A **Discarded choices** : presentation of the most interesting discarded design choices and the motivations behind their rejection.

1.4 Software and machines used

Each query was verified used two different machines with different performances that here we briefly describe:

This document was written over the template Arsclassica Article (https://www.latextemplates.com/template/arsclassica-article) with few adjustments by us.

april 8th	First brief analysis of the requirements.
april 9th	Uppaal and homework presentation.
april 11th	First definition of channels and MT's template.
april 26th	First definition of enemy and turret template (still not synchronized) and shoot to MT modeled with STMTCONTROLLER.
may 1st	Complete definition of all templates and synchronization without STMTCONTROLLER.
may 15th	First definition of queries.
may 17th	Enemy compact version.
may 24th	Re-design of synchronization with clocks.
june 15th	First stochastic version.
june 24th	Started delivery procedures.
june 26th	Started report writing.
july 10th	Finished report writing.

Table 1: Project development timeline

Usage	Software	Versioning
Modeling, simulation and verification	Uppaal	5.0.0
Report writing	TeXstudio	4.6.3
Version	Git	2.40.0.windows.1

Table 2: Software used

Machine name	CPU	RAM	OS	Manufacturing year
Machine 1	AMD Ryzen 5 3500U (2.1 GHz)	8 GB (5,95 GB usable)	Windows 11 Home	2020
Machine 2	11th Gen Intel(R) Core(TM) i5-1135G7 (2.4 GHz)	24 GB (23,7 GB usable)	Windows 11 Pro	2021

Table 3: Machine specifications

2 MODEL DESCRIPTION

We modeled both the Vanilla and the Stochastic version of the game. In particular, we firstly modeled the Vanilla version and then proceeded to modify it to model the stochastic requirements. In this section we provide a detailed explanation of both versions and how the Stochastic one was obtained from the Vanilla one. We clarify that our main purpose in this section is to clarify and support our design choices rather then explain the technical aspect underlying the Uppaal code, which can be read from the templates comments.

2.1 Entities modeling (for Vanilla version)

2.1.1 *Map*

There is no template that models the Map. Each enemy and each turret has a `Cell` constant/variable that represents its actual position on the map (`Cell` is simply a `struct` with two bounded integers):

- an enemy keeps its position updated while moving with the function `next`. If an enemy is outside of the map (because it has not spawned yet or it was killed) its position still has a "feasible" value but a flag prevents turrets to read it (see the following sections for more details);
- a turret can't change its position, it simply uses it to calculate its distance from enemies.

To be clear, red paths structures are "embedded" into the function `next`. Another possibility (implemented in some of the previous versions) would have been to define a `Cell` matrix (or vectors) instead of define the entire structure as a less-readable the sequence of if-else implemented by `next`. We preferred this solution since `next` is constant in time and space however, with a more complex map, this approach could result in difficult to maintain `next` functions.

2.1.2 *Main Tower (MT)*

There is no template to model the Main Tower, it is simply modeled as a variable that enemies decrement in the shot. Originally, an MT template was designed (see the appendix for more details), it was removed in an attempt of optimizing the model, since we understood (after reasoning more on other more crucial design choices) that a so simple and "passive" entity like the MT does not really need to be implemented with a dedicated template.

Decoupling enemy and turret in two templates is necessary to model in a proper and clear way the interleaving between these entities but MT neither needs to trigger other components nor has any non-deterministic behavior. The only one centralized aspect that a dedicated template could have implemented was controlling that MT's life is not `0` before decreasing it, but this can be easily moved in enemy's template (before an enemy shoots, it checks that MT's life is not `0`) without loss of readability.

2.1.3 *Enemy*

Let's explain the design choices beyond each enemy's behavior.

Spawn of an enemy
To model the fact that circles spawn "every x time units" and squares spawn "every y time units" we simply added a parameter `spawningTime` to enemy template. This parameter is simply a delay that must first elapse before an enemy can spawn on the map. By setting it (when we instantiate enemy processes) to `id`×`s` (where `s` is the spawning time defined for that kind of enemy in the requirements), the enemy with id `0` will spawn in the first time unit, the one with id `i` with a slack of `s` time units and so on. Note that, as we explain in the communication section, enemy ids must not overlap each other, therefore, if N enemies are present in the wave, and M of them are circles:

	circle(0)	...	circle(M-1)	square(M)	...	square(N-1)
id:	0	...	M-1	M	...	N-1
`spawningTime`:	0	...	2×(M-1)	0	...	3×(N-1-M)

Table 4: `spawningTime` assignment (using requirements spawning times)

Initialization of an enemy (this paragraph is only to group all the aspects relative to the enemy that will be further explained)
Once an enemy is spawned, it has to:

- initialize its record in the `shoot_table` and update the counter of targetable enemies;
- reset the `trip_time` clock;
- set `chosenPath`.

Move of an enemy
After the speed delay has expired, an enemy has to make a move. In fact, a move is simply an update of the enemy's position with the function `next` and a consequent update of the `shoot_table` record (see the communication section).
The very interesting design choice behind the move of an enemy is how a (deterministic, since Uppaal `random` is not available in symbolic simulation) function like `next` can model the non-deterministic next cell choice an enemy takes in a red paths junction. Simply, the choices an enemy will take in the junctions is non-deterministically determined "a priori" when an enemy spawns ()by calling `initialize` with different parameters): We realized that since during the game there is no event that may change the probability that

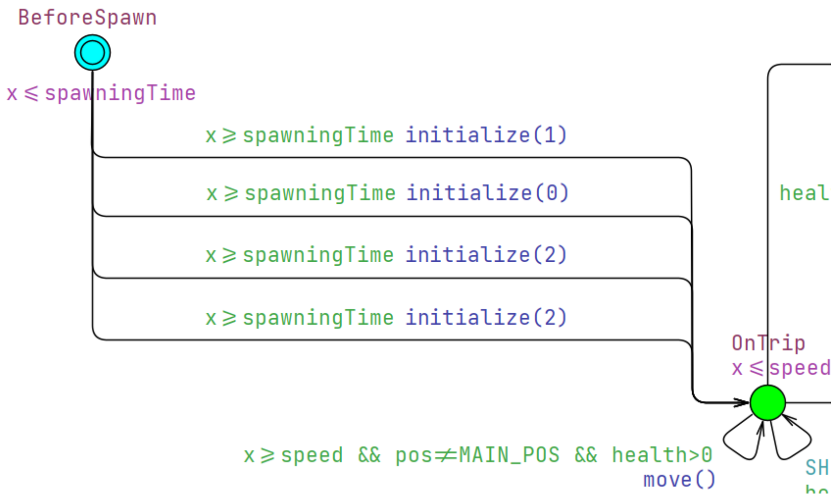

Figure 1: Close-up of how "a priori" non-deterministic path choice is implemented in the enemy template

an enemy takes a certain choice in a junction rather than another one, without loss of generality, these non-deterministic choices can be determined all at once by the time the enemy spawns.
This simple intuition really improved the readability of the enemy template (in the project development timeline we called this new version *compact enemy*) however, it can be convenient only if there are few possible choices.
Note that even if there exist three possible paths, it would be incorrect to model only one transition per-path. These transitions must model the same probability of taking any "sequence of choices". In other words, the probability of taking the choice "under" in the junction (7,4) has a 0.5 of probability to be taken, the other 0.5 is the probability of choosing "up" and then there is a 0.5 probability for each choice in the junction (10,7). If we want to determine these choices "a priori", it is not correct to model any possible path with a single transition, because it would lead the probability of taking any path to $\frac{1}{N}$ (where N is the number of possible paths to the MT).

Dismissal of an enemy Once an enemy has to leave the map, it simply updates properly the `shoot_table` record, the counter of the targetable enemies and the counter of the left enemies.
A match is considered ended if the counter of left enemies is zero (i.e. all enemies spawned and leaved the map). Once the enemy is dismissed, it goes in a location where it waits that the match ends and then goes in an endless self-loop, this choice is in fact more crucial than it might seem:

- it prevents the system deadlock once the match is ended;
- putting a self-loop with no guard would of course prevent deadlock but it would possibly lead the starvation of other enemies (an enemy may arrive in the location and then self-looping indefinitely so other enemies could not move/shoot anymore). With the guard, this situation may happen only once all enemies are dismissed (and so they can't "starve" anymore).

2.1.4 *Turret*

The high-level behavior of a turret (for the design of the communications see the following section) is pretty simple:

1. the turret (in the initial location) is ready to shoot to an enemy as soon as one of them gets into its shooting range (purpose of function `canShoot`). It check other enemies positions by looking into `shoot_table`;

2. once `canShoot` returns true the transition to shoot to an enemy is enabled. If it is performed, the function `target` selects the nearest enemy in the shooting range (in case of ties it follows the requirements rules), and with `shoot` the enemy is shot by putting in the global variable `target_record` the id of the target and the related damage;
3. once the shot is performed the turret waits the delay and then returns in the initial location.

`canShoot` and `target` are optimized to avoid useless scans of the `shoot_table`, in particular:

- `canShoot` :
  - does not selects the enemy to target but simply checks that there is at least one enemy in the shooting range (`target` has this purpose), which is average less complex;
  - while scanning the `shoot_table` it keeps counting how many targetable enemies are found (i.e. enemies that are alive on the map), so it can stop to scan it as soon as it recognizes that no targetable enemy can be found in the following records;
- `target` :
  - first `shoot_table` is scanned to find the first targetable enemy in the shooting range. Then it is considered as the candidate enemy for the shot and the following positions are scanned to find a possibly "better" enemy to shoot;
  - the scan is always stopped as soon as no targetable enemies can be found in the following records of `shoot_table`.

Note that the complexity of both `canShoot` and `target` is linear in the size `shoot_table` length but it is totally independent from the size of the shooting range.

Another improved that could be argued is to let `canShoot` pass the first enemy in the shooting range found to `target`, in this way it has already a candidate enemy. However, since `canShoot` is inside of a guard, it must be side-effects free, therefore it can't change a value outside of its scope.

2.2 Communication modeling description (for Vanilla version)

2.2.1 *How turrets shoot to enemies*

Each enemy has a record in the `shoot_table` structure that it keeps updated. This record contains all the information turrets need to identify which enemy shoot (position, time in which the enemy spawned, kind) and of course a flag to know if that enemy is present on the map.

Once an enemy is targeted, the turret places on the global variable `target_record` the id of the target and the related damage. In the meantime a message to all (targetable) enemies is sent over the broadcast channel `SHOOT_TO_ENEMY`, then each one checks if their id corresponds to the one of the target and if it is, they decrement their life accordingly to the damage and in case their health is now zero or below they leave the map.

An important design choice is that `SHOOT_TO_ENEMY` is an urgent channel. In our vision of the game, when a turret can shoot, time can't elapse. This does not avoid situations where the delay of an enemy is fully elapsed but the move was not performed and the move is performed while the "urgent" transition is enabled by a turret (because in this case the delay has already fully elapsed, so this update does not concern time) but avoids situations where a turret is ready to shoot and it is completely ignored progress of time. In other words, a shot to an enemy must be performed in the same time unit where it is "calculated". We remark that this design choice is not a definition of priority, since the shot to an enemy is not preferred to the other transitions.

2.2.2 *How enemies shoot to the MT*

Since there is no template for the MT, there is apparently no need to use define a channel to "synchronize" the shot since it is nothing more than a decrement of a global variable. However, we still designed an urgent broadcast channel to guarantee the once an enemy can shoot to the MT time can't elapse (the same reason why `SHOOT_TO_ENEMY` is urgent). If we want to make the shot "urgent" is necessary to define the channel as broadcast otherwise, since there is no entity that "receives" that message, a deadlock would happen if an enemy wants to shoot (the enemy is blocked because no entity can receive the message and since it is urgent it blocks the time all other available enemies).

2.3 Enrichment of Vanilla model with stochastic features

To model the stochastic features there are no big change from the Vanilla model but crucial:

- enemy move/after shot delay : it is no more modeled as with a clock but with a simple self-transition (which guard of course does not involve clocks) and `OnTrip` has `rate of exponential` set to `speed:10`. In this way the probability of leaving from the state with the self-transition determines the delay before an enemy's move. The same design is applied to the "after shot" delay);
- turret after shot delay : it is modified with the same principle of the enemy move/after shoot delay (clock is no more used and the rate of exponential is set);
- self-loop in `Dismissed` : [SPIEGARE COME MAI DAVVERO E' STATO RIMOSSO, DATO CHE NON CREDO NON SIA CONCESSO AVERE NESSUNO STATO SENZA INVARIANTI E SENZA RATE OF EXPONENTIAL].

The only delay that is still implemented as a "non-probabilistic" delay is the one related to the spawning time.

Note that there was no change in communication channels since all of them were broadcast also in the Vanilla version (stochastic model can only use broadcast channels) and the motivations behind making them `urgent` are independent from the stochastic nature of the model.

3 VERIFICATION RESULTS

3.1 Vanilla model verification

3.1.1 *Verification without turrets*

To verify the requested query we wrote the following Uppaal queries: Q1 must verify that the system never

	Verified properties	Result [T/F]	Verification time on Machine 1 [s]	Verification time on Machine 2 [s]	Average maximum past-waiting list load
Q1	I	T			
Q2	II	T			
Q3	III, IV	T			
Q4	V	T			

Table 5: Queries without turrets overview

reaches a deadlock state, our query is then simply:

[METTERE QUERY IN TCTL]

Originally, this query aimed only to verify deadlock avoidance only when the match is not ended. In fact, our model can still verify this query if we remove the self-loop on `Dismissed`. Informally, we can also argue that no deadlocks can happen even in this way: all enemies will eventually become `Dismissed` all together. Since once all enemies are `Dismissed` they start the to indefinitely take the self-loop:

[MAGARI METTERE ANCHE QUESTA QUERY IN TCTL]

Q2 aims to verify that all enemies can reach the MT spot:

contenuto...

We interpreted "can reach" in the sense that in any possible path all enemies will reach the MT, this is the reason way this query is not stated with "E...".

Another important motivation behind this formulation is that this query in reality verifies that exists state in any possible path where all enemy have their position in the MT spot all together. This condition is easy to guarantee in our model since the dismissal of an enemy keeps `pos` to the last value it had right before the dismissal. It is like we are "freezing" in the `pos` the fact that the enemy reached the MT spot, but at the end the model will permit to any enemy to reach the MT spot.

Q3 verifies both that each circle and each square satisfy the time constraint:

[METTERE QUERY IN TCTL]

Simply once an enemy is `OnTrip` and reaches the MT spot, it must have satisfied the time constraint for its kind. `trip_time` is a classic clock that counts the time units passed from the initialization (i.e. the time unit when the enemy spawned).

Note that this query would not be satisfied in this form if `SHOOT_TO_MT` was not `urgent`, since this property imposes that time (and so also `trip_time`) can't progress as long as the MT spot in reached in `OnTrip`.

Q4 is probably the most intuitive query:

[METTERE QUERY IN TCTL]

In any state of any possible path, if an enemy is targetable it must be on a red spot.

Note that since `pos` is by default `0,0` before spawning and it is kept to the MT spot once it is reached, even if an enemy is not targetable anymore it will anyway be in a red spot. We put this restriction to the "query scope" because there is no need to check `pos` if an enemy is not present on the map.

3.1.2 *Verification with turrets*

To verify the requested query we wrote the following Uppaal queries:

Q1 is used also for verifying VII since the total absence of deadlocks, in symbolic simulation, can be stated by means of the keyword `deadlock`.

	Verified properties	Result [T/F]	Verification time on Machine 1 [s]	Verification time on Machine 2 [s]	Average maximum past-waiting list load
Q1	VII	T			
Q5	VI	T			

Table 6: Queries with turrets overview

Since we defined a "winning" configuration as a configuration that can never let the MT to be defeated by a wave of six (three circles and three squares) enemies, a query that is true if and only if a configuration is winning is:

mettere query in TCTL

If the MT is defeated in at least one state of one path, the query will not be satisfied (with the configuration chosen).

Note that in the configurations section we also analyze the other kinds of "winning" of them. For brevity, we do not report those queries but they are really similar to Q1 (e.g. to verify the weakly-winning property, it is sufficient to verify the query [METTERE QUERY IN TCTL]).

3.2 Stochastic model verification

4 ANALYSIS OF SELECTED CONFIGURATIONS

4.1 Vanilla version

We further analyzed the default configuration with turrets and four more configurations: First we analyzed

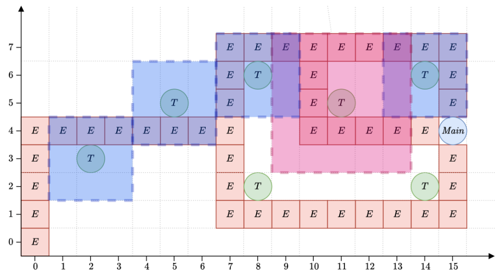

Figure 2: *DownFromCannons* configuration

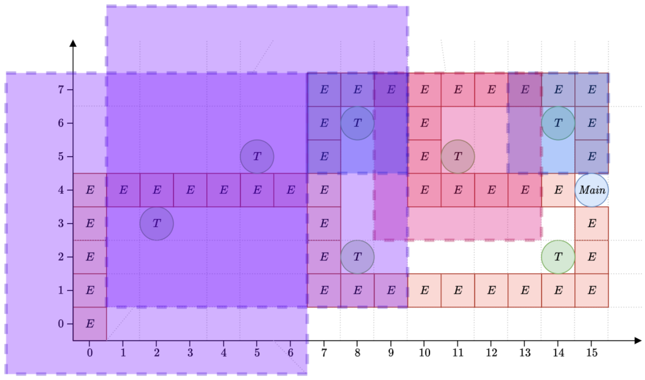

Figure 3: *DownFromSnipers* configuration

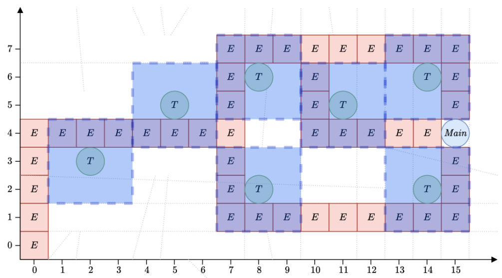

Figure 4: *Cannonphobia* configuration

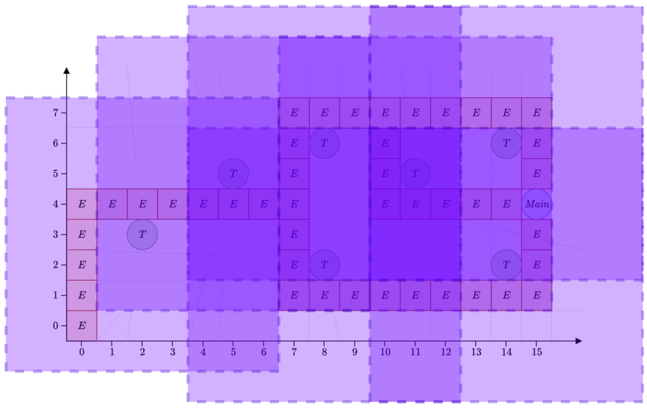

Figure 5: *Sniperphobia* configuration

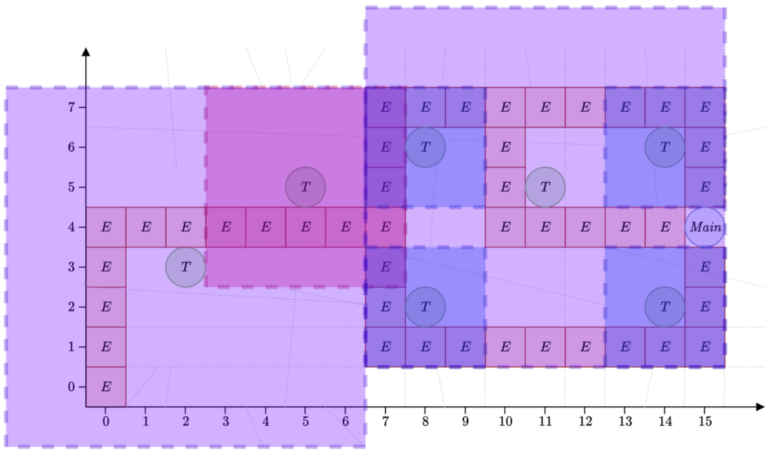

Figure 6: Default configuration

if they are winning and of which kind:

Then (mainly to strain more the model and to explore the Uppaal query syntax), we also wanted to define these metrics to evaluate the "strength" of a configuration (we assume that, like in Uppaal, a boolean predicate is evaluated as 1 if it is *true*, 0 otherwise):

- *ADMT* (*Average Distance from MT*) : it is the minimum average distance that enemies can have to the MT:

	Winning [T/F]	Weakly-winning [T/F]	Strongly-winning [T/F]	Weakly-strongly-winning [T/F]
Default	T	T	F	T
DownFromCannons	F	T	F	T
DownFromSnipers	F	T	F	T
Cannonphobia	T	T	F	T
Sniperphobia	T	T	T	T

Table 7: Vanilla chosen configurations winning overview

$$\mathsf{ADMT} := \min\{\frac{\sum_{e:Enemy} \mathtt{dist}(e.pos, \mathsf{MAIN\_POS})}{\mathsf{MAX\_ENEMIES}}\}$$

- *SE* (*Survived Enemies*) : it is the maximum number of enemies that still were never be killed throughout the whole match:

$$\mathsf{SE} := \max_{matchEnded}\{\textstyle\sum_{e:Enemy}(e.health > 0)\}$$

- *CUPE* (*Completed Upper Paths Enemies*) : it is the maximum number of enemies that reached the MT by choosing "up" in the junction {7,4}:

$$\mathsf{CUPE} := \max\{\textstyle\sum_{e:Enemy}((e.pos == \mathsf{MAIN\_POS}) * (e.chosenPath \neq 2))\}$$

- *CLPE* (*Completed Lower Paths Enemies*) : it is the maximum number of enemies that reached the MT by choosing "down" in the junction {7,4}:

$$\mathsf{CLPE} := \max\{\textstyle\sum_{e:Enemy}((e.pos == \mathsf{MAIN\_POS}) * (e.chosenPath = 2))\}$$

	ADMT	SE	CUPE	CLPE
Default	4	2	2	1
DownFromCannons	1	5	2	5
DownFromSnipers	0	6	2	6
Cannonphobia	4	2	2	2
Sniperphobia	7	0	0	0

Table 8: Vanilla chosen configurations performances

Note that:

- strongly-winning configurations have a SE of 0 (if an enemy has survived it means that it shot the MT, and so the configuration can't be strongly-winning);
- weakly-winning configurations are likely to have lower ADMT since at least of the execution where the MT is defeat, an amount of enemies must arrive at in the MT spot and for the same reason, they are more likely to have higher SE;
- is quite unlikely to have a configuration where $\mathsf{CUPE} > \mathsf{SE}$ or $\mathsf{CLPE} > \mathsf{SE}$, because enemies that reached the MT are really likely to survive (and they will survive of course in configurations like Cannonphobia where the MT spot is outside of any shooting range).

At the end, to judge a configuration we may look for the one with an higher ADMT and a lower SE. CUPE and CLPE and are not better than SE to compare configurations, we defined them to demonstrate that configurations that do not cover parts of the map are more like to see (from the MT spot point of view) enemies coming from those parts. Indeed, DownFromCannons and DownFromSnipers have $\mathsf{CUPE} \leqslant \mathsf{CLPE}$ since they cover more the higher paths of the map, while more "uniform" configurations tend to have $\mathsf{CUPE} = \mathsf{CLPE} = \mathsf{SE}$ (that holds also from a probabilistic point of view; see the section dedicated to the spawn strategy).

4.2 Stochastic version

In this section we aim to revise the performance indexes defined in the previous section by simulating the game in the stochastic version.

First, we simulate (with 10 runs) the trend of SE in three of the four configurations we defined: Surprisingly, Sniperphobia, which was (the only) strongly-winning configuration in the Vanilla version tends to have a worse SE than the others. We see that Cannonphobia tends to have an $\mathsf{SE} \in [620; 651]$ (even worse than

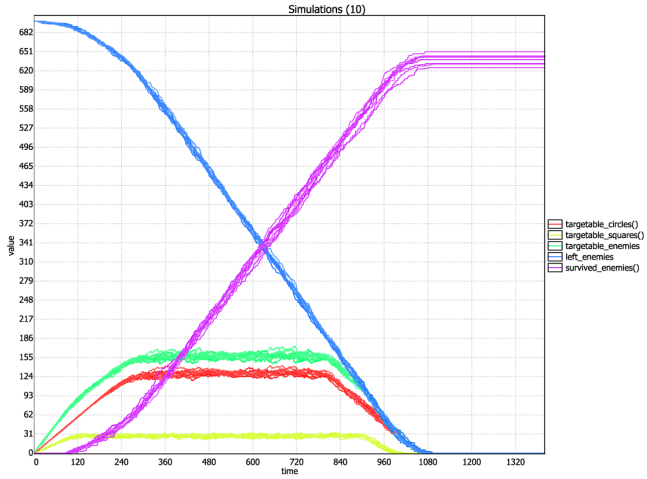

Figure 7: Sniperphobia SE simulation with 10 runs

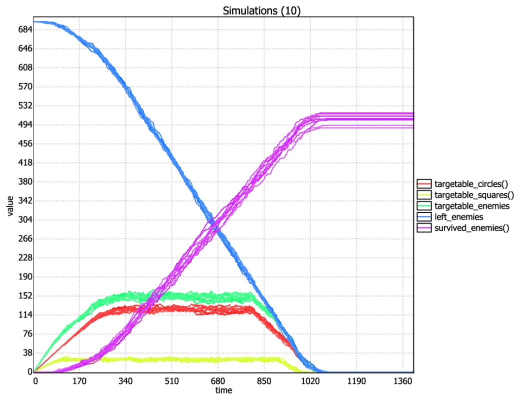

Figure 8: Cannonphobia SE simulation with 10 runs

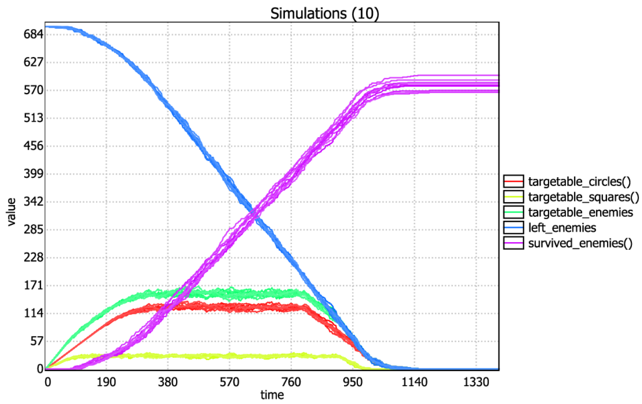

Figure 9: DownFromSnipers SE simulation with 10 runs

the one of DownFromSnipers which was a weakly-winning configuration) while Sniperphobia tends to have an SE $\in [490; 520]$. However, if we analyze the probability distribution of the SE in first 100 of time units we see better performances with winning and strongly-winning configurations: With a large amount of

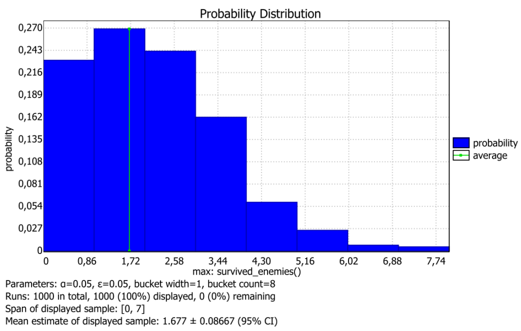

Figure 10: Sniperphobia SE probability distribution in the first 100 time units with 1000 runs

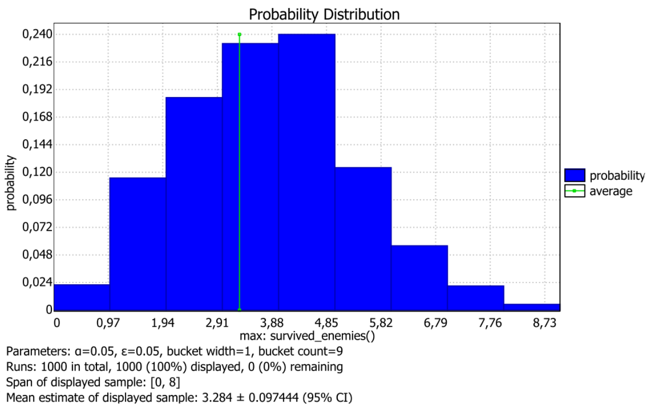

Figure 11: DownFromSnipers SE probability distribution in the first 100 time units with 1000 runs

enemies, turrets with higher delays generally tend to have the worse performances. In fact, if we take the delay of snipers down of 10: We wanted to analyze this kind of unexpected behaviors (i.e. strongly-winning configurations may have a worse SE with large waves) not only to show the trivial evidence that the lower turret delay, the better performances, but to show that a stochastic model checking can reveal behaviors that with exhaustive model checking would have been too heavily computationally complex to verify. With stochastic model checking we were able to show that in larger waves, a turret kind with a lower damage but a lower firing speed tends to provide better performances.

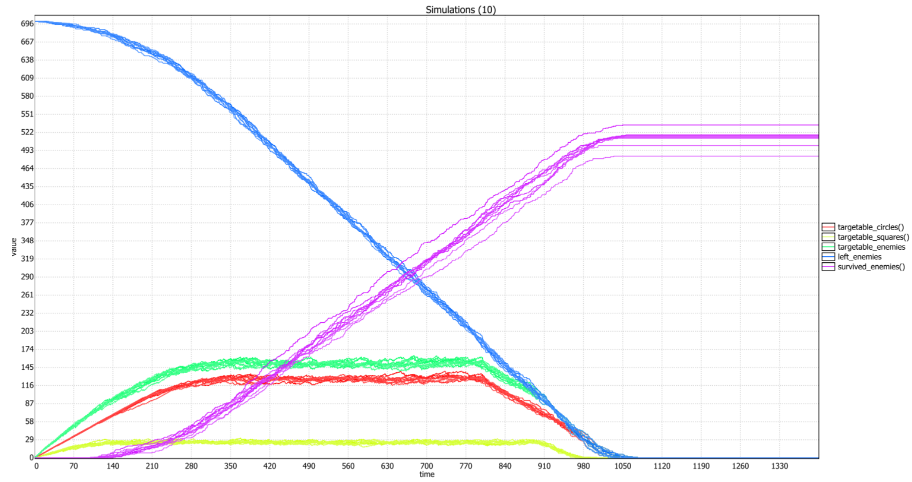

Figure 12: Sniperphobia SE simulation with 10 runs (snipers delay set to 10)

5 CONCLUSIONS

Conclusions.

A DISCARDED CHOICES

We wanted to write this additional section mainly to better clarify the reasons behind our final design choices. Nonetheless, we would like to clarify that our design choices are what we believed to be more efficient and adequate for our interpretation of the game, so some of the discarded can possibly be the best ones in other contexts or with different requirements and also for this reason we wanted to state them in the report.

A.1 MT template

Originally, a template for MT was designed. It was, for its simplicity, the very first one to be designed (`decDamage` decreases MT's life by the value set in a global variable from the firing enemy):

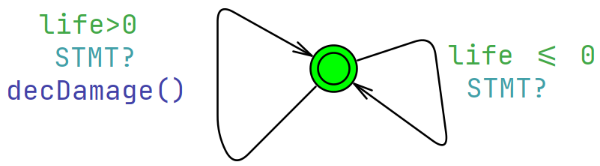

Figure 13: Original MT's template

A.2 Hard-coded enemies paths

The very first enemy version used the concept of `next` but there was no idea on how to model the non-deterministic moves (unless using function `random`, which is not available in symbolic simulation). The only idea was to hard-code vectors of cells representing each straight red path on the map and enemies template would have chosen between them non-deterministically with transitions (once an enemy arrives in the last cell of a path, then it will start to follow non-deterministically one of the "next paths"):

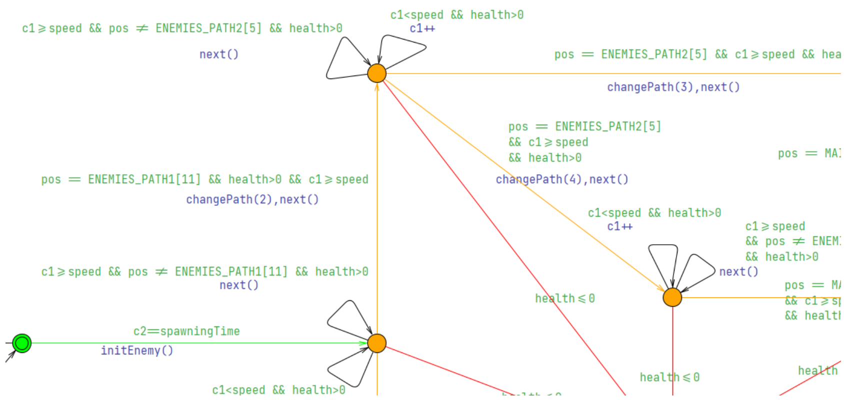

Figure 14: Original non-deterministic path choices example

A.3 Quadratic enemies scanning strategy

The very first turret version used to look for enemies to shoot in this way: for each k from 1 to the `range`, scan `SHOOT_TABLE` to find all the available enemies at exactly k cells of distance; then, choose the best iso-distant available enemy based on the requirements criteria. The worst-case asymptotic complexity of this procedure was $\Theta(k \times MAX\_ENEMIES)$ which was significantly worse than the one of the final version which is $\Theta(MAX\_ENEMIES)$ especially in the average case where there is no enemy inside the shooting range.

A.4 Locks

At the beginning, the first way of synchronizing entities was thought in a classical lock-unlock manner: Once an enemy reaches the MT (for the MT template please see the proper section of the appendix):

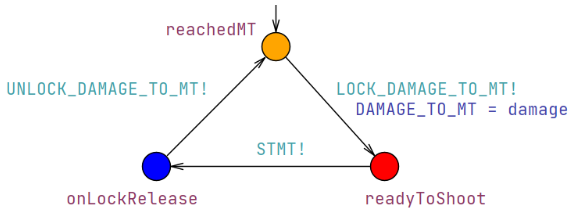

Figure 15: Close-up of the enemy locking STMT channel

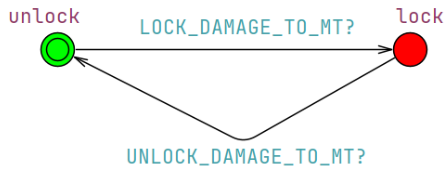

Figure 16: STMTCONTROLLER template

1. the enemy sends a message to STMTCONTROLLER and waits for its reply (more precisely, Uppaal chooses non-deterministically which one of the ready enemies can send the message to the controller);
2. once the controller replies the enemy places in the shared variable the damage for the MT;
3. the enemy shoots to the MT (i.e. sends a message over STMT);
4. the enemy sends a message to STMTCONTROLLER to release the "lock".

We understand that this solution is:

- deadlock-free: soon or later an acquired lock will be released and soon or later a lock request will be accepted;
- not starvation-free: since it is not guaranteed that any enemy that requests a lock will eventually obtain it.

We removed this concept since we understood that a single transition that both changes the global variable and performs the shoot would have produced the same behavior, since, provided that this transition is not synchronized with other enemies, only one of them can perform it in a time instant, therefore there is no possibility that an enemy places the damage for the MT into the shared variable and before sending the shooting message another enemy changes the variable and (or not) shoots to the MT (which would clearly create an undesired behavior).

A.5 Lifetime counter

As we have seen, turrets understand that an enemy is present on the map for a shorter amount of time by looking to its spawning. This idea lets enemies to not keep a counter updated to tell the turrets that they are present on the map for tot. time units. However, this idea is relatively new in the history of the model, since at the beginning, the lifetime of an enemy used to be update in each time unit by the enemy:

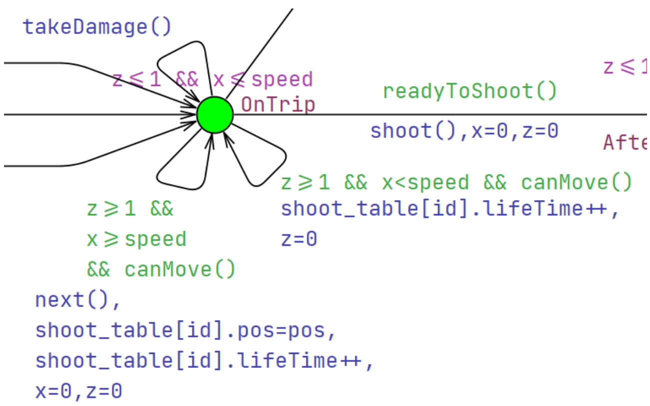

Figure 17: Close-up of of the lifetime counter of an enemy

With clock z at each time unit the lifetime counter would have been updated.

Note that this solution was also used to verify that enemies would have reached the MT spot in no more than MAX_PATH_LENGTH $\times$ speed time units.

REFERENCES

[1] P. San Pietro A. Manini. “The Uppaal Tool”. 2005. URL: https://webeep.polimi.it.

[2] K. G. Larsen G. Behrmann A. David. “A Tutorial on Uppaal 4.0”. In: (2006). URL: https://uppaal.org/texts/new-tutorial.pdf.

[3] I. Lee. *UPPAAL tutorial*. 2009. URL: https://www.seas.upenn.edu/~lee/09cis480/lec-part-4-uppaal-input.pdf.

[4] L. Lestingi. “Statistical Model Checking and Uppaal SMC”. 2021. URL: https://webeep.polimi.it.

[5] F. Corradini M. Bernardo. *Formal Methods for the Design of Real-Time systems*. 2004. URL: https://link.springer.com/book/10.1007/b110123.

[6] P. San Pietro. “Lecture slides of FORMAL METHODS FOR CONCURRENT AND REAL-TIME SYSTEMS course”. 2025. URL: https://webeep.polimi.it.

[7] A. Maggiolo Schettini. *Veryfing Real–Time Systems - The Uppaal Model Checker*. 2007. URL: http://groups.di.unipi.it/~maggiolo/Lucidi_TA/VerifyingTA-Uppaal.